Design – Linux-Kernel

Frieder Bürzele
HfT Stuttgart
Studiengang Informatik
Wintersemester 2007/2008

31. Dezember 2007

Inhaltsverzeichnis

1 Einleitung

Auch wenn es nicht allen bewusst ist, nutzen viele Menschen in irgendeiner Form das Betriebssystem "GNU/Linux"[1]. In vielen elektronischen Geräten läuft eine angepasste Version des Linux-Kernel[2]. Ursprünglich für die i386er[3] Architektur entwickelt und mit relativ wenig Funktionalität wurde innerhalb der letzten 16 Jahren der Linux-Kernel ständig weiter entwickelt und auf etliche weitere Architekturen portiert. Möglich wurde diese Weiterentwicklung aufgrund der freien Verfügbarkeit des gesamten Quellcodes unter den Bedingungen der GPLv2-Lizenz[4]. (Diese Lizenz erlaubt die kommerzielle Nutzung. Wichtigste Bedingung: Anpassungen an Quellcodes müssen, sobald ein Produkt auf den Markt eingeführt wird, genauso wie die ursprünglichen Quellcodes zugänglich sein.)

Heute findet ein meist modifizierter Linux-Kernel[5] in nahezu jedem Bereich seine Anwendung: Beispielsweise im Server-Bereich, auf dem Desktop, auf Mainframes, als Hochleistungs-Cluster, in Set-Top-Boxen, Routern oder auch in Handys. Dabei ist erstaunlich, wie gut der Linux-Kernel auf all diesen Systemen skaliert bzw. funktioniert und wie einfach es dank frei zugänglichen Quellcodes für jeden möglich ist, den Kernel anzupassen. Aufgrund der vielen Funktionalitäten und Möglichkeiten, die der Linux-Kernel bietet, scheint er als Ganzes recht komplex. Ein Beispiel, das von außen enorm komplex aussieht, ist die Unterstützung von mehr als 60 Dateisystemen innerhalb des Linux-Kernel – intern wird dank einer Abstraktionsschicht die Komplexität in die einzelnen Implementierungen der Dateisysteme verlagert. Dank des Design-Konzeptes, der Unterteilung in Subsysteme, der Benutzung von GNU-C[6] [Lov05, p.47] als Programmiersprache, der Kapselung architekturrelevanter- bzw. performancekritischer Funktionen in Inline-Assembler[7] Makros, machen den Einstieg in die Kernel-Programmierung und die Implementierung eigener Funktionalitäten einfacher. Anhand einiger wichtigen Konzepte wird gezeigt, wie relativ einfach neue Funktionalitäten hinzugefügt werden können und welche Mechanismen der Linux-Kernel zur Verfügung stellt, um dieses Vorhaben zu erleichtern.

[1] GNU/Linux deshalb, da Linux nur den Kernel des Betriebssystems darstellt und nur mit Hilfe von Programmen aus dem GNU-Projekt `http://www.gnu.org` ein lauffähiges System bildet.

[2] ein Kernel verwaltet die Ressourcen eines Computers

[3] Prozessorarchitektur, die nach wie vor (2007) am meisten verbreitete Architektur für Desktop Computer

[4] GNU General Public License, version 2

[5] als nicht modifiziert wird das Linux-Kernel Quellcodes-Verzeichnis (`http://www.kernel.org`) von Linus Torvalds genannt, der den Linux-Kernel Anfang der 1990er entworfen und realisiert hat. Bis heute entwickelt er aktiv mit.

[6] spezielle Erweiterungen der GNU-Compiler-Collection (GCC) zu C

[7] Assembler-Quellcodes werden direkt in C-Quellcodes eingebettet

2 Hauptteil

2.1 Einführung

Im folgenden werden ausgewählte Konzepte bzw. Funktionalitäten, die der Linux-Kernel implementiert, vorgestellt. Durch das Verstehen dieser Konzepte sind eigene Funktionalitäten einfacher und mit weniger Aufwand zu realisieren. Die Abbildung 1 zeigt eine Übersicht, über alle Teilbereiche des

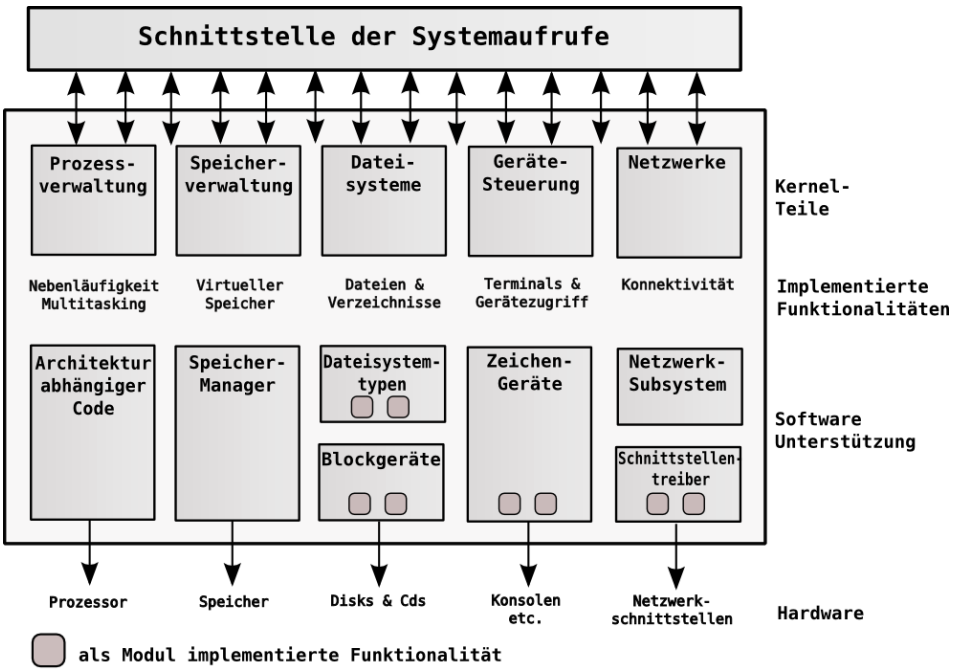

Abbildung 1: Aufgabenblöcke des Linux-Kernels [vgl. RC02]

Linux-Kernel. Die Aufgabenblöcke, wie Speicher-Manager, Dateisystemtypen oder Block-Geräte, werden Subsysteme genannt:

Logisch zusammengehörende Teilsysteme des Kernels sind in Subsysteme untergebracht. Ein Beispiel für ein Subsystem ist das Netzwerk-Subsystem. Praktisch heißt das, dass die Quellcodes für alle Netzwerk-Treiber sich in einem gemeinsamen Quellcode-Verzeichnis befinden: `linux-2.6.23/drivers/net/`. Treiber mit größerem Umfang sind in eigenen Unterverzeichnissen. Die Treiber teilen sich gemeinsam nutzbare Quellcodes. Systemaufrufe[8] bilden die Schnittstelle zu Anwendungen im User-Space[9]

[8] Systemaufrufe sind definierte Schnittstellen, die ein Programm, aus dem User-Space, aufrufen kann. Der Kernel führt dann im Auftrag von diesem Programm eine, zu einem Systemaufruf korrespondierende und im Kernel definierte, Routine aus.

[9] Als User-Space wird der Ausführungsraum für eine Anwendung bezeichnet. Dieser

Hier wird jedoch nur ein kleiner Teil des Linux-Kernel behandelt, weiterführende Literatur ist in der Literaturliste am Ende aufgeführt.

Ein Hinweis: Eine Referenz, zu den vollständigen Quellcodes des Linux-Kernel, wird immer als Pfadangabe dargestellt: `linux-2.6.23/`[10] ist das Verzeichnis welches die ganzen Quellcodes des Linux-Kernel enthält.

2.2 Ressourcenverwaltungsmechanismen

2.2.1 Speicherverwaltung

Die Speicherreservierung im Kernel-Kontext verhält sich etwas anders als im User-Space. Der Linux-Kernel verwendet die physikalische Speicherseite als die grundlegende Einheit zur Speicherverwaltung. Linux stellt einige Low-Level-Funktion zur Verfügung um Speicherseiten zu reservieren:

```
struct page *alloc_pages(gfp_t gfp_mask, unsigned int order)
unsigned long __get_free_pages(gfp_t gfp_mask, unsigned int order)
```

Mit diesen beiden Funktionen ist es möglich, Speicher in Speicherseitengröße zu reservieren. Statt der aus der C-Sprachbibliothek für den User-Space bekannten Funktion `malloc()`, gibt es im Kernel die ähnliche Funktion `kmalloc()`, die auf den vorher vorgestellten Low-Level-Funktionen aufbaut. Mit dieser Funktion kann byteweise Speicher reserviert werden. Im Gegensatz zu `malloc()` gibt `kmalloc()` einen Zeiger auf eine zusammenhängende physikalische Speicherstelle zurück. Der Linux-Kernel stellt die Funktion `vmalloc()` zur Verfügung, um mit deren Hilfe Speicher für User-Space-Prozesse zu reservieren.

Um reservierten Speicher wieder freizugeben, stellt der Kernel auch einige Funktionen zur Verfügung:

```
void free_pages(struct page *page, unsigned int order)
void __free_pages(struct page *page, unsigned int order)
void free_page(unsigned long addr)
void kfree(const void *x)
```

Die vorgestellten Funktionen zur Speicherreservierung haben gemeinsam, dass über den Parameter `gfp_mask` [Lov05, p.251] bestimmte Bit-Flags[11], die sogenannten gfp_mask_flags, zu übergeben sind. Diese Bit-Flags geben beispielsweise an, ob die Funktion bei der Speicherreservierung vom Prozessor suspendiert werden darf, mit welcher Priorität Speicher reserviert werden soll oder ob für einen User-Space-Prozess, Speicher reserviert wird.

Ausführungsraum hat weniger Rechte als der Kernel (Kernel-Space), beispielsweise ist der direkte Zugriff auf die Hardware nicht erlaubt. Ein Beispiel für ein User-Space-Programm, ist ein Webbrowser.

[10] Die Version 2.6.23 ist die aktuelle (Dezember 2007) stabile Version des Linux-Kernels

[11] eine Variable wird auf gesetzte Bits überprüft: Für ein gesetztes Bit wird dann eine bestimmte zugeordnete Funktionalität aktiviert

Mit diesen vorgestellten Funktionen und den dazugehörenden Flags (gfp_mask_flags), ist es möglich, die selben Funktion für die Speicherreservierung im ganzen Kernel zu nutzen und je nach Bedarf anzupassen.

2.2.2 Synchronisierung

Der Linux-Kernel stellt einige Methoden zum Locking[12] zur Verfügung, also zur Vermeidung von Deadlocks[13] und Race-Conditions[14]. Locking bezeichnet den Schutz, der es ermöglicht, der alleinige Nutzer einer bestimmten Ressource, beispielsweise der Speicherstelle einer Datenstruktur, zu sein. Deadlocks können entstehen, wenn zum Beispiel zwei Prozesse auf die jeweils von dem anderen Prozess gelockte Ressource wartet. Als Race-Condition wird im Allgemeinen eine Situation bezeichnet, die auftritt, wenn zwei Prozesse gleichzeitig auf die gleichen Daten zugreifen. Die Folge wären inkonsistente Daten oder defekte Datenstrukturen. Ein einfaches Beispiel ist die Inkrementierung eines Zählers: Prozess 1 liest von einer Speicherstelle, gleichzeitig (bei Systemen mit mehr als einem Prozessor bzw. durch vorherige Suspendierung des Prozesses 1 durch den Prozessor-Scheduler[15]) liest Prozess 2 von der gleichen Speicherstelle. Beide inkrementieren um eins, aber das Ergebnis ist nicht die Erhöhung um insgesamt 2 Zähler, sondern nur um einen, da die Daten nicht vor dem gleichzeitigen Zugriff geschützt wurden. Um dies zu vermeiden, stellt der Linux-Kernel folgende Mechanismen zur Verfügung:

- **Spinlock**:
  Spinlocks bieten die Möglichkeit eine Ressource zu locken. Will ein anderer Prozess auf dieselben Daten zugreifen, muss dieser solange "aktiv Warten" (spin) bis der Prozess, der den Lock hält, diesen wieder frei gibt. Spinlocks werden verwendet, wenn sicher gestellt ist das der Lock sehr schnell wieder freigegeben wird (deterministisch), da der wartende Spinlock ständig nachfragt und so einen Overhead[16] verursacht. Spinlocks werden in einem Kontext, verwendet, in welchem ein Prozess nicht in einen Ruhezustand versetzt werden darf, zum Beispiel in einem Interrupt-Kontext. Die zu verwendenden Schnittstellen sind in `linux-2.6.23/include/linux/spinlock.h` definiert. Die einfache Benutzung eines Spinlocks sieht folgendermaßen aus:

  ```
  spinlock_t mein_lock;
  mein_lock = SPIN_LOCK_UNLOCKED;  /* spinlock initialisieren */
  spin_lock(&mein_lock);           /* Lock (Sperre) setzen */
  ```

[12] dt.: Sperren
[13] dt.: Verklemmung
[14] dt.: Wettlaufsituation
[15] eine Funktion des Kernels. Diese plant wie lange jeder Prozess den Prozessor verwenden darf
[16] dt.: Mehraufwand

```
/* Implementation eines kritischen Abschnitts ... */

spin_unlock(&mein_lock);          /* Lock (Sperre) freigeben */
```

- **Semaphore**:
  Semaphore ermöglichen ebenfalls das Locken von Ressourcen. Ist eine Ressource gelockt und will ein weiterer Prozess auf diese Ressource zugreifen, wird der anfragende Prozess, statt, wie bei den Spinlocks aktiv zu warten, in einen Ruhezustand versetzt und nach Freigabe der Ressource wieder aktiviert. Ein Semaphore wird durch die `struct semaphore`-Struktur repräsentiert und in `linux-2.6.23/include/asm/semaphore.h` definiert. Ein Beispiel für die Benutzung eines Semaphores:

```
/* ein Semaphore definieren und deklarieren */
static DECLARE_MUTEX(mein_semaphore);

/* Versuch das Semaphore zu erhalten ... */
if(down_interruptible(&mein_semaphore)){
    /* der Versuch das Semaphore zu erhalten
     * ist gescheitert. Fehlerbehandlung ...
     */
}

/* Implementation eines kritischen Abschnitts ... */

/* Das angegebene Semaphore wieder freigeben */
up(&mein_semaphore);
```

Viele Synchronisierungsprobleme treten nur bei Computer mit mehr als einem Prozessor auf. Grundsätzlich sollte deshalb, um Probleme zu vermeiden, immer von Computer mit mehr als einem Prozessor ausgegangen werden.

Um die Atomarität[17] sicherzustellen, sind Teile der vorgestellten Methoden mit Hilfe von architekturspezifischer Hardwareunterstützung implementiert. Durch die Kapselung, innerhalb dieser Methoden, bietet Linux eine einheitliche Struktur, um Deadlocks bzw. Race-Conditions wirkungsvoll zu vermeiden. Somit entfällt die aufwändige Implementierung eigener Mechanismen.

[17] eine Aktion wird entweder ganz oder gar nicht durchgeführt

2.2.3 Interruptbehandlung

Mit Hilfe von Interrupts (Unterbrechungen) kann die Hardware mit dem Prozessor kommunizieren. Ein einfaches Beispiel ist die Eingabe von Zeichen über die Tastatur: Die Eingabe wird von dem Steuerungschip der Tastatur bemerkt. Der Steuerungschip meldet durch ein elektrisches Signal (Interrupt) dem Prozessor, dass neue Zeichen zur Verarbeitung bereitstehen. Der Prozessor reagiert auf den Interrupt und ruft die entsprechende, vom Betriebssystem definierte Routine auf, den Interrupt-Handler.

Interrupt Handler sind in zwei Bereiche aufgeteilt, nämlich erstens in die Top-Halves (obere Hälfte) und zweitens in die Bottom-Halves (untere Hälfte)

- **Top-Halves**:
  Hier wird der unmittelbare, durch die Hardware ausgelöste, Interrupt behandelt. Daten, beispielsweise eines Geräte-Puffers, werden in den Hauptspeicher geschrieben. Diese Verarbeitung sollte so schnell wie möglich und möglichst deterministisch sein. Die Top-Half ist vollständig im Interrupt-Handler implementiert.

- **Bottom-Halves**:
  Die in der Top-Half angelegten Daten werden verarbeitet. Hier ist es nicht so wichtig, wie in der Top-Half, das die Abarbeitung so schnell wie möglich durchgeführt wird.

Das Auftreten eines Interrupts ist asynchron und der Prozess, der gerade auf dem Prozessor läuft, wird unterbrochen. Aufgrund dieser ungewollten Unterbrechung wird der Interrupt-Handler in Top und Bottom aufgeteilt. Bottom-Halves werden zu einem günstigeren Zeitpunkt in der Zukunft ausgeführt.

Der Linux-Kernel stellt folgende Mechanismen zur Bearbeitung von Bottom-Halves zur Verfügung, nämlich Softirqs, Tasklets und Work-Queues:

- **Softirqs**:
  Softirqs werden statisch zum Zeitpunkt der Übersetzung des Kernel-Quellcodes belegt. Sie werden durch diese, in `linux-2.6.23/include/linux/interrupt.h` definierte, Struktur dargestellt:

```
struct softirq_action
{
    void (*action)(struct softirq_action *);
    void *data;
};
```

Diese Struktur wird in einem Array mit 32 Einträgen in `linux-2.6.23/kernel/softirq.c` initialisiert:

```
static struct softirq_action softirq_vec[32];
```

Der erste Eintrag `*action` in der Struktur repräsentiert einen Funktionszeiger auf die Softirq-Handler Funktion. Die Definition des Prototyps sieht folgendermaßen aus:

```
void softirg_handler(struct *softirq_action);
```

Der zweite Eintrag `*data` der Struktur repräsentiert die Daten, die an den Softirq-Handler übergeben werden sollen. Anstatt, wie anzunehmen `data` zu übergeben, wird die ganze Struktur übergeben. Somit wird ein Umschreiben [Lov05, p.143] des Quellcodes vermieden, falls es Änderungen an der Struktur geben sollte. (Diese Technik wird in vielen Bereichen des Linux-Kernels verwendet.):

```
mein_softirq->action(mein_softirq);
```

Um einen Softirq-Handler ausführen zu können, muss dieser vorher, meistens durch den Interrupt-Handler (Top-Half), markiert werden. Der `ksoftirqd`-Kernel-Thread führt dann, zu einem späteren günstigen Zeitpunkt, diesen markierten Softirq-Handler aus.

- **Tasklets**:
  Tasklets bauen auf den Softirqs auf. Tasklets werden von zwei Softirqs repräsentiert. `HI_SOFTIRQ` und `TASKLET_SOFTIRQ`. Sie unterscheiden sich nur in der Priorität. `HI_SOFTIRQ` hat eine höhere Priorität. Ein Tasklet wird durch die folgende Struktur, ebenfalls in `linux-2.6.23/include/linux/interrupt.h` definiert, repräsentiert:

```
struct tasklet_struct
{
    struct tasklet_struct *next;  /* nächstes Tasklet in der Liste */
    unsigned long state;          /* Zustand des Tasklets */
    atomic_t count;
    void (*func)(unsigned long);  /* Zeiger: Tasklet-Handler Funktion */
    unsigned long data;           /* Parameter für Tasklet-Handler */
};
```

Im Gegensatz zu Softirqs können Tasklets sowohl statisch – bei Übersetzung des Quellcodes – als auch dynamisch zur Laufzeit erzeugt werden. Um ein Tasklet statisch zu erzeugen und zu initialisieren, stellt der Linux-Kernel 2 Makros zur Verfügung.

```
#define DECLARE_TASKLET(name, func, data) \
struct tasklet_struct name = { NULL, 0, ATOMIC_INIT(0), func, data }

#define DECLARE_TASKLET_DISABLED(name, func, data) \
struct tasklet_struct name = { NULL, 0, ATOMIC_INIT(1), func, data }
```

Der einzige Unterschied zwischen den beiden Makros ist, dass `DECLARE_TASKLET_DISABLED`, das Tasklet nicht aktiviert. Um eine dynamisch erzeugte `tasklet_struct` Struktur zu initialisieren, wird folgende Funktion verwendet:

```
void tasklet_init(struct tasklet_struct *t,
        void (*func)(unsigned long), unsigned long data);
```

Nachfolgend zu statisch und dynamisch jeweils ein Beispiel:

statisch:
`DECLARE_TASKLET(mein_tasklet_name,mein_tasklet_handler,meine_daten);`

dynamisch:
`tasklet_init(mein_tasklet_zeiger,mein_tasklet_handler,meine_daten);`

Es stehen noch ein Handvoll Funktion zum Einplanen, Aktivieren und Deaktiveren von Tasklets bereit. Ausgeführt werden die Tasklets, da sie ja auf den Softirqs aufbauen, durch `ksoftirqd`-Kernel-Thread.

- **Work-Queues**:
  Die Work-Queue verfolgt einen anderen Ansatz als die bisher vorgestellten Mechanismen. In dieser Bottom-Half wird die Arbeit entweder mit Hilfe eines neu erzeugten eigenen Worker-Typs oder in dem so genannten *event*-Worker-Typ abgearbeitet.

  Dieser standardmäßig zur Verfügung stehende *event*-Typ, wird durch die `workqueue_struct`-Struktur repräsentiert. Diese Struktur enthält ein Array von `cpu_workqueue_struct`-Strukturen. Die Größe des Arrays hängt von der Anzahl der vorhandenen Prozessoren eines Systems ab. Es gibt eine `workqueue_struct`-Struktur pro Typ. Für die zurückgestellte Aufgaben wird die `work_struct`-Struktur verwendet.[18] Die Abbildung 2 zeigt den Zusammenhang der einzelnen Strukturen.

  Dadurch das die Arbeit in Kernel-Threads ausgeführt wird, hat die Work-Queue alle Vorteile des Prozesskontextes. Der wichtigste Vorteil ist hierbei, dass die Bottom-Half vom Prozessor in einen Ruhezustand versetzt werden darf und später wieder aktiviert werden

[18] eigene Worker-Typen werden natürlich auch von diesen Strukturen repräsentiert

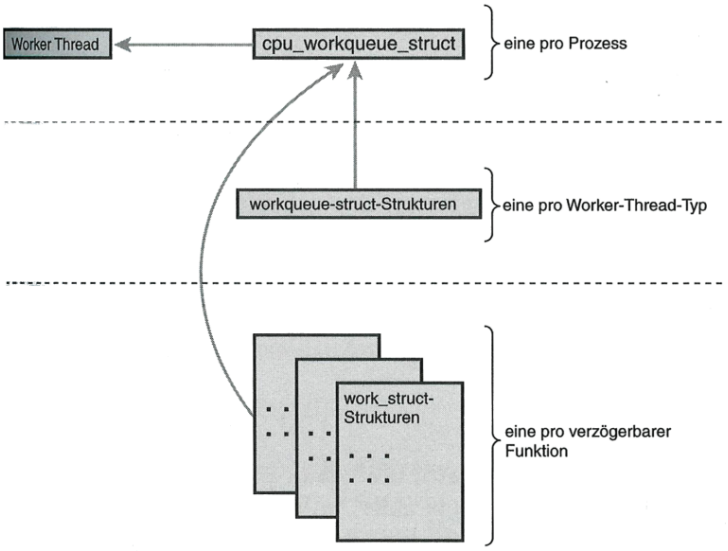

Abbildung 2: Zusammenhang zwischen den verschiedenen Work-Queue-Strukturen [vgl. Lov05, p.159]

kann. Wenn eine hohe Performance benötigt wird können auch spezielle, wie zu Anfang erwähnt, eigene Kernel-Threads angelegt werden. Die Strukturen sind in `linux-2.6.23/kernel/workqueue.c` und `linux-2.6.23/kernel/workqueue.h` definiert.

Die Entscheidung, welcher Mechanismus verwendet werden soll, hängt stark vom Anwendungskontext ab. Softirqs benötigen den meisten Aufwand, bieten aber die beste Skalierbarkeit. Sobald es möglich sein soll, die zurückzustellende Aufgabe in einen Ruhezustand zu versetzen, ist die Work-Queue die einzigste Wahl. Ansonsten sollte ein Tasklet für den Treiber verwendet werden. [Lov05, p.163]

2.3 Konzepte ausgewählter Systeme

2.3.1 Nachladbare Module

Der Linux-Kernel unterstützt die Möglichkeit optionale Teile des Kernels in Module auszulagern. Diese Module können, zur Laufzeit, in den aktiven Kernel dynamisch nachgeladen werden. Das heißt: Die kompilierten Module werden zum laufenden Kernel dazugelinkt und bilden dann, mit dem Kernel, eine Ausführungseinheit. Module werden meist durch ein “intelligentes” User-Space-Programm nach Bedarf geladen und entladen. Explizit kann dies auch über die User-Space-Programme `insmod, rmmod` von der Kommandozeile aus durchgeführt werden:

`insmod <modul_name>`

Das Programm `insmod` um ein Modul zu laden und das Programm `rmmod` um ein Modul wieder aus dem Kernel-Prozess zu entfernen.

`rmmod <modul_name>`

Durch dieses Konzept ist es möglich, ein minimalen Basis-Kernel zu erstellen und optionale Funktionalitäten oder Treiber bei Bedarf zu laden. Diese Modularität des eigentlich monolithischen[19] Kernel erlaubt es den Linux-Kernel auch in Eingebetteten-Systemen[20] einzusetzen.

Die zwei wichtigsten Funktionen, die jedes Modul zur Verfügung stellen muss sind [Lov05, p.356]:

`<modul_name>_init()` und `<modul_name>_exit()`

`<modul_name>` ist der Name des Moduls (der Name ist eigentlich egal, aber es ist sinnvoll, diese Funktionen entsprechend des Modulnames zu benennen). In der `<modul_name>_init()` Funktion werden üblicherweise Ressourcen belegt und Datenstrukturen initialisiert. Der Rückgabewert gibt Aufschluss darüber, ob das Modul erfolgreich geladen wurde oder nicht.

Die `<modul_name>_exit()` Funktion stellt sicher, dass beim Entfernen des Moduls, sich beispielsweise die Hardware (bei einem Treibermodul) in einem konsistenten Zustand befindet und belegte Ressourcen (beispielsweise belegter Speicher) wieder frei gegeben werden.

Module können nur Kernelfunktionen benutzen, die exportiert wurden. Will ein Modul beispielsweise auf die Kernelfunktion `get_quadrat_zahl()` zugreifen, muss diese folgendermaßen exportiert werden:

```
int get_quadrat_zahl(int x)
{
   return x * x;
}
/* exportieren der zuvor definierten Funktion */
EXPORT_SYMBOL(get_quadrat_zahl);
```

Durch Einbinden der C-Headerdatei (Datei, in welcher der Prototyp der zu verwendenden Funktion deklariert ist) ist es nun möglich die Funktion `get_quadrat_zahl()` im selbst erstellten Modul zu nützen. Nachfolgend eine kurze Beispielimplementation eines Moduls, welches die zuvor deklarierte Funktion `get_quadrat_zahl()` aufruft:

[19] Monolithisch bedeutet hier, dass sich alle Funktionalität eigentlich innerhalb einer festen Ausführungseinheit befindet

[20] ein Computersystem das in ein elektronisches Gerät eingebunden ist und dieses steuert oder überwacht

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

/* in dieser Datei sollte der Prototyp der Funktion
 * get_quadrat_zahl deklariert worden sein */
#include <linux/get_quadrat.h>

static int quadrat_init(void)
{
    printk("Initialisierung des Moduls\n");
    /* exportierte Funktion get_quadrat_zahl() aufrufen */
    printk("%d\n",get_quadrat_zahl(4));
    return 0;
}

static void quadrat_exit(void)
{
    printk("Beenden des Quadrat-Moduls\n");
}

module_init(quadrat_init);
module_exit(quadrat_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Frieder B.");
```

Über das Makro `module_init()` wird die `quadrat_init()`-Funktion als Einstiegsfunktion des Moduls registriert. Mit Hilfe des `module_exit`-Makro wird festgelegt, welche Funktion beim Beenden des Moduls aufgerufen wird. In den letzten beiden Makros, die in der Beispielimplementation benutzt werden, wird die Lizenz für das Modul, bzw. der Name des Autors angegeben. Eine nützliche Makro-Familie ist das Parameter-Framework. Mit dessen Hilfe können Parameter, eines Moduls, während der Laufzeit über das sysfs-Dateisystem[21] verändert werden oder beim Ladevorgang des Moduls übergeben werden. Das Makro auf dem das Parameter-Framework aufbaut heißt:

```
/* name -- Variable in die ein möglicher Wert gespeichert wird
 * type -- Datentyp der name-Variable: zum Beispiel: int, char, bool
 * perm -- Zugriffsrechte falls der Zugriff über sysfs ermöglicht
```

[21] ein virtuelles-Dateisystem (je nach Zugriff werden Dateien angelegt). Um dieses System benutzen zu können, muss es wie ein normales Dateisystem eingebunden werden. Meistens ist es unter `/sys/` eingebunden.

```
 *         werden soll */
module_param(name, type, perm)
```

Es kann beispielsweise folgendermaßen verwendet werden:

```
int faehigkeit_aktivieren = 0;
module_param(faehigkeit_aktivieren, bool, 0644);
```

Über das eingebundene sysfs-Dateisystem kann auf die zuvor definierte Eigenschaft, sofern die richtigen Zugriffsrechte gegeben sind, zugegriffen werden:

```
echo "1" > /sys/module/<module_name>/parameters/faehigkeit_aktivieren
```

Mit dieser Eingabe in der Kommandozeile wird der Variable `faehigkeit_aktivieren` der Wert `1` zugewiesen.

Die Organisation des Modul Quellcodes, innerhalb des Linux-Kernel-Tree[22], erfolgt entweder direkt im passenden Subsystem-Verzeichnis oder bei größerem Umfang in einem eigenen Unterverzeichnis (innerhalb des Subsystem-Verzeichnis). Das Modulkonzept des Linux-Kernels ermöglicht, auf sehr einfache Art und Weise, Quellcode in einer Einheit zu kapseln und neue Funktionalitäten, wenn gewünscht zur Laufzeit, in den Kernel zu laden. Genauso besteht die Möglichkeit, ein Modul zur Übersetzungszeit des Linux-Kernel statisch einzubinden.

2.3.2 Das virtuelle Dateisystem (VFS)

Das VFS[23] ist im Dateisystem-Subsystem des Kernels implementiert. Es ist die Abstraktionsschicht zwischen den einzelnen Dateisystemen und den Anwendungen im User-Space (vgl. Abbildung 3). Jedes im Kernel verfügbare Dateisystem ist auf diese Schnittstelle hin implementiert. User-Space Programme können somit einfach mit Systemaufrufen wie `open(), read()` oder `write()`, unabhängig von dem tatsächlich darunter liegenden Dateisystem, auf Daten zugreifen.

Das VFS ist objektorientiert. Eine Handvoll Datenstrukturen stellen das Dateisystemmodell dar. Die 4 Hauptstrukturen (Objekte) [Lov05, p.281ff] des VFS sind:

- **Das Superblock-Objekt**:
  Dieses Objekt zeigt auf ein bestimmtes eingebundenes Dateisystem. Es wird durch "`struct super_block`"-Struktur dargestellt.
- **Inode-Objekt**:
  Das Inode-Objekt repräsentiert eine bestimmte Datei. ("`struct inode`"-Struktur)

[22] bezeichnet die baumförmige Struktur des Linux-Kernel-Quellcodes Verzeichnisses
[23] engl.: Abk. für Virtual-File-System

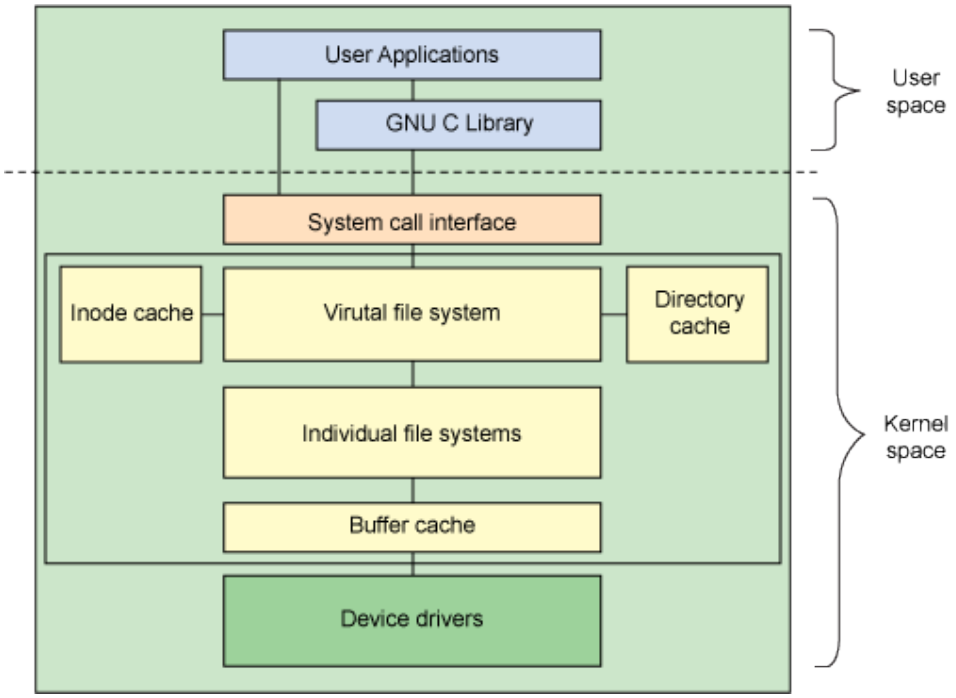

Abbildung 3: Architektonische Sicht auf die Linux-Dateisystem Komponenten [vgl. Jon07]

- **Dentry-Objekt**:
  Das Dentry-Objekt stellt nicht ein Verzeichnis, sondern nur einen einzelnen Bestandteil eines Verzeichnispfades, dar. ("`struct dentry`"-Struktur)
- **File-Objekt**:
  Das File-Objekt repräsentiert eine geöffnete Datei (eine geöffnete Datei ist mit einem Prozess verbunden). ("`struct file`"-Struktur)

Ein spezielles Directory-Objekt gibt es nicht, da das VFS Verzeichnisse als normale Dateien betrachtet. Das Superblock-Objekt speichert Dateisystem spezifische Information, beispielsweise die maximale Dateigröße.

Für jedes dieser 4 Hauptobjekte gibt es ein dazugehörendes Operationen-Objekt. Diese Objekte beinhalten Zeiger auf Funktionen, die das jeweilige Objekt verändern dürfen. Das Superblock-Operationen-Objekt enthält Zeiger auf Funktionen, die bestimmte vom VFS vorgegebene Funktionen implementiert. Diese zu implementierenden Funktionen sind Low-Level-Methoden, die bestimmte Operationen am Dateisystem vornehmen. Die Methoden sind spezifisch für jedes Dateisystem implementiert. Die Struktur eines Superblock-Objektes sieht wie folgt aus (gekürzt):

```
struct super_block {
```

```
    struct list_head s_list;  /* Keep this first */
    dev_t            s_dev;   /* search index; _not_ kdev_t */
    unsigned long    s_blocksize;
    unsigned char    s_blocksize_bits;
    unsigned char    s_dirt;
    unsigned long long s_maxbytes;
    struct file_system_type *s_type;
    const struct super_operations *s_op;
    unsigned long    s_flags;
    unsigned long    s_magic;
    struct dentry    *s_root;
    struct rw_semaphore s_umount;
    struct mutex    s_lock;
    (...)
};
```

Die in der `super_block`-Struktur verwendete Variable `s_maxbytes` gibt die maximale Dateigröße an. Der Typ des Dateisystems wird mit Hilfe der Struktur "`struct file_system_type *s_type`" dargestellt. Die wichtigste Struktur in diesem Superblock-Objekt ist der Zeiger `*s_export_op`. Dieser Zeiger zeigt auf eine "`struct export_operations`"-Struktur, welche Zeiger auf Methoden enthält, die mit diesem Objekt zusammenarbeiten. Um dieses Objekt verwenden zu können müssen die dazugehörigen Methoden des Superblock-Operationen-Objekt implementiert werden:

```
struct super_operations {
    struct inode *(*alloc_inode)(struct super_block *sb);
    void  (*destroy_inode)(struct inode *);
    void  (*read_inode) (struct inode *);
    void  (*dirty_inode) (struct inode *);
    int   (*write_inode) (struct inode *, int);
    void  (*put_inode) (struct inode *);
    void  (*drop_inode) (struct inode *);
    void  (*delete_inode) (struct inode *);
    void  (*put_super) (struct super_block *);
    void  (*write_super) (struct super_block *);
    (...)
};
```

Kurze Erläuterung zu ausgewählten Funktions-Zeigern:
Der "`void (*read_inode) (struct inode *)`" Zeiger, zeigt auf eine Methode die das Einlesen, der als Parameter übergebenen Inode, in eine zuvor angelegte Inode-Struktur durchführt. "`void (*delete_inode) (struct inode *)`" implementiert die Methode zum Löschen einer Inode von der Festplatte. Durch Aufrufen von "`void (*write_super) (struct`

`super_block *)`" werden Änderungen, die im Arbeitsspeicher an dem Superblock-Objekt durchgeführt wurden, auf die Festplatte geschrieben (Synchronisierung).

Für die anderen Hauptstrukturen gibt es auch dazugehörende Operationen-Objekte:

```
struct inode_operations;
struct dentry_operations;
struct file_operations;
```

Für eine genauere Betrachtung der VFS-Objekte wird auf die Literaturliste verwiesen. Das VFS verleiht dem Linux-Kernel eine flexible Schnittstelle, um neue Dateisysteme relativ einfach hinzuzufügen. Dateisysteme können natürlich auch als Module implementiert werden. Die vorgestellten Strukturen befinden sich in: `linux-2.6.23/fs/include/linux/fs.h`.

3 Zusammenfassung

Trotz der Komplexität, die der Linux-Kernel mittlerweile aufweist, besteht nach einiger Einarbeitung die Möglichkeit, eigene neue Funktionen zum Kernel hinzuzufügen. Die Kapselung von architekturspezifischen Details, auf die durch definierte Schnittstellen, einheitlich zugegriffen werden kann, ermöglicht es neuen Quellcode plattformübergreifend zu schreiben. Dank der guten Dokumentation[24] des Kernels und der vielen aktiven Kernel-Entwickler, die über die gemeinsame Linux-Kernel-Mailingliste (LKML)[25] direkt erreicht werden können, besteht die einfache Möglichkeit Fragen zu stellen und eigene Quellcodes mit anderen Entwicklern zu diskutieren. Solange die Form des "Gebens und Nehmens" eingehalten wird (wie bereits zu Anfang erwähnt, ist das Dank der Lizenzierung sichergestellt) steht der weiteren Entwicklung des Linux-Kernel nichts im Wege. Der Linux-Kernel hat gute Chancen eine noch weitere Verbreitung zu erfahren und ein Zuhause in den verschiedensten elektronischen Geräten zu finden.

Literatur

[Jon07] M. Tim Jones. Anatomy of the linux file system, 2007. URL: `http://www.ibm.com/developerworks/linux/library/l-linux-filesystem/` Abgerufen am 28.12.2007.

[Lov05] Robert Love. *Linux-Kernel-Handbuch.* Addison-Wesley, 2005.

[24] jeder Linux-Kernel-Tree enthält das Verzeichnis `Documentation/`, in diesem ist die aktuellste Dokumentation zu finden

[25] `http://lkml.org`

[RC02] Alessandro Rubini and Jonathan Corbet. *Linux-Gerätetreiber, 2. Auflage*. O'Reilly, 2002. URL: `http://www.oreilly.de/german/freebooks/linuxdrive2ger/ldr_0101.jpg` Abgerufen am 27.12.2007.