Diplomarbeit

zum Thema

Generische Portierung von Linux-Gerätetreibern auf die DROPS-Architektur

Christian Helmuth

TU Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

31. Juli 2001

Selbständigkeitserklärung

Hiermit erkläre ich, daß ich diese Arbeit selbständig und nur mit den zugelassenen und aufgeführten Hilfsmitteln erstellt habe.

Dresden, 31. Juli 2001

Christian Helmuth

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Aufbau der Arbeit . . . 2
1.2 Danksagung . . . 2

2 Gerätetreiber und Betriebssysteme **3**
2.1 Das DROPS-Projekt . . . 3
2.2 Gerätetreiber . . . 5
2.3 Geräteklassen . . . 6
2.4 Die Kernumgebung . . . 7
2.4.1 Ablaufsteuerung . . . 7
2.4.2 Ressourcen . . . 9
2.4.3 Ein/Ausgabe-Steuerung . . . 9
2.5 Der Linuxkern . . . 9
2.5.1 Linux 2.4 Gerätetreiber und -klassen . . . 10
2.5.2 Linux 2.4 Kernumgebung . . . 11
2.6 Verwandte Arbeiten . . . 15

3 Entwurf **17**
3.1 Architektur . . . 17
3.2 Zentrale Ein/Ausgabe-Unterstützung . . . 18
3.3 Emulation der Kernumgebung . . . 19
3.3.1 Threadstruktur . . . 19
3.3.2 Aufbau der Emulation . . . 20
3.3.3 Allgemeine DDE Funktionalität . . . 21
3.3.4 Spezifische DDE Funktionalität — `sound` . . . 25

4 Implementierung **28**
4.1 Der I/O Server . . . 28
4.1.1 Ressourcenverwaltung . . . 28
4.1.2 PCI Support . . . 29
4.1.3 Interrupt Behandlung . . . 30

4.2 Bibliotheken für Linux-Gerätetreiber . . . 30
4.2.1 Allgemeine (`common`) Bibliothek . . . 30
4.2.2 `sound` Bibliothek . . . 33
4.3 Fallbeispiel: Soundtreiber `es1371.c` . . . 34

5 Leistungsbewertung **36**

6 Zusammenfassung und Ausblick **37**

A Schnittstellen-Definition: I/O Server **38**

B Glossar **40**

Abbildungsverzeichnis

2.1 DROPS-Architektur (aus [BBH$^+$98]) . . . 3
2.2 UNIX Style (monolithisch) . . . 4
2.3 Separater Treiber-Server (Mikrokern) . . . 4
2.4 Colocated Treiber (high-speed Applikationen) . . . 4
2.5 Prozeß- und Interruptebene (nach [Mar99]) . . . 6
2.6 Linux Geräteklassen . . . 10
2.7 Prozeß- und Interruptebene im Linuxkern . . . 13
2.8 Auszug des Inhalts von `drivers/` . . . 15

3.1 Aufbau des I/O Servers . . . 19
3.2 Speicherpools und Dataspaces (nach [L$^+$99] Abb. 4) . . . 22
3.3 Aufbau eines Gerätetreibers . . . 26
3.4 Endgütlige DDE Architektur . . . 27

4.1 Aufbau der `schedule()` Funktion . . . 31
4.2 Interrupt Thread . . . 32
4.3 *Deferred Activity* Thread . . . 32
4.4 Warteschlangen-Synchronisation (durch Signale unterbrechbar) . . . 34
4.5 Warteschlangen-Synchronisation in `es1371.c` . . . 35

Tabellenverzeichnis

3.1 Funktionalität des DDE 18
3.2 Speichermanagement unter Linux 21

Kapitel 1

Einleitung

Moderne Computersysteme zeichnen sich durch die Fähigkeit aus, verschiedenste Anforderungen in den unterschiedlichsten Bereichen erfüllen zu können. Ein Hauptgrund dafür ist die einfache Konfiguration durch individuelle Auswahl von Systemkomponenten. Hierbei ist auch der Anschaffungspreis ein entscheidender Faktor, weshalb viele Hersteller unterschiedliche Lösungen anbieten.

Diese Vielfalt an Hardware bewirkt auch, daß immer neue Programme zu ihrer Ansteuerung entwickelt werden müssen. Diese *Gerätetreiber* werden oft vom Hersteller selbst zur Verfügung gestellt, sind aber stets betriebssystemspezifisch und selten werden die Programm-Quellen offengelegt.

Für Forschungssysteme wie das mikrokernbasierte *Dresden Real-Time Operating System* existieren keine Gerätetreiber der Hersteller. Sie sind aber trotzdem nötig, um auch Geräte ansteuern zu können, die im Gegensatz zur Standard-Hardware wie Prozessor, DMA-Controller, PIC usw. von System zu System variieren. Die selbständige Entwicklung von Treibern für jedes Gerät ist aus zwei Gründen keine Lösung: **1.** sind umfangreiche Spezifikationen der Hardware, wenn überhaupt verfügbar, teuer und mit rechtlichen Einschränkungen für die Weiterverwendung verbunden, und **2.** sind Entwicklung und Test von Gerätesteuerprogrammen sehr zeitaufwendig.

Eine bessere Lösung ist die Portierung von Gerätetreibern anderer Betriebssysteme auf die Zielarchitektur mit dem Anspruch, den Originaltreiber möglichst nicht zu verändern, um den Wartungsaufwand gering zu halten. Arbeiten wie [Sta96] haben bewiesen, daß es möglich ist, den Gerätetreiber aus dem monolithischem Originalsystem herauszulösen und unverändert in eine mikrokernbasierte Architektur zu integrieren.

Der nächste logische Schritt ist eine Generalisierung dieses treiberspezifischen Prozesses für (nahezu) alle Treiber eines Betriebssystems. Mit diesem Thema setzt sich diese Arbeit auseinander und beschreibt den Prozeß der *generischen Gerätetreiberportierung* vom monolithischen Linuxkern auf das mikrokernbasierte DROPS.

1.1 Aufbau der Arbeit

Die Arbeit ist in fünf Kapitel gegliedert und soll den Entwicklungsprozeß von der Bestandsaufnahme und den allgemeinen Betrachtungen über Entwurf und Implementierung bis zu einem abschließenden Ausblick auf zukünftige Erweiterungen und Anwendungen vollständig darstellen.

Kapitel 2 beschäftigt sich nach einer knappen Vorstellung des DROPS-Projektes mit allgemeinen Betrachtungen zur Rolle der Gerätetreiber im Betriebssystem und einer Konkretisierung für Linux [T+01]. Außerdem werden verwandte Projekte kurz vorgestellt.

Im nächsten Kapitel wird eine geeignete Architektur für Gerätetreiber unter DROPS entworfen. Aufgaben und Aufbau der beteiligten Komponenten werden beschrieben, wobei besondere Beachtung der Umgebung — DROPS bzw. *Common L4 Environment* — und ihren spezifischen Anforderungen und Möglichkeiten gilt.

Das Kapitel 4 beschreibt die Implementierung der entworfenen Komponenten und deren Zusammenspiel allgemein und an einem konkreten Beispiel. Darauf folgt eine Abschätzung zur erwarteten Leistung der Gerätetreiber im Vergleich mit dem Originalsystem.

Die gewonnen Erkenntnisse werden schließlich in Kapitel 6 zusammengefaßt und ein Überblick über den Stand der Implementierung gegeben. Es wird weiterhin ein Ausblick auf mögliche Erweiterungen und weiterführende Entwicklungen geboten.

1.2 Danksagung

An dieser Stelle möchte ich mich herzlich bei jedem bedanken, der mich bei der Fertigstellung dieser Arbeit unterstützte, insbesondere bei meinen Betreuern Prof. Hermann Härtig und Lars Reuther. Außerdem gilt mein Dank auch meinen Freunden in der „Schlüterstraße“ und anderswo.

Kapitel 2

Gerätetreiber und Betriebssysteme

2.1 Das DROPS-Projekt

An der TU Dresden, Professur Betriebssysteme, konzentriert sich die Forschungsarbeit auf ein Betriebssystem mit Multi-Server-Architektur, das Anwendungen mit *QoS*-Anforderungen unterstützt. Das ***D**resden **R**eal-Time **Op**erating **S**ystem* (DROPS) nutzt den DROPS-Mikrokern, einen Mikrokern der 2. Generation, für die Realisierung dieses Ziels.

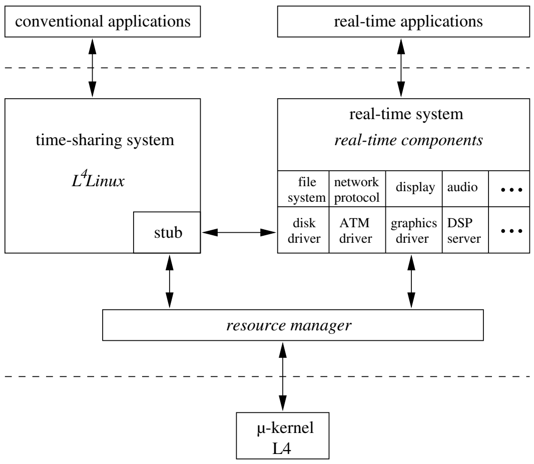

Abbildung 2.1: DROPS-Architektur (aus [BBH+98])

Die DROPS-Architektur unterscheidet sich aufgrund der Mikrokern-Basis von traditionellen monolithischen Betriebssystemen. Hauptunterschied ist hierbei die Plazierung der Systemkomponenten bzw. Gerätetreiber:

Monolithischer Kern Abb. 2.2 illustriert den traditionellen Ansatz, bei dem der Gerätetreiber Bestandteil des Betriebssystemkerns ist und somit im *privilegierten Modus* der CPU und im *Kernadreßraum* ausgeführt wird. Dieser Ansatz liegt nahezu allen UNIX-Systemen zugrunde.

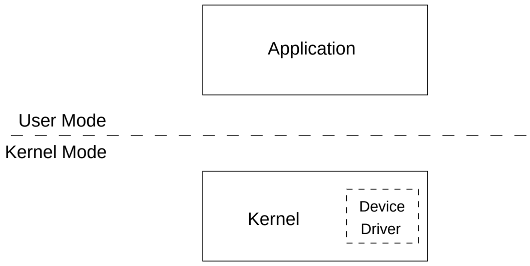

Abbildung 2.2: UNIX Style (monolithisch)

Mikrokerne verringern den im *privilegierten* Prozessormodus ausgeführten Programmcode auf ein Minimum. Weitergehende Funktionalitäten werden in *User Mode Server* ausgelagert, so auch Gerätetreiber (Abb. 2.3). Die Server sind durch Adreßraumgrenzen geschützt und kommunizieren auf der Basis von *Inter-Process Communication (IPC)*.

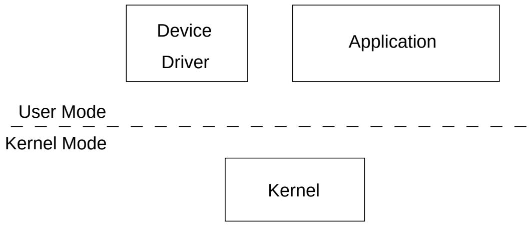

Abbildung 2.3: Separater Treiber-Server (Mikrokern)

Colocation Eine Lösung für Applikationen mit hohen Leistungsansprüchen ist in Abb. 2.4 dargestellt. Der Gerätetreiber ist hier Bestandteil der Applikation und der Kommunikationsaufwand zwischen den beiden Komponenten wird somit minimiert, da weder der Systemkern einbezogen wird noch IPC nötig ist.

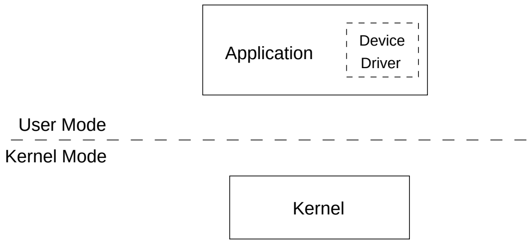

Abbildung 2.4: Colocated Treiber (high-speed Applikationen)

Für DROPS ist der erste Ansatz, den Gerätetreiber in den Kern zu integrieren, nicht passend, da der Mikrokern, die Basis des Systems, *nicht* erweitert werden soll. Es bietet sich also an, wie im zweiten Ansatz beschrieben, jeden Gerätetreiber in einen eigenen Server auszulagern und eine *schnelle* IPC zur Kommunikation zu verwenden.

Man sollte aber die Möglichkeit der *Colocation* nicht vollständig verwerfen und den Gerätetreibern für DROPS erlauben, als Programm-Modul im Adreßraum der Applikation abzulaufen. Es ist damit möglich, Geräte mit potentiell nur einem Klienten, z. B. Netzwerk-Protokollstack und -gerätetreiber bzw. Konsole und Videotreiber, *colocated* zu konfigurieren und die Leistung zu verbessern.

Gerätetreiber unter DROPS sollen also eigenständige Module mit einer definierten Schnittstelle sein, die je nach Anspruch bzw. Konfiguration über IPC oder lokale Prozeduraufrufe genutzt wird.

2.2 Gerätetreiber

Das Betriebssystem abstrahiert die Schnittstelle zur wirklich vorhandenen Hardware in einem Computersystem und stellt eine *virtuelle Maschine* mit definierten Eigenschaften zur Verfügung. Ein Beispiel für eine Schnittstellendefinition ist der *POSIX* Standard.

Die Aufgaben, welche das Betriebssystem zu lösen hat sind vielfältig:

1. Verwaltung von Tasks
2. Verwaltung von Hardware-Ressourcen
3. Ablaufsteuerung
4. Zugriffsschutz & Ausnahmebehandlung
5. Ein/Ausgabe-Steuerung

Im Kontext der Gerätetreiber-Portierung sind insbesondere die Punkte 2) und 5) interessant. Es ist also zu klären, wie das Betriebssystem mit den Hardware-Ressourcen umgeht und welche Mechanismen die Ein/Ausgabe unterstützen. Außer diesen Punkten darf man auch den Einfluß der *Ablaufsteuerung* auf Gerätetreiber nicht vernachlässigen.

Auf diese Themen gehe ich in den nächsten Abschnitten näher ein, beginne nun aber mit einer genaueren Beschreibung von Gerätetreiber-Software.

Das „Taschenbuch der Informatik“ definiert Treiber-Software so ([W$^+$95] S. 295):

> *Die Gerätebedienroutinen (Gerätetreiber) übernehmen die Eingabe bzw. Ausgabe einer gerätespezifischen Dateneinheit, z. B. eines Zeichens oder eines Blockes. Dabei beachten sie das von der Gerätesteuerung festgelegte Protokoll (z. B. Ausgabe von Kommandos, Abfrage und Auswertung des Gerätestatus, Behandlung von Fertigmeldungen, Übergabe bzw. Übernahme von Nutzdaten).*

Das bedeutet also, daß Gerätetreiber eine Softwareschicht zwischen Applikationen bzw. Kernkomponenten (z. B. Netzwerkprotokoll-Stack) und dem wirklichen Gerät darstellen. Als integraler Bestandteil des Betriebssystems sind sie eng mit diesem verbunden und werden normalerweise speziell auf eine bestimmte Systemumgebung zugeschnitten.

Eine nicht geringe Eigenschaft der Gerätetreiber ist die Schutzfunktion, die diese darstellen. Es soll also nicht möglich sein, ein über einen Treiber angesteuertes Gerät durch falsche Benutzung zu beschädigen oder sogar zu zerstören.

Der Gerätetreiber behandelt Anfragen von Nutzerprogrammen (*system calls*) und Hardwareereignisse (*interrupts*). Aus diesem Grund kann man die Treiberfunktionalität grob in zwei Schichten — *Prozeß-* und *Interrupt-Ebene* — unterteilen (Abb. 2.5). Da die beiden Schichten einen gemeinsamen Status teilen und nicht zwingend sequentiell ausgeführt werden, müssen Zugriffe auf den *shared* Bereich synchronisiert werden.

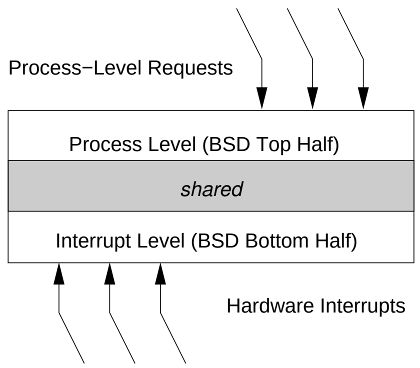

Abbildung 2.5: Prozeß- und Interruptebene (nach [Mar99])

Wie die Synchronisation durchgesetzt wird, ist systemabhängig. Häufig basieren Lösungen auf Warteschlangen, Locks und Unterbrechungszulassung bzw. -verbot. Das Sperren von Interrupts ist in Uniprozessor-Systemen die einfachste Möglichkeit zu synchronisieren.

In Multiprozessor-Betriebssystemen dagegen muß man andere Lösungen finden. Die Serialisierung von Systemrufen, so daß nur einer aktiv zu einem Zeitpunkt (Linux 2.0) ist, ist eine Möglichkeit, Aktivitäten auf Prozeßebene zu synchronisieren. Eine andere ist, den Kern *multi-threaded* (Solaris, Linux 2.4) zu entwerfen. Intelligente Sperren sind hier sehr wichtig, da mehrere Aktivitäten der Prozeß- und Interruptebenen auf unterschiedlichen Prozessoren ablaufen können.

Soll nun derselbe Treiber in verschiedenen Umgebungen genutzt werden, ist es wahrscheinlich, daß die erwarteten Kernfunktionalitäten nicht zur Verfügung stehen bzw. ein anderes Verhalten aufweisen.

2.3 Geräteklassen

Die Geräte in einem Computersystem lassen sich traditionell nach ihrem Aufbau und dem daraus resultierendem Zugriff in zwei Kategorien einordnen:

Blockgeräte „speichern“ Blöcke fester Größe, denen eine eindeutige Adresse zugeordnet wird. Typische Blockgrößen sind Zweierpotenzen von 512 bis 32768 Bytes. Jeder Block kann individuell für einen Lese/Schreibzugriff adressiert werden, d. h. Blockgeräte bieten eine *seek* Operation an. Beispiele für diese Geräteklasse sind Speichermedien wie Festplatten, Disketten und CD-ROMs.

Zeichengeräte hingegen arbeiten mit einem Zeichenstrom, d. h. es gibt keine Blöcke und keine individuelle Adressierung. Lese/Schreiboperationen greifen immer auf das *nächste* Zeichen im Strom zu und eine *seek* Operation wird nicht implementiert. In dieser Klasse findet man Geräte wie Scanner, Drucker und Soundkarten.

Da diese Klassifizierung aber nicht *perfekt* ist, existieren einige Geräte die nicht klar einzuordnen sind. So liefern z. B. Uhren keinen Zeichenstrom, unterstützen aber auch keine adressierbaren Blöcke. Trotz dieser Schwäche hat sich eine grobe Einteilung in Block- und Zeichengeräte in vielen Betriebssystemen durchgesetzt.

Weiterführende Unterteilungen der Klassen sind meist systemspezifisch und in die Betrachtungen einzubeziehen (siehe Abschnitt 2.5.1). So kann ein Gerätetreiber für einen DSP (Soundkarte) — obwohl auch ein Zeichengerät — eine weitaus umfangreichere Kernumgebung erfordern als einer für die serielle Schnittstelle.

An dieser Stelle spielen auch *Subsysteme* eine große Rolle. Treiber für SCSI-Geräte z. B. benötigen viele Funktionalitäten aus dem SCSI-Subsystem, das Warteschlangen verwaltet und Synchronisation durchführt. Ein IDE-Treiber hingegen sollte eine schlankere Umgebung voraussetzen, schon allein aus dem Fakt heraus, daß die angesteuerten Endgeräte nicht so vielfältig wie bei SCSI sein können.

Die Adresse von Geräten (zumindest in UNIX Systemen) setzt sich aus mehreren Teilen zusammen. So muß beim Zugriff der Gerätetyp — *char* oder *block* — der Treiber in Form einer *major* Nummer und die Gerätenummer — *minor number* — angegeben werden.

2.4 Die Kernumgebung

2.4.1 Ablaufsteuerung

Ein sehr wichtiger Aspekt der Kernumgebung ist das *Threading Modell*, d. h. welche Semantik haben Synchronisation und Scheduling im Betriebssystem. Dabei ist zu klären, ob bzw. wann ein Thread blockieren kann und was mit seinem Zustand geschieht falls er blockiert.

Non-blocking Ein mögliches Design für die Kernumgebung ist es, allen Nutzerprozessen nur einen Kernstack zur Verfügung zu stellen. Tritt also der Fall ein, daß mehrere Prozesse einen Systemruf durchführen, kann immer nur jeweils ein Prozeß den Kern betreten (und auf Kerndaten zugreifen bzw. privilegierte Instruktionen ausführen). Alle weiteren „Anwärter“ blockieren im User Mode, d. h. ihre Ausführung wird solange unterbrochen bis der Kerneintritt erlaubt wird. Die Bezeichnung *non-blocking* begründet sich auf der Tatsache, daß ein Prozeßkontext, der im Kern blockiert, z. B. weil eine Ressource nicht verfügbar ist, den gesamten Status verliert und den Systemruf neu starten muß.

Eine solche Kernumgebung ist nur in Systemen sinnvoll, die das Blockieren im Kern allgemein vermeiden wollen und den Anspruch haben, Systemrufe stets *vollständig* durchzuführen. Ein Beispiel ist der Exokernel.

Blocking Die zweite (und häufigere) Möglichkeit ist, jeder Nutzeraktivität neben dem Nutzerstack einen Kernstack zur Verfügung zu stellen. Blockiert ein Prozeßkontext im Kern, wird sein Status auf dem dedizierten Kernstack gespeichert. Eine Fortsetzung des Prozesses ist nun mit den Informationen auf dem Stack ohne Probleme möglich. *Blocking* Betriebssysteme sind z. B. Windows NT und die meisten UNIX Varianten incl. Linux.

Ich werde mich bei den weiteren Ausführungen mit Betriebssystemen auseinandersetzen, welche die zweite Variante nutzen, da sie, wie erwähnt, weitaus häufiger anzutreffen ist. Wenn es mehreren Ausführungspfaden erlaubt ist, „gleichzeitig“ den Kern zu betreten — *Reentrance*, muß auch das *Scheduling* von Aktivitäten im Kern genauer betrachtet und geklärt werden, wie die Verdrängung (*Preemption*) von Prozessen, d. h. die Freigabe der CPU, erfolgt.

Preemptive Scheduling Diese Strategie erlaubt es einer zentralen Instanz, einem laufenden Prozeß zu (nahezu) jedem Zeitpunkt den Prozessor zu entziehen und einem anderen Prozeß zuzuteilen. Die temporäre Unterbrechung des Prozesses ist für diesen transparent.

Preemptives Scheduling ist vor allem für Echtzeit-Betriebssysteme interessant, da höher priorisierte Prozesse nach dem Deblockieren, d. h. wenn sie in die Bereit-Warteschlange eingekettet werden, solche mit niederer Priorität von der CPU verdrängen können.

Non-Preemptive Scheduling Jede Aktivität wird solange ausgeführt bis sie blockiert oder den Prozessor freigibt. Diese Eigenschaft erlaubt es, Annahmen über eine zeitlich begrenzte Konsistenz von Daten zu machen, d. h. wenn nur ein Thread aktiv ist, müssen Datenmanipulationen nicht unter explizit gegenseitigem Ausschluß stattfinden[1].

Dieses ist das traditionelle UNIX-Kernmodell und außer wenigen Ausnahmen neuerer Entwicklungen die vorherrschende Strategie.

Ich möchte nochmals darauf hinweisen, daß es sich hier um das Scheduling von Aktivitäten, welche den Kern betreten haben, handelt. Andere Nutzeraktivitäten und solche die den Kern verlassen unterliegen meist einer anderen Schedulingstrategie.

[1]Diese Annahmen gelten in Uniprozessor-Systemen *ohne* Beachtung von Interferenz mit Hardwareereignissen.

2.4.2 Ressourcen

Die in diesem Kontext interessanten Systemressourcen betreffen die Ein/Ausgabe, sind also die Schnittstelle zur Hardware. Im einzelnen sind das E/A-Adreßbereiche, DMA-Kanäle und Interrupts.

Da in einem Computersystem eine große Anzahl von Erweiterungskarten und Geräten existieren kann, werden die E/A-Ressourcen intelligent verwaltet. Die Geräte müssen, um mit den Gerätetreibern bzw. der CPU kommunizieren zu können, Register exportieren. Zu diesem Zweck werden die Register in einen Adreßraum der CPU eingeblendet (*Mapping*). Welcher Adreßraum hierfür verwendet wird, hängt von der Hardware-Architektur ab. In PC-Systemen existieren dafür zwei mögliche Wege: der E/A-Adreßraum (Ports) und der Speicheradreßraum (Memory Mapped I/O).

Das Betriebssystem muß nun garantieren, daß keine E/A-Bereiche (egal in welchem Adreßraum) von zwei Gerätetreibern verwendet werden, und über die Belegungen „Buch führen“. Sollten trotzdem unterschiedliche Gerätetreiber sich überschneidende Bereiche anfordern, liegt entweder ein Konfigurations- bzw. Implementierungsfehler oder ein echter Hardware-Konflikt vor, denn die Zuordnung von Adressen zu Geräteregistern ist eindeutig.

In PC-Systemen existiert eine weitere Ressource, die so verwaltet wird — DMA-Kanäle für ISA-Geräte. Die DMA-Controller stellen 7 Kanäle[2] für direkte Übertragung von Daten zwischen Hauptspeicher und ISA-Bus zur Verfügung. Auch hier ist die Bedingung, daß jedem Kanal nur ein Gerät zugeordnet wird. Diese muß vom Betriebssystem durchgesetzt werden.

2.4.3 Ein/Ausgabe-Steuerung

Die E/A-Steuerung ist Aufgabe des Gerätetreibers. Dabei können allgemeine Aufgaben in *Subsysteme* ausgelagert sein, die verschiedene Treiber einer Treiberklasse gemeinsam nutzen.

Bei der Betrachtung der E/A-Steuerung ist also immer ein mögliches Subsystem zusätzlich zum Gerätetreiber selbst einzubeziehen. Solche Subsysteme existieren in vielen Kernen für SCSI-, IDE- und Sound-Geräte, aber auch für sog. *Terminalgeräte*, z. B. serielle Schnittstelle oder Tastatur.

2.5 Der Linuxkern

An der Professur Betriebssysteme der TU Dresden wurden in der Vergangenheit schon mehrere Arbeiten durchgeführt, denen Gerätetreiber aus der Linuxumgebung als Grundlage dienten [Sta96, Meh96, Pau98]. Daran erkennt man, daß sich Linux aufgrund der freien Verfügbarkeit des Quelltextes und der weiten Verbreitung sehr gut für forschungsorientierte Arbeit eignet.

Deshalb wurde auch für diese Arbeit als Referenzumgebung der Kern des Betriebssystems Linux in der Version 2.4 ausgewählt. Vorweg ist aber zu erwähnen, daß der Linuxkern in den

[2] Es sind eigentlich 8 DMA Kanäle, aber Kanal 4 wird ähnlich dem Prinzip der kaskadierten PICs verwendet, um Aktivitäten am Slave-Controller zu indizieren.

vergangenen 3 Jahren seit der letzten Arbeit stetig weiterentwickelt wurde und die Erkenntnisse aus den Vorarbeiten nicht 1:1 auf die Version 2.4 übertragbar sind.

2.5.1 Linux 2.4 Gerätetreiber und -klassen

Betrachtet man den Quelltext des Linuxkerns, entfallen knapp 50 Prozent auf das `drivers/` Unterverzeichnis, enthalten also Programm-Module, mit deren Hilfe auf die Hardware zugegriffen wird. Bestandteil des Gerätetreiberanteils des Kerns sind auch umfassendere Subsysteme wie z. B. das SCSI-Subsystem.

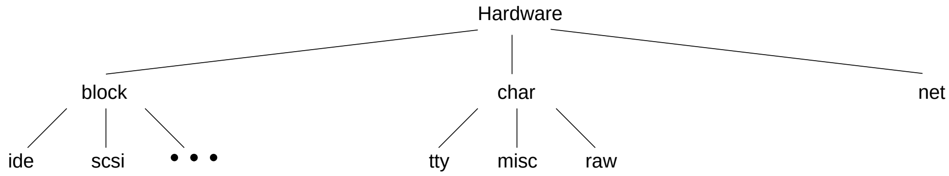

Abbildung 2.6: Linux Geräteklassen

Abb. 2.6 verdeutlicht die Einteilung der Gerätetreiber unter Linux. Auf der obersten Ebene wird in die beiden besprochenen Klassen *block* und *char* sowie in Netzgeräte unterteilt. Da der Zugriff auf die letztgenannten nur indirekt über BSD-Sockets (und den Netzwerk-Protokollstack) stattfindet, werden sie in eine eigene Gruppe ausgelagert.

Blockgeräte unter Linux können verschiedenster Natur sein. Es werden SCSI-Blockgeräte unterstützt und solche an der IDE-Schnittstelle. Weiterhin existieren auch Gerätetreiber für proprietäre Lösungen, wie z. B. ältere Schnittstellen für CD-ROMs.

Zeichengeräte werden in *tty*, *misc* und *raw* Geräte unterteilt. Genau genommen sind Geräte der ersten beiden Gruppen auch *raw*. Es wird aber noch eine Softwareschicht zwischen Gerätetreiber und Schnittstelle gelegt, die im Falle von *tty* Terminalgeräte ansteuert und für *misc* Hardware wie z. B. eine PS/2-Maus.

Bei der Identifikation der benötigten Subsysteme ist diese Unterteilung nicht ausreichend. Es ist zwar gut zu erkennen, daß eine Terminal-Treiber nicht ohne das Terminal-Subsystem, bestehend aus *Registry* und *Line Discipline*, funktioniert, ein *raw* Gerät wie ein SCSI-Streamer aber benötigt das SCSI-Subsystem bzw. bestimmte Komponenten dessen.

Eine einheitliche Unterteilung läßt sich also nur ansatzweise bzw. in Schritten durchsetzen:

1. Welche Schnittstelle exportiert der Gerätetreiber?

2. Welche Subsysteme werden zusätzlich benötigt?

Die Subsysteme können auch mehrere Treiber mit unterschiedlicher Schnittstelle gleichzeitig unterstützen. Das oben erwähnte Beispiel beschreibt diesen Fall: SCSI-Streamer und SCSI-Festplatte in einem System benutzen dasselbe Subsystem, exportieren aber verschiedene Schnittstellen.

2.5.2 Linux 2.4 Kernumgebung

Ablaufsteuerung

Das *Threading* Modell des Linuxkerns beruht auf dem blockierenden Schema, und Aktivitäten im Kern sind nicht verdrängbar. Natürlich können eintretende Hardware-Ereignisse — Interrupts — eine sofortige Preemption zur Folge haben, sofern diese nicht deaktiviert wurden[3].

Um den Kern attraktiver bzw. leistungsfähiger im Serverbereich zu gestalten, wurde großer Wert auf maximale Parallelität in Multiprozessor-Systemen gelegt und somit die in Abschnitt 2.2 erwähnten *intelligenten Sperren* weitreichend eingesetzt.

Die Synchronisation im Linuxkern stützt sich auf grundlegende Mechanismen und erweiterte Funktionen. Die grundlegenden sind:

Semaphore Die Semaphore in Linux entsprechen der klassischen Definition, d. h. ist der Semaphore nicht frei, wird die Aktivität bei Anforderung in eine Warteschlange eingekettet und gibt den Prozessor frei. Wird die Ressource wieder freigegeben, kann auch die Aktivität fortgesetzt werden. Semaphore dienen der Synchronisation von Aktivitäten auf der Prozeßebene.

Sperren von Interrupts Hiermit ist es möglich Aktivitäten auf einem Prozessor zu serialisieren, denn wenn keine Unterbrechungen erlaubt sind, müssen diese nicht asynchron bearbeitet werden. Auf diesem Wege synchronisiert man Prozeß- und Interruptebene auf einem Prozessor.

Spin Locks Diese Art der Synchronisation ist nur in Systemen mit *Symmetric Multi Processing (SMP)*, also mehreren parallel arbeitenden Prozessoren und „echter" Parallelität von Aktivitäten, interessant. Die Anforderung einer Sperre hat zur Folge, daß der Thread solange aktiv wartet — *busy waiting* — bis die Sperre frei ist und diese dann belegt.

Hier zeigt sich auch der Sinn der Einschränkung auf Multiprozessor-Systeme, denn Busy-Waiting auf den Abschluß einer parallelen Aktion auf *einem* Prozessor ist ein eindeutiger Deadlock in nicht preemptiven Umgebungen wie Linux.

Semaphore und Spinlocks in Linux sind nicht rekursiv, ein nochmaliges Anfordern der gleichen Ressource führt also zu einem Deadlock.

Verwendet man diese Funktionalitäten sinnvoll und *intelligent*, läßt sich ein hohes Maß an Parallelität erreichen. Davon profitiert z. B. der überarbeitete Netzwerkprotokoll-Stack unter SMP.

Außer diesen grundlegenden Mechanismen existieren weitergehende Funktionalitäten, die den Ablauf im Kern beeinflussen. Hierbei werden verschiedene mögliche Szenarien sehr gut

[3]Die Bearbeitung des Ereignisses wird vom Interrupt-Handler, der sog. *Linux-Top-Half*, unter Nutzung des Kernstacks des aktiven Prozesses durchgeführt.

auf abstrakte Datentypen und Funktionen abgebildet. Zu beachten ist aber, auf welcher *Ebene* man sich gerade befindet:

Prozeßebene Aktivitäten der Prozeßebene sind, wie oben erwähnt, nicht durch andere Prozeßaktivitäten verdrängbar, d. h. die Prozessorfreigabe muß freiwillig erfolgen. Diesem Zweck dienen die `schedule()` Funktion und zwei Felder in der Task-Struktur, welche den *process control block* repräsentiert. Die genannten Felder sind `state` für den Status des Prozesses (*running* oder *(un)interruptible*[4]) und `policy` für eine gewünschte Schedulingstrategie (interessant ist das `yield` Flag).

Ein Prozeßkontext kann nun freiwillig den Prozessor abgeben, in dem in seinem `policy` Feld das `yield` Flag gesetzt und anschließend `schedule()` aufgerufen wird. Andere Aktivitäten der Prozeßebene können nun die CPU belegen.

Während der Prozeß bei dem beschriebenen Vorgehen *bereit* bleibt, sein Zustand sich also nicht ändert, kann er aber auch blockieren, in dem vor dem `schedule()` Aufruf der Prozeßzustand geändert wird. In [BC01] S. 75 ist hierzu zu lesen:

> *Processes in a* `TASK_INTERRUPTIBLE` *or* `TASK_UNINTERRUPTIBLE` *state are subdivided into many classes, each of which corresponds to a specific event. In this case, the process state does not provide enough information to retrieve the process quickly, so it is neccessary to introduce additional lists of processes. These additional lists are called* wait queues.

Ein Prozeßkontext kann also auf ein bestimmtes Ereignis warten, wenn er in die korrespondierende *wait queue* eingekettet und ein `schedule()` eingeleitet wird.

Zu diesem Zweck existiert die Funktion `sleep_on()`, welche den beschriebenen Vorgang transparent macht. D. h. kehrt ein Aufruf dieser Funktion zurück, wurde der gesamte Zyklus bestehend aus Einketten, Blockieren, „Aufwachen" und Ausketten durchlaufen. Der Prozeßkontext befindet sich nun im *bereit* Zustand. Das Aufwecken übernimmt die `wake_up()` Funktion.

Es ist hierbei egal, ob ein anderer Prozeßkontext aufweckt oder ein Interruptkontext, beide Szenarien existieren und werden im Kern verwendet. Dabei werden aber nicht in jedem Fall die genannten Funktionen genutzt. Manche Gerätetreiber simulieren den Mechanismus selbständig (siehe Abschnitt 4.3).

Interruptebene Aktivitäten auf dieser Ebene haben keine korrespondierende Task-Struktur, da sie keinem Prozeß zugeordnet sind, sondern vielmehr auf *Hardware-Ereignisse* reagieren. Aus diesem Grund werden sie auch bevorzugt ausgeführt, d. h. Prozeßaktivitäten werden erst nach Ausführung aller Aktivitäten auf der Interruptebene fortgesetzt.

Aber auch auf der Interruptebene sind nicht alle Threads gleich priorisiert. So kann der *Interrupt-Handler* bei zugelassenen Unterbrechungen jede Aktivität von der CPU verdrängen.

[4] *interruptible* bedeutet hier blockiert aber durch *Signale* unterbrechbar; *uninterruptible* ist auch ein Blockierungszustand, aber meist an einer E/A-Operation und deshalb *nicht* durch Signale unterbrechbar.

Für den Zeitraum der Bearbeitung eines Interrupts sind weitere Unterbrechungen auf dieser CPU oft[5] gesperrt, so daß die nötigen Aktionen in drei Gruppen unterteilt werden:

Kritisch sind *interrupt acknowledgement* und Reprogrammierung des PIC bzw. Gerätecontrollers. Unterbrechungen *dürfen* nicht auftreten.

Nicht kritisch sind Aktionen, die Datenstrukturen aktualisieren und schnell beendet werden können. Unterbrechungen können zugelassen werden[6].

Nicht kritisch & aufschiebbar sind Aktionen wie langwierige Kopierperationen innerhalb des Hauptspeichers, die möglicherweise blockierende Funktionen benötigen.

Im Linuxkern werden Aktionen der ersten beiden Gruppen unter dem Begriff *top half* zusammengefaßt und sind Bestandteil des registrierten *Interrupt-Handlers*. Die letzte Gruppe macht die *bottom half* (BH) aus, deren Ausführung zu einem späteren Zeitpunkt stattfinden kann. Abb. 2.7 präzisiert den in Abschnitt 2.2 beschriebenen Aufbau für Linux-Gerätetreiber.

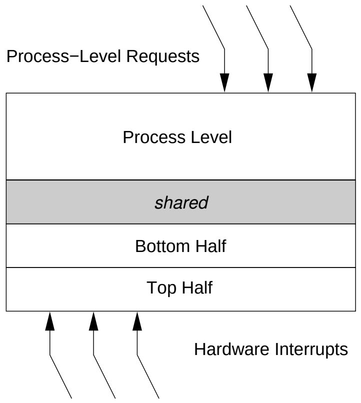

Abbildung 2.7: Prozeß- und Interruptebene im Linuxkern

Die Aktivitäten der Interuptebene können nun direkt von einem Interrupt angestoßen werden und somit in der Top Half liegen oder zu einem späteren Zeitpunkt (durch die Top Half angestoßen) als Bottom Half ausgeführt werden.

Während der fortschreitenden Entwicklung des Linuxkerns wurde dieses Konzept stetig ausgefeilt und neben den *old-style* Bottom Halves existieren auch neuere *deferred activities*:

Bottom Halves implementieren die Ursprungsidee, werden dabei aber streng serialisiert. D. h. auch in SMP-Systemen ist stets nur eine BH zu einem Zeitpunkt aktiv, um der Annahme zu entsprechen, daß keine weitere BH parallel Daten verändert.

Diese Funktionalität stammt noch aus der nicht-SMP Zeit von Linux und ist, wie man

[5]Es ist geräte- bzw. gerätetreiberabhängig ob weitere Unterbrechungen während der Bearbeitung zugelassen werden (`SA_INTERRUPT Flag`).

[6]Unterbrechungen derselben Quelle können aber nicht auftreten, da diese *maskiert* ist.

leicht erkennen kann, nicht besonders gut geeeignet für Plattformen mit mehreren Prozessoren.

Tasklets erweitern das BH-Konzept dahingehend, daß nun Typen von *deferred activities* existieren, welche serialisiert werden. Es können also in Multiprozessor-Systemen mehrere Tasklets gleichzeitig ausgeführt werden, wenn sie nicht vom gleichen Typ sind.
Tatsächlich sind alle *old-style* Bottom Halves ein Tasklettyp in Linux 2.4, so ist ihre Serialisierung bewahrt und Parallelität in Bezug auf *new-style* Tasklets gewährt.

Softirqs repräsentieren die Idee, daß *deferred activities* vollständig parallel ablaufen können, also insbesondere auch verschiedene Instanzen ein und derselben Aktivität. Diese Freiheit erzwingt aber *reentrante* Funktionen und intelligente Sperren.
Wie vorab schon erwähnt, ist die Komponente, welche von der SMP-Fähigkeit der Version 2.4 des Linuxkerns hauptsächlich profitiert, der Netzwerk-Protokollstack. Beide Pfade — Senden und Empfangen — werden hier als *softirqs* implementiert (siehe [T$^+$01] Datei `net/core/dev.c`: `net_rx_action()` bzw. `net_tx_action()`), sind also reentrante Funktionen.

In Zukunft sollen die BHs allmählich verschwinden und durch Tasklets ersetzt werden. Die *softirqs* sollen aber wichtigen Kernaktivitäten vorbehalten bleiben und sind bisher nur für den TCP/IP-Stack geplant.

Ressourcen

Der Linuxkern verwaltet Ressourcen mit Hilfe eines übersichtlichen und allgemein gehaltenen Konzepts — dem *Arbitrary resource management*. Dieses ist Bestandteil des architekturunabhängigen Teils des Kerns und kann somit für jede Architektur speziell konkretisiert werden. PC-Systeme verwalten mit dieser Kernkomponente zwei Adreßräume — E/A-Ports und E/A-Speicher (siehe auch Abschnitt 2.4.2).

Die Schnittstelle zur Komponente besteht aus Funktionen zur Anforderung und Freigabe von Bereichen des jeweiligen Adreßraumes. Weiterhin ist noch eine Funktion zur Überprüfung der Verfügbarkeit einer Ressource vorgesehen.

Zusätzlich zu diesem Konzept verwaltet der Kern mit einer ähnlichen Schnittstelle ISA DMA Kanäle und Interrupts, wobei letztere durchaus von mehreren Gerätetreibern gleichzeitig genutzt werden können — *shared interrupts*. In diesem Falle werden die bei Anforderung des Interrupts registrierten Interrupt-Handler der Reihe nach aufgerufen[7].

[7] In [BC01] S. 127: *Notice that the kernel cannot break the loop* [that executes all ISRs] *as soon as one ISR has claimed the interrupt because another device on the same IRQ line might need to be serviced.* (ISR = Interrupt Service Routine)

Ein/Ausgabe-Steuerung

Wie erwähnt finden sich alle gerätetreiber-spezifischen Komponenten im `drivers/` Unterverzeichnis der Quellen des Kerns — die Subsysteme und die Gerätetreiber selbst (Abb. 2.8).

```
...
drwxr-sr-x    3 krishna  krishna      4096 Jul  6 17:48 block/
drwxr-sr-x    2 krishna  krishna      4096 Jul  6 17:48 cdrom/
drwxr-sr-x    9 krishna  krishna      4096 Jul  6 17:48 char/
...
drwxr-sr-x   13 krishna  krishna      8192 Jul  6 17:48 net/
...
drwxr-sr-x    4 krishna  krishna      8192 Jul  6 17:48 scsi/
drwxr-sr-x    3 krishna  krishna      4096 Jul  6 17:48 sgi/
drwxr-sr-x    5 krishna  krishna      4096 Jul  6 17:48 sound/
...
```

Abbildung 2.8: Auszug des Inhalts von `drivers/`

2.6 Verwandte Arbeiten

[Sta96] René Stange beschäftigte sich in seiner Diplomarbeit am Lehrstuhl Betriebssysteme der TU Dresden mit der systematischen Übertragung von Linux-Gerätetreibern (Version 1.2) auf den Mikrokern L3. Ausgehend von der Motivation, ohne aufwendige Entwicklung eine Vielzahl von Geräten zu unterstützen, ist es gelungen die Linux-Umgebung im Zielsystem nachzubilden.

Leider wurden in der Arbeit nur Netzwerkgeräte-Treiber betrachtet, also eine spezifische Lösung geschaffen. [Sta96] hatte auch schon erkannt, daß *„das Linux-System spezifische Treiberschnittstellen für die verschiedenen Geräteklassen verwendet, weshalb keine Allgemeinbehandlung erfolgen kann“*.

Dieser Treiber wurde später auch auf L4 portiert.

[Meh96] Basierend auf [Sta96] portierte Frank Mehnert den Linux-SCSI-Gerätetreiber auf L3 und beschränkte sich damit von vornherein auf eine Geräteklasse. Grundlage war die Linuxversion 2.0.

In seiner späteren Diplomarbeit verwendete er die Ergebnisse für die Entwicklung eines *zusagenfähigen SCSI-Subsystems für DROPS* und schuf damit unter anderem einen Port des SCSI-Treibers für L4.

Beide Arbeiten sind geräteklassen-spezifische Lösungen für Linuxtreiber im Quellcode.

[Mar99] Kevin Van Maren entwickelte an der University of Utah ein *Device Driver Framework* für den FLUKE Mikrokern. Das Framework erlaubt es Gerätetreibern aus dem OSKit [Flu], auf dem Mikrokern abzulaufen. Damit stellt es einen Rahmen für den *OSKit Glue code* zur Verfügung.

Leider ist die Funktionalität auf die im OSKit enhaltenen Gerätetreiber beschränkt. Außerdem ist die Implementierung der Arbeit unvollständig und zur Zeit nicht verfügbar.

[GJP$^+$00] Der Anspruch, den SawMill Linux erfüllen soll, ist sehr gute Konfigurierbarkeit für unterschiedlichste Anforderungen. Um dieses Ziel zu erreichen, wird das Betriebssystem in wiederverwendbare Komponenten zerlegt, z.B. Filesysteme und Netzwerk-Protokollstack, die unter Nutzung von allgemeinen Komponenten, wie Speicher- und Taskmanager, auf der Basis eines Mikrokerns ablaufen.

Gerätetreiber unter SawMill Linux sind auch (mit Einschränkungen) wiederverwendbare Komponenten des Systems. Damit ist diese Architektur Drops sehr ähnlich. Lösungen für Linux *char*, *block* und *net* Treiber existieren in unterschiedlichem Umfang.

Kapitel 3

Entwurf

Das Ziel dieser Arbeit ist der Entwurf einer Umgebung für Linux-Gerätetreiber innerhalb DROPS. Die Gerätetreiber sollen als unmodifizierte Quellprogramme übernommen werden. Außerdem ist es Aufgabe der Umgebung, Konflikte verschiedener Treiber zu vermeiden und die Hardwareressourcen zu verwalten.

Während des Entwurfs der Umgebung müssen viele verschiedene Aspekte in Betracht gezogen werden, wie Synchronisation, Speicherverwaltung, gemeinsame Ressourcen und Interrupt-Zustellung. Andere Spezifikationen für Gerätetreiber-Umgebungen, z. B. [UDI99], sind deshalb sehr umfassend und beinhalten detailierte Beschreibungen der Schnittstellen und beteiligten Module.

Hier soll ein generalisierter Prozeß zur Entwicklung einer Gerätetreiber-Umgebung für mikrokernbasierte Architekturen wie DROPS von der Betrachtung der allgemeinen Eigenschaften des Originalsystems bis zu spezifischen Lösungen erarbeitet werden.

Im weiteren Text werde ich die Umgebung mit *DDE — Device Driver Environment* abgekürzt benennen.

3.1 Architektur

Wie in Abschnitt 2.1 beschrieben, ist es der Ansatz von DROPS, System-Komponenten (und Applikationen) als selbständige Server auf dem Mikrokern ablaufen zu lassen. Es liegen also in diesem Themenbereich Adreßraumgrenzen zwischen den Gerätetreibern und anderen DROPS-Servern, die durch geeignete Mechanismen des Kerns — *Inter-Process Communication (IPC)* überwunden werden können[1].

Das DDE muß den Gerätetreibern die benötigte Funktionalität aus dem Ursprungssystem zur Verfügung stellen und allen Annahmen entsprechen. Es ist offensichtlich, daß eine vollständige Emulation nicht möglich oder sinnvoll ist. Deshalb beschränkt sich das DDE auf grundlegende Funktionen, ohne welche die Treiber nicht funktionieren würden, und unterteilt

[1] Diese allgemeine Annahme kann durch bestimmte Konfigurationen überwunden werden (Abschnitt 2.1: Colocation).

diese in *zentralisierte* und *dezentralisierbare*.

Diese Unterscheidung macht es möglich, kritische bzw. zu synchronisierende Zugriffe auf globale Ressourcen in eine separate Komponente auszulagern — zentralisieren, die eine geeignete IPC-Schnittstelle anbietet.

zentralisiert	dezentralisierbar
Ressourcenverwaltung für E/A-Ports und E/A-Speicherbereiche sowie ISA DMA	Speichermanagement
Interrupt-Controller-Programmierung und Zustellung	Synchronisation und „Scheduling“ treiberlokaler Aktivitäten
PCI Bus Support	spezifische Funktionen der Originalumgebung

Tabelle 3.1: Funktionalität des DDE

Ich werde nun auf die DDE-Komponenten eingehen, welche die genannten Funktionen zur Verfügung stellen.

3.2 Zentrale Ein/Ausgabe-Unterstützung

In jedem Betriebssystem existiert eine Komponente, die Hardwareressourcen zentral verwaltet, um Konflikte bei Zugriffen zu vermeiden. Wie aus Tab. 3.1 erkenntlich ist, sollten auch die Programmierung des Interrupt-Controllers sowie PCI-Bus-Zugriffe zentral gekapselt sein.

Würde man jeder Treiberinstanz wahlfreien Zugriff gestatten, wären Konflikte vorprogrammiert, da viele Operationen nicht atomar (also inherent sequentiell) durchführbar sind und mehrere Schritte benötigen.

Ein Beispiel ist die Konfiguration eines Gerätes am PCI-Bus. Hierbei werden *nacheinander* zwei 32-Bit Ports — Adreß- und Datenport — genutzt: **1.** Geräteregister addressieren durch Schreiboperation in Adreßport und **2.** Konfigurationsregister lesen/schreiben durch Lese/Schreiboperation auf Datenport.

Im DROPS DDE ist die benötigte Komponente der *I/O Server*:

Ressourcenverwaltung Der I/O Server erlaubt es, Ressourcen anzufordern und freizugeben. Zugriffe auf nicht zugewiesene Ressourcen sind nicht erlaubt.
Obwohl diese Komponente unabhängig von den tatsächlich verwendeten Gerätetreibern ist, habe ich hier die Schnittstelle von Linux übernommen, da sie genügend abstrakt bzw. allgemeingültig ist, um auch Linux-fremde Treiber unterstützen zu können.

PCI Support Der PCI-Bus-Zugriff bzw. der Zugriff auf den *PCI Configuration Space* wird hinter einer abstrakten Schnittstelle verborgen, die es erlaubt synchronisierte gerätespezifische Konfigurationen durchzuführen.
Auch hier war Linux das Vorbild.

Interrupt-Controller Bestandteil des I/O Servers ist eine zentralisierte Interruptlogik-Behandlung, welche die Zustellung von Interrupts per IPC anbietet. Dieses Konzept wurde 1:1 aus [LH00] übernommen.

Ein Vorteil des I/O Servers ist die Möglichkeit, systemweite Politiken für Ressourcen mit Hilfe einer zentralen Instanz durchzusetzen. Ich gehe hier aber nicht näher auf diese Möglichkeit ein, da für DROPS noch kein Sicherheitskonzept entwickelt wurde und dieses nicht Gegenstand der Arbeit sein soll.

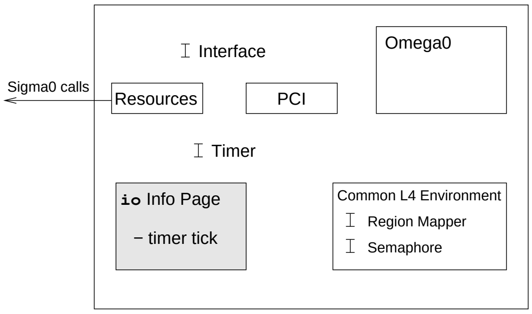

Abbildung 3.1: Aufbau des I/O Servers

Dieser Entwurf ist ein I/O Server mit dem in Abb. 3.1 dargestellten Aufbau. Die Schnittstellen werden von zwei Threads zur Verfügung gestellt, die sich beim Namensdienst als *ioserver* und *omega0* registrieren. Im Omega0-Teil ist der weitere Ablauf und das Threading wie in [LH00] beschrieben. Auf der anderen Seite erfüllt der *ioserver*-Thread Anfragen betreffend Ressourcen-Management und PCI.

Eine weitere Aktivität innerhalb des Servers ist ein *Timer*, welcher mit Hilfe der Zeitangabe des Mikrokerns einen *Timer-Tick* auf der *IO Info Page* zur Verfügung stellt. Diese Informations-Seite wird jedem Klienten während der Initialisierung eingeblendet und unterstützt eine DDE-weite Zeitbasis.

Der I/O Server stützt sich außerdem auf Funktionalitäten des *Common L4 Environment* und beinhaltet deshalb auch dessen lokale Umgebung.

3.3 Emulation der Kernumgebung

3.3.1 Threadstruktur

Am Anfang stellt sich die Frage, welche Nebenläufigkeiten in den Gerätetreibern existieren *können*. Es ist hier leider nicht möglich, eine allgemeingültige Antwort zu geben, da diese stark von der Geräteklasse abhängig ist. Soll also eine Lösung geschaffen werden, die jede mögliche Anforderung aller Treiber erfüllt, ist es schwer eine *Ein-Thread* Lösung (wie in [Sta96, Meh96]) zu entwerfen.

Potentiell nebenläufig sind zuerst alle Nutzeranfragen der Prozeßebene. Hier ist es Aufgabe des Treibers, Sperren zu nutzen bzw. jeweils nur eine Anfrage zuzulassen. Es ist auch möglich, daß ein Gerätetreiber mehrere Schnittstellen exportiert, welche auch parallel nutzbar sein sollen. Ein Beispiel sind Treiber für Soundkarten, die neben einer Schnittstelle für den Signalprozessor (PCM-Daten) auch eine solche für den Mixer-Chip exportieren.

Neben diesen Aktivitäten treten Hardware-Interrupts und zugehörige *deferred activities* auf, die auch als potentiell nebenläufig zu betrachten sind. Es ergibt sich folgende Struktur:

1. n Threads für Nutzerschnittstellen
2. ein Thread für *deferred activities*[2]
3. ein Interrupt-Thread

Vorteil dieser Struktur ist die Möglichkeit, jeder Aktivität eine eigene Thread-Priorität zuordnen zu können. So kann der Interrupt-Thread sofort auf auftretende Unterbrechungen reagieren und am Gerät die Verarbeitung anzeigen, um eine kleine Latenz zu erreichen. Die *deferred activities* sind weniger hoch priorisiert und von der ISR verdrängbar. Ähnlich verhält es sich mit den Schnittstellen-Threads, welche die geringste Priorität besitzen.

Dies emuliert das Verhalten im Linuxkern ziemlich genau, demnach eine Prozeß-Aktivität den Kern erst nach vollständiger Abarbeitung anstehender Aufgaben verläßt.

Der Nachteil ist die erhöhte Anzahl von L4-Threads im Vergleich zu den früheren Lösungen, deren Mehraufwand aber nicht sehr groß sein sollte. Außerdem unterstützen alle Gerätetreiber aus Linux inherent Nebenläufigkeiten, was auch in Hinblick auf SMP-Fiasco-Systeme interessant erscheint.

3.3.2 Aufbau der Emulation

Betrachtet man welche Funktionen des Kerns verschiedene Gerätetreiber verwenden, ist zu erkennen, daß die Ansprüche von Treibern innerhalb einer Geräteklasse sehr ähnlich sind. Ein Teil dieser Funktionen wird von *allen* Gerätetreibern genutzt. Es ist also, möglich die Kernfunktionalität in drei Gruppen zu unterteilen:

allgemein In diese Gruppe gehören die Speichermanagement-Funktionen, Synchronisation und Scheduling sowie Ressourcen-Management, PCI Unterstützung und Interrupt-Behandlung, welche jeder Gerätetreiber voraussetzt.

geräteklassen-spezifisch Klassenspezifische Funktionen verbergen sich hinter den einzelnen Subsystemen, z. B. SCSI und Sound, und den Schnittstellen zu Nutzer- oder Kernkomponenten wie dem Netzwerk-Protokollstack.

speziell sind Kernfunktionen wie z. B. die *slabs*, die selten direkt verwendet werden, aber trotzdem potentiell genutzt werden könnten.

[2] Unter SMP-Fiasco würde es sich möglicherweise anbieten, mehrere Threads zu verwenden.

Diese Aufteilung der Funktionalitäten ermöglicht eine vereinfachte Konfiguration nach den Bedürfnissen der jeweiligen Treiberklasse. Außerdem könnte durch Verkürzung von Aufrufpfaden ein Leistungsgewinn erzielt werden.

3.3.3 Allgemeine DDE Funktionalität

Speichermanagement

Gerätetreibern unter Linux stehen diverse Funktionen zur Allokation und Freigabe von Speicherblöcken mit festgelegten Eigenschaften und unterschiedlicher Granularität zur Verfügung.

Granularität	Anordnung phys. Seiten	Schnittstelle	Bezeichnung
unbestimmt	fortlaufend	`kmalloc()`	kernel memory
		`kfree()`	(kmem)
2^n Speicherseiten	fortlaufend	`get_pages()`	kernel memory pages
		`free_pages()`	
unbestimmt	fortlaufend	`kmem_cache_alloc()`	kernel memory cache
		`kmem_cache_free()`	(slabs)
unbestimmt	unbestimmt[a]	`vmalloc()`	virtual kernel memory
		`vfree()`	(vmem)

[a]aber fortlaufend im virtuellen Adreßraum

Tabelle 3.2: Speichermanagement unter Linux

Nach Tab. 3.2 können Seiten von Speicherbereichen, die mit `vmalloc()` allokiert wurden, frei im physischen Speicher verteilt sein, solange sie virtuell fortlaufend sind. Die *virtuellen* Speicherblöcke können aufgrund dieser Eigenschaft nicht für DMA verwendet werden.

Kernel Memory hingegen kann für DMA-Transfers genutzt werden. Somit müssen die verbleibenden drei Gruppen Blöcke allokieren, die fortlaufend im physischen Adreßraum auch über Seitengrenzen hinaus sind. Weiterhin muß ein schneller Weg existieren, eine virtuelle in eine physische Adresse und umgekehrt umzusetzen, da Peripheriegeräte nicht mit virtuellen Adressen umgehen können.

Ich werde hier nicht näher auf die *slabs* eingehen, welche eine spezielle Funktionalität zur Verfügung stellen, die selten direkt genutzt wird. Slabs werden in [Bon94] genau beschrieben, hier nur soviel: Slabs implementieren einen Objekt-Cache, sind also eine Optimierung für gleichförmige Speicherallokationen[3].

Da die Schnittstelle zum Speichermanagement von Linux vorgegeben wird, bleibt nun noch zu klären, welchen Weg das DDE einschlägt um die genannten Anforderungen zu erfüllen.

vmem Ich beginne mit dem virtuellen Kernspeicher, der die geringsten Ansprüche hat. Das DDE muß also einen *virtuellen Speicherpool* zur Verfügung stellen, aus dem Schritt für Schritt

[3]Im Linuxkern baut selbst das einfache `kmalloc()` auf Slabs der Größen 32, 64, 128 usw. Bytes auf. Dies wurde hier *nicht* übernommen, um die Slab-Implementierung in eine eigene Spezial-Bibliothek auslagern zu können.

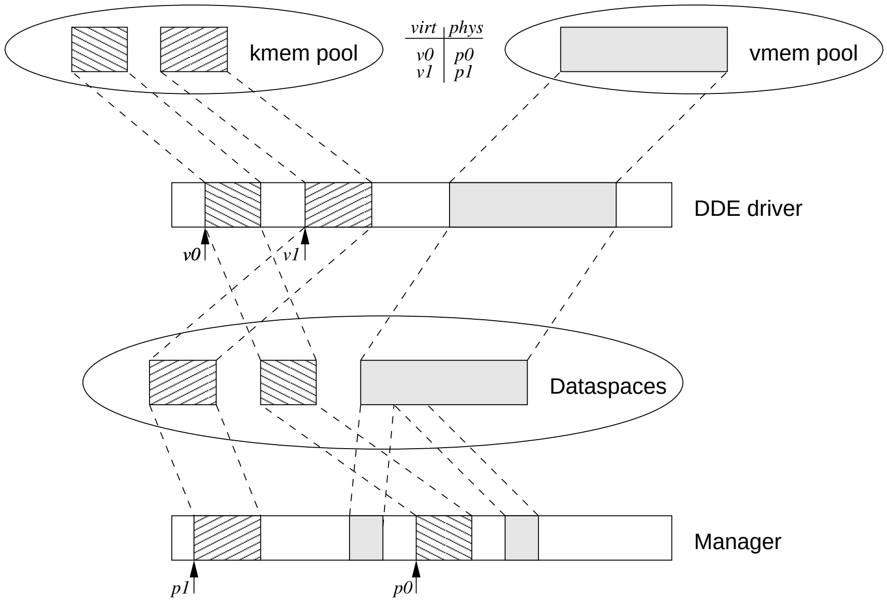

Abbildung 3.2: Speicherpools und Dataspaces (nach [L+99] Abb. 4)

Stücke (*chunks*) herausgeschnitten werden. Reicht ein *chunk* über eine Seitengrenze hinaus, ist die Anordnung der dahinterliegenden physischen Seiten nicht relevant. Die *Kacheln* des virtuellen Speicherpools müssen auch nicht über ihre ganze Lebensdauer im Speicher gehalten werden, sind also nicht *gepinnt*.

Ausgehend von [L+99] und den genannten Anforderungen[4] läßt sich Abb. 3.2 (zuerst nur die grauen Blöcke) ableiten. Danach besteht der gesamte *vmem pool* aus einem *dataspace*, dessen Speicherseiten keine speziellen Anforderungen erfüllen müssen. Seiten des Dataspace werden bei Benutzung eingeblendet, können aber später auch wieder entzogen werden.

kmem Der „normale" Kernspeicher hingegen stellt andere Ansprüche. Wie oben erwähnt, muß für jede virtuelle Adresse eine Abbildung auf eine physische existieren und die Seiten stets im physischen Speicher gehalten — *gepinnt* — werden. Allokationen von Chunks des kmem über Seitengrenzen hinaus müssen korrespondierende, physisch fortlaufende Speicherseiten garantieren — *physically contiguous memory*.

Die Lösung im DDE ist in Abb. 3.2 (linke Seite, schraffierte Blöcke) dargestellt. Der *Kern-Speicherpool* besteht hierbei aus mehreren Dataspaces, die bei Bedarf angefordert und dem Pool zugeführt werden können. Hier erfüllt jeder einzelne Dataspace die Bedingungen an kmem Chunks: gepinnt, fortlaufend und bekannte Adreßabbildung.

Diese Lösung ist vorteilhaft, da die Allokation eines großen, physisch fortlaufenden Speicherbereiches, wenn zum Startzeitpunkt überhaupt möglich, sehr statisch ist. Allokiert man initial

[4]Eine Anforderung ist natürlich, daß der *Manager* ein Dataspace-Manager für Speicher ist, der bei dieser Darstellung ein 1:1 Mapping physischen Speichers in seinem Adreßraum hält.

nur einen kleinen Bereich und fordert bei Bedarf neue Dataspaces an, ist der „Verschnitt“ dieser speziellen Speicherseiten nicht so groß. An dieser Stelle ist in Betracht zu ziehen, daß *jeder* Treiber-Server diese Bereiche benötigt, eine Verschwendung von Speicher also mehrfach ins Gewicht fällt.

Die physische Anfangsadresse wird bei Anforderung eines Dataspaces beim *Manager* erfragt und lokal verwaltet, damit bei einer Adreßumsetzung keine IPC nötig wird.

Speicherallokationen mit Seitengranularität (`get_pages()`) werden ähnlich gehandhabt. So wird eine Allokation auf die Anforderung eines passenden Dataspaces bzw. eine Freigabe auf die Rückgabe dessen abgebildet. Auch für diese Bereiche muß die Adreßabbildung ($v \leftrightarrow p$) geführt werden.

Synchronisation und Scheduling

Linux Spinlocks und Semaphore werden auf die Implementierungen des *L4 Environment* abgebildet — L4-Semaphore und L4-Locks. Das Sperren von Interrupts zur Synchronisation ist natürlich *nicht* passend für das DDE.

Hier ist der Ansatz aus [HHW98] viel interessanter, bei dem die `cli()/sti()` Paare auf eine Implementierung mit Sperren abgebildet werden. Der `interrupt_lock` wird bei einem Aufruf von `cli()` angefordert und bei `sti()` wieder freigegeben. Damit verhält sich das „Sperren von Unterbrechungen“ wie ein *globaler Mutex* und emuliert sehr gut die Originalumgebung.

Der `interrupt_lock` ist hierbei treiberlokal, d. h. in jedem Treiber-Server existiert eine solche Sperre. Das ist möglich, da alle Aktionen, bei denen Unterbrechungen wirklich ausgeschaltet sein müssen, vom Omega0-Teil des I/O Servers gekapselt werden und nur die Synchronisations-Funktion emuliert werden muß.

Bei dem Entwurf der weiterführenden Synchronisations- und Scheduling-Mechanismen muß wieder nach den beiden Ebenen im Gerätetreiber unterschieden werden:

Prozeßebene Für Schnittstellen-Threads, welche die Prozeßebene im DDE ausmachen, muß nun eine geeignete `schedule()` Implementierung entworfen werden. Die Möglichkeit, das Linux-Scheduling mit *user level threads* durchzuführen, scheidet aufgrund des Aufwand-Nutzen-Verhältnisses aus. Es ist nämlich nicht zu erwarten, daß viele Prozeßaktivitäten gleichzeitig den Treiber nutzen.

Es gibt aber eine einfachere Möglichkeit, wenn man die Stellen in den Programmquellen betrachtet, an denen von den Gerätetreibern `schedule()` aufgerufen wird. Hierbei ergeben sich zwei Anwendungen: Einmal wird von der Aktivität der Prozessor freigegeben (*yield*) und zum anderen aus irgendeinem Grund blockiert. Der Unterschied besteht darin, daß im ersten Fall der Prozeßzustand *nicht* verändert wird[5]. Ein Aufruf von `schedule()` sollte hier nur eine Prozessorfreigabe zur Folge haben — `l4_yield()`.

[5] Es wird hier (selten) das `yield` Flag gesetzt.

Wie im Abschnitt 2.5.2 (Ablaufsteuerung) beschrieben, kann aber auch der zweite Fall eintreten und die Aktivität den Prozeßzustand ändern, um zu blockieren. Das Deblockieren ist nur dann möglich, wenn der Prozeßkontext zuvor in eine *wait queue* eingekettet wurde.

Hier muß sich der Thread „schlafen legen" bis das erwartete Ereignis eingetreten ist, also ein anderer Thread „aufweckt". Dieses Verhalten läßt sich am besten mit einem kontextlokalen, binären Semaphore erreichen, der mit 0 inititalisiert wird. So kann über die Task-Struktur in `schedule()` der Semaphore angefordert und beim `wake_up()` auf die Waitqueue freigegeben werden.

An dieser Stelle zeigt sich ein Nachteil in unserer Umgebung: Die Task-Struktur umfaßt etwa 1700 Bytes und jedem Prozeßkontext wird eine Instanz zugeordnet. Andererseits werden im DDE nur wenige Elemente der Struktur wirklich benötigt — es wird also Speicher „verschwendet". Leider ist ein *Aufräumen* des `task_struct` Typen sehr aufwendig, wenn überhaupt machbar, da viele Funktionen, die als *shortcuts* dienen, bestimmte Felder der Struktur ansprechen und auch geändert werden müßten.

Die Änderungen würden sich stetig fortsetzen und die Anpassung an neue Kern-Versionen erschweren. An dieser Stelle muß noch nach einer passenden Lösung gesucht werden.

Interruptebene Auf der Interruptebene befinden sich in diesem Entwurf zwei Threads — *deferred activities* und *interrupt handler*. Der letztere meldet sich bei Bedarf für den Empfang von Nachrichten bei Eintreten eines bestimmten Interrupts beim *I/O Server* an. Trifft eine Nachricht ein, wird die registrierte ISR aufgerufen und anschließend die Verarbeitung bestätigt[6] bzw. die erneute Empfangsbereitschaft signalisiert.

Deferred activities können wie beschrieben durch die ISR angestoßen werden. Die Behandlung führt ein dedizierter Thread durch, der entweder anstehende Aufgaben erfüllt (Funktionen aufruft) oder an einem Semaphore blockiert.

Abbildung auf den I/O Server

Alle Funktionen, die im I/O Server zentralisiert wurden, müssen nun über IPC-Stubs aufgerufen werden. Das betrifft die oben angesprochene Interrupt-Behandlung genauso wie das Ressourcenmanagement und die PCI-Zugriffe. Da bei den beiden letztgenannten Komponenten die Linux-Schnittstelle weitgehend übernommen wurde, wird beim Funktionsaufruf einfach die passende I/O-Schnittstellen-Funktion aufgerufen.

Die Unterbrechungs-Behandlung wurde im vorangehenden Abschnitt schon erläutert.

Linux-Spezialitäten

Als spezielle Anforderungen von Linux-Gerätetreibern kann man *Timer* und möglicherweise auch das `/proc` Dateisystem ansehen, wobei letzteres nicht zwingend emuliert werden muß (lee-

[6] Bei der Unterstützung von *shared interrupts* ist das Verhalten etwas anders. Diese Funktionalität wird aber bisher noch nicht vom DDE unterstützt. Der Grund dafür ist, daß bisher keine Einigung über eine einheitliche Methode zum Umgang mit *shared interrupts* in der L4 Umgebung getroffen wurde.

re Funktionsrümpfe). Eine echte Emulation ist nicht besonders aufwendig und stellt (zumindest bei einigen Gerätetreibern) interessante Informationen bzw. Konfigurationsmöglichkeiten zur Verfügung.

Timer innerhalb Linux sind sogenannte *one-shot timer*, d. h. zum angegebenen Zeitpunkt (in `jiffy` ticks) wird die übergebene Funktion *einmal* ausgeführt. Die Schnittstelle besteht aus drei Funktionen, welche auf der Timerliste arbeiten: `add_timer()`, `del_timer()` und `mod_timer()` (um einen Timer zu modifizieren, falls dieser nicht schon bearbeitet wird).

Die Timer können von einem weiteren Thread ausgeführt werden, der auf den jeweils nächsten Timer-Zeitpunkt wartet und danach die Funktion aufruft. Dieser Thread muß dem eingangs entworfenen Modell noch hinzugefügt werden (siehe Abb. 3.3).

3.3.4 Spezifische DDE Funktionalität — `sound`

Die treiberspezifischen Komponenten der lokalen Umgebungen können von sehr unterschiedlichem Umfang sein und erfordern eine genaue Analyse der entsprechenden Gerätetreiber und Subsysteme. Deshalb gebe ich an dieser Stelle keinen allgemeingültigen Entwurf sondern beschreibe beispielhaft eine `sound` Umgebung.

Schnittstelle

Soundtreiber unter Linux exportieren *char* Schnittstellen für den Digital-Analog-Wandler, den Mixerchip und die MIDI-Komponente. Wieviele Schnittstellen angeboten werden, ist vom Gerätetreiber und dem tatsächlichen Gerät abhängig, da z. B. auch mehrere DSPs zur Signalumwandlung auf der Steckkarte sein können.

Die Linux-Sound-Schnittstelle ist kompatibel zum *Open Sound System*, einer platformunabhängigen Schnittstelle zu Audio-Geräten.

Eine zentrale Komponente — `soundcore` — übernimmt eine Koordinationsrolle und alle Audio-Gerätetreiber sind unter derselben *major number* erreichbar. Soll nun ein D/A-Wandler geöffnet werden, wird die `open` Routine der zentralen Komponente aufgerufen und die „wahre“ Schnittstelle als Rückgabewert geliefert. Der `soundcore` repräsentiert das *Sound Subsystem*.

Die eigentlichen Gerätetreiber registrieren sich beim Koordinator über eine definierte Schnittstelle, d. h. sie registrieren die Schnittstellen, welche sie exportieren möchten, z. B. `register_sound_dsp()` für einen D/A-A/D-Wandler.

Die Abbildung auf DROPS ist also intuitiv und originalgetreu: ein `soundcore` Thread (Koordinator) und ein Thread je Schnittstelle des registrierten Gerätetreibers. Als mögliche Optimierung bietet sich an, den Koordinator-Thread einzusparen, in dem jeder Schnittstellen-Thread beim zentralen Namensdienst registriert ist und sich potentielle Klienten direkt an die Threads wenden.

Hier ist die Frage, wie der Namensdienst unter DROPS in Zukunft aussehen wird und ob alle Informationen zentral verwaltet werden sollen. Da darüber keine Informationen existieren und eine spätere Anpassung geringen Aufwand bedeutet, habe ich mich für den Koordinator entschieden.

Für DROPS-Komponenten sollte jeder Thread eine an das OSS angelehnte IPC-Schnittstelle (möglicherweise unter Nutzung des DSI [LHR01]) exportieren.

Weitere Subsystem-Funktionalitäten

In Linux 2.4 existieren zwei Typen von Soundtreibern. Einmal sind das die sog. *native* Treiber, die direkt zum Linuxkern gehören. Daneben gibt es noch das OSS/Free-Paket, das einen Teil der kommerziell vertriebenen *Open Sound System* Gerätetreiber beinhaltet und von dessen Autor gepflegt wird.

Die OSS/Free-Treiber sind auch Bestandteil der Standard-Linuxquellen, setzen aber eine andere Umgebung voraus. Deshalb existiert eine Software-Schicht zwischen dem eigentlichen `soundcore` und den Treibern, die zum einen die Umgebung zur Verfügung stellt und zum anderen eine Abbildung auf das *native* Sound-Subsystem ist.

Es müssen also, wenn man Gerätetreiber aus OSS/Free im DDE verwenden möchte, nicht nur diese, sondern auch deren Subsystem-Wrapper in den Treiber-Server eingebunden werden.

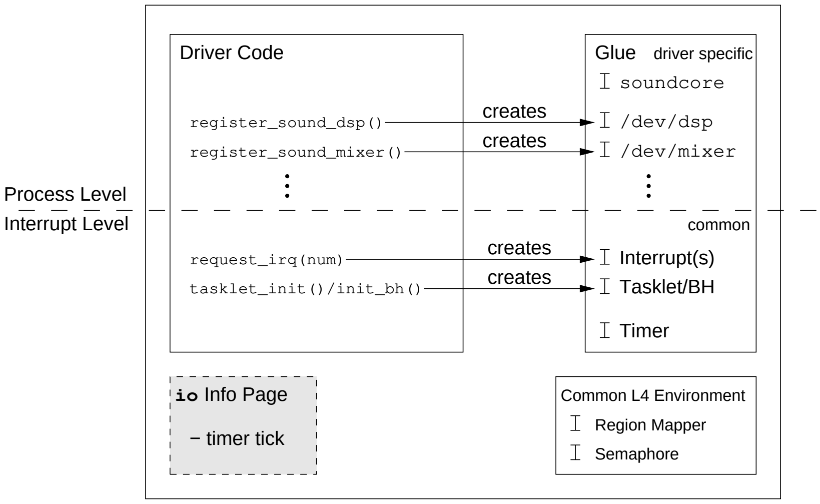

Abbildung 3.3: Aufbau eines Gerätetreibers

Den Aufbau eines Treiber-Servers illustriert Abb. 3.3. Auch hier ist (wie beim I/O Server) das lokale *Common L4 Environment* Bestandteil jedes Servers.

Zusammenfassend ergibt sich die in Abb. 3.4 dargestellte Architektur. Die einzige Systemkomponente, welche im privilegierten Prozessormodus mit eigenem Adreßraum abläuft, ist der Mikrokern.

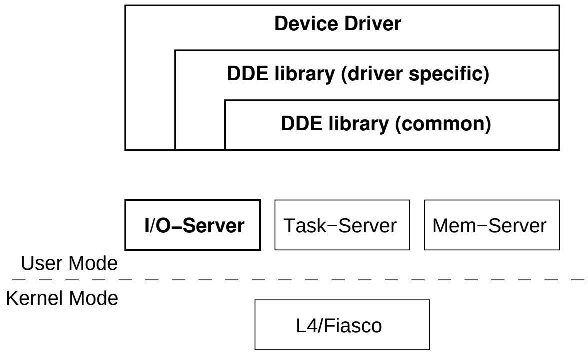

Abbildung 3.4: Endgütlige DDE Architektur

Darüber liegt eine Schicht von Laufzeitkomponenten ausgeführt als User-Mode-Server, die das *Common L4 Environment* [Reu01] unterstützen und damit die minimale Grundlage für Applikationen unter L4/Fiasco bzw. DROPS bilden. Bestandteil dieser Schicht ist der I/O Server neben Komponenten wie z. B. einem Speicher-Server, der physischen Speicher in Form von *Dataspaces* [L$^+$99] zur Verfügung stellt.

Oberhalb dieses *Runtime Layers* sind systemnahe Komponenten angesiedelt, z. B. Dateisysteme, Gerätetreiber oder der Netzwerk-Protokollstack. Der in Abb. 3.4 dargestellte Gerätetreiber besteht wie in Abschnitt 3.3 beschrieben aus drei Teilen und läuft als eigenständiger Server auf dem Mikrokern. Es kann hier auch eine Verschmelzung mit anderen Komponenten stattfinden (siehe 2.1: Colocation), um die Leistung zu steigern.

Im folgenden Kapitel werde ich auf die programmtechnische Umsetzung der Entwurfsideen und die Lösung dabei auftauchender Probleme eingehen.

Kapitel 4

Implementierung

Dieses Kapitel beschreibt die Implementierungsphase für die beiden DDE-Komponenten — I/O Server [Hel01a] und Linux-Bibliotheken [Hel01b]. Aus diesem Grund habe ich es in zwei große Abschnitte unterteilt und beginne mit den Ausführungen zum I/O Server.

4.1 Der I/O Server

Neben den folgenden Erklärungen liegt die Definition der Schnittstelle des I/O Servers (IDL) in Anhang A vor.

4.1.1 Ressourcenverwaltung

Wie in Abschnitt 3.2 entworfen, existiert im I/O Server ein Modul zur Verwaltung der Systemressourcen. Für alle drei Ressourcen — E/A-Ports und -Speicher sowie ISA DMA-Kanäle — ist je ein Schnittstellen-Kommando zur Anforderung und Freigabe vorgesehen.

Hierbei wird jede Ressource nur einmal vergeben und alle weiteren Anforderungen bis zur Freigabe durch den aktuellen Besitzer scheitern. Unterstützt wird diese Vorgabe durch die Annahme, daß z. B. *Memory Mapped I/O* Bereiche initial nur dem I/O Server zur Verfügung stehen. Dieser kann diese Bereiche — Speicherseiten — einem Klienten auf Anforderung mit Hilfe der Mechanismen des Mikrokerns einblenden. Dieser kann nun wahlfrei darauf zugreifen. Einen ähnlichen Mechanismus nutzt man für Bereiche im E/A-Port-Adreßraum[1].

Bei der zentralen Verwaltung von ISA DMA-Kanälen ergeben sich weitere Probleme, und ich gehe hier nur kurz darauf ein, da der DMA-Controller mit wenigen Ausnahmen, z. B. Floppy, in modernen Systemen — obwohl vorhanden — nicht mehr verwendet wird und meist andere Lösungen, wie *Programmed I/O*, existieren.

Der DMA-Controller in PC-Systemen kann in verschiedenen Modi verwendet werden, wobei nur der *CASCADE* Modus ohne Reprogrammierung des Controllers nach jedem Transfer verwendet werden kann. Da DMA-Kanäle über Ports programmiert werden, ergibt sich

[1] Dieser Mechanismus (*IO-Flexpages*) ist zwar für alle L4 Versionen vorgesehen, aber bisher noch nicht implementiert worden.

hier ein ähnliches Problem wie bei der Programmierung von Interrupt-Controllern (siehe Abschnitt 4.1.3).

Um eine sichere Programmierung des DMA-Controllers durchzusetzen, müßte diese im I/O Server stattfinden und über eine geeignete Schnittstelle verwendbar sein. Da aber häufige Reprogrammierung, bei Floppy z. B. maximal nach 512 transferierten Bytes, möglichst schnell sein sollte, um die Vorteile eines DMA-Transfers nicht zu neutralisieren, scheint diese Lösung nicht passend.

Eine Ausnahme ist hier der *CASCADE* Modus (auch ISA Busmastering genannt). Bei diesem Modus übernimmt das Gerät die Programmierung des Controllers wenn es den ISA-Bus belegt. Der DMA-Controller muß von CPU-Seite nur initial in diesen Modus geschaltet werden und eine weitere, explizite Synchronisation von Port-Zugriffen ist nicht nötig, da diese die Hardware verbirgt.

4.1.2 PCI Support

Der Linuxkern beinhaltet ein PCI-Subsystem aus zwei Teilen — einem architekturabhängigen und einem allgemeinen, plattformübergreifenden. Da auch in der Version 2.4 des Kerns eine eindeutig definierte Scnittstelle zu diesem existiert und nur wenige Anforderungen an die Kernumgebung gestellt werden, kann man das Subsystem einfach aus dem Kern herauslösen und in einer schmalen Emulationsumgebung verwenden.

Diese Eigenschaft macht sich der I/O Server zunutze und nutzt eine interne Bibliothek bestehend aus unmodifizierten Linuxquellen des Subsystems und der notwendigen Emulationsumgebung für Zugriffe auf den PCI-Bus und Verwaltung der angeschlossenen Geräte.

Der Umfang der Emulation beschränkt sich hierbei auf einfache Speicherallokation und -freigabe sowie Prozeduren zum Aufruf des I/O Ressourcen-Management.

Die kerninterne Schnittstelle des PCI-Subssystem wurde für den I/O Server 1:1 übernommen, und es werden somit alle Funktionen, die Linux unterstützt, exportiert. Weiterhin wurde ein Datentyp für ein PCI-Gerät nach dem Originalvorbild definiert, der alle relevanten Informationen zur Verfügung stellt.

Geräteabstraktion Der Konfigurationsbereich eines PCI-Gerätes beinhaltet Informationen wie Index und Funktionsnummer, Hersteller- und Geräteidentifikator usw, aber auch Adressen eingeblendeter Speicherbereiche (MMIO) oder Ports. Diese Informationen stellt der `l4io_pci_dev_t` Datentyp bereit. Außerdem erhält jedes Gerät ein *Handle*, um bei Zugriffen referenziert werden zu können (`l4io_pdev_t`).

Auffinden der Geräte Ein angeschlossenes Gerät kann durch Angabe verschiedener Identifikatoren lokalisiert werden. Üblich sind hier Hersteller und Typ oder die Geräteklasse. Bei einem `pci_find` Aufruf wird die gesamte Informationsstruktur als Ergebnis zurückgegeben. Diese Funktionen initiieren keinen Zugriff auf den Bus, denn die Informationen hält das Subsystem in einer Datenbank im Speicher.

PCI Konfigurationsbereich Zugriffe auf diesen Bereich können in Byte, Wort oder Doppelwort Granularität stattfinden und Lese- oder Schreiboperationen sein. Hierbei wird vorallem auf den Bereich außerhalb des Headers (die ersten 16 Doppelworte) zugegriffen, da dieser gerätespezifisch ist. `l4io_read/write` Aufrufe erfordern die Angabe des *Handles* für das gwünschte Gerät.

Weitere Funktionen unterstützen die Konfiguration und Initialisierung der PCI-Geräte für Busmastering und Power Management. Auch hier wird das *Handle* zur Identifikation des Gerätes verwendet.

4.1.3 Interrupt Behandlung

Der Teil des I/O Servers, der für die Interrupt-Behandlung und Controller-Programmierung zuständig ist, stützt sich auf die Erkenntnisse und die Referenzimplementierung nach [LH00].

Bestandteil des Servers ist somit eine weitere interne Bibliothek aus leicht[2] modifizierten Quellen des `omega0` Servers und einer minimalen Emulationsumgebung. Der I/O Server bietet damit die Omega0-Schnittstelle an und kann den eigenständigen Original-Server vollständig transparent ersetzen.

4.2 Bibliotheken für Linux-Gerätetreiber

Die lokale Emulationsumgebung für Linuxgerätetreiber stellen die `dde_test` Bibliotheken zur Verfügung. Ergebnis dieser Arbeit sind die allgemeinen Funktionen in `dde_test-common` und das Sound-Subsystem in `dde_test-sound`. Im folgenden werde ich die Implementierung dieser Komponenten beschreiben.

4.2.1 Allgemeine (`common`) Bibliothek

Speichermanagement

Zur Realisierung der beiden Speicherpools wurde die LMM[3] Bibliothek aus dem OSKit verwendet, die es erlaubt, Pools mit einer oder mehreren Regionen zu verwalten.

Im Falle des vmem Pools wird also initial eine Region angelegt und mit einem angeforderten Dataspace für Hauptspeicher assoziiert. Die Abbildung Adreßraum-Region – Dataspace wird nach [L$^+$99] von einer dedizierten Instanz — *region mapper* — innerhalb der L4-Task durchgeführt. Der Dataspace kann zu jedem Zeitpunkt „Lücken" enthalten, d. h. Seitenfehler beim Zugriff auf Adressen innerhalb der vmem Region sind erlaubt und werden vom *region mapper* aufgelöst.

[2]Der Quellcode des Omega0-Servers wurde umstrukturiert. Durch die Auslagerung der Threadinitialisierung in ein eigenes Modul kann nun entweder die Eigenimplementierung (Original-Server) oder die Threadbibliothek als Bestandteil von [Reu01] verwendet werden. Weiterhin wurde die Spezifikation des Servers dahingehend erweitert, daß der Schnittstellen-Thread nicht zwingend lthread 0 sein muß. Die Änderungen sind schon Bestandteil des Originalpakets `omega0` und stellen somit keinen selbständigen Patch dar.

[3]list-based memory manager

Der kmem Pool enthält zu Beginn auch nur eine Region, deren Dataspace aber gepinnten und physisch fortlaufenden Speicher enthält. Stellt die LMM Bibliothek bei einer Allokation fest, daß nicht genügend Speicher im Pool ist, wird eine sog. `morecore()` Funktion aufgerufen. Diese kann nun neue Bereiche in den Pool einfügen. In unserem Fall wird eine neue Region mit assoziiertem Dataspace angelegt und dem Pool zugeführt.

Eine neue Region muß angelegt werden, damit der LMM nicht aufgrund virtuell fortlaufender Adressen Bereiche allokiert, die Dataspace-Grenzen überschneiden. Dann wäre die *physically contiguous* Eigenschaft nicht in jedem Fall garantiert.

Neben dem kmem LMM Pool wird noch eine Tabelle geführt, die je Dataspace-Anfangsadresse eine Abbildung der virtuellen auf die physische Adresse verwaltet. Die physischen Adressen der gepinnten Speicherbereiche können beim *dataspace manager* erfragt werden, was auch beim Anlegen der neuen LMM Region getan wird.

Die Linux-Funktionen `virt_to_phys()` und `phys_to_virt()` arbeiten nun mit dieser Tabelle.

Scheduling

Die Semantik der Scheduling-Funktionen wurde dem entworfenen Schema angepaßt.

```
void schedule(void)
{
    switch (current->state) {
    case TASK_RUNNING:
        /* release CPU on any way */
        l4_yield();
        break;

    case TASK_UNINTERRUPTIBLE:
    case TASK_INTERRUPTIBLE:
        /* lock yourself on current semaphore */
        l4_semaphore_down(&current->dde_sem);
        break;

    default:
        ...
    }
}
```

Abbildung 4.1: Aufbau der `schedule()` Funktion

Die `schedule()` Funktion hat nun den in Abb. 4.1 dargestellten Aufbau. Ein korrespondierendes `wake_up()` ruft `l4_semaphore_up(&process->dde_sem)` auf einem Waitqueue-Element auf und der Prozeßkontext deblockiert.

Außer dieser Standard-Variante existiert noch eine `schedule_timeout(to)` Funktion, die nach maximal `to` Zeitschritten zurückkehrt. Die Implementierung basiert auf dem normalen Scheduling und einem Timer, der nach `to` ein modifiziertes `wake_up()` aufruft.

Interrupt-Behandlung

Die Interrupt-Behandlung wird in Linux von der ISR durchgeführt, die mittels `request_irq(irq, isr)` registriert wird. Im DDE wird daraufhin ein Thread angestoßen, der sich an eine Omega0-Instanz wendet und Zustellung von Unterbrechungs-Ereignissen anfordert.

```
void irq_thread()
{
    ...
    for (;;) {
        /* wait for interrupt */
        omega0_request(IRQ, ...);

        /* handle interrupt */
        IRQ_handler(IRQ, ...);
    }
    ...
}
```

Abbildung 4.2: Interrupt Thread

Beim Eintreffen eines Ereignisses wird die ISR (`IRQ_handler`) aufgerufen. Der ISR-Thread ist in Abb. 4.2 dargestellt.

Deferred Activities

Die Implementierung für *deferred activities* ist ähnlich. Auch hier wird zu Beginn ein Thread angestoßen, der aber sofort an einem Semaphore blockiert, welcher mit 0 initialisiert wurde.

```
void softirq_thread()
{
    ...
    for (;;) {
        /* softirq _consumer_ */
        l4_semaphore_down(&softirq_sema);

        /* low-priority tasks only if no high-priority available */
        if (!tasklet_hi_action())
            tasklet_action();
    }
    ...
}
```

Abbildung 4.3: *Deferred Activity* Thread

Stößt die ISR nun eine Bottom Half an, deblockiert der Thread und behandelt alle anstehenden Aufgaben. Danach legt er sich wieder schlafen und das ganze beginnt von vorn.

An dieser Stelle unterstützt das DDE bisher nur zwei Tasklet*prioritäten*, es wird also keine „echte“ *softirq* Semantik durchgeführt. Das ist aber für Gerätetreiber im Linux 2.4 hinreichend[4].

[4] Eine Änderung in Hinblick auf Softirqs wäre hier nur nötig, wenn der TCP/IP Protokollstack in der

Die beiden Prioritäten werden durch die Pfade `HI_SOFTIRQ` und `TASKLET_SOFTIRQ` repräsentiert, wobei erstere in `tasklet_hi_action()` für hoch priorisierte *deferred activities*, z. B. *old-style* BHs, ausgeführt werden und letztere in `tasklet_action()` für normal priorisierte.

Abbildung auf den I/O Server

Alle benötigten Funktionen, die der I/O Server bereitstellt werden durch Aufrufe von `io` IPC-Stubs implementiert. Hierbei sind die vorn genannten Typen (Abschnitt 4.1) in Linuxinterne zu konvertieren.

Linux Timer

Die One-Shot Timer werden, wie im Entwurf beschrieben von einem separaten Thread ausgeführt. Sind keine Aufgaben zu erfüllen, schläft der Thread (IPC Empfangsoperation), verbraucht also keine CPU Zeit. Die Timer-Listen wurden leicht verändert aus Linux übernommen (siehe [Hel01b] `lib/src/common/time.c`).

4.2.2 `sound` Bibliothek

Die `sound` Komponenten wurden bisher noch nicht vollständig von den Treibern separiert und auch die Schnittstellen-Threads existieren bisher noch nicht. Aus dem Entwurf ist aber leicht abzuleiten, wie die Bibliothek implementiert werden kann.

Den Gerätetreibern muß also ein *Sound Subsystem* zur Verfügung gestellt werden, daß die `soundcore` Schnittstelle implementiert, aber die benötigten Schnittstellen-Threads erzeugt. Der Haupt-Thread zum Startzeitpunkt ist der `soundcore` Thread. Dieser ruft die Initialisierungsroutine des Gerätetreibers auf und meldet sich danach beim Namensdienst als Soundkoordinator an.

Während der Initialisierung werden nun benötigte Threads erzeugt, z. B. erzeugt ein `register_sound_dsp()` Aufruf einen Thread für ein DSP-Gerät (Abb. 3.3). Nutzerapplikationen können sich nun über eine *OSS*-ähnliche Schnittstelle an das Gerät wenden. Dabei verbirgt die `common` Bibliothek die Synchronisation der Threads.

Emulationsumgebung ablaufen soll, da diese Komponente die einzige ist, welche ebensolche *softirqs* direkt nutzt.

4.3 Fallbeispiel: Soundtreiber `es1371.c`

Als Testgerät wurde eine Soundblaster 128 PCI Soundkarte benutzt, die auf dem es1371 Chipsatz basiert. Der korrespondierende Linux-Treiber ist `es1371.c`. Ich werde an dieser Stelle auf eine Besonderheit, den Treiber betreffend, eingehen.

Diese betrifft die Synchronisation von Prozeßkontexten über Warteschlangen. Wie ich in Abschnitt 2.5.2 beschrieben habe, existieren dazu Kernfunktionen (`sleep_on()`) deren Aufbau in Abb. 4.4 dargestellt ist.

```
void interruptible_sleep_on(wait_queue_head_t *q)
{
    ...

    current->state = TASK_INTERRUPTIBLE;

    wq_write_lock_irqsave(&q->lock,flags);
    __add_wait_queue(q, &wait);
    wq_write_unlock(&q->lock);

    schedule();

    wq_write_lock_irq(&q->lock);
    __remove_wait_queue(q, &wait);
    wq_write_unlock_irqrestore(&q->lock,flags);
}
```

Abbildung 4.4: Warteschlangen-Synchronisation (durch Signale unterbrechbar)

Der Prozeßkontext ändert seinen Zustand in *blockiert* und reiht sich in die *wait queue* ein. Anschließend wird `schedule()` aufgerufen, um die Blockierung einzuleiten und auf das Ereignis, das die Warteschlange repräsentiert, zu warten. Nach dem Aufwecken entfernt sich der Kontext selbständig aus der *wait queue*.

Der es1371 Treiber verwendet diese Funktion aber nicht, sondern bildet das Verhalten selbständig nach (Abb. 4.5). „Prophylaktisch" wird er anfangs in die Warteschlange eingereiht und am Ende der Funktion wieder entfernt. Eine Blockierung wird nur eingeleitet, wenn ein bestimmtes Ereignis nicht eingetreten ist.

Würde die Emulation an dieser Stelle erwarten, daß *wait queue* Synchronisation *nur* über die besprochene Schnittstelle durchgeführt wird, wäre der Treiber nicht lauffähig. Da aber im DDE die `schedule()` Funktion nachgebildet wird, führt auch dieser *allgemeinere* Fall zu einem funktionierenden Gerätetreiber.

Diese Besonderheit beweist, daß eine Emulation nur als bedingt allgemeingültig angesehen werden kann, da nicht abzusehen ist, welche Spezialsituationen eintreten können bzw. wieviel Phantasie einige Programmierer haben. Eine genaue Analyse des Kerns und stichprobenartig auch der Gerätetreiber selbst sollte diese Schwäche minimieren.

```
ssize_t es1371_read(...)
{
    DECLARE_WAITQUEUE(wait, current);
    ...
    add_wait_queue(&s->dma_adc.wait, &wait);
    while (count > 0) {
        ...
        if (cnt <= 0) {
            ...
            __set_current_state(TASK_INTERRUPTIBLE);
            schedule();
            if (signal_pending(current)) {
                ret = -ERESTARTSYS;
                break;
            }
            continue;
        }
        ...
    }
    remove_wait_queue(&s->dma_adc.wait, &wait);
    ...
}
```

Abbildung 4.5: Warteschlangen-Synchronisation in `es1371.c`

Kapitel 5

Leistungsbewertung

Da die Implementierung noch nicht vollständig abgeschlossen wurde, kann ich an dieser Stelle nur Abschätzungen zur Performance der Gerätetreiber innerhalb des DDE geben.

Ein Faktor, der die Leistung — genauer die Latenz — beeinflußt, ist der Ansatz eines Omega0-Servers. Da in Mikrokernen Unterbrechungsereignisse per IPC zugestellt werden, sind Verzögerungen bis zum *acknowledgement* in diesen Systemen größer als bei monolithischen Ansätzen. Der Omega0-Server verschärft dies, in dem er noch eine Indirektionsstufe zwischen Interruptquelle und Treiber einführt. Somit sind für jeden Interrupt zwei IPC-Nachrichten bis zum Treiber-Server nötig.

Die Synchronisation im DDE basiert auf den standard Lock- und Semaphore-Implementierungen. Diese nutzen *atomare* Operationen und im Blockierungsfall IPC. Dadurch ist zumindest das Blockieren und Aufwecken aufwendiger als in Linux — es bedarf zwei IPC-Nachrichten.

Weiterhin ist eine zusätzliche Verzögerung zu erwarten, wenn der Kern-Speicherpool vollständig belegt ist. Hier müssen von der `morecore()` Funktion neue Dataspaces angefordert und dem Pool zugeführt werden. Dieser Vorgang beinhaltet mehrere IPCs innerhalb der L4-Task (region mapper) und nach außen (dataspace manager). Vermindern kann man diese Auswirkungen, in dem man eine Analyse des Speicherbedarfes des Gerätertreiber-Servers durchführt und die Größe des initialen Pools entsprechend den Ergebnissen anpaßt.

Insgesamt gesehen ist der Faktor *IPC* sehr wichtig in mikrokernbasierten Systemen. Da der L4- bzw. Fiasco-Mikrokern eine *schnelle* IPC zur Verfügung stellt, ist zu erwarten, daß der Einfluß der Architektur auf die Leistung nicht sehr groß ist.

Kapitel 6

Zusammenfassung und Ausblick

Im Rahmen dieser Diplomarbeit wurde eine Umgebung für Linux-Gerätetreiber unter DROPS entworfen. Dabei wurde der Weg von der genauen Analyse der Originalumgebung bis zur Implementierung der Laufzeitkomponenten beschrieben, um einen generalisierten Prozeß zur Entwicklung solcher Umgebungen aufzuzeichnen. Als Fallbeispiel dienten Gerätetreiber für Soundkarten unter Linux.

Es zeigte sich im Laufe der Arbeit, daß die Portierung machbar ist, eine Leistungsmessung wurde aber aufgrund der unvollständigen Implementierung nicht durchgeführt. Ich konnte hier nur aus den Vorarbeiten schließen, daß die Performance-Einbußen in einem vernünftigen Rahmen liegen werden.

Für die Zukunft dieses Projektes stellt sich primär die Aufgabe, die Implementierung zur vervollständigen und auf andere Geräteklassen zu erweitern. Danach kann eine Leistungsanalyse stattfinden, um die im letzten Absatz genannte Annahme zu überprüfen.

Ein anderer Aspekt für die Umgebung wäre es, nicht nur Gerätetreiber zu portieren, sondern auch andere Komponenten, z. B. den TCP/IP-Stack aus dem Linuxkern herauszulösen und unter DROPS zu verwenden.

Zuletzt ist es noch interessant, welche Auswirkungen bzw. Vorteile ein funktionierendes SMP-Fiasco System auf die entwickelte Umgebung hat, denn an dieser Stelle sollte sich das *multi-threaded* Konzept auszahlen.

Anhang A

Schnittstellen-Definition: I/O Server

This can be found in the `io` package [Hel01a] (`io/idl/io.idl`).

```
/*
 * CORBA IDL based function interfaces for L4 systems
 */
module l4
{
    /*
     * L4 io server interface
     */
    interface io
    {
        /*
         * Miscellaneous Services
         */

        /*
         * register new io client
         */
        long register_client(in l4_io_drv_t type);

        /*
         * unregister io client
         */
        long unregister_client();

        /*
         * initiate mapping of io info page
         */
        long map_info(out fpage info);

        /*
         * Resource Allocation
         */

        /*
         * register for exclusive use of IO port region
         */
        long request_region(in unsigned long addr,
                            in unsigned long len);

        /*
         * release IO port region
         */
        long release_region(in unsigned long addr,
                            in unsigned long len);

        /*
         * register for exclusive use of ‘IO memory’ region
         */
         ...

        /*
         * PCI Services
```

```
         */

        /*
         * search for PCI device by vendor/device id
         */
        long pci_find_device(in unsigned short vendor_id,
                             in unsigned short device_id,
                             in l4_io_pdev_t start_at,
                             out l4_io_pci_dev_t pci_dev);
         ...

        /*
         * read configuration BYTE registers of PCI device
         */
        long pci_read_config_byte(in l4_io_pdev_t pdev,
                                  in long offset, out octet val);
         ...

        /*
         * write configuration BYTE registers of PCI device
         */
        long pci_write_config_byte(in l4_io_pdev_t pdev,
                                   in long offset, in octet val);
         ...
    };
};
```

This can be found in the `omega0` package (`l4/omega0/client.h`).

```
/* attach to an irq line */
extern int omega0_attach(omega0_irqdesc_t desc);

/* detach from an irq line */
extern int omega0_detach(omega0_irqdesc_t desc);

/* request for certain actions */
extern int omega0_request(int handle, omega0_request_t action);
```

This can be found in the `io` [Hel01a] (`l4/io/types.h`) and `omega0` packages (`l4/omega0/client.h`).

```
/* l4io types */

typedef unsigned long l4io_pdev_t;

typedef struct {
    unsigned long   devfn;       /* encoded device & function index */
    unsigned short  vendor;
    unsigned short  device;
    unsigned short  sub_vendor;
    unsigned short  sub_device;
    unsigned long   class;       /* 3 bytes: (base,sub,prog-if) */

    unsigned long   irq;
    l4io_res_t res[12];          /* resource regions used by device:
                                  * 0-5  standard PCI regions (base addresses)
                                  * 6    expansion ROM
                                  * 7-10 unused for devices */
    char            name[80];
    char            slot_name[8];

    l4io_pdev_t     handle;      /* handle for this device */
} l4io_pci_dev_t;

/* omega0 types (look there ...) */

typedef struct {
   ...
} omega0_irqdesc_t;

typedef struct {
   ...
} omega0_request_t;
```

Anhang B

Glossar

BH (bottom half) siehe *deferred activities*

deferred activities *verzögerte Aktivitäten* sind von Unterbrechungen angestoßene Kernaktivitäten außerhalb der ISR. Linux z. B. unterscheidet diese in Bottom Halves, Tasklets und Softirqs.

IPC (InterProcess Communication; engl. Interprozeßkommunikation) Mechanismus des Datenaustausches zwischen verschiedenen Programmen (in unterschiedlichen Adreßräumen)

ISR (Interrupt Service Routine) Funktion, die bei Auftreten einer Unterbrechung zur Behandlung dieser aufgerufen wird

Mikrokern Betriebssystemkerne, deren Funktionalität auf den sog. *Nucleus* beschränkt wurde. Es wird also eine minimale Basis im privilegierten Prozessormodus ausgeführt und alle anderen Systemkomponenten sind Mikrokern-Server im Nutzermodus.

PCM (Pulse Code Modulation) Verfahren zur Digitalisierung analoger Daten

QoS (Quality of Service) Die Möglichkeit Leistungsangaben in Datenkommunikationssystemen zu treffen.

Terminal Ein/Ausgabegerät für Computer mit Tastatur und Monitor

Literaturverzeichnis

[BBH+98] Robert Baumgartl, Martin Borriss, Hermann Härtig, Claude-Joachim Hamann, Michael Hohmuth, Lars Reuther, Sebastian Schönberg, and Jean Wolter. Dresden Realtime Operating System. In *Proceedings of the First Workshop on System Design Automation (SDA'98*, pages 205–212, Dresden, March 1998.

[BC01] Daniel P. Bovet and Marco Cesati. *Understanding the LINUX Kernel*. O'Reilly & Associates, Inc., 1st edition, January 2001.

[Bon94] Jeff Bonwick. The Slab Allocator: An Object-Caching Kernel Memory Allocator. Technical report, Sun Microsystems, 1994.

[Flu] Flux Research Group. *The OSKit*. University of Utah.
Available from URL: `http://www.cs.utah.edu/flux/oskit/`.

[GJP+00] Alain Gefflaut, Trent Jaeger, Yoonho Park, Jochen Liedtke, Kevin Elphinstone, Volkmar Uhlig, Jonathon Tidswell, Luke Deller, and Lars Reuther. The SawMill Multiserver Approach. In *ACM SIGOPS European Workshop*. IBM T.J. Watson Research Center, University of Karlsruhe, TU Dresden, 2000.
URL: `http://www.research.ibm.com/sawmill/`.

[Hel01a] Christian Helmuth. I/O Server Source Code, 2001. First `io` Reference Implementation.

[Hel01b] Christian Helmuth. Linux Device Driver Environment Source Code, 2001. First `dde_test` Reference Implementation.

[HHW98] Hermann Härtig, Michael Hohmuth, and Jean Wolter. Taming Linux. In *Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time Systems PART '98*, Adelaide, Australia, 1998.

[L+99] Jochen Liedtke et al. A Framework for VM Diversity. Technical report, IBM T. J. Watson Research Center, 1999.

[LH00] Jork Löser and Michael Hohmuth. Omega0: A portable interface to interrupt hardware for L4 systems. In *Workshop on Common Mikrokernel System Platforms*, January 2000. Jewel Edition.

[LHR01] Jork Löser, Hermann Härtig, and Lars Reuther. A Streaming Interface for Real-Time Interprocess Communication. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Schloss Elmau, Bavaria, Germany, May 2001.

[Mar99] Kevin Van Maren. The FLUKE Device Driver Framework. Master's thesis, University of Utah, 1999.

[Meh96] Frank Mehnert. Portierung des SCSI-Gerätetreibers von Linux auf L3. Belegarbeit, TU Dresden, 1996. In German.
Available from URL: `http://wwwos.inf.tu-dresden.de/L4/`.

[Pau98] Torsten Paul. Entkopplung von Echtzeit- und Timesharing-Aktivitäten am Beispiel einer echtzeitfähigen Darstellungskomponente in Drops. Diplomarbeit, TU Dresden, 1998. In German.
Available from URL: `http://wwwos.inf.tu-dresden.de/L4/`.

[Reu01] Lars Reuther. Towards A Common Environment For L4 Applications. Unpublished, 2001.

[Sta96] René Stange. Systematische Übertragung von Gerätetreibern von einem monolithischen Betriebssystem auf eine mikrokernbasierte Architektur. Diplomarbeit, TU Dresden, 1996. In German.
Available from URL: `http://wwwos.inf.tu-dresden.de/L4/`.

[T+01] Linus Torvalds et al. Linux Kernel Source Code, 2001. Version 2.4.2,
Available from URL: `ftp://ftp.kernel.org`.

[UDI99] Project UDI: Uniform Driver Interface, 1999.
Available from URL: `http://www.project-UDI.org`.

[W+95] Werner et al., editors. *Taschenbuch der Informatik*. 1995.