Design – Linux-Kernel
rohe Rohfassung

Frieder Bürzele
HfT Stuttgart
Studiengang Informatik
Wintersemester 2007/2008

20. Oktober 2007

Inhaltsverzeichnis

1 Einleitung

Viele Menschen können mit dem Begriff “Linux” nichts anfangen aber nutzen dieses System in irgendeiner Form trotzdem. In vielen elektronischnen Geräten läuft eine angepasste Version des Linux Kernel. Ursprünglich für die i386er [1] Architektur entwickelt und mit relativ wenig Funktionalität wurde innerhalb der letzen 16 Jahren der Linux-Kernel ständig weiter entwickelt und auf etliche weitere Architekturen portiert. Möglich wurde so eine “Evolution” aufgrund der freien Verfügbarkeit des Quellcodes unter der GPLv2 [2].

Heute findet Torvalds [3] Kernel in nahezu jedem Bereich seine Anwendung: Beispielsweise im Serverbereich, auf dem Desktop, auf Mainframes, als hochleistungs Cluster, in Set-Top-Boxen, Routern oder auch in Handys. Dabei ist erstaunlich wie gut der Kernel auf all diesen System skaliert bzw. funktioniert und wie einfach es dank frei zugänglichen Quellcodes für jeden möglich ist den Kernel anzupassen. Aufgrund der vielen Features und Möglichkeiten die der Linux-Kernel bietet scheint er als Ganzes recht Komplex. Trotzdem wird dank des Designskonzeptes, Unterteilung des Kernels in Subsysteme, der Einstieg in die Kernelprogrammierung und die Implementierung eigener Features erleichtert. Im Laufe dieser Ausarbeitung wird anhand einiger Konzepte gezeigt wie einfach neue Funktionen hinzugefügt werden können und welche Mechanismen das erleichtern.

2 Hauptteil

2.1 Einführung

2.1.1 Was sind Subsysteme

Ein Subsystem fasst Teilbereiche des Linux-Kernel zusammen. Beispiele für Subsysteme befinden sich in dem TODO fake-Dateisystem sysfs. In der aktuellen Kernel-Revision[4] sind das die folgenden: `block bus class devices firmware fs kernel module power`. Um den genauen Aufbau und Zusammenhang eines Subsystems zu verstehen müssen zuererst einige Begriffe erklärt werden.

- kobjects: ist eine struktur die Grundlegende Möglichkeiten für die Erzeugung einer Objekt-Hierarchie repräsentiert:

```
struct kobject {
    const char * k_name;
    char name[KOBJ_NAME_LEN];
    struct kref kref;
```

[1] Prozessorarchitektur, die nach wie vor (2007) meist verwendete Architektur für Desktop Computer

[2] GNU General Public License, version 2

[3] Linus Torvalds, ursprünglicher Schöpfer des Linux-Kernels

[4] 2.6.23.1

```
    struct list_head entry;
    struct kobject * parent;
    struct kset * kset;
    struct kobj_type * ktype;
    struct dentry * dentry;
    wait_queue_head_t poll;
};
```

Diese Struktur wird innerhalb einer anderen Struktur eingebettet. Beispielsweise innerhalb einer Klasse von gleichartigen Gerätetreibern Die Hierachie kommmt dadurch zustande das jedes kobject seinen parent kennt. Diese Struktur wird innerhalb einer anderen Struktur eingebettet. Beispielsweise

- ktypes: beschreiben das Standardverhalten von gleichartigen kobjects.

  ```
  struct kobj_type {
     void (*release)(struct kobject *);
     struct sysfs_ops * sysfs_ops;
     struct attribute ** default_attrs;
  };
  ```

- ksets: sind eine art Container für einen Satz von gleichartigen Objekten

  ```
  struct kset {
     struct subsystem * subsys;
     struct kobj_type * ktype;
     struct list_head list;
     spinlock_t list_lock;
     struct kobject kobj;
     struct kset_uevent_ops * uevent_ops;
  };
  ```

  beispielsweise alle Block-Geräte[5]

Ein Subsystem verweist dabei auf eine ksets struktur.

```
struct subsystem {
   struct kset kset;
   struct rw_semaphore rwsem;
};
```

2.1.2 Zusammenarbeiten der Subsysteme

Wie greifen nun die Subsysteme ineinander? TODO

[5] Geräte wie Festplatten, cdrom, ...

2.2 Eweiterbarkeit ausgewählter Systeme

2.2.1 Das virtuelle File-System (VFS)

Das VFS ist Subsystem des Kernels das alle Schnittstellen eines Dateisystems implementiert. Es ist die Abstraktionsschicht zwischen den einzelnen Dateisystemen und den Anwendungen im user-space. Jedes im kernel verfügbare Dateisystem ins auf diese Schnittstelle hin implementiert. Userspace Programme können somit einfach mit Systemcalls wie open(), read(), write() unabhängig vom tatsächlichen darunterliegenden Dateisystem auf Daten zugreifen.

Das VFS ist objektorientiert. Eine handvoll Datenstrukturen stellt das Datei-Modell dar. Die vier Hauptstrukturen(Obkekten) des VFS sind:

- Das Superblock-Objekt: es stellt ein bestimmtes gemountetes Dateisystem dar.
- Das Inode-Objekt repräsentiert ein bestimmtes File.
- Das Dentry-Objekt, repräsentiert ein nicht ein Verzeichnis sondern nur einen einzelnen Bestandteil eines Pfades.
- Das File-Objekt repräsentiert eine geöffnete Datei (eine geöffnete Datei ist mit einem Prozess verbunden)

Ein spezielles Directory-Objekt gibt es nicht, da das VFS Verzeichnisse als normale Dateien betrachtet.

Das Superblock-Objekt speichert hier Filesystem spezifische Information. Beispielsweise die maximale Dateigröße. Das Superblock-Operationen-Objekt enthält Zeiger auf Funtionen die bestimmte vom VFS vorgegebenen Funktionen implementiert. Diese zu implementierenden Funktionen sind low-level-Methoden die bestimmte Operationen am Dateisystem vornehmen also spezifisch.

Desweiteren gibt es auch für das Inode-Objekt ein Inode-Operationen-Objekt. Das Inode-Objekt bildet jede Datei in einem File-System ab. Das Operationen-Objekt enthält wieder Funktionszeiger welche die auf die tatsächlichen File-System spezifischen Implementierungen der nötigen Funktion zeigt. Dasselbe gilt auch für das File und denry-objekt und die dazugehörigen Operationen-Objekte. TODO

Das VFS erlaubt es auf relativ einfache Art-und Weise neue Dateisysteme in den Linux-Kernel zu integrieren. An dieser Stelle wird auch gezeigt das trotz der fehlenden Sprachunterstützung in C für OOP [6] ein Objektorienter Ansatz erfolgreich realiert wurde.

2.2.2 Das module konzept

Der Linux-Kernel unterstüzt die Möglichkeit Code zur Laufzeit in Form von nachladbaren Modulen in zum laufenden Kernel dynamisch hinzuzulinken. Durch dieses Konzept ist es möglich ein minimales Basissystem zu erstellen und die

[6] Abk. für Objekt orientierte Programmierung

andere optionale Funktionen oder Treiber bei Bedarf zu laden. Diese Modularität des eigentlich monolitischnen[7] Kernel erlaubt es den Linux kernel auch auf embedded systemem laufen zu lassen

Die zwei wichtigsten Funktionen die jedes Modul implementieren muss ist zum einen die `MODUL_NAME_init()` Funktion. In dieser Funktion werden üblicherweise Ressourcen belegt und Datenstrukturen initialisiert. Der Rückgabewert gibt Aufschluss darüber ob das Modul erfolgreich geladen wurde oder nicht. Zum anderen die `MODUL_NAME_exit()` Funktion, welche sicherstellt das sich (bei einem Treibermodul) die Hardware in einem konsistenten Zustand befindet und gibt belegte Resourcen wieder frei beispielweise belgter Speicher.

Module köennen nur Kernelfunktionen benutzen die exportiert wurden. Will ein Modul beispielsweise auf die eine Kernelfunktion `foo()` zugreifen muss diese zuvor mit
`EXPORT_SYMBOL(foo)`
exportiert worden sein. Jetzt ist es möglich durch Einbinden der Headerdatei (wo der Prototyp der Funktion `foo()` deklariert ist) diese in seinem Modul zu nützen. Die Organisation des Quellcodes erfolgt bei kleinen Modulen (header + sourcefile) innerhalb des linux-kernel trees direkt im passenden Subsystem Verzeichnis bei größerem Umfang wird für das Modul ein eigenes Unterverzeichnis angelegt. Das Modulkonzept ermöglicht auf sehr einfach Art und Weise Funktionalität zum Kernel hinzuzufügen.

2.2.3 Synchronisierung Mechanismen

Der Linux-Kernel bietet von Haus aus einige Methoden zum Locking. Also zur Vermeidung von Deadlocks und Race-Conditions. Locking bezeichnet den Schutz der es ermöglicht der alleinige Nutzer einer bestimmten Ressource (bsplw. Daten) zu sein. Deadlocks können entstehen wenn beispielsweise zwei Prozesse auf die die jeweils von dem anderen gelockten Resource warten. Unter Race-Conditions versteht man im Allgemeinen die Situation wenn zwei Prozesse gleichzeitig die auf die gleichen Daten zugreifen. Ein einfaches Beispiel ist schon die eine einfache Inkrementierung eines Counters: Prozess 1 ließt von Speicherstelle, gleichzeitig ließt Prozess 2 von der gleichen Speicherstelle. Beide Inkrementieren um eins, aber das Ergebnis ist nicht die Erhöhung um 2 Zähler sondern nur um einen. Um dies zu vermeiden gibt es im Linux-Kernel folgende Mechanismen:

- Spinlocks: bieten die Möglichkeit eine Ressource zu locken. Wenn jetzt ein anderer Prozess auf dieselben Daten zugreifen will muss er solange auf der stelle treten (spin) bis der vorherige Prozess den Lock entfernt hat. Spinlocks werden verwendet wenn sicher gestellt ist das der Lock sehr schnell wieder freigegeben wird (deterministisch) da der wartende Prozess dauernd nachfrägt und so einen Overhead verursacht. Sie werden dort verwendet wo ein Prozess sich nicht schlafen darf, z.B. in einem Interrupt-Kontext.

[7] TODO was is monolitisch

- Semaphore: ermöglichen ebenfalls das Locken von Ressourcen. Ist ein Ressource gelockt und ein weiterer Prozess will auf diese Zugreifen, wird der anfragende Prozess anstatt wie bei den Spinlocks in einen Schlafzustand versetzt und dann sobald die Ressource wieder freigegeben ist geweckt.

Die vorgestellten Methoden sind Architekturspezifisch Hardware unterstützend implementiert um die Atomarität [8] sicherzustellen. Durch diese einheitlichen Methoden bietet Linux eine Infrastruktur um Deadlocks bzw. Race-Conditions [9] wirkungsvoll zu vermeiden.

2.3 Beispiel Implementierungen

TODO

2.3.1 Austauschbarer CPU-Scheduler

TODO

2.3.2 Austauschbare IO-Scheduler

TODO

2.3.3 Interrupt Handler

Interrupt Handler sind aufgeteilt in zwei Bereiche. Erstens in Top-Halves und in Botton-Halves.

- Top-Halves: Hier wird der unmittelbare Interrupt behandelt. Daten zB von Block-Gerät Buffer werden in Haupt-Speicher geschrieben.
- Botton-Halves: hier werden die im Top-Halves angelegten Daten verarbeitet

Das Auftreten eines Interrupts ist asynchron und unterbricht den Prozess der gerade läuft. Aufgrund dieser ungewollten Unterbrechung wird der Intrupt-Handler in Top und Bottom aufgeteilt. Botton-Halves werden zu einem günstigern Zeitpunkt in der Zukunft ausgeführt. TODO

3 Zusammenfassung

Trotz der Komplexität die der Linux-Kernel mittlerweile aufweist besteht die Möglichkeit nach einiger Einarbeitung eigene neue Funktionen zum Kernel hinzuzufügen. Dank der guten Dokumentation des Kernels und der Möglichkeit "seinen Beitrag" zu leisten in Form von eigenen Code-Beiträgen steht einer weiterern "Evolution" des Linux-Kernel nichts im Wege.

[8] Alles oder nichts

[9] können nur bei sogenannten SMP Systemen auftreten, also Systeme die mehr als eine CPU haben

Der Linux-Kernel gute Chancen eine noch weitere Verbreitung zu finden und ein Zuhause in den verschiedensten Geräten zu haben.

Literatur

missing TODO