Georg-Simon-Ohm-Hochschule
Fakultät Informatik

WS 2012/13

Bachelorarbeit

Implementierung eines Dateisystems und einer RAM-Disk für das Lehrbetriebssystem ULIX-i386

mit Literate Programming

Vorgelegt von Liviu Beraru
Matrikelnummer 2137329
Erstprüfer Prof. Dr. rer. pol. Ralf Ulrich Kern
Zweitprüfer Dipl.-Math. Dipl.-Inform. Hans-Georg Eßer
Ausgabedatum Nürnberg, den 1. Oktober 2012
Abgabedatum Nürnberg, den 18. Februar 2013

I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be works of literature. Hence, my title: "Literate Programming."

Donald Knuth [9]

Eidesstattliche Erklärung

Ich, Liviu Beraru, Matrikel-Nr. 2137329, versichere hiermit, dass ich meine Bachelorarbeit mit dem Thema

> Implementierung eines Dateisystems und einer RAM-Disk für das Lehrbetriebssystem ULIX-i386

selbstständig verfasst, nicht anderweitig für Prüfungszwecke vorgelegt, und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, wobei ich alle wörtlichen und sinngemäßen Zitate als solche gekennzeichnet habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Nürnberg, den 18. Februar 2013

LIVIU BERARU

Inhaltsverzeichnis

Abbildungsverzeichnis xi

1 Einleitung 1
- 1.1 Literate Programming . . . 1
  - 1.1.1 noweb . . . 3
  - 1.1.2 Große Projekte . . . 4
- 1.2 ULIX . . . 5

2 UlixFS 7
- 2.1 Begriffe . . . 7
  - 2.1.1 Block . . . 7
  - 2.1.2 Disk . . . 8
  - 2.1.3 Datei . . . 8
- 2.2 Aufbau des Dateisystems . . . 8
  - 2.2.1 Bootblock . . . 9
  - 2.2.2 Superblock . . . 10
  - 2.2.3 Inode- und Block-Bitmap . . . 11
  - 2.2.4 Inodetabelle . . . 13
  - 2.2.5 Datenregion . . . 13
- 2.3 Inodes . . . 13
- 2.4 Datei Modus (`mode_t`) . . . 16
  - 2.4.1 Dateityp . . . 16
  - 2.4.2 Zugriffsrechte . . . 16
- 2.5 Verzeichnisse . . . 17
- 2.6 Verzeichnisbaum . . . 18

3 Gerätemodell 21
- 3.1 Geräte und Einheiten . . . 21
  - 3.1.1 Major- und Minor-Nummern (`dev_t`) . . . 21
- 3.2 Treiberschnittstelle . . . 22
- 3.3 Treibertabelle . . . 25
  - 3.3.1 Generische Geräte-Funktionen . . . 27
- 3.4 RAM-Disk Treiber . . . 30
  - 3.4.1 Was ist eine RAM-Disk? . . . 31
  - 3.4.2 Öffentliche Schnittstelle . . . 31
  - 3.4.3 Verwendung . . . 31
  - 3.4.4 Interaktion mit ULIX . . . 33
  - 3.4.5 RAM-Disk-Tabelle . . . 33
  - 3.4.6 Implementierung . . . 34

4 mkfs.ulixfs **41**
4.1 Argumente einlesen . . . 42
4.2 Dateisystem generieren . . . 44
4.2.1 Einträge des Superblocks ausrechnen . . . 45
4.2.2 Diskregionen schreiben . . . 48
4.3 Dateien in das Dateisystem einfügen . . . 54
4.3.1 Funktionen für Diskregionen . . . 58
4.3.2 Inode-Funktionen . . . 60
4.3.3 Block-Funktionen . . . 62

5 Implementierung **67**
5.1 Blöcke und Bitmaps . . . 68
5.1.1 Blöcke lesen und schreiben . . . 68
5.1.2 Superblock lesen . . . 69
5.1.3 Bitmap-Operationen . . . 70
5.2 Inodes . . . 74
5.2.1 Inodes allokieren und löschen . . . 75
5.2.2 Inodes lesen und schreiben . . . 75
5.3 Dateitabelle . . . 77
5.3.1 Struktur der Dateitabelle . . . 78
5.3.2 Freien Eintrag finden . . . 80
5.3.3 Inode laden und Dateideskriptor erzeugen . . . 81
5.3.4 Eintrag löschen . . . 82
5.3.5 Eintrag lesen und schreiben . . . 82
5.3.6 Zugriffsposition setzen . . . 84
5.3.7 Dateieintrag sichern . . . 85
5.3.8 root lesen . . . 85
5.3.9 root setzen . . . 86
5.4 Einhängen . . . 87
5.4.1 Einhängen . . . 88
5.4.2 Einhängepunkt abfragen . . . 89
5.4.3 Aushängen . . . 89
5.5 Dateipfade . . . 90
5.5.1 Dateiname im Verzeichnis suchen . . . 91
5.5.2 Pfad nach Inodenummer auflösen . . . 93
5.6 Dateien öffnen und schließen . . . 97
5.7 Lesen und Schreiben . . . 99
5.7.1 Datei lesen . . . 99
5.7.2 Datei schreiben . . . 103

A POSIX Hearderdateien **107**
A.1 Datentypen (`types.h`) . . . 107
A.2 Status (`stat.h`) . . . 108
A.3 Standard Makros (`stddef.h`) . . . 111
A.4 Dateikonstanten (`fcntl.h`) . . . 112

B Integration in ULIX **113**
B.1 Multiboot Header . . . 113

B.2 Test Modul 114
B.2.1 Hilfsfunktionen 116
B.2.2 Dateisystem initialisieren und testen 119

Literaturverzeichnis **125**

Index **127**

Abbildungsverzeichnis

1.1 Workflow mit `noweb` 4
1.2 Größeres Projekt mit Literate Programming 5

2.1 Aufbau des Dateisystems ULIXFS 9
2.2 Superblock Beispiel 11
2.3 Bitmap Beispiel 12
2.4 Inode des ULIXFS Dateisystems 14
2.5 Einteilung der Bits im `mode` Feld 16
2.6 Inhalt einer Verzeichnisdatei 18
2.7 Verzeichnisbaum als Netz von Verweisen 19

3.1 Geräte und Einheiten 21
3.2 Der Typ `dev_t` 22
3.3 Treiberarchitektur 22
3.4 Treiberarchitektur 23
3.5 Tabelle virtueller Funktionen 26

5.1 Hierarchie der Module in ULIXFS 67
5.2 Dateitabelle 80
5.3 Einhängetabelle 87
5.4 Zerlegen eines Pfads 95
5.5 Datei lesen – Variablen 103

A.1 Überprüfung des Dateityps 110
A.2 Überprüfung der Zugriffsrechte 111

1 Einleitung

In dieser Bachelorarbeit implementieren wir das Dateisystem ULIXFS, das in dem Betriebssystem ULIX zum Einsatz kommt. ULIXFS wird im Kapitel 2 vorgestellt und im Kapitel 5 implementiert. Zusätzlich entwickeln wir eine allgemeine Treiberschnittstelle im Kapitel 3 und den RAM-Disk-Treiber als Implementierung dieser Schnittstelle im Abschnitt 3.4. Zur Generierung einer RAM-Disk-Image schreiben wir ein Programm namens `mkfs.ulixfs` im Kapitel 4. Im Anhang B.2 findet sich der Code, der die RAM-Disk in ULIX lädt und der die wichtigsten Funktionen von ULIXFS getestet.

Für die Implementierung dieser Komponenten und Erstellung der Dokumentation, die unsere Bachelorarbeit ausmacht, verwenden wir die Programmiertechnik „Literate Programming“, die im folgenden Abschnitt kurz beschrieben wird.

1.1 Literate Programming

The practitioner of literate programming can be regarded as an essayist, whose main concern is with exposition and excellence of style.

Donald Knuth [9]

1978 wurde Donald Knuth von Tony Hoare gefragt, ob er das Programm TEX veröffentlichen möchte. Seine Begründung war, dass es zu der Zeit nur wenig große Programme gab, die man als Lernmaterial benutzen konnte. Die meisten Programme, die in den Universitäten als Lernmaterial dienten, waren zu kurz. TEX, damals noch in Pascal geschrieben, war laut Hoare ein guter Kandidat für ein Vorzeige-Programm aus dem „echten“ Leben und Knuth sollte es in Buch- oder Artikelform veröffentlichen.

Knuth meinte dazu, der Programmcode von TEX würde nicht die Qualität haben, die man für eine Veröffentlichung benötige. Als bekannter Professor für Informatik, wollte er keine Programme veröffentlichen, die nicht bestimmte Qualitätsanforderungen erfüllen. Knuth schreibt in [9] dazu:

> Hoare's suggestion was actually rather terrifying to me, and I'm sure he knew that he was posing quite a challenge. As a professor of computer science, I was quite comfortable publishing papers about toy problems that could be polished up nicely and presented in an elegant manner; but I had no idea how to take a piece of real software, with all the compromises necessary to make it useful to a large class of people on a wide variety of systems, and to open it up to public scrutiny. How could a supposedly respectable academic, like me, reveal the way he actually writes large programs? And could a large program be made intelligible?

Er begann daraufhin an einem System zu arbeiten, womit man auch einen großen Programmcode verständlich machen kann. Er stand vor der Frage: wie kann man ein Programm darstellen, dass man es wie ein Buch lesen kann? Wie kann man Programmcode nicht nur in einer verständlicher Weise, sondern auch mit einer hohen typografischen Qualität präsentieren?

1981 veröffentlicht Knuth „WEB“, ein Literate Programming System, das aus zwei Komponenten besteht:

1. Die Programmiersprache WEB, mit der sich „Literate Programs“ schreiben lassen. Eine Literate-Programming-Datei besteht aus Dokumentation und Programmcode gemischt. Der Programmcode wird speziell markiert. Für die Dokumentation wir TEX verwendet.
2. Zwei Tools: `weave` und `tangle`. Mit `weave` lässt sich aus einer Literate-Programming-Datei der TEX-Code und mit `tangle` der Pascal-Code extrahieren, denn die erste Version von WEB verwendete noch Pascal. Siehe Abbildung 1.1, Seite 4.

In einer Literate-Programming-Datei ist die Reihenfolge der Programm-Abschnitte nicht an der Syntax der verwendeten Programmiersprache gebunden, sondern folgt der inneren Logik und Notwendigkeit, die vom Programmierer bestimmt wird. Dieser geht dabei wie ein Essayist vor, der ein literarisches Werk schreibt. Knuth spricht in diesem Kontext auch von einem „stream of consciousness“, das die Reihenfolge der Ausführungen bestimmt. Er formuliert das in [9] wie folgt:

> When I tear up the first draft of a program and start over, my second draft usually considers things in almost the same order as the first one did. Sometimes the “correct” order is top-down, sometimes it is bottom-up, and sometimes it’s a mixture; but always it’s an order that makes sense on expository grounds.
>
> Thus the WEB language allows a person to express programs in a “stream of consciousness” order. TANGLE is able to scramble everything up into the arrangement that a PASCAL compiler demands. This feature of WEB is perhaps its greatest asset; it makes a WEB-written program much more readable than the same program written purely in PASCAL, even if the latter program is well commented. And the fact that there’s no need to be hung up on the question of top-down versus bottom- up – since a programmer can now view a large program as a web, to be explored in a psychologically correct order – is perhaps the greatest lesson I have learned from my recent experiences.

Mit WEB entstand das erste Literate-Programming-System, das die weitere Entwicklung dieser Programmiertechnik bestimmen sollte. Aus WEB lassen sich drei Eigenschaften entnehmen, die ein Literate-Programming-System ausmachen:

1. Programmcode und Dokumentation können in einer Literate-Programming-Datei beliebig gemischt werden.
2. Lesbare Dokumentation mit Inhaltsverzeichnis, Index etc. kann automatisch erstellt werden. Die Dokumentation wird mit der Komponente `weave` extrahiert.
3. Programm-Abschnitte können in beliebiger Reihenfolge angeordnet und kombiniert werden. Dabei extrahiert die Komponente `tangle` den Programmcode so, dass er von einem Compiler akzeptiert wird.

1.1.1 noweb

Das System WEB von Knuth war auf Pascal und TEX abgestimmt. Später entwickelt Knuth das System CWEB, das mit der Programmiersprache C statt Pascal arbeitet. Diese Entwicklung veranlasste Norman Ramsey, 1989 bis 1999 ein neues System namens *noweb* – von Norman-WEB – zu entwickeln, das auf Sprachunabhängigkeit setzt und zudem LATEX und andere Ausgabeformate wie HTML unterstützt. noweb hat dieselben Komponenten wie WEB, aber `noweave` und `notangle` genannt. Die Syntax von noweb ist, im Gegensatz zu CWEB, sehr einfach. Das folgende Beispiel zeigt, wie man mit noweb einen Code-Abschnitt definiert:

```
Den Algorithmus definieren wir folgenderweise:
<<Binäre Suche>>=
int binary_search
(int list[], int left, int right, int target) {
    while (left <= right) {
        <<Berechne mittlere Position>>
        <<Entscheide den nächsten Schritt>>
    }
    return -1;
}
@
```

Dieser Code fängt mit einem Beschreibungstext an, der in diesem Beispiel aus einem kurzen Satz besteht. Danach wird ein Code-Abschnitt („chunk“) mit der Syntax „`<<name>>=`“ und Namen „Binäre Suche“ definiert. In diesem Abschnitt wird eine Funktion definiert, die wiederum zwei Code-Abschnitte verwendet: „Berechne mittlere Position“ und „Entscheide den nächsten Schritt“. Da es sich um Literate Programming handelt, ist es unwichtig, wo diese Code-Abschnitte definiert werden. Die Definition von „Berechne mittlere Position“ kann vorher oder nachher erfolgen, je nachdem, wie der Programmierer es für sinnvoll hält. Die Definition des Abschnitts wird mit `@` beendet. Der „literarische“ Programmierer könnte diesen Abschnitt wie folgt definieren:

```
In jedem Schritt berechnen wir die mittlere
Position zwischen \texttt{left} und \texttt{right}.
<<Berechne mittlere Position>>=
int middle = low + (low + hight)/2;
@
```

Was man merkt ist, dass dieser Code-Abschnitt frei an einer beliebigen Stelle in der Literate-Programming-Datei definiert wird: sein Kontext und Inhalt wird allein von der inneren Logik („stream of consciousness“) des Programmierers bestimmt. Darüberhinaus verwendet der Abschnitt LATEX-Befehle, die sich auf die Ausgabe auswirken.

Die Abbildung 1.1 zeigt den Workflow mit noweb. Angenommen, die Literate-Programming-Datei hat den Namen `prog.nw`, so kann man die LATEX Dokumentation wie folgt mit dem Tool `noweave` extrahieren:

```
noweave -delay prog.nw > prog.tex
```

Es wird dabei die Datei `prog.tex` erzeugt, die später in einer anderen LATEX-Datei inkludiert werden kann. Lässt man die Option `-delay` weg, so ergänzt `noweave` die Ausgabe von `prog.nw` mit den LATEX-Befehlen `\documentclass`, `\begin{document}` und `\end{document}`. Somit könnte man `prog.tex` mit `pdflatex` direkt kompilieren.

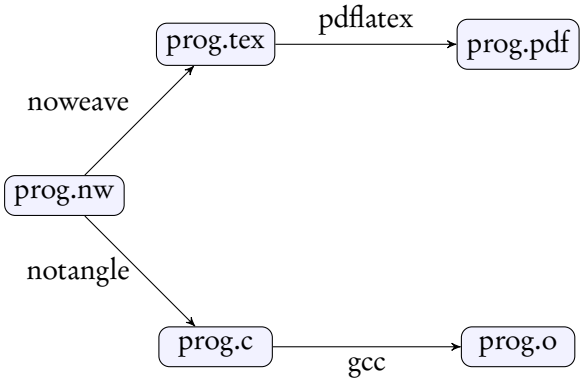

Abbildung 1.1: Workflow mit `noweb`: Dateien und Tools.

Um die C-Datei aus `prog.nw` zu extrahieren, benutzt man das Tool wie folgt:

```
notangle -R"Binäre Suche" prog.nw > prog.c
```

Damit wird der Code-Abschnitt „Binäre Suche“ zusammengebaut und in die Ausgabedatei `prog.c` geschrieben. Diese Datei kann dann mit einem C-Compiler wie `gcc` kompiliert werden (siehe Abbildung 1.1).

Mehr Details und eine ausführlichere Dokumentation finden sich auch in [12] und [13].

1.1.2 Große Projekte

Eine berechtigte Frage ist die folgende: Eignet sich Literate Programming für größere Projekte? Diese Frage beantworten wir zuversichtlich mit Ja. Man betrachte dazu die Abbildung 1.2, die einen Ausschnitt aus unserer Bachelorarbeit darstellt, die mit Literate Programming geschrieben wurde. Auf der rechten Seite befindet sich eine Reihe von C-Dateien, die kompiliert werden. Auf der linken Seite befindet sich eine Reihe von LaTeX-Dateien, die in einer übergeordneten Datei namens `bachelorarbeit.tex` inkludiert werden. Kompiliert man `bachelorarbeit.tex`, so erhält man die Dokumentation unserer Arbeit in PDF Version.

Die übliche Vorgehensweise in der Softwareentwicklung ist die Programmdateien auf der rechten Seite direkt zu schreiben und die Dokumentationsdateien auf der linken Seite entweder getrennt zu schreiben oder mittels Tools wie Javadoc oder Doxygen aus den C-Dateien zu extrahieren (siehe [11] und [3]). Die Vorgehensweise mit Literate Programming ist gemeinsame Dokumentations- und Programmdateien zu erstellen (mittlere Spalte in Abbildung 1.2) und daraus die anderen Dateien zu extrahieren. Dafür lassen sich Makefiles gut verwenden.

Wie man aus der Abbildung 1.2 entnehmen kann, können in der Hauptdokumentationsdatei noch andere Dateien inkludiert werden, die nicht aus Literate-Programming-Dateien extrahiert werden (Datei `Begriffe.tex`). In diesen Dateien werden Konzepte entwickelt, das Projekt vorgestellt usw. Auf diese Weise sind große Projekte nicht nur möglich, sondern Literate Programming eröffnet neue Wege für eine viel tiefere und übersichtlichere Dokumentation eines Projektes, als es sonst möglich ist. Denn hier kann die ganze Fülle an Funktionalität, die LaTeX anzubieten hat, verwendet werden.

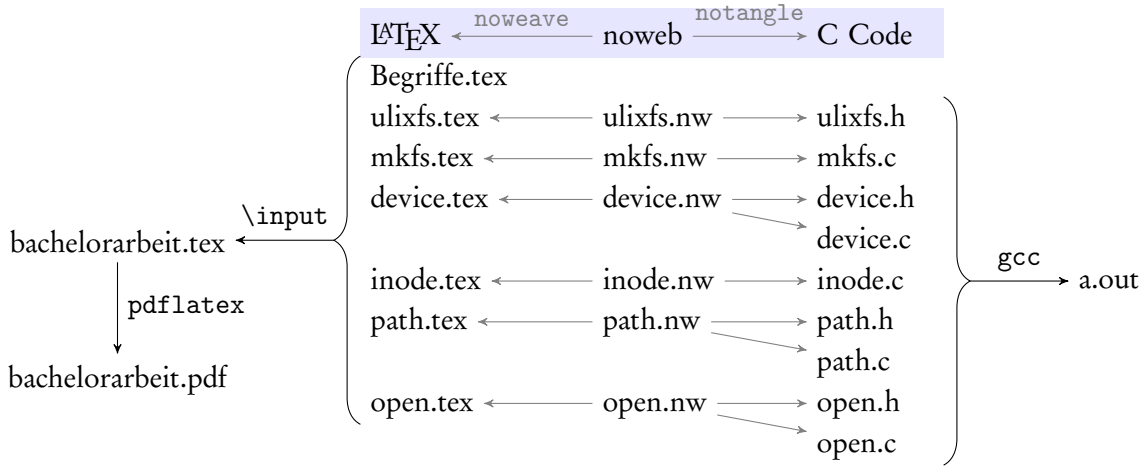

Abbildung 1.2: Aufbau eines größeren Projekts mit Literate Programming.

Für mehr Informationen bzgl. Literate Programming verweisen wir im Internet auf die Webseiten [4], die eine große Anzahl von Artikeln und Dokumentationen enthält, und auf [5], wo viele Beispielprogramme nachgeschlagen werden können.

1.2 ULIX

ULIX-i386 ist ein Unix-artiges Lehrbetriebssystem für die Intelarchitektur i386. Es wird für Lehrzwecke an der Erlanger Universität, Fakultät Informatik, entwickelt. Aktueller Betreuer ist Herr Hans-Georg Eßer, der das Projekt ULIX von Professor Felix Freiling übernommen hat. ULIX-i386 (siehe [6]) befindet sich noch in einer frühen Phase der Entwicklung.

Das besondere an ULIX ist nicht, dass es ein Lehrbetriebssystem ist, denn es gibt auch andere Lehrbetriebssysteme, wie z. B. Minix, sondern, dass ULIX mit Literate Programming geschrieben wird. Das erlaubt eine viel übersichtlichere Dokumentation des Betriebssystems, wie man auch aus dieser Ausarbeitung entnehmen kann – da sie auch mit Literate Programming geschrieben ist.

2 UlixFS

In diesem Kapitel werden wir das Dateisystem ULIXFS, das wir für das Betriebssystem ULIX entwickelt haben, vorstellen. Zugleich werden wir eine C Headerdatei namens `ulixfs.h` erstellen, die die Beschreibung und Struktur des Dateisystems beinhaltet. Sie ist die zentrale Headerdatei von ULIXFS und hat die folgende Struktur:

7 ⟨*ulixfs.h* 7⟩≡

```
#ifndef ULIXFS_H
#define ULIXFS_H

#include "sys/types.h"

⟨Dateisystem Konstanten 8⟩
⟨Superblock Definition 10b⟩
⟨Inode Definition 13⟩
⟨Verzeichnisstruktur 17b⟩

#endif
```

Die Headerdatei, die am Anfang inkludiert wird, enthält ein paar Typdefinitionen, die wir später brauchen werden. Sie ist im Anhang A.1 auf der Seite 107 beschreiben.

ULIXFS orientiert sich stark an den Unix-artigen Dateisystemen und besonders an dem Dateisystem, das vom Betriebssystem Minix verwendet wird. Siehe [14], insbesondere Kapitel 5, wo das Minix-Dateisystem beschrieben wird. In dieser Ausarbeitung werden wir ein geeigneten Stellen auf die Unterschiede zum Dateisystem von Minix eingehen.

2.1 Begriffe

Bevor wir mit der Beschreibung des Dateisystems ULIXFS anfangen, möchten wir ein paar Begriffe festlegen.

2.1.1 Block

Unter einem *Block* verstehen wir eine zusammenhängende Reihe von Bytes fester, gleicher Länge. Die Bytes innerhalb eines Blocks können durch Angabe eines Index eindeutig adressiert werden. Wir sprechen vom ersten Byte, zweiten Bytes usw. Die Indizierung der Bytes innerhalb eines Block fängt bei Null an: Das erste erste Byte hat den Index Null.

Auf der Ebene der C-Syntax ist ein Block ein Array. Für das Dateisystem ULIXFS werden wir eine einheitliche feste Blockgröße festlegen, die überall in dem Dateisystem verwendet wird.

2.1.2 Disk

Wir werden in dieser Ausarbeitung immer wieder von einer „Disk“ sprechen. Unter dem Begriff *Disk* verstehen wir allgemein ein Speichermedium, das die folgenden Eigenschaften besitzt:

1. Es kann nur blockweise gelesen und geschrieben werden.
2. Die Blöcke können durch Angabe eines Index eindeutig adressiert werden.
3. Der Index eines Blocks kann frei gewählt werden, jedoch darf er nicht kleiner Null oder größer als die maximale Blocknummer sein (wahlfreier Zugriff).

Man kann auch sagen, dass eine Disk eine indizierte Reihe von Blöcken ist.

Es ist dabei nicht wichtig, um welches konkrete Gerät es sich handelt, welche Geometrie oder physikalische Größe es hat – lediglich die genannten Eigenschaften sind relevant. Alles, worauf wir blockweise zugreifen können, das eine blockorientierte Schnittstelle hat, nennen wir eine Disk. Eine DVD, eine Hard-Disk, ein USB-Stick: Alle sind aus unserer Sicht eine Disk. Eine Tastatur hingegen ist keine Disk, da sie nicht blockweise gelesen werden kann, sondern nur byteweise.

Man könnte auch sagen, eine Disk ist einfach nur ein *Blockgerät*. Wir sprechen aber von einer Disk, da das Wort uns intuitiver scheint. Den Begriff eines Geräts werden wir später im Abschnitt 3.1 (Seite 21) einführen.

2.1.3 Datei

Aus der Sicht des Dateisystems (Systemsicht), besteht eine *Datei* aus zwei getrennten Komponenten:

- Metadaten über die Datei, wie ihre Größe, Typ, Zugriffsrechte usw. Die Struktur, die diese Metadaten enthält nennen wir, in Anlehnung an die Unix-Kultur, einen *Inode*. Wie wir sehen werden, enthält ein Inode keine Dateinamen. Siehe den Abschnitt 2.3, Seite 13.
- Inhalt der Datei.

2.2 Aufbau des Dateisystems

Das Dateisystem ULIXFS existiert als Inhalt einer Disk oder Teil (Partition) einer Disk, wenn das physikalische Speichermedium größer als das Dateisystem ist. Im Abschnitt 2.1.2 haben wir die Disk als blockorientiertes Gerät charakterisiert. Hardware-Disks haben verschiedene Einheiten, wie z. B. Sektoren, die man für den Zugriff verwenden kann. Diese Einheiten sind nicht immer für alle Geräte gleich groß. In ULIXFS abstrahieren wir von den konkreten physikalischen Geräten und von deren unterschiedlichen Sektorgrößen und definieren eine einheitliche Blockgröße. Wie definieren diese Größe in der Headerdatei `ulixfs.h` wie folgt:

8 ⟨*Dateisystem Konstanten 8*⟩≡ (7) 9▸
```
#define BLOCK_SIZE 1024
```

Es ist die Aufgabe der einzelnen Disk-Treiber, zwischen dieser Blockgröße, die vom Dateisystem benutzt wird, und den tatsächlichen Sektorgrößen zu übersetzen.

Eine Diskussion der optimalen Blockgröße findet sich in [14, S.235, 509-511]. In Minix kann man die Blockgröße während der Installation auswählen, die Standardeinstellung ist 4 KB (siehe [1]). Wir haben eine feste Größe von 1 KB gewählt, da wir eher kleine Dateien erwarten.

Eine Größe, die sich aus der Blockgröße in Bytes ableiten lässt, ist die Anzahl der Bits in einem Block. Wir werden später diese Größe brauchen und definieren sie deshalb in `ulixfs.h` als Makro:

9 ⟨*Dateisystem Konstanten* 8⟩+≡ (7) ◁8 10a▷

```
#define BITS_PER_BLOCK (BLOCK_SIZE << 3)
```

Das ist nichts anderes als die Anzahl der Bytes in einem Block multipliziert mit 8, der Anzahl der Bits in einem Byte.

Nach der Einteilung der Disk in Blöcke, ist die nächste Einteilung der Disk in funktionale Regionen oder Bereiche, die zur Verwaltung des Dateisystems dienen. Diese Regionen enthalten verschiedene Komponenten des Dateisystems, wie z. B. den Superblock und die Datenblöcke, die den Inhalt der verwalteten Dateien beinhalten. Eine Übersicht der Regionen ist in der Abbildung 2.1 dargestellt. Das Dateisystem besteht aus genau diesen Diskregionen.

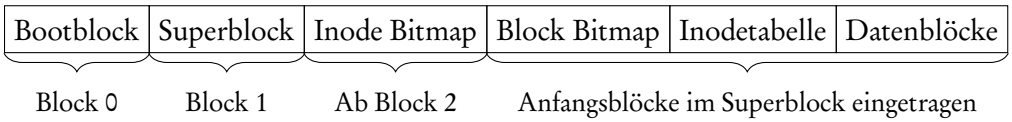

Abbildung 2.1: Aufbau des Dateisystems ULIXFS. Die angegebenen Diskregionen folgen unmittelbar nacheinander auf der Disk, beginnend mit dem ersten Byte im ersten Block.

Die Diskregionen in Abbildung 2.1 enthalten alle Informationen, die zum Lesen, zum Schreiben und für die Buchhaltung des Dateisystems notwendig sind. Diese Informationen sind allerdings geräteunabhängig: Es werden keine Informationen über die darunterliegende Hardware gespeichert. Ob es sich um einen USB-Stick oder eine DVD oder einfache Datei innerhalb eines anderen Dateisystems handelt, das kann man am Dateisystem selbst nicht ablesen.

Jede Diskregion belegt eine ganzzahlige Anzahl von Blöcken.

In den nächsten Abschnitten werden wir jede Region des Dateisystems vorstellen und entsprechende Einträge in der Headerdatei `ulixfs.h` schreiben.

2.2.1 Bootblock

Jedes Dateisystem besitzt einen Bootblock, dieser belegt im Dateisystem den ersten Block (mit Index 0). Der Bootblock dient dazu, einen so genannten Bootloader zu speichern. Das ist ein Programm, das ein Betriebssystem findet, in den Speicher lädt und ausführt. Mit diesem Programm werden wir uns nicht weiter beschäftigen.

2.2.2 Superblock

Der Superblock folgt unmittelbar nach dem Bootblock und belegt ebenfalls einen Block, auch wenn seine tatsächliche Größe kleiner als ein Block ist. Seine feste Position auf der Disk legen wir in der Headerdatei fest:

10a ⟨*Dateisystem Konstanten 8*⟩+≡ (7) ◁9 11▷
```
#define SUPER_BLOCK    1
```

Damit sagen wir, dass der Superblock sich im Block mit Index 1 befindet, also der nächste Block nach dem Bootblock, der den Index 0 hat.

Der Superblock enhält Metadaten zum Dateisystem. Er ist eine Tabelle, mit deren Hilfe wir die Anfangspositionen (Blocknummern) aller anderen Regionen finden können. Zusätzlich befinden sich in der Tabelle Größenangaben, die zur Verwaltung und Identifikation des Dateisystems notwendig sind, wie z. B. die Anzahl der Inodes und die gesamte Anzahl von Blöcken im Dateisystem.

Um diese Beschreibungstabelle eindeutig lesen zu können, deklarieren wir in der Headerdatei des Dateisystems eine C-Struktur, die der Struktur des Superblocks entspricht.

10b ⟨*Superblock Definition* 10b⟩≡ (7)
```
typedef struct superblock {
    int     magic;    // signature
    ino_t   inodes;   // how many inodes
    block_t imap;     // begin of Inode-Bitmap
    block_t bmap;     // begin of blocks bitmap
    block_t itable;   // begin of inode table
    block_t data;     // first data block
    block_t nblocks;  // total blocks
} superblock;
```

Die Reihenfolge und Größe jedes Eintrages in dieser Struktur entsprechen deren Reihenfolge und Struktur auf der Disk. Laden wir also aus einer Disk den zweiten Block (den Superblock) in den Speicher, so können wir die Anzahl der Inodes im Dateisystems aus dem zweiten Eintrag auslesen. Ein hypothetischer Code könnte also wie folgt aussehen:

```
char block[BLOCK_SIZE] = { 0 };
device_read(block, BLOCK_SIZE);
superblock *super = (superblock *) block;
ino_t inodes = super->inodes;
```

Schauen wir uns als nächstes an, was die einzelnen Einträge bedeuten.

magic Das ist die Signatur des Dateisystems (magic number), eine Konstante analog zur Signatur eines Datenformats. Mit ihrer Hilfe können wir das ULIXFS Dateisystem von anderen Dateisystemen unterscheiden und auch mehrere ULIXFS-Versionen voneinander. Wir definieren sie weiter unten als eine 4-Byte Zahl, die wir aus den ASCII-Codes von „ULX0“ bilden (von „ULIXFS Version 0“).

inodes Hier steht die Anzahl der Inodes im Dateisystem (siehe Abschnitt 2.3).

imap Dieser Eintrag enthält den Anfangsblock der Inode-Bitmap (siehe Abschnitt 2.2.3). Der Name kommt von „Inode-Bitmap“.

bmap Dieses Feld enthält die erste Blocknummer der Block-Bitmap (siehe Abschnitt 2.2.3). Die Bezeichnung kommt von „Block-Bitmap“.

itable Hier findet man die erste Blocknummer der Inodetabelle (siehe Abschnitt 2.2.4).

data Dieses Feld enthält die erste Blocknummer der Datenregion (siehe Abschnitt 2.2.5).

nblocks Hieraus kann man die gesamte Anzahl der Blöcke entnehmen, die das Dateisystem ausmachen. Multipliziert man diese Zahl mit `BLOCK_SIZE`, ergibt sich die Anzahl von Bytes des ganzen Dateisystems, inklusive alle Diskregionen. Subtrahiert man von `nblocks` den Eintrag im `data`-Feld, so erhält man dir Größe der Datenregion in Blöcken.

Mit der Definition unseres Superblocks weichen wir von Minix ab. In Minix enthält der Superblock hauptsächlich eine Liste von Größen: Wie groß ist die Inode-Bitmap, wie groß ist die Inode-Tabelle usw. Wir betrachten den Superblock analog zum Inhaltsverzeichnis eines Buches: Wo fängt die Inode-Bitmap an, wo fängt die Datenregion an usw. Wir finden, die Anfangsblöcke sind praktischer und intuitiver als deren Größe.

Die „magische Zahl“ und Signatur des Dateisystems definieren wir wie folgt:

11 ⟨*Dateisystem Konstanten 8*⟩+≡ (7) ◁10a 14a▷

```
#define MAGIC_NO 0x554C5830
//magic number:      U L X 0
```

Die Abbildung 2.2 zeigt einen Superblock mit konkreten Zahlen. Im Abschnitt 4.2.1 ab der Seite 45 wird gezeigt, wie man die Einträge des Superblocks festlegen kann.

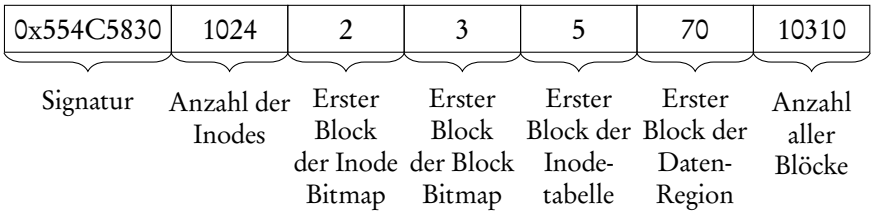

Abbildung 2.2: Beispiel für einen Superblock des Dateisystems ULIXFS.

2.2.3 Inode- und Block-Bitmap

Für die Verwaltung der freien Inode- und Blocknummern verwendet ULIXFS zwei Tabellen, wo jeder Eintrag den Zustand einer bestimmten Inode- bzw. Blocknummer speichert. Der Zustand „belegt“ wird durch eine 1 vermerkt, „unbelegt“ (frei) mit einer `0`. Da wir lediglich diese zwei Zustände speichern, eignet sich als Tabelleneintrag ein einziges Bit.

Eine solche Tabelle, die aus einzelnen Bits besteht, nennt man eine *Bitmap* oder Bitvektor. Die Bits einer Bitmap, die aus mit n Bits besteht, werden durchgehend „von links nach rechts“ durchnummeriert, wie die Einträge eines Arrays mit n Elementen. Die Nummerierung der einzelnen Bits fängt bei Null an und mit dem linkesten Bit im ersten Byte. Innerhalb eines Bytes, hat das erste Bit

also die Wertigkeit 2^7 und das letzte Bit die Wertigkeit 2^0. Diese Anordnung der Bits, die einem Array ähnelt, finden wir intuitiver (eine Bitmap heißt auch „Bit-Array“).

Die zwei Bitmaps belegen den Raum nach dem Superblock (siehe Abbildung 2.1, Seite 9).

2.2.3.1 Inode-Bitmap

Die Inode-Bitmap beginnt im nächsten Block nach dem Superblock – das heißt im Block mit Index 2, denn der Bootblock hat den Index `0` und der Superblock befindet sich am Index 1 (siehe Abbildung 2.1, Seite 9). Sie speichert den Zustand der einzelnen Inodenummern und besteht aus $n + 1$ Bits, wobei n die Anzahl der Inodes im Dateisystem ist, die auch im Superblock vermerkt ist (Feld `inodes`). Das eine zusätzliche Bit ist das Bit mit Index `0`, das dem Inode `0` entspricht (mehr zum Thema Inodes im Abschnitt 2.3, Seite 13). Jedem Bit wird eine Indexnummer in der Inode-Bitmap zugewiesen. Diese Indexnummer ist gleich der Inodenummer, die dort abgebildet wird. Möchte man also prüfen, ob der Inode mit Nummmer 15 frei ist, so schaut man sich das Bit am Index 15 an.

1	1	0	0	0	0	0	0	...

Abbildung 2.3: Beispiel für den Anfang einer Bitmap des Dateisystems ULIXFS. Dargestellt wird das erste Byte.

Betrachten wir als Beispiel die Abbildung 2.3. Dargestellt wird das erste Byte einer Bitmap. Interpretieren wir das als die Inode-Bitmap, so kann man aus der Abbildung entnehmen, dass die Inodes mit Nummern `0` und 1 belegt sind, da die ganz links liegenden zwei Bits auf 1 gesetzt sind – diese Bits haben nämlich die Indizes `0` und 1. Die folgenden sechs Inodenummern sind frei, da die entsprechenden Bits auf `0` gesetzt sind.

Die Inodes mit Nummern `0` und 1 sind in der Inode-Bitmap immer belegt: Inode `0`, weil er nie verwendet werden soll, und Inode 1 (*root*-Inode), weil jedes Dateisystem mindestens ein Root-Verzeichniss hat. Dies hat zur Folge, dass auf einem „frischen“ Dateisystem, wo nur die Inodes `0` und 1 belegt sind, das erste Byte der Inode-Bitmap den hexadezimalen Wert `0xC0` hat.

2.2.3.2 Block-Bitmap

Die Block-Bitmap funktioniert analog zur Inode-Bitmap mit dem Unterschied, dass hier die Zustände der Datenblöcke gespeichert sind (siehe auch die Abbildung 2.1, Seite 9). Das erste (linkeste) Bit steht also für den ersten Datenblock, der im Superblock als Start der Datenregion angegeben wird. Hat dieser Startblock die Nummer (Index) n, so steht das Bit am Index i für den Block mit Nummer $n + i$. Die Blöcke vor der Datenregion werden in dieser Bitmap nicht abgebildet.

Nehmen wir wieder die Abbildung 2.3 als Beispiel und interpretieren sie diesmal als eine Block-Bitmap, so entnehmen wir daraus, dass die ersten zwei Datenblöcke belegt sind. Hat der erste Datenblock die Nummer `70`, so stehen die beiden linken Bits für die Blöcke `70` und `71`.

2.2.4 Inodetabelle

Die Inodetabelle ist die Tabelle aller möglichen Inodes im Dateisystem, einschließlich des Inodes mit Nummer 0, der nicht verwendet wird (mehr zum Thema Inodes im Abschnitt 2.3). Es gibt also für n Inodes im Dateisystem $n+1$ Einträge.

Auf der C-Code-Ebene ist die Inodetabelle ein Array von `inode`-Strukturen. Der Inode mit Nummer x befindet sich am Index x in diesem Array. Möchte man also den Inode mit Nummer 5 einlesen, so liest man aus diesem Array den Eintrag am Index 5.

2.2.5 Datenregion

Die Datenregion enthält die eigentlichen Daten, den Inhalt der Dateien und der Verzeichnisse. Sie hat keine bestimmte Struktur, außer, dass sie blockweise indiziert ist. Der erste Datenblock ist in dem Superblock eingetragen (Feld `data`). Siehe dazu die Struktur `superblock`, die wir oben im Abschnitt 2.2.2 (Seite 10) definiert haben.

2.3 Inodes

Der Inode ist eine Datenstruktur fester Größe, die aus zwei Teilen besteht:

1. Metadaten über eine bestimmte Datei, wie z. B. Ihre Größe, Zugriffsrechte usw.
2. Eine Liste von Blocknummern, wo die Daten (Inhalt) der Datei sich befinden.

Das Dateisystem ULIXFS enthält nach seiner Generierung eine feste Anzahl von Inodes, die sich alle in der Inodetabelle befinden. Diese Anzahl ist im Superblock vermerkt und sie bestimmt die maximale Anzahl von Dateien im Dateisystem.

Bei der Wahl der Felder der Inode-Struktur haben wir uns an den Unix-Konzepten orientiert und besonders am Minix-Inode (siehe dazu [14, S. 555ff]). Die Deklaration dieser Datenstruktur tragen wir in die Headerdatei `ulixfs.h` ein.

13 ⟨*Inode Definition* 13⟩≡ (7)

```
typedef struct inode {
//metadata
    off_t   fsize;
    mode_t  mode;
    nlink_t nlinks;
    uid_t   uid;
    gid_t   gid;
    time_t  atime; // last access time
    time_t  ctime; // inode change time
    time_t  mtime; // data modification time
//block numbers of content
    block_t block[INODE_BLOCKS];
} inode;
```

Die Datei `sys/types.h` enthält alle Typdefinitionen, die wir für die Inode-Felder verwendet haben (siehe Anhang A.1, Seite 107). Die Anzahl der Blöcke am Ende müssen wir noch definieren:

14a ⟨*Dateisystem Konstanten 8*⟩+≡ (7) ◁11 14b▷
```
#define INODE_BLOCKS 10
```

Das ist die maximale Anzahl von Blocknummern, die in einem Inode gespeichert werden können. Daraus können wir eine wichtige Größe ableiten: die maximale Dateigröße in Bytes. Da wir maximal 10 Blöcke speichern können, ist die maximale Dateigröße $10 \cdot 1024 = 10240$ Bytes (ein Block ist 1024 Bytes groß). Wir definieren diese Größe in dem `ulixfs.h` Header, um spärere Berechnungen zu vermeiden:

14b ⟨*Dateisystem Konstanten 8*⟩+≡ (7) ◁14a 14c▷
```
#define MAX_FILE_SIZE (INODE_BLOCKS * BLOCK_SIZE)
```

Im Unterschied zum Minix-Inode, verwendet ULIXFS keine indirekte Blöcke. Wir haben trotz dieser Einschränkung die Anzahl der Blocknummer im Inode so gewählt, dass eine zukünftige Implementierung von ULIXFS das Minix-Modell problemlos übernehmen kann: die ersten sieben Blocknummer könnten als direkte und die restlichen drei als indirekte Blocknummern verwendet werden. Dazu müsste man an der obigen Definition des Inodes nichts mehr ändern. Siehe [14, S. 555] für mehr Informationen zum Thema Inode in Minix.

Die Datentypen der einzelnen Struktur-Felder haben wir so gewählt, dass eine Inode-Struktur auf einem 32-Bit System insgesamt 64 Byte groß ist. Die Idee dahinter ist, dass in einem Diskblock der Größe `BLOCK_SIZE`, die ebenfalls eine Zweierpotenz ist, eine ganzzahlige Anzahl von Inodes passen soll. Somit müssen wir nicht Inodes über die Grenzen eines Blocks aufteilen, sondern können einen Block als ein Array von `inode`-Strukturen betrachten und die einzelnen Inodes leicht per Index adressieren. Definieren wir die Größe eines Inodes und die Anzahl von Inodes in einem Block als Makros, damit wir sie später kürzer schreiben können:

14c ⟨*Dateisystem Konstanten 8*⟩+≡ (7) ◁14b 15▷
```
#define INODE_SIZE       (sizeof(inode))
#define INODES_PER_BLOCK (BLOCK_SIZE/INODE_SIZE)
```

Datenlänge in Bytes
Modus (Dateityp, Rechte)
Anzahl Links
Benutzer ID
Gruppe ID
Zeit des letzten Zugriffs
Zeit der letzten Inode-Änderung
Zeit der letzten Daten-Änderung
Liste der Datenblöcke (10 Blocknummern)

Abbildung 2.4: Inode des ULIXFS Dateisystems. Ein Inode ist 64 Bytes groß.

Wir besprechen nun die Felder eines Inodes. Siehe dazu auch die Abbildung 2.4.

fsize Das ist die Dateilänge in Bytes, die wir schon hatten. Die Länge bezieht sich auf den Inhalt der Datei, der sich in der Datenregion befindet. Die Bezeichnung kommt von „file size“.

mode Hier speichern wir den Dateityp und die Zugriffsrechte der Datei. Für die Gestaltung und Wahl dieser Zugriffsrechte benutzen wir das alte Unix-Modell. Siehe weiter unten den Abschnitt 2.4, wo wir dieses Feld näher betrachten.

nlinks Dieses Feld speichert die Anzahl der sogenannten „Hard Links“ zu einer Datei. Es ist mindestens gleich 1, wenn die Datei in mindestens einem Verzeichnis eingetragen ist. Für jeden weiteren Eintrag in einem Verzeichnis, wird dieses Feld inkrementiert.

uid Hier findet man die Benutzer-ID desjenigen Benutzers, der diese Datei besitzt. Die Bezeichnung kommt von „user id“.

gid Hier steht die ID derjenigen Gruppe von Benutzern, die Zugriffsrechte für die Datei hat. Die Bezeichnung kommt von „group id“.

atime Hier wird immer die Zeit eingetragen, zu der zuletzt auf diese Datei zugegriffen wurde. Die Bezeichnung kommt von „last access time“.

ctime Hier wird die letzte Zeit eingetragen, zu der der Inode selbst verändert wurde. Die Bezeichnung kommt von „inode change time“.

mtime Die Zeit, zu der der Inhalt der Datei durch Schreibzugriff verändert wurde. Die Bezeichnung kommt von „modification time“.

block Das ist vielleicht das wichtigste Feld: Ein Array von Blocknummern, wo die Daten zu finden sind. Die Blocknummern werden in der Reihenfolge eingetragen, die der Reihenfolge der Daten entspricht. Wenn man also die Datei sequentiell lesen will, muss man alle Blöcke der Reihe nach lesen, beginnend mit dem ersten. Dabei muss man die Dateilänge beachten und die richtige Anzahl von Blöcken lesen.

Die `uid` und `gid` Felder enthalten sogenannte Identifikationsnummern (ID) des Datei-Besitzers und der Gruppe, die Zugriffsrechte auf die Datei hat. Wir werden in dieser Ausarbeitung nicht auf das Thema Benutzer und Gruppen eingehen, wir werden aber in der `ulixfs.h` Datei bestimmte IDs definieren, die immer vorhanden ist: die Benutzer-ID und Gruppen-ID des root-Benutzers bzw. der root-Gruppe.

15 ⟨*Dateisystem Konstanten* 8⟩+≡ (7) ◁14c 17a▷

```
#define ROOT_UID    0
#define ROOT_GID    0
```

Ein Inode also, der dem root-Benutzer gehört wird immer das Feld `uid` auf Null gesetzt haben. Ein Inode, auf den die root-Gruppe Zugriffsrechte hat, wird immer das `gid` Feld auf Null gesetzt haben.

2.4 Datei Modus (`mode_t`)

Das Feld `mode` in der `inode` Struktur ist ein 16 Bit großes Bitmuster. Dieses Muster enthält in den einzelnen Bits oder Gruppen von Bits verschiedene Informationen wie Zugriffsrechte und Dateityp, wie in der Abbildung 2.5 dargestellt.

Wir besprechen im Folgenden die Bedeutungen der einzelnen Bits dieses Feldes. Im Anhang A.2 werden wir einige POSIX-Makros und Konstanten definieren, mit deren Hilfe verschiedene Informationen aus dem Feld `mode` extrahiert werden können.

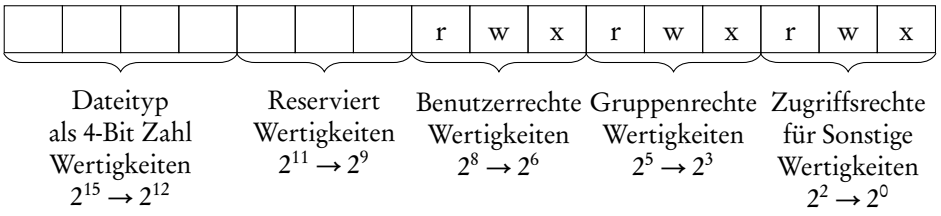

Abbildung 2.5: Einteilung der 16 Bits in dem `mode`-Feld eines Inodes. Die Buchstaben ‘r’, ‘w’ uns ‘x’ stehen jeweils für „read“ (Leserechte), „write“ (Schreibrechte) und „execute“ (Ausführungsrechte).

2.4.1 Dateityp

Der Dateityp wird in den vier höchstwertigen Bits des Typs `mode_t` als eine 4-Bit ganze Zahl gespeichert. Mit diesen vier Bits können wir $2^4 = 16$ mögliche Dateitypen darstellen. Eine Festlegung der genauen Nummer für jeden Dateityp erfolgt in der Datei `sys/stat.h`, die wir im Anhang A.2 (Seite 108) definieren. Dort werden auch Makros eingeführt, mit deren Hilfe die einzelnen Informationen aus dem `mode`-Feld extrahiert werden können.

2.4.2 Zugriffsrechte

Die Zugriffsrechte für eine Datei sind in den neun niedrigstwertigen Bits des Typs `mode_t` vermerkt, wie in der Abbildung 2.5 (Seite 16) dargestellt. Wir unterteilen sie nach dem Unix-Modell in drei Arten:

1. Leserechte: ob die Datei gelesen werden darf.
2. Schreibrechte: ob die Datei beschrieben werden darf.
3. Ausführungsrechte: ob die Datei ausgeführt oder ein Verzeichnis durchsucht werden darf.

Diese Zugriffsrechte gelten immer für eine Kategorie von Benutzern. Wir unterscheiden zwischen drei Kategorien:

1. Eigentümer der Datei. Seine ID ist im Feld `uid` eingetragen.
2. Gruppe von Benutzern, die Zugriffsrechte auf die Datei haben. Die ID dieser Gruppe ist im Feld `gid` eingetragen.
3. Alle anderen Benutzer.

2.5 Verzeichnisse

Ein Verzeichnis ist nichts anderes als eine Datei, deren Inhalt eine bestimmte Struktur hat. Diese Struktur besteht aus einer Liste von Verzeichniseinträgen. In ULIXFS, hat ein Verzeichniseintrag eine feste Größe und besteht aus zwei Elementen: aus einer Inodenummer und aus einem Dateinamen, der der Inodenummer in diesem Verzeichnis gegeben wird. Siehe dazu auch die Abbildung 2.6 auf Seite 18. Die Größe des zweiten Elements (Dateiname) aus einem Verzeichniseintrag bestimmt die maximale Länge eines Dateinamens. Diese Größe legen wir in ULIXFS auf 60 Bytes fest. Die größe der Inodenummer legen wir auf vier Bytes fest (auf einem 32-Bit System).

Diese Werte wurden so gewählt, damit ein Verzeichniseintrag eine Größe der Form 2^n (also eine Zweierpotenz) hat, in diesem Fall $4 + 60 = 64$ Bytes. Wir wählen (wie bei den Inodes) eine Zweierpotenz als Größe, weil diese eine optimale Unterbringung in Blöcken erlaubt.

Wir definieren die Größe des Dateinamens in einem Verzeichnis wie folgt:

17a ⟨*Dateisystem Konstanten 8*⟩+≡ (7) ◁15 17c▷
```
#define FNAME_SIZE 60
```

Da wir einen Namen wie einen C-String verwenden, beträgt die tatsächliche maximale Namenslänge ein Byte weniger, da ein Byte für das abschließende Null-Byte reserviert ist. Wir hätten dieses Byte für einen zusätzlichen Buchstaben frei lassen und eine Namenslänge von 60 Byte verwenden können. Das hätte für die Anwender dieser Datenstruktur bedeutet, dass sie entsprechend größere Arrays verwenden. Wir haben uns entschieden, die Null-Terminierung selber zu übernehmen.

Nun kommen wir zur Definition eines Verzeichniseintrags:

17b ⟨*Verzeichnisstruktur 17b*⟩≡ (7)
```
typedef struct dir_entry {
    ino_t ino;
    char  name[FNAME_SIZE];
} dir_entry;
```

Der Datentyp `ino_t` (von „inode type“) wird in der Headerdatei `sys/types.h` definiert, die wir im Anhang A.1 beschreiben. Die Größe eines Verzeichniseintrags und die Anzahl der Verzeichniseinträge in einem Block machen wir ebenfalls in `ulixfs.h` bekannt, allerdings in einer dynamischen Form:

17c ⟨*Dateisystem Konstanten 8*⟩+≡ (7) ◁17a 18a▷
```
#define DIR_ENTRY_SIZE (sizeof(dir_entry))
#define DIR_ENTRIES_PER_BLOCK (BLOCK_SIZE/DIR_ENTRY_SIZE)
```

Jedes Verzeichnis hat mindestens zwei Einträge. Die dort eingetragenen Dateinamen sind zuerst „.“ (ein Punkt) und dann „..“ (zwei Punkte). Der erste Eintrag („.“) hat dieselbe Inodenummer wie das Verzeichnis selbst, ist also ein Verweis (Hard Link) des Verzeichnisses auf sich selbst. Der zweite Eintrag („..“) hat die Inodenummer des Vaterverzeichnisses, desjenigen Verzeichnisses, das einen Eintrag für dieses Verzeichniss hat. Siehe das Beispiel in Abbildung 2.6.

			1	‘.’											
			1	‘.’	‘.’										
			2	‘b’	‘i’	‘n’									
			5	‘u’	‘s’	‘r’									
			12	‘h’	‘o’	‘m’	‘e’								
			15	‘d’	‘a’	‘t’	‘a’	‘.’	‘t’	‘x’	‘t’				

Inode-Nummern Dateinamen

Abbildung 2.6: Inhalt einer Verzeichnisdatei, hier des Root-Verzeichnisses. Die Zeilen sind als eine einzige Reihe von Bytes zu lesen, von oben nach unten und von links nach rechts. Die nicht belegten Einträge enthalten die Zahl Null. Buchstaben sind statt deren ASCII-Codes angegeben. Jedes Kästchen entspricht einem Byte aus der Verzeichnisdatei. Die Länge des Dateinamens wurde aus Platzgründen auf zwölf Bytes reduziert.

2.6 Verzeichnisbaum

Wie in Unix auch, sind die Dateien in ULIXFS in einer Baumstruktur organisiert.[1] Der Baum hat eine einzige Wurzel: das *Root*-Verzeichnis. Das ist der Inode mit Nummer 1, der erste gültige Inode. Wir legen seine Nummer in der Headerdatei `ulixfs.h` fest:

18a ⟨*Dateisystem Konstanten 8*⟩+≡ (7) ◁17c 18b▷
```
#define ROOT_INO    1
```

Der Inode mit Nummer `0` ist ungültig.

In diesem Root-Verzeichnis mit Inodenummer 1 gibt es Einträge für Kindverzeichnisse und Dateien. Die Kindverzeichnisse enthalten wieder Einträge für weitere Kindverzeichnisse und Dateien. Es entsteht somit ein Baum von Verzeichnissen und Dateien. Jeder Pfad kann von der Wurzel aus zu einem Blatt und umgekehrt durchlaufen werden.

Der Eintrag „..“ zeigt im Root-Verzeichnis wie der Eintrag „.“ auf das Root-Verzeichnis selbst. Mit anderen Worten, das Vaterverzeichnis von root ist root selbst.

Eine Datei im Verzeichnisbaum ist durch einen oder mehreren Pfaden gekennzeichnet (falls es Hard Links auf die Datei gibt). Der Pfad besteht aus einer Konkatenation von Knotennamen, die von der Wurzel aus durchlaufen werden müssen, um den gekennzeichneten Knoten zu erreichen. Der Pfad fängt mit dem Root-Verzeichnis an und endet mit dem Datei- oder Verzeichnisknoten.

In einer Pfadangabe werden die einzelnen Knotennamen mit einem Schrägstrich getrennt. Dieses Trennzeichen machen wir öffentlich in `ulixfs.h` wie folgt:

18b ⟨*Dateisystem Konstanten 8*⟩+≡ (7) ◁18a
```
#define PATH_SEP ('/')
```

[1] Die Verzeichnishierarchie ist kein Baum im Sinne der Graphentheorie, da nicht kreisfrei.

Das Root-Verzeichnis hat einen leeren Namen, da es in keinem anderen Verzeichnis mit einem Namen eingetragen ist.

Jeder Pfad kann zu einer Inodenummer aufgelöst werden. Siehe dazu den Abschnitt 5.5.2, Seite 93.

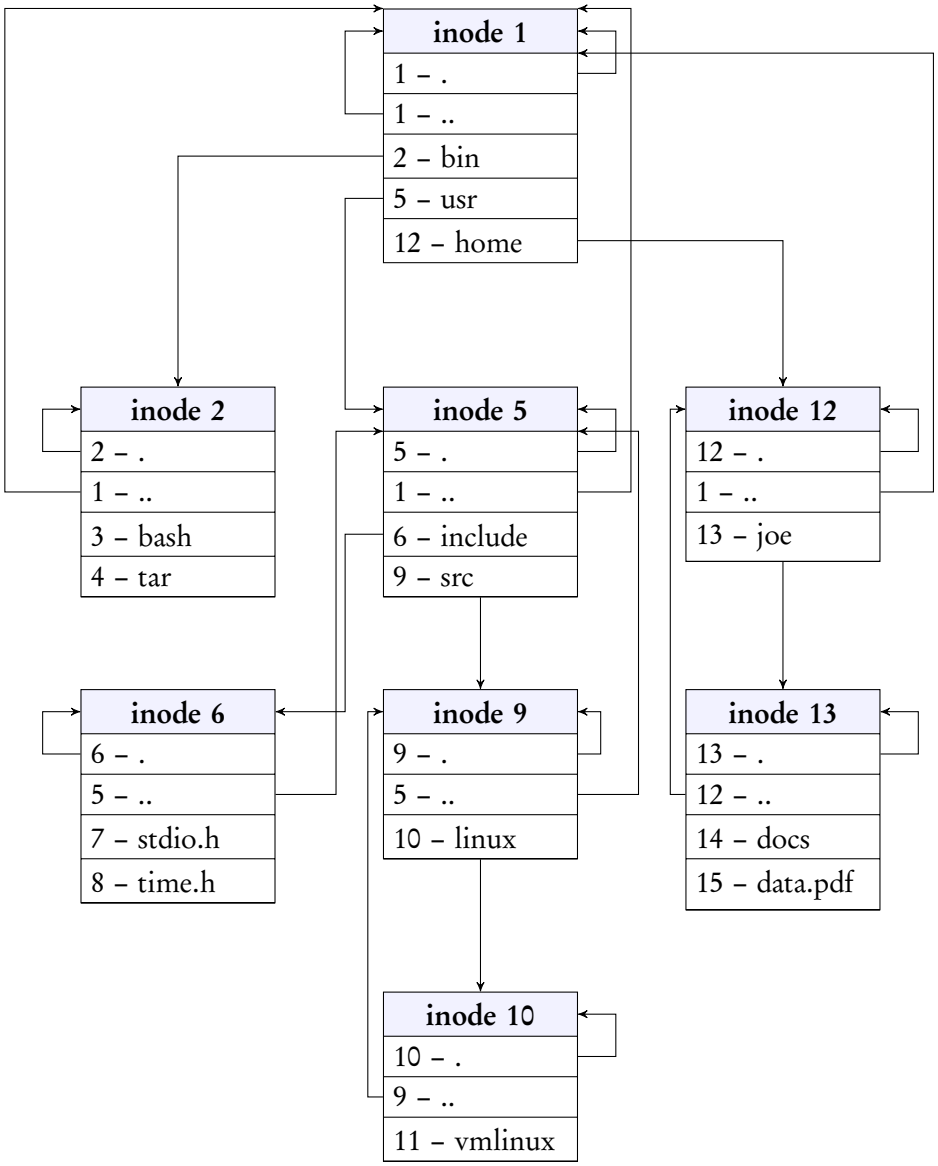

Abbildung 2.7: Verzeichnisbaum als Netz von Verweisen in der Datenregion. In jedem Knoten wird unter der Inodenummer der Inhalt derjenigen Verzeichnisdatei gezeigt, die zum Inode gehört. Die Inode-Felder werden nicht dargestellt.

Betrachtet man in der Datenregion den Inhalt der Verzeichnisdateien, die den Verzeichnisbaum ausmachen, so erhält man das Netz aus der Abbildung 2.7. Dort kann man gut sehen, dass die Dateinamen lediglich als Inhalt einer Verzeichnisdatei vorhanden sind, nämlich als Eintrag im Vaterverzeichnis.

Wie würden wir in der Abbildung 2.7 nach der Inodenummer der Datei mit dem Pfad `/home/joe/`

`data.pdf` suchen? Wir zerlegen den Pfad in Knotennamen und suchen in jedem Knoten nach dem nächsten Namen, bis wir zum letzten Knoten angekommen sind. Die Schritte sind wie folgt:

1. Wir fangen mit der Verzeichnisdatei des Root-Verzeichnisses an, bzw. mit der Inodenummer 1 (root).
2. In der Inodetabelle (die in der Abbildung nicht gezeigt ist) lesen wir den Inode 1 und lesen anhand der darin gespeicherten Blockliste Block für Block den Inhalt des Root-Verzeichnisses.
3. In jedem Block suchen wir nach einem Eintrag mit dem Namen `home` (siehe Abbildung 2.7).
4. Wir finden die Zuordnung von `home` zum Inode 12.
5. Wir lesen aus der Inodetabelle den Inode 12 und anhand der dort gespeicherten Blocknummern suchen wir in jedem Verzeichniseintrag in der Datenregion nach dem Namen `joe`.
6. Wir finden die Zuordnung von `joe` zum Inode 13.
7. Wir lesen den Inode 13 aus der Inodetabelle. In den dort eingetragenen Datenblöcken suchen wir nach dem Verzeichniseintrag mit dem Namen `data.pdf`.
8. Wir finden die Zuordnung von `data.pdf` zum Inode 15 und sind dabei am Ende des Pfades angekommen.

Nun können wir den Inhalt der Datei anhand der Blockliste im Inode 15 lesen oder schreiben.

3 Gerätemodell

3.1 Geräte und Einheiten

Wir haben im Abschnitt 2.1.2 (Seite 8) eine abstrakte „Disk“ vorgestellt, die aus einer Reihe von Blöcken von Bytes besteht.

Nun werden wir von dem Disk-Konzept weiter abstrahieren und von „Geräten“ sprechen. Den Unterschied zu einer Disk ist, dass eine Disk aus einer statischen Reihe von Bytes besteht – sie hat eine feste, unveränderliche Größe. Ein Gerät hingegen kann nicht nur aus einer festen Anzahl von Bytes oder Blöcken bestehen, sondern auch aus einem Fluss von Bytes, wie z. B. ein Netzwerkgerät, das in sich keine Bytes speichert, sondern nur Bytes überträgt. Wir können immer noch von einem Gerät einen Block von Bytes verlangen. Diese Bytes müssen aber nicht bereits im Gerät vorliegen, sondern es ist auch möglich, dass das Gerät diese Bytes zunächst erzeugen muss.

Jeder Gerätetyp kann in einem Rechner mehrfach auftauchen, wir sprechen dann von *Einheiten*: ein Rechnersystem kann mehrere Festplatten haben mit jeweils mehreren Partitionen, mehrere optische Laufwerke, mehrere USB-Anschlüsse, mehrere Netzwerkanschlüsse usw. Wir sprechen jedoch jeweils vom Festplattengerät, optischen Gerät, USB Gerät, Netzwerkgerät usw. Siehe dazu die Abbildung 3.1 auf Seite 21.

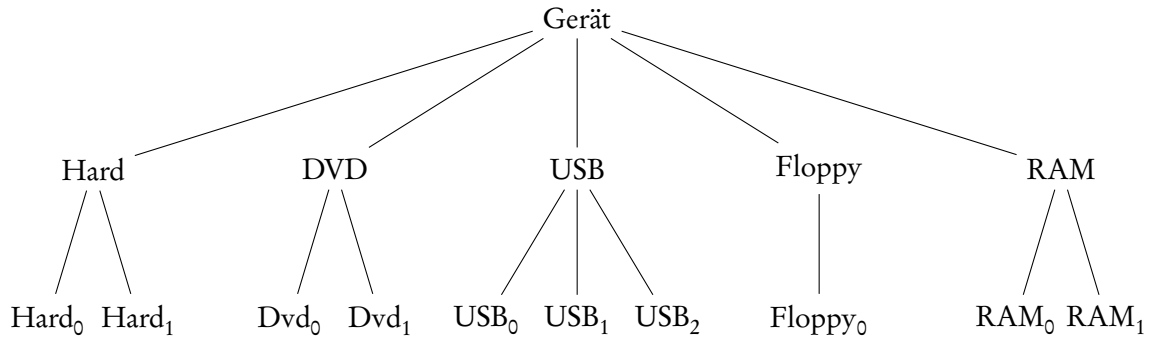

Abbildung 3.1: Geräte und Einheiten. Jedes Gerät kann aus mehreren Einheiten des selben Typs bestehen.

3.1.1 Major- und Minor-Nummern (`dev_t`)

Um die Gerätetypen voneinander unterscheiden zu können, wird jedem Gerätetyp eine Nummer gegeben. Diese Nummer nennen wir in Anlehnung an Unix eine „Major-Nummer“ (major number).

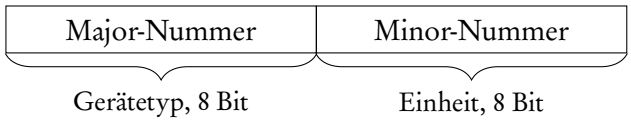

Abbildung 3.2: Der Typ `dev_t` speichert die Gerätenummer in zwei Bytes.

Innerhalb jedes Gerätetyps unterscheiden wir zwischen verschiedenen Einheiten anhand einer anderen Nummer: die Minor-Nummer (minor number).

Zur Angabe von *Gerätenummern* werden wir immer den Typ `dev_t` verwenden, der in der Datei `sys/types.h` (Anhang A.1) deklariert und in der Abbildung 3.2 dargestellt ist.

3.2 Treiberschnittstelle

Im Kapitel 2 haben wir das Dateisystem ULIXFS vorgestellt. Das war die erste Komponente eines Dateisystems: die Spezifikation. Der zweite Teil ist die Realisierung (Implementierung) der erstellten Spezifikation. Es stellt sich nun die Frage, wie wir weiter vorgehen. Gehen wir einmal davon aus, dass die „Disk“, die das Dateisystem enthält, sich im Speicher befindet, in einer sogenannten RAM-Disk. Die „Disk“ wäre ein großes Array, das wir entsprechend dem Aufbau des Dateisystems aufteilen (siehe Abschnitt 2.2). Dazu würden wir eine Menge von C-Funktionen schreiben, die mit diesem großen Array arbeiten: Inodes lesen, Blöcke schreiben usw. Das wäre unsere Implementierung des Dateisystems.

Stellen wir uns aber dann vor, dass ULIX einen Disktreiber bekommt, und dass das Dateisystem auf einer Hard-Disk gespeichert wird. Wir müssten in diesem Fall alle unsere Funktionen neu schreiben, um sie zu befähigen, von der Hard-Disk zu lesen, nicht nur aus dem Speicher. Und dann kommt noch ein DVD-Treiber dazu und das System liegt auf einer DVD, nicht mehr auf Hard-Disk oder im Speicher. Wir müssten ein drittesmal alle Funktionen neu schreiben, damit sie vom DVD lesen. Diese Architektur ist in der Abbildung 3.3 dargestellt.

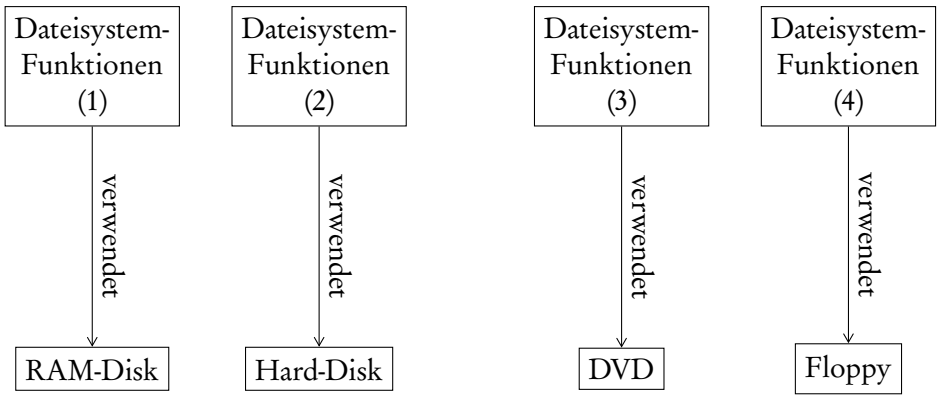

Abbildung 3.3: Treiberarchitektur. Für jedes Gerät gibt es eine getrennte Implementierung der Dateisystem-Funktionen. Für 4 Geräte gibt es 4 Implementierungen.

Wie man schnell erahnt, ist diese Architektur langfristig mehr als ungünstig. Wir müssen daher ein anderes Konzept entwickeln. Das Problem beruht auf der starken Verbindung zwischen den Funk-

tionen, die dem Dateisystem als solches spezifisch sind, und dem genauen Speichermedium, worauf das Dateisystem liegt. Trennen wir diese, so können wir die Funktionen des Dateisystems getrennt von diskspezifischen Funktionen schreiben und sie nur dann ändern, wenn wir das Dateisystem selbst ändern, nicht aber, wenn wir das Speichermedium wechseln.

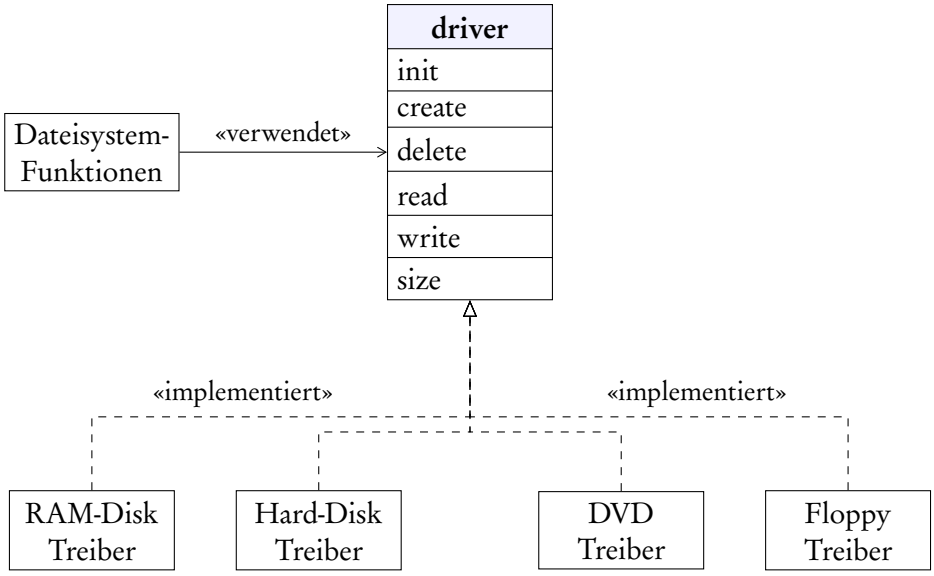

Abbildung 3.4: Treiberarchitektur. Das Dateisystem verwendet allgemeine (abstrahierte) Treiber-Funktionen, die von allen konkreten Treibern implementiert werden.

Dies können wir dadurch erreichen, dass wir ein Konzept aus der objektorientierten Programmierung verwenden, auch wenn es in der Programmiersprache C keine syntaktische Unterstützung hat: *Vererbung*. Dieses Konzept ermöglicht uns, einem wichtigen Prinzip der Softwareentwicklung zu folgen: „program to an interface, not an implementation“ (siehe [8, S. 17-18]). Die Grundidee wird in der Abbildung 3.4 dargestellt. Dort gibt es nur eine einzige Implementierung der Dateisystem-Funktionen. Diese Funktionen verwenden eine allgemeine Schnittstelle, die weiter entscheidet, welches konkretes Gerät angesprochen wird. Diejenigen Funktionen, die ein konkretes Gerät steuern und der Schnittstelle entsprechen, machen einen sogenannten *Treiber* aus.

Diese Schnittstelle kann in C dadurch realisiert werden, dass wir zunächst eine Struktur schreiben, die aus einer Menge von Funktionszeigern besteht. Die Struktur entspricht in den objektorientierten Sprachen einem Interface. Die darin enthaltenen Funktionszeiger entsprechen der einzelnen Methoden (Funktionsprototypen), die das Interface zur Verfügung stellt. Diese Schnittstelle werden wir in einer Headerdadei namens `device.h` schreiben, die zusätzlich alle relevanten Informationen über Geräte (devices) enthält. Die Headerdadei hat den folgenden Aufbau:

23 ⟨*device.h* 23⟩≡

```
#ifndef DEVICE_H
#define DEVICE_H

#include "sys/types.h"

⟨Geräte Konstanten 24b⟩
⟨Major-Nummern 26b⟩
⟨Treiberstruktur 24a⟩
⟨Makros 27b⟩
```

⟨*Generische Treiberschnittstelle* 27a⟩

```
#endif
```

Nun kommen wir zu den Operationen, die von einem Gerätetreiber zur Verfügung gestellt werden müssen, um die Schnittstelle zu erfüllen.

24a ⟨*Treiberstruktur* 24a⟩≡ (23)

```
typedef struct driver {
    int     (*init)   (void);
    dev_t   (*create) (size_t nbytes);
    dev_t   (*delete) (dev_t device);
    ssize_t (*read)   (dev_t device, off_t start, char buffer[], size_t nbytes);
    ssize_t (*write)  (dev_t device, off_t start, char buffer[], size_t nbytes);
    size_t  (*size)   (dev_t minor);
} driver;
```

Ein *Gerätetreiber* besteht aus einer Menge von Funktionen, deren Signaturen den einzelnen Elementen der Struktur `driver` entsprechen. Die verwendeten Typen werden im Anhang A.1 (Seite 107) vorgestellt.

Diejenigen Funktionen, die einen Rückgabewert vom Typ `dev_t` verwenden, müssen einen Fehlerfall signalisieren können. Dies können sie nicht durch Rückgabe eines negativen Wertes, denn `dev_t` ist ein vorzeichenloser Typ (siehe seine Definition im Anhang A.1). Wir müssen also eine bestimmte Gerätenummer reservieren, die wir als Fehlersignal verwenden können. Wir nutzen dafür die `0`.

24b ⟨*Geräte Konstanten* 24b⟩≡ (23) 26a▸

```
#define NO_DEV ((dev_t) 0) // no such device
```

Die einzelnen Operationen haben die folgenden Bedeutungen:

`init` Das ist die Initialisierungsfunktion des Gerätes und soll aufgerufen werden, bevor andere Operationen benutzt werden. Diese Funktion soll dafür verwendet werden, um interne Datenstrukturen zu initialisieren usw. Falls die Initialisierung fehlschlägt, wird ein negativer Wert zurückgegeben. In diesem Fall sollen alle anderen Operationen fehlschlagen.

`create` Diese Funktion erzeugt eine Einheit innerhalb des Gerätes. Argument dazu ist die gewünschte Größe in Bytes der neuen Einheit. Diese Größe kann danach mit der Funktion `size` abgefragt werden und muss nicht mit der Argument-Größe übereinstimmen, falls z. B. das Gerät keine feste Größe haben kann oder falls es auf dem Gerät nicht genug Platz für so viele Bytes gibt. Rückgabewert ist die Gerätenummer der neuen Einheit oder `NO_DEV`, falls diese Operation fehlgeschlagen ist. Siehe dazu den Abschnitt 3.1.

`delete` Eine Einheit, die zuvor mittels `create` erzeugt wurde, kann mit der Operation `delete` wieder gelöscht werden. Argument ist eine Gerätenummer, die von `create` zurückgegeben wurde. Rückgabewert ist dieselbe Gerätenummer, falls die Operation erfolgreich war oder `NO_DEV`, falls nicht.

`read` Diese Funktion erlaubt das Lesen aus dem Gerät. Argumente sind die Gerätenummer der Einheit, die Startadresse in Bytes auf der Einheit (`start`), der Puffer, wo der gelesene Inhalt gespeichert werden soll (`buffer`) und die Anzahl von Bytes, die gelesen werden sollen (`nbytes`). Rückgabewert ist die Anzahl von Bytes, die erfolgreich gelesen wurden. Ein negativer Wert ist ein Fehlersignal.

Angenommen, der `read`-Pointer zeigt auf die noch nicht implementierte Funktion `rd_read`, dann müsste man den dritten Block aus der Einheit mit Nummer 1 wie folgt ablesen können:

```
char buffer[BLOCK_SIZE] = { 0 };
dev_t device = MKDEV(RAM_MAJOR, 1);
ssize_t bytes
    = rd_read(device, 2*BLOCK_SIZE, buffer, BLOCK_SIZE);
if (bytes < 0){...}
```

`write` Mit dieser Funktion kann man eine Einheit beschreiben. Die Startadresse in Bytes auf der Einheit wird mit dem Argument `start` angegeben. Der Puffer, der auf das Gerät geschrieben werden soll, wird mit dem Argument `buffer` übergeben. Das letzte Argument `nbytes` besagt, wie viele Bytes aus dem Puffer geschrieben werden sollen. Rückgabewert der Funktion ist die Anzahl der Bytes, die erfolgreich geschrieben wurden. Ein negativer Rückgabewert ist ein Fehlersignal. Man merkt, dass die `read`- und `write`-Operationen dieselbe Signatur haben. Sie unterscheiden sich lediglich in der Schreibrichtung.

`size` Mit dieser Funktion kann die Größe in Bytes einer Einheit abgefragt werden. Argument ist die Gerätenummer vom Typ `dev_t` (siehe Abschnitt 3.1.1). Eine nicht initialisierte Einheit hat die Größe `0`.

Ein Beispiel für die Verwendung dieser Geräte-Operationen kann man im Abschnitt 3.4.3 (Seite 31) finden.

3.3 Treibertabelle

Wir haben in der Struktur `driver` festgelegt, welche Operationen muss ein Gerätetreiber implementieren, um in unser Modell zu passen, bzw. unsere Treiberschnittstelle zu realisieren. Nehmen wir an, wir haben eine Menge von Funktionen geschrieben, die der Treiberschnittstelle entsprechen und die eine RAM-Disk implementieren. Es stellt sich nun die Frage, wie wir diese Funktionen verwenden, um sie als eine Implementierung der Treiberschnittstelle zu kennzeichnen. Um die Analogie mit objektorientierten Sprachen weiter zu verwenden, würden wir in Java die Funktionen (Methoden) in einer Klasse kapseln und diese Klasse als eine Unterklasse von `Driver` deklarieren, also ungefähr so:

```
class RAMDriver extends Driver {
    public int init() {...}
    public int create(int size) {...}
    public int delete(int device) {...}
    ...
}
```

Diesen Code (Java mit C Dialekt) würde der Compiler so übersetzen, dass er für diese Klasse eine sogenannte „Tabelle virtueller Methoden“ erstellt, die jedem Methodennamen die Adresse des entsprechenden Codes zuordnet (siehe [19]). Er würde praktisch eine Tabelle von Funktionszeigern erstellen mit zusätzlichen Informationen wie Klassenname, Angaben zur Vererbung etc. Siehe dazu das Beispiel in der Abbildung 3.5.

Treiber	init	create	delete	read	write
RAM-Disk	`rd_init`	`rd_create`	`rd_delete`	`rd_read`	`rd_write`
Hard-Disk	`hard_init`	`hard_create`	`hard_delete`	`hard_read`	`hard_write`
DVD	`dvd_init`	`dvd_create`	`dvd_delete`	`dvd_read`	`dvd_write`
Floppy	`flpy_init`	`flpy_create`	`flpy_delete`	`flpy_read`	`flpy_write`

Abbildung 3.5: Tabelle von hypothetischen Treiber-Funktionen, die eine generische Schnittstelle implementieren.

Eine solche Tabelle werden wir auch in der Datei `device.c` schreiben. Wir können sie leicht realisieren, indem wir ein Array von `driver`-Strukturen verwenden. Die Tabellenzeilen sind die einzelnen Array-Einträge und die Spalten die Strukturfelder. Bevor wir das machen, überlegen wir uns die Größe des Arrays, denn in C müssen wir diese Größe kennen. Sie hängt davon ab, wieviele Geräte wir unterstützen wollen. Wir werden uns an der Minix-Definition in [14, S. 598,686,1019] (Diskussion der `dmap` Tabelle) orientieren und diese Größe auf 32 setzen.

26a ⟨*Geräte Konstanten* 24b⟩+≡ (23) ◁24b

```
#define NR_DEVICES  32
```

Die Indizes der einzelnen Treiber-Einträge müssen zusätzlich festgelegt werden. Dafür verwenden wir die Major-Nummer des jeweiligen Gerätes.

An dieser Stelle definieren wir diejenigen Major-Nummern, die uns an diesem Punkt bekannt sind. Hier ist die Stelle im Code, wo neue Nummern hinzugefügt werden sollen.

26b ⟨*Major-Nummern* 26b⟩≡ (23)

```
/* major numbers of devices and their drivers */
#define NODEV_MAJOR   0  // no device
#define RAM_MAJOR     1  // ramdisk
```

Die erste Nummer ist eine reservierte Nummer: Gerätetyp `0` wollen wir als Fehlersignal reservieren. Die zweite Nummer geben wir dem RAM-Treiber, den wir später im Abschnitt 3.4 implementieren werden. In Minix sind diese Nummern in der Datei include/minix/dmap.h deklariert. Für die Linux-Definitionen siehe [7].

Angenommen, die Tabelle heißt `dmap` und wir wollen die Funktion `open` dieses Treibers aufrufen, so müssten wir Code wie den folgenden schreiben:

```
dmap[RAM_MAJOR].open();
```

Nun, das ist keine schöne Syntax, die wir dem Benutzer unserer Schnittstelle anbieten wollen, auch wenn sie kurz und praktisch ist. Darüberhinaus erlaubt sie direkten Zugriff auf die Tabelle, was nicht gerade wünschenswert ist, denn so könnte man die Tabelleneinträge leicht manipulieren, wie der folgende Code zeigt:

```
dmap[RAM_MAJOR].read = dmap[RAM_MAJOR].write;
```

Um die Tabelle von solchen Manipulationen zu schützen, werden wir die Tabelle als `static` in einer eigenen Modul-Datei deklarieren, in `device.c`. Bevor wir das machen, deklarieren wir noch in der Headerdadei `device.h` diejenigen Funktionen, die unsere generische öffentliche Schnittstelle zu den Treiberfunktionen darstellen. Diese Funktionen sollen vom Dateisystem und von anderen Komponenten verwendet werden, um die Treiberfunktionen anzusprechen.

27a ⟨*Generische Treiberschnittstelle* 27a⟩≡ (23)

```
/* generic interface to the device drivers */

int     dev_init   (dev_t device);
dev_t   dev_create (dev_t device, size_t nbytes);
dev_t   dev_delete (dev_t device);
ssize_t dev_read   (dev_t device, off_t start, char buffer[], size_t nbytes);
ssize_t dev_write  (dev_t device, off_t start, char buffer[], size_t nbytes);
size_t  dev_size   (dev_t device);
const char* dev_name (dev_t device);
```

Diese Funktionen sind nichts anderes als die Funktionen aus der `driver`-Struktur, die wir leicht umbenannt haben und die wir, falls erforderlich, mit einem zusätzlichen Argument erweitert haben: die Gerätenummer bzw. die Major-Nummer `device`, die wir als Tabellenindex verwenden wollen. Zusätzlich haben wir die Funktion `dev_name` eingefügt. Sie erlaubt, den Namen eines Treibers oder eines Gerätes zu lesen. Der Name wird in der Tabelle eingetragen und ist nicht Bestandteil der `driver`-Struktur.

Es ist praktisch, an dieser Stelle ein paar Makros zu definieren, die auf der einen Seite diese Major- und Minor-Nummer aus einem `dev_t`-Argument extrahieren und auf der anderen Seite einen `dev_t`-Typ aus einer Major- und Minor-Nummer ausrechnen. Das sind dieselben Makros wie in dem Linux-Header <linux/kdev_t.h>.

27b ⟨*Makros* 27b⟩≡ (23)

```
#define MAJOR(dev)      ((dev)>>8)
#define MINOR(dev)      ((dev) & 0xFF)
#define MKDEV(ma,mi)    ((ma)<<8 | (mi))
```

3.3.1 Generische Geräte-Funktionen

Nun kommen wir zu dem Punkt, die oben deklarierten generischen Funktionen und die Treibertabelle zu implementieren. Dies erfolgt in der Datei `device.c`, die den folgenden Aufbau hat:

27c ⟨*device.c* 27c⟩≡

```
#include "device.h"
#include "stddef.h"

⟨Inkludierte Deklarationen der Treiberfunktionen 28a⟩
⟨Definition der Treibertabelle 28b⟩
⟨Implementierung der generischen Treiberschnittstelle 29a⟩
```

Am Anfang inkludieren wir die Headerdatei `stddef.h`, die im Anhang A.3 (Seite 111) vorgestellt wird. Sie definiert unter anderem den NULL-Pointer, den wir verwenden werden. Danach inkludieren wir diejenigen Headerdateien, die Treiberfunktionen nach dem Muster der `driver`-Struktur deklarieren, denn diese Funktionen wollen wir ja in der Tabelle eintragen.

28a ⟨*Inkludierte Deklarationen der Treiberfunktionen* 28a⟩≡ (27c)

```
#include "ramdisk-driv.h"
```

Die Treibertabelle definieren wir statisch (sichtbar nur in der `device.c` Datei), damit sie gegen unerwünschte Manipulationen geschützt ist. Gleich bei der Definition tragen wir den einzigen Treiber ein, den wir kennen und den wir im Abschnitt 3.4 ab der Seite 30 implementieren.

28b ⟨*Definition der Treibertabelle* 28b⟩≡ (27c)

```
static
struct {const char *name;
        driver      driv;}
dmap [NR_DEVICES] =
{
  [RAM_MAJOR]=
    {
      "ramdisk",
      {rd_init, rd_create, rd_delete, rd_read, rd_write, rd_size}
    }
};
```

Diese Definition bedarf vielleicht einer Erklärung. Um die Struktur `driver` mit einem Namen zu erweitern, kapseln wir sie in einer anderen anonymen Struktur, die aus zwei Elementen besteht: einem C-String (der Name) und einer `driver` Strukur. Das Array `dmap` besteht aus solchen anonymen Strukturen. Bei der Initialisierung verwenden wir die C99-Syntax die uns erlaubt, gezielt bestimmte Indizes zu belegen.

Hier ist die Stelle, wo man neue Treiber eintragen kann. Der Workflow für einen neuen Treiber ist der folgende:

1. Schreibe Funktionen, deren Signaturen den Funktionszeigern aus der `driver` Struktur entsprechen.
2. Deklariere eine neue Major-Nummer in `device.h`. Bedenke, dass sie als Index verwendet werden soll, also muss sie positiv und kleiner als `NR_DEVICES` sein.
3. Deklariere diese Funktionen in einer Headerdatei und inkludiere diese Headerdatei hier in `device.c`, wie wir es oben gemacht haben.
4. Im Array `dmap` am selben Index wie die oben definierte Major-Nummer, trage die geschriebenen Funktionen ein. Verwende dafür die C99-Syntax (direkte Indexangabe), die wir verwendet haben nach dem folgenden Muster:

   ```
   [MAJOR_NO]=
   {
       "Treibername",
       {Funktionsnamen...}
   }
   ```

Die Reihenfolge der Funktionsnamen entspricht der Reihenfolge der Funktionszeiger in der `driver` Struktur.

Nun kommen wir zur Implementierung der generischen Funktionen, die in der Abbildung 3.4 auf Seite 23 angedeutet werden (dort werden noch die Funktionsnamen der `driver`-Struktur verwendet).

Wir fangen mit einer Hilfsfunktion an. Sie prüft, ob eine Zahl einen gültigen Index in dem Treiberarray darstellt, d.h. ob sie eine gültige Major-Nummer ist. Wir werden diese Funktion in jeder der nachfolgenden Funktionen verwenden, wenn wir die Major-Nummer prüfen.

29a ⟨*Implementierung der generischen Treiberschnittstelle* 29a⟩≡ (27c) 29b▷

```
inline static int inrange(int major)
{
    return (major > NODEV_MAJOR && major < NR_DEVICES);
}
```

Die Funktionen tun nicht viel und sind sehr ähnlich zueinander. Zuerst extrahieren sie die Major-Nummer aus einer Gerätenummer und dann prüfen sie, ob diese Major-Nummer einen gültigen Index in der Treibertabelle darstellt. Ist es so und ist die entsprechende Funktion in der Tabelle definiert (kein NULL-Pointer), so wird die Funktion mit den übrigen Argumenten aufgerufen, sonst wird ein Fehlersignal (Errorcode) zurückgegeben. Man kann hier ein Entwurfsmuster erahnen: „Delegate“ oder auch „Adapter“, das wir auch mit der Sprache C verwenden können – von Entwurfsmustern spricht man im Kontext der objektorientierten Sprachen (siehe [8]). Da die Funktionen praktisch dieselbe Logik implementieren, können wir sie alle am Stück angeben.

29b ⟨*Implementierung der generischen Treiberschnittstelle* 29a⟩+≡ (27c) ◁29a

```
const char* dev_name(dev_t device)
{
    int major = MAJOR(device);
    if (inrange(major) && dmap[major].name) {
        return dmap[major].name;
    } else {
        return NULL;
    }
}

int dev_init(dev_t device)
{
    int major = MAJOR(device);
    if (inrange(major) && dmap[major].driv.init) {
        return dmap[major].driv.init();
    } else {
        return -1;
    }
}

dev_t dev_create(dev_t device, size_t nbytes)
{
    int major = MAJOR(device);
    if (inrange(major) && dmap[major].driv.create) {
        return dmap[major].driv.create(nbytes);
```

```
    } else {
        return NO_DEV;
    }
}

dev_t dev_delete(dev_t device)
{
    int major = MAJOR(device);
    if (inrange(major) && dmap[major].driv.delete) {
        return dmap[major].driv.delete(device);
    } else {
        return NO_DEV;
    }
}

ssize_t dev_read(dev_t device, off_t start, char buffer[], size_t nbytes)
{
    int major = MAJOR(device);
    if (inrange(major) && dmap[major].driv.read) {
        return dmap[major].driv.read(device, start, buffer, nbytes);
    } else {
        return -1;
    }
}

ssize_t dev_write(dev_t device, off_t start, char buffer[], size_t nbytes)
{
    int major = MAJOR(device);
    if (inrange(major) && dmap[major].driv.write) {
        return dmap[major].driv.write(device, start, buffer, nbytes);
    } else {
        return -1;
    }
}

size_t dev_size(dev_t device)
{
    int major = MAJOR(device);
    if (inrange(major) && dmap[major].driv.size) {
        return dmap[major].driv.size(device);
    } else {
        return -1;
    }
}
```

3.4 RAM-Disk Treiber

Im Abschnitt 3.2 haben wir eine Treiberschnittstelle vorgestellt, die die Operationen eines abstrakten Gerätes spezifiziert. Die Schnittstelle wurde in der Struktur `driver` konkretisiert. In diesem Abschnitt werden wir den RAM-Disk-Treiber implementieren.

3.4.1 Was ist eine RAM-Disk?

Wie der Name schon andeutet, eine RAM-Disk ist eine Disk, die sich im RAM oder Speicher befindet. Als Disk, so wie im Abschnitt 2.1.2 beschrieben, ist die RAM-Disk eine Reihe von Blöcken im Speicher, also praktisch ein Array, das man mit der Disk-Schnittstelle ansprechen kann. Der Inhalt des Arrays und wie er interpretiert wird, ist nicht Sache der RAM-Disk selber, denn eine Disk weiß nicht, was sie enthält. Ihre Aufgabe ist, den Zugriff auf die Speicherbereiche so zu regulieren, dass man die Disk-Operationen simuliert.

3.4.2 Öffentliche Schnittstelle

Die öffentliche Schnittstelle zu den RAM-Disk Operationen schreiben wir in einer Headerdatei namens `ramdisk-driv.h` (von ramdisk driver). Die deklarierten Funktionen entsprechen den Einträgen der `driver` Struktur. Zusätzlich wird die Funktion `rd_load` deklariert, die spezifisch zu diesem Modul ist.

31 ⟨*ramdisk-driv.h* 31⟩≡

```
#ifndef RAMDISK_DRIV_H
#define RAMDISK_DRIV_H

#include "sys/types.h"

⟨RAM-Disk Konstanten 33b⟩

int     rd_init   (void);
dev_t   rd_create (size_t nbytes);
dev_t   rd_delete (dev_t device);
ssize_t rd_read   (dev_t device, off_t start, char buffer[], size_t nbytes);
ssize_t rd_write  (dev_t device, off_t start, char buffer[], size_t nbytes);
size_t  rd_size   (dev_t device);

dev_t   rd_load   (void *mem, size_t size);

#endif
```

3.4.3 Verwendung

Wenn man mit solchen Modulen zu tun hat, wie wir in diesem Abschnitt implementieren werden, ist es praktisch, wenn man ein Beispiel für die Verwendung des Moduls sieht, denn es können bestimmte Fehler auftreten, wenn man das Modul nicht richtig verwendet. Diese Fehler liegen meistens daran, dass die Funktionen in falscher Reihenfolge aufgerufen werden.

Das Hauptziel in der Arbeit mit einem Gerätemodul wie der RAM-Disk-Treiber ist das Gerät zu lesen und zu beschreiben. Um dies in dem Fall unseres RAM-Disk-Treibers zu erreichen, sind die folgenden Schritte zu befolgen, wobei ihre Reihenfolge zu beachten ist:

1. Das Modul initialisieren mit der Funktion `rd_init`.

2. Eine RAM-Disk erzeugen. Dafür verwendet man die Funktion `rd_create`. Der Rückgabewert dieser Funktion ist eine Gerätenummer, die in den folgenden Schritten benutzt wird. Diese Nummer enthält die Major- und Minor-Nummer der erzeugten RAM-Disk.
3. Eventuell die Größe der erzeugten RAM-Disk mit der Funktion `rd_size` abfragen.
4. Nun können die Funktionen `rd_read` und `rd_write` verwendet werden, um die RAM-Disk zu lesen und zu beschreiben.
5. Als letztes kann man mit `rd_delete` die RAM-Disk löschen.

Wir zeigen nun ein konkretes Beispiel, wie diese Schritte implementiert werden können. Dafür verwenden wir, um das Beispiel allgemeiner zu halten, nicht die konkreten RAM-Disk-Funktionen, sondern die allgemeinen Geräte-Funktionen, so wie sie in der Headerdatei `device.h` im Abschnitt 3.2 definiert sind. Es sind fast dieselben Funktionsnamen, wenn man „rd_“ mit „dev_“ ersetzt.

```
#include "device.h"
#define SIZE 16

int ramdisk_example(void)
{
    dev_t device = MKDEV(RAM_MAJOR, 0);
    if (dev_init(device) < 0) {
        return -1;
    }

    dev_t devno;
    if ( (devno = dev_create(device, SIZE)) == NO_DEV ) {
        return -1;
    }

    char buffer1[SIZE] = { 0 };
    char buffer2[SIZE] = { 0 };

    int i;
    for (i = 0; i < SIZE; i++) {
        buffer1[i] = 'A' + i;
    }

    if (dev_write(devno, 0, buffer1, SIZE) < 0) {
        dev_delete(devno);
        return -1;
    }

    if (dev_read(devno, 0, buffer2, SIZE) < 0) {
        dev_delete(devno);
        return -1;
    }

    dev_delete(devno);

    return 0;
}
```

Die Funktion verwendet die allgemeine Geräteschnittstelle, um zuerst eine RAM-Disk der Größe 16 Bytes zu erzeugen. Danach wird das Array `buffer1` mit den ersten 16 Buchstaben gefüllt und in die RAM-Disk geschrieben. Von dort aus wird es in das Array `buffer2` gelesen. Am Ende haben die zwei Arrays denselben Inhalt.

3.4.4 Interaktion mit ULIX

Der RAM-Disk-Treiber verwendet ein Minimum von Funktionen, die in dem ULIX Code schon implementiert wurden. Das sind insbesondere Funktionen zum Speicher allokieren und frei geben. Ein paar dieser Funktionen sind in der Headerdatei `ulix.h` deklariert, allerdings noch nicht alle. Darüberhinaus enthält `ulix.h` bestimmte Typdefinitionen wie `size_t`, die wir ebenfalls im Anhang A.1 deklariert haben. Bis die ULIX-Headerdateien sich stabilisieren – ULIX befindet sich ja noch in starker Entwicklung – werden wir die Kollision der Typdefinitionen durch eine unschöne Lösung vermeiden: In der Datei `ramdisk-driv.c`, die den RAM-Disk Treiber implementiert, deklarieren wir selber die benötigten Funktionen und verzichten auf die Headerdatei `ulix.h`. Ziel für die Zukunft soll sein, diese Funktion-Deklarationen durch Inkludieren von entsprechenden Headerdateien zu ersetzen.

33a ⟨*Deklaration der Ulix Funktionen* 33a⟩≡ (34a)

```
/* replace this with Ulix-Headers */
typedef unsigned int uint;
extern void *memcpy(void *dest, const void *src, size_t count);

// soll wegen posix size_t sein, nicht uint
extern void* kmalloc (uint size);
extern void  kfree   (void* pointer);
/* end of ulix imports */
```

3.4.5 RAM-Disk-Tabelle

Der RAM-Disk Treiber soll mehrere RAM-Disks unterstützen, die durch Minor-Nummern identifiziert sind (siehe Abschnitt 3.1.1, Seite 21). Wir haben hier die Möglichkeit, entweder eine feste oder eine variable Anzahl von RAM-Disks zu verwenden. Eine feste Anzahl von 16 RAM-Disks ist einfacher zu realisieren und so entscheiden wir uns für diese Möglichkeit.

33b ⟨*RAM-Disk Konstanten* 33b⟩≡ (31)

```
#define MAX_RAMDISKS 16
```

Diese 16 mögliche RAM-Disks kann man leicht in einer Tabelle verwalten. Eine Tabelle heißt, wie bei der Treibertabelle auch, ein Array von Strukturen, die verschiedene Verwaltungsinformationen über eine RAM-Disk speichert. Diese Tabelle werden wir lokal in der Datei `ramdisk-driv.c` deklarieren, damit sie von Manipulationen von außen geschützt ist.

33c ⟨*Deklaration der RAM-Disk Tabelle* 33c⟩≡ (34a)

```
static struct {
    char   *ram;
    size_t size;
} ramdisk [MAX_RAMDISKS];
```

Die Tabelle (Array) `ramdisk` enthält genau 16 Zeilen: eine Zeile für jede mögliche RAM-Disk. In jedem Eintrag stehen die folgenden Informationen:

1. die Startadresse des Speicherbereichs der RAM-Disk (Feld `ram`)
2. seine Größe in Bytes (Feld `size`)

Der Datentyp `size_t`, der für die Größe des Speicherbereichs verwendet wird, ist in der POSIX-Datei `sys/types.h` deklariert (Anhang A.1, Seite 107).

Die Tabelle wird mit der Minor-Nummer einer RAM-Disk indiziert: Hat eine RAM-Disk die Minor-Nummer x, so befindet sich diese RAM-Disk am Index x in der Tabelle.

Ein RAM-Disk-Eintrag in der Tabelle ist dann frei, wenn das `ram` Feld, dass auf den entsprechenden Speicherbereich zeigt, ein Nullpointer ist.

3.4.6 Implementierung

34a ⟨*ramdisk-driv.c* 34a⟩≡

```
#include "stddef.h"
#include "device.h"
#include "ramdisk-driv.h"

⟨Deklaration der Ulix Funktionen 33a⟩
⟨Deklaration der RAM-Disk Tabelle 33c⟩
⟨RAM-Disk Funktionen 34b⟩
```

Die RAM-Disk-Funktionen sind diejenigen, die wir in der öffentlichen Schnittstelle deklariert haben und ein paar Hilfsfunktionen. Wir werden sie alle nacheinander in den nächsten Abschnitten implementieren.

3.4.6.1 Freien Eintrag finden

Eine Hilfsfunktion, die wir oft verwenden werden, ist einen freien Eintrag in der RAM-Disk-Tabelle zu finden.

34b ⟨*RAM-Disk Funktionen* 34b⟩≡ (34a) 35a▷

```
static
int findfree(void)
{
    int i;
    for (i = 0; i < MAX_RAMDISKS; i++) {
        if (ramdisk[i].ram == NULL) {
            return i;
        }
    }
    return -1;
}
```

Die Funktion tut nichts anderes als linear alle Tabelleneinträge zu durchsuchen. Falls sie einen freien Eintrag findet, gibt sie seinen Index zurück. Falls nicht, gibt sie −1 zurück, einen ungültigen Index.

3.4.6.2 Initialisieren

Das RAM-Disk Modul zu initialisieren bedeutet, alle Einträge in der RAM-Disk-Tabelle freizugeben. Man könnte meinen, dass das sowieso der Fall ist, wenn das Modul in den Speicher geladen wird, denn die Tabellendeklaration (Array `ramdisk`) wird vom Compiler so übersetzt, dass sie anfangs mit Nullen gefüllt ist. Man müßte für die Initialisierung nichts machen, also praktisch die Funktion `rd_init` leer lassen. Falls aber ein Benutzer unser Modul verwendet und dann zu einem späteren Zeitpunkt wieder diese Funktion aufruft, bleiben mit einer leeren Funktion alle Tabelleneinträge unverändert und, bei einer vollen Tabelle, können auch nach dem Aufruf von `rd_init` keine neuen RAM-Disks allokiert werden. Das wollen wir natürlich nicht.

Wir implementieren deshalb die `rd_init` Funktion so, dass sie über alle Tabelleneinträge läuft und jede freigibt. Bevor wir das machen, schreiben wir eine kleine Hilfsfunktion, die einen Eintrag in der Tabelle bzw. den dort eingetragenen Speicherbereich freigibt. Für die Speicherfreigabe wird die ULIX-Funktion `kfree` verwendet. Falls der angegebene Index falsch ist, wird −1 zurückgegeben, sonst wird der Speicher an diesem Index freigegeben, seine Adresse und Größe auf Null gesetzt und der Index zurückgegeben.

35a ⟨*RAM-Disk Funktionen* 34b⟩+≡ (34a) ◁34b 35b▷

```
static
int ram_free(int minor)
{
    if (minor < 0 || minor >= MAX_RAMDISKS) {
        return -1;
    }

    if (ramdisk[minor].ram) {
        kfree(ramdisk[minor].ram);
        ramdisk[minor].ram = NULL;
    }
    ramdisk[minor].size = 0;
    return minor;
}
```

Damit können wir die Initialisierungsfunktion sehr kurz schreiben. Es wird für jeden Eintrag in der RAM-Disk-Tabelle die obige Funktion aufgerufen.

35b ⟨*RAM-Disk Funktionen* 34b⟩+≡ (34a) ◁35a 36a▷

```
int rd_init(void)
{
    int i;
    for (i = 0; i < MAX_RAMDISKS; i++) {
        ram_free(i);
    }
    return 0;
}
```

3.4.6.3 Erzeugen und Löschen

Eine neue RAM-Disk bestimmter Größe zu erzeugen heißt, einen Speicherbereich dieser Größe zu allokieren. Wir suchen zuerst nach einem freien Platz in der RAM-Disk-Tabelle. Falls wir keinen finden, geben wir `NO_DEV` zurück (kein Gerät mehr frei). Falls wir einen finden, versuchen wir Speicher zu allokieren. Falls die Allokierung erfolgreich ist, speichern wir die Adresse des allokierten Speicherbereichs und seine Größe in der Tabelle, berechnen die entsprechende Gerätenummer und geben sie zurück. Wenn wir die Gerätenummer berechnen, verwenden wir den gefundenen Index als Minor-Nummer und `RAM_MAJOR` als Major-Nummer. Alle Werte und Makros sind in der Headerdatei `device.h` deklariert (Abschnitt 3.3, Seite 25).

36a ⟨*RAM-Disk Funktionen* 34b⟩+≡ (34a) ◁35b 36b▷

```
dev_t rd_create(size_t nbytes)
{
    int minor = findfree();
    if (minor < 0) {
        return NO_DEV;
    }

    void *ram = kmalloc(nbytes);
    if (ram) {
        ramdisk[minor].ram = ram;
        ramdisk[minor].size = nbytes;
        return MKDEV(RAM_MAJOR,minor);
    } else {
        return NO_DEV;
    }
}
```

Die entgegengesetzte Funktion, `rd_delete`, ist einfacher, denn sie gibt einfach den Speicher frei, der einem Index in der RAM-Disk-Tabelle entspricht.

36b ⟨*RAM-Disk Funktionen* 34b⟩+≡ (34a) ◁36a 37▷

```
dev_t rd_delete(dev_t device)
{
    int minor = MINOR(device);
    if (ram_free(minor) < 0)
        return NO_DEV;
    else
        return device;
}
```

3.4.6.4 Lesen und Schreiben

Nun kommen wir zu den Hauptfunktionen eines Gerätes: Lesen und Schreiben. Diese Operationen implementieren wir als Lesen und Schreiben in einem Array und dafür verwenden wir die ULIX-Funktion `memcpy`, die analog zur Standardfunktion mit demselben Namen funktioniert. Die Lese- und Schreibfunktionen sind sehr ähnlich: sie unterscheiden sich lediglich in der Schreibrichtung. Es liegt daher nahe, eine gemeinsame Funktion zu schreiben, die anhand eines zusätzlichen

Parameters diese Richtung entscheidet. Diese Funktion nimmt sonst alle anderen Parameter der `read`- und `write`-Operationen:

- `device`: Die Gerätenummer der RAM-Disk. Davon wird die Minor-Nummer extrahiert, die einen gültigen Index in der RAM-Disk-Tabelle sein muss.
- `start`: Byteindex in der RAM-Disk, wo die Transferoperation anfangen soll. Dieser Index muss größer gleich Null und kleiner als die Größe der RAM-Disk sein.
- `nbytes`: Wieviele Bytes sollen übertragen werden.
- `buffer`: Quelle- oder Zielpuffer der Daten, die übertragen werden.
- `op`: Ob es gelesen oder geschrieben werden soll.

Rückgabewert ist entweder die Anzahl von Bytes, die erfolgreich kopiert wurden, oder -1, falls eine Überprüfung der Parameter fehlschlägt.

Falls alle Parameter fehlerfrei sind, berechnet die Funktion, wieviele Bytes können tatsächlich übertragen werden: Gibt es ab dem Index `start` bis zum Ende der RAM-Disk weniger Bytes als im `nbytes` angegeben, so wird dieser Parameter entsprechend erniedrigt. Danach entscheidet die Funktion anhand des Parameters `op`, in welche Richtung müssen die Bytes kopiert werden. Ist dieser Parameter gleich `READ`, so wird aus der RAM-Disk in den Puffer `buffer` kopiert, ist `op` gleich `WRITE`, wird aus dem Puffer in die RAM-Disk kopiert. Diese zwei Werte werden unten definiert.

37 ⟨*RAM-Disk Funktionen* 34b⟩+≡ (34a) ◁36b 38a▷

```
#define READ  1
#define WRITE 2

static
ssize_t rd_read_write
(dev_t device, off_t start, char buffer[], size_t nbytes, int op)
{
    int minor = MINOR(device);
    if (minor < 0 || minor >= MAX_RAMDISKS) {
        return -1;
    }
    if (! ramdisk[minor].ram) {
        return -1;
    }
    if (start >= ramdisk[minor].size) {
        return -1;
    }

    if (start + nbytes > ramdisk[minor].size) {
        nbytes = ramdisk[minor].size - start;
    }

    switch (op) {
        case READ:
            memcpy(buffer, ramdisk[minor].ram + start, nbytes);
            break;
        case WRITE:
            memcpy(ramdisk[minor].ram + start, buffer, nbytes);
            break;
```

```
            default:
                return -1;
        }
        return nbytes;
    }
```

Damit können wir die Lese- und Schreibfunktionen einfach implementieren:

38a ⟨*RAM-Disk Funktionen* 34b⟩+≡ (34a) ◁37 38b▷

```
    ssize_t rd_read
    (dev_t device, off_t start, char buffer[], size_t nbytes)
    {
        return rd_read_write(device, start, buffer, nbytes, READ);
    }

    ssize_t rd_write
    (dev_t device, off_t start, char buffer[], size_t nbytes)
    {
        return rd_read_write(device, start, buffer, nbytes, WRITE);
    }
```

3.4.6.5 Größe ermitteln

Die Größe in Bytes einer RAM-Disk kann mit der Funktion `rd_size` ermittelt werden kann. Diese Funktion nimmt ein Argument: die Gerätenummer der RAM-Disk, deren Größe abgefragt wird. Ihr Rückgabewert ist die Größe in Bytes oder -1, falls die Gerätenummer bzw. die dort angegebene Minor-Nummer keiner gültigen RAM-Disk entspricht. Das kann entweder dann passieren, wenn die Minor-Nummer kein gültiger Index ist oder wenn der Eintrag an diesem Index keinen Zeiger auf einen Speicherbereich enthält (das Feld `ram` ist Null).

38b ⟨*RAM-Disk Funktionen* 34b⟩+≡ (34a) ◁38a 39▷

```
    size_t rd_size(dev_t device)
    {
        int minor = MINOR(device);
        if (minor < 0 || minor >= MAX_RAMDISKS) {
            return 0;
        }
        if (! ramdisk[minor].ram) {
            return 0;
        }

        return ramdisk[minor].size;
    }
```

3.4.6.6 Speicherinhalt in RAM-Disk laden

Die nächste Funktion ist nicht Teil der Geräteschnittstelle, sondern spezifisch zum RAM-Disk-Treiber. Die Funktion wird dazu verwendet, eine neue RAM-Disk zu allokieren und mit Inhalt

aus dem Speicher zu belegen. Ihr Verhalten kann auf die Funktionen `rd_create` und `rd_write` reduziert werden und so wird sie auch implementiert. Wir werden diese Funktion im Anhang B.2 (Seite 114) verwenden, um die sogenannte `initrd` („initial ramdisk“) oder `initramfs` („initial ram file system“) in eine neue Ramdisk zu kopieren.

Argumente der Funktion sind ein Pointer `mem` und die Größe des Speicherbereichs, auf den `mem` zeigt. Der Inhalt in `mem` wird in eine neue RAM-Disk kopiert. Rückgabewert ist die Gerätenummer der neuen RAM-Disk oder `NO_DEV`, falls während des Kopiervorgangs ein Fehler auftritt. Mögliche Fehlerquellen sind: es kann keine neue RAM-Disk allokiert werden (RAM-Disk-Tabelle ist voll), die Größe `size` des Speicherbereichs ist zu groß und führt dazu, dass die ULIX-Funktion `kmalloc` fehlschlägt.

39 ⟨*RAM-Disk Funktionen* 34b⟩+≡ (34a) ◁38b

```
dev_t rd_load(void *mem, size_t size)
{
    dev_t dev = rd_create(size);
    if (dev == NO_DEV) {
        return NO_DEV;
    }

    if (rd_size(dev) < size) {
        rd_delete(dev);
        return NO_DEV;
    }
    rd_write(dev, 0, mem, size);

    return dev;
}
```

Um die neue RAM-Disk freizugeben, verwendet man die Funktion `rd_delete`.

4 mkfs.ulixfs

Zu jedem Dateisystem gehört ein Werkzeug, mit dem man einen Datenträger „formatieren“ kann, d.h. seinen Inhalt so zu gestallten, dass er die Struktur eines neuen Dateisystems besitzt. In unserem Fall bedeutet formatieren den Inhalt aus der Abbildung 2.1 (Seite 9) in einer Datei zu erzeugen. Dazu werden wir in diesem Kapitel die Betriebssystemprogrammierung verlassen und ein C Programm namens `mkfs.ulixfs` schreiben. Die folgenden Beispiele zeigen verschiedene Möglichkeiten, wie das Programm ausgeführt werden kann:

```
./mkfs.ulixfs
./mkfs.ulixfs -o disk.img
./mkfs.ulixfs file.txt init.sh sys/stat.h
./mkfs.ulixfs -i 1024 -o disk.img -v file.txt init.sh sys/stat.h
```

Wie man aus diesen Beispielen entnehmen kann, sind alle Optionen und Argumente fakultativ. Der erste Aufruf verwendet die Standardwerte für alle Optionen und gibt keine Dateien an. Es wird eine Datei namens „ulixfs.img“ erzeugt, die das Dateisystem ULIXFS speichert. Das Dateisystem enthält 64 Inodes und hat keine Dateien außer dem root-Verzeichniss. Der zweite Aufruf spezifiziert mit der Option `-o` die Ausgabedatei „disk.img“ statt dem Standardwert „ulixfs.img“. Der dritte Aufruf gibt eine Liste von Dateien an, die in das Dateisystem eingefügt werden sollen und lässt sonst alle Optionen unverändert. Der vierte Aufruf spezifiziert die Anzahl der Inodes, die Ausgabedatei, die Ausgabe von bestimmten Nachrichten und gibt auch eine Liste von Dateinamen an.

Die Programmoptionen haben die folgenden Bedeutungen:

- $-i$ die Anzahl von Inodes, die das Dateisystem verwalten soll. Diese Anzahl bestimmt die maximale Anzahl von Dateien und Verzeichnissen im System, das root-Verzeichniss mit einberechnet. Standardwert ist auf 64 Inodes gesetzt.
- $-o$ Ausgabedatei. Standardwert ist „ulixfs.img“.
- $-v$: wird diese Option angegeben, so werden während des Programmablaufs verschiedene Informationen ausgegeben.
- Argumente: Eine Liste von Dateien, die in das neue Dateisystem kopiert werden sollen. Die Liste muss nur reguläre Dateien enthalten, keine Verzeichnisse. Alle diese Dateien werden in das root-Verzeichnis des neuen Dateisystem kopiert. Wir eine Datei als Pfad mit Verzeichnisanteil angegeben, so wird der Verzeichnisanteil ignoriert und nur der letzte Dateiname verwendet.

Das Programm verwendet POSIX-Funktionen und wurde ausschließlich unter Linux getestet.

Wir werden dieses Programm in einer einzigen C-Datei schreiben: `mkfs.ulixfs.c`. Sie hat den folgenden Aufbau:

42 ⟨*mkfs.ulixfs.c* 42⟩≡

```
#include "ulixfs.h"
#include "sys/stat.h"
#include <stdio.h>
#include <stdlib.h>   // atoi(), abort()
#include <unistd.h>   // getopt(), close(), write()
#include <sys/stat.h> // open()
#include <fcntl.h>    // O_WRONLY
#include <string.h>   // memcpy
#include <time.h>     // time()
#include <sys/mman.h> // mmap()
#include <libgen.h>   // basename(), posix 2003

⟨Programmdaten 43b⟩
⟨Makros 48a⟩
⟨Funktionsdeklarationen 43a⟩

int main(int argc, char **argv)
{
    int ind = parseargs(argc, argv);

    if (generate_clean_fs() < 0) {
        fprintf(stderr, "Could not generate file system.\n");
        return 1;
    }

    insert_files(argc, argv, ind);

    return 0;
}

⟨Programmfunktionen 43c⟩
```

Das Programm hat im Großen zwei Teile: das neue Dateisystem generieren und die Liste der Dateien in das System kopieren. Der erste Teil wir von der Funktion `generate_clean_fs` und der zweite Teil von der Funktion `insert_files` übernommen. Vor diesen Teilen werden die Programmargumente eingelesen.

Eine stilistische Sache wäre vielleicht zu bemerken: die `main` Funktion, so wie wir sie angegeben haben ist ein Beispiel für „Programming by Intention“: Bevor wir die verwendete Funktionen implementiert haben, schreiben wir ihren Kontext und Verwendung und machen uns so einen Eindruck, von dem was wir später programmieren wollen.

4.1 Argumente einlesen

Das erste, was das Programm tut, ist die Argumente einzulesen. Um die `main` Funktion sauber zu halten, verwenden wir für die Auswertung der Argumente eine getrennte Funktion namens

`parseargs`:

43a ⟨*Funktionsdeklarationen* 43a⟩≡ (42) 44a▷
```
static int parseargs(int argc, char **argv);
```

Diese Funktion übernimmt dieselben Argumente wie `main` (Anzahl der Programmargumente und die Argumentenliste) und gibt den Index in der Argumentenliste zurück, wo die Liste der Dateien anfängt. Ist dieser Rückgabewert größer gleich die Anzahl der Argumente `argc`, so wurden keine Dateien angegeben.

Die Funktion setzt die Felder einer Optionen-Struktur die wir global wie folgt definieren:

43b ⟨*Programmdaten* 43b⟩≡ (42)
```
static struct {
    int   inodes;
    char *output;
    off_t fsize;
    int   verbose;
} options = {
    64,
    "ulixfs.img",
    0,
    0
};
```

Die Struktur enthält Einträge für die Anzahl der Inodes, die vom Dateisystem verwaltet werden sollen, für den Namen der Ausgabedatei, für ihre Größe in Bytes, nach dem sie geschrieben wurde und ob das Programm „verbose“ sein soll, also ob verschiede Mitteilungen ausgegeben werden sollen. Die Variable `options`, die gleich angelegt wird, enthält Startwerte für diese Daten. Die Funktion `parseargs`, die die meisten dieser Variablen setzt, verwendet die POSIX-Funktion `getopt` um die Programmargumente auszuwerten. Die Verwendung dieser Funktion ist mehr oder weniger standard und wird hier nicht weiter erklärt.

43c ⟨*Programmfunktionen* 43c⟩≡ (42) 44b▷
```
int parseargs(int argc, char **argv)
{
    char opts[] = "i:o:v"; // option characters
    opterr  = 0;           // no error messages from getopt
    int opt = 0;           // the option
    int inodes  = 0;

    while ((opt = getopt(argc, argv, opts)) != -1) {
        switch(opt) {
            case 'i':
                inodes = atoi(optarg);
                if (inodes > 0) {
                    options.inodes = inodes;
                } else {
                    fprintf(stderr,
                    "Ignoring wrong inode number...\n");
                }
                break;
            case 'o':
```

```
                options.output = optarg;
                break;
            case 'v':
                options.verbose++;
                break;
            case '?':
                if (optopt == 'i') {
                    fprintf(stderr,
                    "Option i requires a number of inodes\n");
                } else if (optopt == 'o'){
                    fprintf(stderr,
                    "Option o requires an output file name\n");
                } else {
                    fprintf(stderr, "Unknown option %c\n", optopt);
                }
                break;
            default:
                abort();
        }
    }

    return optind;
}
```

4.2 Dateisystem generieren

Um das Dateisystem zu generieren ist mehr Aufwand notwendig, als zum Parsen den Programmoptionen. Die Grundidee dabei ist die Diskregionen, die im Abschnitt 2.2 (Seite 8) beschrieben sind, nacheinander abhängig von der Anzahl der Inodes zu berechnen und in die Ausgabedatei zu schreiben. Bevor die einzelnen Diskregionen ausgeschrieben werden, werden die Einträge der wichtigsten Komponente des Dateisystem ausgerechnet: der Superblock.

Der ganze Prozess wird in einer Funktion gesteuert, die wiederum in `main` aufgerufen wird:

44a ⟨*Funktionsdeklarationen* 43a⟩+≡ (42) ◁43a 47▷

```
static int generate_clean_fs(void);
```

Sie hat den folgenden Inhalt:

44b ⟨*Programmfunktionen* 43c⟩+≡ (42) ◁43c 48b▷

```
int generate_clean_fs(void)
{
    superblock super = { 0 };
    make_super(&super);

    int fd = open(options.output, O_WRONLY | O_CREAT | O_TRUNC,
                  S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH);
    if (fd == -1) {
        perror(options.output);
        return -1;
    }
```

```
    write_bootblock   (fd);
    write_superblock  (fd, &super);
    write_inode_map   (fd, &super);
    write_block_map   (fd, &super);
    write_inode_table (fd, &super);
    write_data_region (fd, &super);

    close(fd);
    options.fsize = (super.nblocks * BLOCK_SIZE);

    if (options.verbose) {
        printf("Generated UlixFS in file %s...\n", options.output);
        printf("Superblock content:\n\n");
        print_superblock(&super);
    }

    return 0;
}
```

Diese Funktion lässt zuerst die Einträge des Superblock festlegen (siehe unten). Die verwendete Struktur `superblock` ist in `ulixfs.h` deklariert (siehe Kapitel 2). Danach öffnet die Funktion die Ausgabedatei und, nach einer Überprüfung der erfolgreichen Dateiöffnung, ruft sie für jede Diskregion eine entsprechende Funktion, die die jeweilige Region in die Datei schreibt. Nachdem alle Diskregionen geschrieben wurden, wird die Ausgabedatei geschlossen und die Größe in Bytes des Dateisystems in die `options`-Struktur geschrieben, damit wir sie später verwenden können. Falls der Benutzer die Option `-v` angegeben hat, wird der Superblock ausgegeben.

Wenn diese Funktion fertig ist, wurde die Ausgabedatei vollständig geschrieben und geschlossen. Ihr Name und ihre Größe in Bytes stehen in der `options` Struktur.

4.2.1 Einträge des Superblocks ausrechnen

Das schwierigste in diesem Programm ist die Berechnung der Einträge im Superblock aus der Anzahl der Inodes, die als Programmoption übergeben wird. Wir machen daher eine Beispielrechnung durch, um uns das Verfahren näher zu bringen. Siehe auch den Abschnitt 2.2.2 (Seite 10), wo der Superblock vorgestellt wird.

Um die Anzahl der Bytes zu berechnen, die von x Bits belegt werden, verwenden wir die folgende Formel:

$$\text{bytes}(x) = (x - 1) \div 8 + 1 \tag{4.1}$$

Um die Anzahl der Blöcke, die von b Bytes belegt werden, verwenden wir die folgende Formel:

$$\text{blocks}(b) = (b - 1) \div \texttt{BLOCK\_SIZE} + 1 \tag{4.2}$$

Wir verwenden in unserem Beispiel die folgenden Größen:

- $I = 256$ Inodes
- $B = 1024$ Bytes pro Block
- $\overline{I} = 64$ Größe in Bytes einer Inode-Struktur
- $X = I + 1 = 257$ die Anzahl der tatsächlich vorhandenen Inodes in der Inode-Bitmap und Inodetabelle.

Ziel ist alle Felder der Struktur `superblock` zu bestimmen.

magic Die Signatur des Dateisystems ist in `ulixfs.h` fest definiert und kann übernommen werden.

inodes Die Anzahl I der Inodes haben wir auf 256 gesetzt und kann übernommen werden.

imap Wie im Abschnitt 2.2.3.1 (Seite 12) vorgestellt, befindet sich die Inode-Bitmap im Block mit Index $i = 2$, nach dem Bootblock (Index 0) und nach dem Superblock (Index 1). Genauer gesagt, sie befindet sich im Block $s + 1$, wenn der Superblock sich im Block s befindet. Die Blocknummer des Superblocks ist ebenfalls in `ulixfs.h` fest definiert.

bmap Während die Inode-Bitmap eine feste Position $i = 2$ hat, hängt der Anfangsblock b der Block-Bitmap von der Größe der Inode-Bitmap ab, die wir mit $\overline{i}$ notieren. Diese können wir mit (4.1) und (4.2) leicht berechnen. Die $X = 257$ Bits in der Inode-Bitmap passen in 33 Bytes:

$$\text{bytes}(257) = (257 - 1) \div 8 + 1 = 33$$

Diese Bytes passen wiederum in einem Block:

$$\overline{i} = \text{blocks}(33) = (33 - 1) \div 1024 + 1 = 1$$

Die Block-Bitmap fängt also im nächsten Block nach dem Anfangsblock der Inode-Bitmap an, im Block 3. Allgemein können wir die Formel benutzen:

$$\overline{i} = \text{blocks}(\text{bytes}(X)) \tag{4.3}$$

$$b = i + \overline{i} \tag{4.4}$$

Mit b Anfangsblock der Block-Bitmap, i Anfangsblock der Inode-Bitmap, $\overline{i}$ Größe in Blöcken der Inode-Bitmap.

itable Der Startblock der Inodetabelle hängt vom Startblock b der Block-Bitmap und von deren Größe ab, die wir mit $\overline{b}$ notieren. Siehe zur Orientierung die Abbildung 2.1 auf Seite 9. Diese Größe müssen wir berechnen.

Jeder Inode hat eine Liste von zehn Blocknummern, die auf Datenblöcke in der Datenregion verweisen (siehe auch Abschnitt 2.3, Seite 13). Für $I = 256$ Inodes, haben wir also $I \cdot 10 = 256 \cdot 10 = 2560$ mögliche Datenblöcke, die in den Inodes eingetragen werden können. Diese 2560 Blocknummern werden in der Block-Bitmap auf 2560 Bits abgebildet. Diese belegen nach (4.1) 320 Bytes:

$$\text{bytes}(2560) = (2560 - 1) \div 8 + 1 = 320$$

Diese Bytes belegen nach (4.2) einen Block:

$$\overline{b} = \text{blocks}(320) = (320-1) \div B + 1 = (320-1) \div 1024 + 1 = 1$$

Die Inodetabelle fängt also in dem nächten Block nach der Block-Bitmap an, im Block 4. Allgemein können wir den Startblock t der Inodetabelle mit der folgenden Formel berechnen:

$$\overline{b} = \text{blocks}(\text{bytes}(I \cdot 10)) \tag{4.5}$$

$$t = b + \overline{b} \tag{4.6}$$

Mit $\overline{b}$ Größe in Blöcken der Block-Bitmap und t Anfangsblock der Inodetabelle.

data Die Datenregion fängt nach der Inodetabelle an und ihr Startblock d hängt somit von deren Größe ab, die wir mit $\overline{t}$ notieren. Diese Größe müssen wir zuerst berechnen.

Die Inodetabelle enthält eine Liste von $X = 257$ `inode`-Strukturen. Diese belegen 16448 Bytes:

$$X \cdot \overline{I} = 257 \cdot 64 = 16448$$

Diese Bytes belegen wiederum nach (4.2) 17 Blöcke:

$$\overline{t} = \text{blocks}(16448) = (16448-1) \div B + 1 = 16447 \div 1024 + 1 = 17$$

Die Datenregion fängt also 17 Blöcke nach der Inodetabelle an, im Block $d = 4 + 17 = 21$. Allgemein können wir die folgende Formel benutzen, um den Startblock d der Datenregion zu berechnen:

$$\overline{t} = \text{blocks}(X \cdot \overline{I}) \tag{4.7}$$

$$d = t + \overline{t} \tag{4.8}$$

Mit t Anfangsblock der Inodetabelle, $\overline{t}$ Größe in Blöcken der Inodetabelle, $\overline{I}$ Größe in Bytes einer `inode`-Struktur.

nblocks Für die Gesamtzahl aller Blöcke im Dateisystem brauchen wir nur noch die Größe in Blöcken der Datenregion, mit $\overline{d}$ notiert. Diese hängt nur von der Anzahl I der Inodes ab und lautet $I \cdot 10$, da jedem Inode zehn Datenblöcke zugewiesen werden (Inode mit Nummer 0 belegt keine Datenblöcke). In unserem Beispiel heißt es $\overline{d} = I \cdot 10 = 2560$ Blöcke in der Datenregion. Damit lässt sich die Anzahl n aller Blöcke, die das Dateisystem ausmachen, als Summe der Größen aller Diskregionen berechnen.

$$n = \underbrace{1}_{\text{Bootblock}} + \underbrace{1}_{\text{Superblock}} + \underbrace{\overline{i}}_{\text{Inode-Bitmap}} + \underbrace{\overline{b}}_{\text{Block-Bitmap}} + \underbrace{\overline{t}}_{\text{Inodetabelle}} + \underbrace{\overline{d}}_{\text{Datenregion}}$$

In unserem Fall ist

$$n = \underbrace{1}_{\text{Bootblock}} + \underbrace{1}_{\text{Superblock}} + \underbrace{1}_{\text{Inode-Bitmap}} + \underbrace{1}_{\text{Block-Bitmap}} + \underbrace{17}_{\text{Inodetabelle}} + \underbrace{2560}_{\text{Datenregion}} = 2581$$

Die Disk ist also ca. 2,5 MB groß.

In der Abbildung 2.2 auf Seite 11 wird ein Superblock dargestellt, dessen Einträge für 1024 Inodes ausgerechnet wurden.

Nun können wir diese Berechnungen in eine Funktion übernehmen:

47 ⟨*Funktionsdeklarationen* 43a⟩+≡ (42) ◁44a 48c▷

```
static void make_super(superblock *super);
```

Diese Funktion liest die Anzahl der Inodes aus der `options` Struktur und berechnet alle Felder der übergebenen `superblock`-Struktur so wie wir oben gezeigt haben. Die Funktion übersetzt praktisch alle unsere Berechnungen der Reihe nach in die C-Sprache.

Zuerst definieren wir die Formel (4.1) und (4.2) als Makros:

48a ⟨*Makros* 48a⟩≡ (42)
```
#define BYTES(bits)   (((bits) - 1)/ 8 + 1)
#define BLOCKS(bytes) (((bytes)- 1)/ BLOCK_SIZE + 1)
```

48b ⟨*Programmfunktionen* 43c⟩+≡ (42) ◁44b 49a▷
```
void make_super(superblock *super)
{
    super->magic    = MAGIC_NO;
    super->inodes   = options.inodes;
    super->imap     = SUPER_BLOCK + 1;

    int I = options.inodes; // Anzahl Inodes
    int X = I + 1;

    int imap_size   = BLOCKS(BYTES(X));
    super->bmap     = super->imap + imap_size;

    int bmap_size   = BLOCKS(BYTES(I * INODE_BLOCKS));
    super->itable   = super->bmap + bmap_size;

    int itable_size = BLOCKS(X * INODE_SIZE);
    super->data     = super->itable + itable_size;

    int data_size   = (I * 10);
    super->nblocks  =
        1 + 1 + imap_size + bmap_size + itable_size + data_size;
}
```

4.2.2 Diskregionen schreiben

Nun schreiben wir alle Funktionen der Reihe nach, die jeweils eine Diskregion in die Ausgabedatei schreiben. Sie werden alle in der Funktion `generate_clean_fs` aufgerufen. Alle diese Funktionen nehmen als Argument einen ganzzahligen Dateideskriptor und eventuell einen Pointer auf eine Superblock-Struktur. Der Dateideskriptor wurde in `generate_clean_fs` generiert und wird dazu verwendet, in die Ausgabedatei zu schreiben. Diese Funktionen verwenden nicht die POSIX-Funktion `lseek` um die Schreibeposition in der Ausgabedatei zu setzen. Sie müssen daher in der richtigen Reihenfolge aufgerufen werden. Dies wird in `generate_clean_fs` sichergestellt.

4.2.2.1 Bootblock schreiben

Der Bootblock ist ein Block voll mit Nullbytes. Daher ist die entsprechende Funktion einfach.

48c ⟨*Funktionsdeklarationen* 43a⟩+≡ (42) ◁47 49b▷
```
static void write_bootblock(int fd);
```

49a ⟨*Programmfunktionen* 43c⟩+≡ (42) ◁48b 49c▷
```
void write_bootblock(int fd)
{
    char block[BLOCK_SIZE] = { 0 };
    write(fd, block, BLOCK_SIZE);
}
```

4.2.2.2 Superblock schreiben

Der Superblock lässt sich auch ziemlich einfach schreiben indem man zuerst einen Block voll mit Nullbytes allokiert und dann den Superblock am Anfang des Block kopiert. Anschließend wird der ganze Block in die Datei geschrieben.

Die entsprechende Funktion hat zwei Argumente: einen Dateideskriptor für die Ausgabedatei und einen Pointer auf die `superblock`-Struktur, die geschrieben werden soll.

49b ⟨*Funktionsdeklarationen* 43a⟩+≡ (42) ◁48c 49d▷
```
static void write_superblock(int fd, superblock *super);
```

Und hier die Funktion:

49c ⟨*Programmfunktionen* 43c⟩+≡ (42) ◁49a 50a▷
```
void write_superblock(int fd, superblock *super)
{
    char block[BLOCK_SIZE] = { 0 };
    memcpy(block, super, sizeof(superblock));
    write(fd, block, BLOCK_SIZE);
}
```

4.2.2.3 Inode-Bitmap schreiben

Die Inode-Bitmap hat nicht, wie der Superblock oder der Bootblock, eine feste Größe, sondern ihre Größe hängt von der Anzahl der Inodes ab. Es müssen also eventuell mehrere Blöcke geschrieben werden. Die Idee dabei ist, in einem frischen System sind nur zwei Inodes belegt: Inode 0 (unbenutzt) und Inode 1. Das ergibt im ersten Byte der Inode-Bitmap den hexadezimalen Wert `0xC0` (siehe auch Abschnitt 2.2.3.1, Seite 12). Wir konstruieren den ersten Block der Inode-Bitmap, indem wir das erste Byte eines Nullblocks auf diesen Wert setzen. Danach setzen wir das Byte wieder auf Null und schreiben die anderen Blöcke, falls notwendig. Die Größe in Blöcken der Inode-Bitmap können wir aus dem Superblock entnehmen, indem wir den Startblock der Inode-Bitmap aus dem Startblock der Block-Bitmap subtrahieren.

Das wird die Aufgabe einer getrennten Funktion sein:

49d ⟨*Funktionsdeklarationen* 43a⟩+≡ (42) ◁49b 50b▷
```
static void write_inode_map(int fd, superblock *super);
```

Sie nimmt, wie `write_superblock` auch, zwei Argumente: den Dateideskriptor der Ausgabedatei und den Superblock des Dateisystems.

50a ⟨*Programmfunktionen* 43c⟩+≡ (42) ◁49c 51a▷

```
void write_inode_map(int fd, superblock *super)
{
    char block[BLOCK_SIZE] = {0xC0, 0, 0, 0};
    write(fd, block, BLOCK_SIZE);
    block[0] = 0;

    int imap_blocks = super->bmap - super->imap;
    int i = 1;

    while (i < imap_blocks) {
        write(fd, block, BLOCK_SIZE);
        i++;
    }
}
```

4.2.2.4 Block-Bitmap schreiben

Die Block-Bitmap wird ähnlich wie die Inode-Bitmap geschrieben. Auf einem neuen Dateisystem sind diejenige Datenblöcke besetzt, die dem root-Verzeichniss gehören. Das root-Verzeichnis besitzt, wie jedes Verzeichnis, zwei Einträge mit den Namen „.“ und „..“, beide assoziiert mit derselben Inodenummer 1 (siehe auch den Abschnitt 2.5, Seite 17).

Schauen wir in die `ulixfs.h` Datei, so können wir dort ablesen, dass ein Verzeichnis-Eintrag (Struktur `dir_entry`) 64 Bytes groß ist. Zwei solche Einträge brauchen 128 Bytes, die in einem einzigen Block passen, denn ein Block ist 1024 Bytes groß. Das root-Verzeichnis braucht also auf einem frischen Dateisystem einen einzigen Datenblock für die Starteinträge „.“ und „..“. Möchten wir in der Block-Bitmap den ersten Datenblock als belegt markieren, so reicht es, das erste Byte dieser Bitmap auf den Wert `0x80` zu setzen (binär: `10000000`).

Man könnte hier meinen, dass das root-Verzeichnis nur dann einen einzigen Datenblock braucht, wenn zwei Verzeichnis-Einträge in einem Block passen. Würde man deren Größe ändern, oder Blockgröße kleiner machen, so könnte man nicht mehr davon ausgehen, dass zwei Einträge in einem einzigen Block passen und wir müssten eigentlich die Anzahl der Datenblöcke allgemein berechnen. Das ist natürlich ein berechtigtes Argument. Wir gehen aber davon aus, dass wir die Blockgröße nicht (wesentlich) kleiner machen. Insbesondere, werden wir die Blockgröße nicht kleiner als 128 Bytes setzen.

Die Funktion, die in `generate_clean_fs` aufgerufen wird und die eine „frische“ Block-Bitmap in die Ausgabedatei schreibt, hat die folgende Signatur, analog zu den anderen Funktionen:

50b ⟨*Funktionsdeklarationen* 43a⟩+≡ (42) ◁49d 51b▷

```
static void write_block_map(int fd, superblock *super);
```

Sie funktioniert ähnlich wie `write_inode_map`, die wir ober besprochen haben: Es wird ein Nullblock allokiert und das erste Byte davon wird auf `0x80` gesetzt (das erste Bit auf 1). Dann wird dieser Block in die Ausgabedatei geschrieben und das erste Byte wieder auf `0` gesetzt (keine weitere

Datenblöcke sind belegt). Fall andere Blöcke notwendig sind, werden sie geschrieben. Die Größe in Blöcken der Block-Bitmap wird aus dem Superblock entnommen: sie ist die Differenz zwischen dem Startblock der Inodetabelle und dem Startblock der Block-Bitmap. Siehe zur Orientierung die Abbildung 2.1 auf Seite 9 und die Abbildung 2.2 auf der Seite 11.

51a ⟨*Programmfunktionen* 43c⟩+≡ (42) ◁50a 51c▷

```
void write_block_map(int fd, superblock *super)
{
    char block[BLOCK_SIZE] = {0x80, 0, 0, 0};
    write(fd, block, BLOCK_SIZE);
    block[0] = 0;

    int bmap_blocks = super->itable - super->bmap;
    int i = 1;

    while (i < bmap_blocks) {
        write(fd, block, BLOCK_SIZE);
        i++;
    }
}
```

4.2.2.5 Inodetabelle schreiben

Nun kommen wir zu der Funktion, die die Inodetabelle in die Ausgabedatei schreibt. Auf einem neuen Dateisystem, ist nur der root-Inode gesetzt. Wir schreiben daher zuerst eine Funktion, die diesen Inode mit Werten belegt.

51b ⟨*Funktionsdeklarationen* 43a⟩+≡ (42) ◁50b 52a▷

```
static void make_root(inode *root, superblock *super);
```

Die Funktion erledigt die folgenden Aufgaben: Das `mode` Feld des Inodes wird so gesetzt, dass der Dateityp auf ein Verzeichniss hindeutet (`S_IFDIR`) und die Zugriffsrechte in Unix-Schreibweise als „rwxr-xr-x“ dargestellt werden können (siehe Abschnitt 2.4.2, Seite 16 und Abbildung 2.5, Seite 16). Die dafür verwendeten Makros befinden sich in der Datei `sys/stat.h`, die wir im Anhang A.2 beschreiben.

Die Dateigröße (Feld `fsize`) setzen wir auf zwei mal die Größe eines Verzeichniseintrages, denn das Verzeichnis enthält ja zwei Einträge („.“ und „..“). Die Anzahl von Hard-Links setzen wir auf zwei, denn die zwei Einträge verweisen auf das root-Verzeichnis selbst. Benutzer-ID und Gruppen-ID setzen wir auf die ID des root-Benutzers und root-Gruppe. Für die Zugriffszeiten verwenden wir die POSIX-Funktion `time`: Sie gibt die aktuelle Zeit auf unserem System zurück, auf dem wir die Datei erzeugen. Als leztes setzen wir die erste Blocknummer des root-Inodes auf den ersten Block in der Datenregion. Dort befinden sich die zwei Verzeichniseinträge (siehe auch Abschnitt 2.3, Seite 13, wo die Inode-Felder vorgestellt werden).

Hier ist die Funktion:

51c ⟨*Programmfunktionen* 43c⟩+≡ (42) ◁51a 52b▷

```
void make_root(inode *root, superblock *super)
{
    root->fsize = 2 * DIR_ENTRY_SIZE;
```

```
    root->mode =
      S_IFDIR | S_IRWXU | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH;
    root->nlinks = 2;
    root->uid = ROOT_UID;
    root->gid = ROOT_GID;

    time_t t = time(NULL);
    root->atime = t;
    root->ctime = t;
    root->mtime = t;

    root->block[0] = super->data;
}
```

Nun können wir mit der Funktion `write_inode_table` fortfahren. Sie schreibt in die Ausgabedatei die Inodetabelle und hat die folgende Signatur:

52a ⟨*Funktionsdeklarationen* 43a⟩+≡ (42) ◁51b 53a▷

```
static void write_inode_table(int fd, superblock *super);
```

Die Idee bei der Implementierung der Funktion ist ähnlich wie bei der Implementierung von `write_block_map`: wir allokieren einen Nullblock und setzen dort den root-Inode, nachdem wir mit `make_root` mit Daten gefüllt haben. Danach schreiben wir den Block in die Ausgabedatei, löschen den root-Inode und schreiben den selben Block so oft wie notwendig.

52b ⟨*Programmfunktionen* 43c⟩+≡ (42) ◁51c 53b▷

```
void write_inode_table(int fd, superblock *super)
{
    inode root  = { 0 };
    inode empty = { 0 };
    make_root(&root, super);

    char block[BLOCK_SIZE] = { 0 };
    inode *inodes = (inode*) block;
    inodes[ROOT_INO] = root;

    write(fd, block, BLOCK_SIZE);

    inodes[ROOT_INO] = empty;
    int itable_blocks = super->data - super->itable;

    int i = 1;
    while (i < itable_blocks) {
       write(fd, block, BLOCK_SIZE);
       i++;
    }
}
```

Eine Bemerkung zur Variable `inodes`: Sie ist ein Zeiger auf ein Array von `inode`-Strukturen, erhält aber die Adresse von `block`. So können wir die im Block liegenden Inodes bequem per Index

ansprechen und müssen nicht Funktionen wie `memcpy` verwenden. Die Variable `empty` ist ein leerer Inode und wird dazu verwendet, den root-Inode wieder zu löschen.

Die Zuweisungen wie „`inodes[ROOT_INO] = root`“ könnten überraschen: sie sind in C gültige Variable-Zuweisungen. Es wird in diesem Fall eine Variable vom Typ `struct inode` einer anderen Variable vom selben Typ zugewiesen. Dabei werden alle Inhalte einer Struktur kopiert, wie bei einer normalen Variable.

4.2.2.6 Datenregion schreiben

Die letzte Diskregion, die Datenregion, enthält die zwei Standardeinträge „.“ und „..“ des root-Verzeichnisses. Die Struktur und Größe dieser Einträge wird durch die Struktur `dir_entry` in der Headerdatei `ulixfs.h` angegeben. Zum schreiben dieser Strukturen und der nachfolgenden Blöcke verwenden wir die folgende Funktion:

53a ⟨*Funktionsdeklarationen* 43a⟩+≡ (42) ◁52a 54▷

```
static void write_data_region(int fd, superblock* super);
```

Sie nimmt wie die vorigen Funktionen auch, zwei Argumente: den Dateideskriptor der Ausgabedatei und einen Pointer auf die Superblock-Struktur. Sie funktioniert analog zu den anderen ähnlichen Funktionen: Zuerst, initialisiert die Funktion drei Strukturen vom Typ `dir_entry`: eine für den „.“-Eintrag, eine für den „..“-Eintrag und eine leere, zum löschen der anderen. Dann wird ein Nullblock allokiert und in der Variable `dirs` in ein Array von `dir_entry` Strukturen konvertiert. In diesem Array werden die ersten Einträge gesetzt. Somit haben wir den ersten Block eines frischen Dateisystems konstruiert. Nachdem wir den Block in die Ausgabedatei geschrieben haben, setzen wir wieder die ersten Einträge auf `empty` und schreiben die weiteren Blöcke.

53b ⟨*Programmfunktionen* 43c⟩+≡ (42) ◁52b 55a▷

```
void write_data_region(int fd, superblock* super)
{
    dir_entry dot   = {ROOT_INO, "." };
    dir_entry ddot  = {ROOT_INO, ".."};
    dir_entry empty = {0,        ""  };

    char block[BLOCK_SIZE] = { 0 };
    dir_entry *dirs = (dir_entry*) block;

    dirs[0] = dot;
    dirs[1] = ddot;

    write(fd, block, BLOCK_SIZE);
    dirs[0] = empty;
    dirs[1] = empty;

    int data_blocks = super->nblocks - super->data;
    int i = 1;

    while (i < data_blocks) {
        write(fd, block, BLOCK_SIZE);
        i++;
    }
```

```
}
```

4.3 Dateien in das Dateisystem einfügen

Bis zu diesem Punkt haben wir ein neues Dateisystem generiert. Nun folgt der zweite Teil des Programms: alle angegebenen Dateien in das Dateisystem einfügen. Die wichtigste Frage dabei ist: wie wollen wir die Ausgabedatei, die das neue Dateisystem enthält, verändern? Würden wir z. B. die „high level“ Funktionen `fread` und `fwrite` verwenden, so würden wir die Größe der Datei ändern, was nicht erwünscht ist. Es bleiben zwei Möglichkeiten: entweder nutzen wir wie bisher die „low level“ Funktionen `read` und `write`, oder wir verwenden eine POSIX-Technik namens „memory mapping“. Die erste Möglichkeit scheint allgemeiner zu sein, ist aber mit zusätzlichem Aufwand verbunden. Die zweite Möglichkeit bedeutet, dass wir die Datei auf ein Array abbilden und alle Dateioperationen mit Array-Operationen ersetzen. Diese Möglichkeit scheint uns einfacher und intuitiver, besonders wenn man mit strukturierten Dateien fester Größe arbeitet, wie in unserem Fall.

Der ganze Prozess, der die Argumentdateien in das Dateisystem einfügt, wird von einer Funktion gesteuert, die den zweiten Teil von `main` ausmacht. Sie hat die folgende Signatur:

54 ⟨*Funktionsdeklarationen* 43a⟩+≡ (42) ◁53a 55b▷

```
static void insert_files(int argc, char **argv, int ind);
```

Argumente sind: Anzahl `argc` der Programmargumente, so wie der `main` übergeben, die Liste `argv` aller Programmargumente und der Index `ind`, wo die Dateiliste in `argv` anfägt.

Zuerst überprüft die Funktion, ob es überhaupt Dateinamen gibt und falls nicht, beendet sie sich, denn die weitere Arbeit ist sinnlos. Dann versucht sie, die Ausgabedatei zu öffnen und beendet sich mit einer Fehlermeldung, falls die Öffnung fehlschlägt. Falls aber die Datei erfolgreich geöffnet wurde, wird mit Hilfe der POSIX-Funktion `mmap` eine „memory map“ eingerichtet. Wenn diese Abbildung nicht fehlschlägt, steht in dem Pointer `filesystem` die Adresse des Speicherbereichs, auf den die Ausgabedatei abgebildet wurde. Ab diesem Punkt kann die Ausgabedatei geschloßen werden und alle Lese- und Schreiboperationen auf den `filesystem` Pointer bewirken Lese- und Schreibeoperationen auf die Ausgabedatei direkt. Es ist als ob diese Datei im Speicher geladen wurde, ohne dass dies tatsächlich der Fall ist.

Die Ausgabedatei so in den Speicher abzubilden hat einige Vorteile. Einer davon ist, dass man den Superblock und andere Diskregionen sehr einfach lesen und schreiben kann. Es reicht z. B. der folgende Code um den Superblock aus der Datei zu laden, einen Wert zu setzen und wieder in die Datei zu schreiben:

```
superblock *super = (superblock *)(filesystem + BLOCK_SIZE);
super->inodes = 0;
```

Tatsächlich wird hier die Ausgabedatei bearbeitet, wir benutzen aber statt der expliziten Funktionen `read` und `write` die Array-Syntax, die viele Operationen einfacher macht.

Der Hauptteil der Funktion, nachdem der Pointer `filesystem` auf die Ausgabedatei abgebildet wurde, läuft in einer `while`-Schleife über alle Dateinamen und ruft für jede Datei die Funktion `insert_file` auf. Wenn die Schleife fertig ist, wird die „memory map“ wieder gelöscht.

Hier ist der Code dieser Funktion. Da sie praktisch aus wenigen Operationen besteht (Datei öffnen, memory map einrichten, wiederholt `insert_file` aufrufen, memory map löschen), können wir sie am Stück angeben:

55a ⟨*Programmfunktionen* 43c⟩+≡ (42) ◁53b 56b▷

```
void insert_files(int argc, char **argv, int ind)
{
    if (ind >= argc) {
        return;
    }

    int fd;
    if ((fd = open(options.output, O_RDWR)) == -1) {
        perror(options.output);
        return;
    }
    // the memory map
    void *filesystem = mmap
          (NULL, options.fsize, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    if(filesystem == MAP_FAILED) {
        perror(options.output);
        close(fd);
        return;
    }

    close(fd);

    while (ind < argc) {
        insert_file(filesystem, argv[ind]);
        ind++;
    }

    munmap(filesystem, options.fsize);
}
```

Auf Details zu den Funktionen `mmap` und `munmap` werden wir hier nicht eingehen. Auf einem Linux-System kann die entsprechende Dokumentation (man page) durch Eingabe von „man 2 mmap“ in einem Terminal gelesen werden.

Kommen wir nun zu der Implementierung der Funktion `insert_file` (ohne „s“ am Ende, da Singular). Wir sind im Programmverlauf zu dem Punkt angekommen, dass `main` die Ausgabedatei geschrieben hat und dann `insert_files` aufgerufen hat. Diese hat die „memory map“ im Pointer `filesystem` eingerichtet und jetzt wird `insert_file` aufgerufen um eine einzige Datei in das Dateisystem einzufügen.

Diese Funktion hat zwei Parameter: ein Pointer auf das ganze Dateisystem und den Name der Datei, die eingefügt werden soll. Hinter dem Dateisystem-Pointer steht die Ausgabedatei.

55b ⟨*Funktionsdeklarationen* 43a⟩+≡ (42) ◁54 56a▷

```
static void insert_file(void *filesystem, char *fname);
```

Bevor wir diese Funktion implementieren, ist es zu bedenken, dass nicht alle Dateien können in das Dateisystem ULIXFS eingefügt werden: Wegen dem Aufbau eines Inodes ist die maximale Dateigröße 10 KB (siehe Abschnitt 2.3). Darüberhinaus muss die Datei lesbar und eine reguläre Datei sein. Es ist daher praktisch, eine Hilfsfunktion zu schreiben, die diese Einschränkungen überprüft:

56a ⟨*Funktionsdeklarationen* 43a⟩+≡ (42) ◁55b 58a▷
```
static int open_file(char *fname, off_t *fsize);
```

Sie überprüft zuerst, ob die Datei lesbar und regulär ist (kein Verzeichnis z. B.) und ob ihre Größe die maximale Dateigröße in ULIXFS nicht überschreitet. Falls das alles gilt, versucht sie die Datei zu öffnen und gibt den entsprechenden Dateideskriptor, sonst −1 zurück. Die Funktion verwendet die POSIX-Funktion `stat` um die Dateiattribute auszlesen.

56b ⟨*Programmfunktionen* 43c⟩+≡ (42) ◁55a 57▷
```
int open_file(char *fname, off_t *fsize)
{
    struct stat st;
    if (stat(fname, &st) == -1) {
        return -1;
    }

    if (! S_ISREG(st.st_mode)) {
        fprintf(stderr, "%s ist not a regular file\n", fname);
        return -1;
    }

    if (st.st_size > MAX_FILE_SIZE) {
        fprintf(stderr, "File %s too big\n", fname);
        return -1;
    }

    int fd = open(fname, O_RDONLY);
    if (fd < 0) {
        fprintf(stderr, "Cannot open file %s\n", fname);
        return -1;
    }
    *fsize = st.st_size;

    return fd;
}
```

Nun können wir die Funktion `insert_file` schreiben. Sie ist wieder ein Beispiel von „Programming by Intention“: Wir verwenden Funktionen, die wir noch nicht geschrieben haben und bestimmen somit auf einfacher Weise, welche Funktionen wir noch brauchen, deren Kontext und deren Verwendung. Dadurch, dass wir diese noch nicht implementierte Funktionen schon in ihrem Kontext setzen, machen wir uns zugleich eine Idee darüber, was sie leisten sollen und welche Rückgabewerte sie haben. Wir versuchen auch, diese Funktionen so zu gestalten, dass sie auf der

gleichen Abstraktionsebene arbeiten: wir arbeiten z. B. mit Inodes, setzen aber keine Inode-Felder – das soll in den verwendeten Funktionen stattfinden.

57 ⟨*Programmfunktionen* 43c⟩+≡ (42) ◁56b 58b▷

```
void insert_file(void *filesystem, char *fname)
{
    off_t fsize = 0;
    int   fd = open_file(fname, &fsize);
    if (fd < 0) {
        return;
    }

    ino_t inodeno = allocate_inode(filesystem);
    if (inodeno <= ROOT_INO) {
        return;
    }

    inode *ino = get_inode(filesystem, inodeno);

    int blocks = fill_inode_blocks(filesystem, ino, fsize);
    if (blocks < 0) {
        return;
    }

    int blockno;
    for (blockno = 0; blockno < blocks; blockno++) {
        copy_datablock(fd, blockno, filesystem, ino->block[blockno]);
    }
    close(fd);

    insert_dir_entry(filesystem, inodeno, fname);
    if (options.verbose) {
        printf("Inserted file %s with inode %lu...\n", fname, inodeno);
    }
}
```

Mit dieser Programmiertechnik haben wir auch die logische Struktur und die Schritte der Funktion `insert_file` angegeben: nachdem sie die Datei geöffnet hat, allokiert sie einen neuen Inode mit der Funktion `allocate_inode`. Wir erwarten von dieser Funktion, dass sie eine neue Inodenummer in der Inode-Bitmap findet, dass sie den gefundenen Inode in der Inodetabelle initialisiert und dass sie am Ende die neue Inodenummer zurückgibt (siehe Abschnitt 4.3.2.3, Seite 62). Die Initialisierung des neuen Inodes sollte keine Blocknummer setzen, denn das machen wir später im Abschnitt 4.3.3.2 (Seite 63). Danach laden wir den neuen Inode aus dem Dateisystem mit der Funktion `get_inode`, die einen Inode-Pointer zurückgibt. Wir erwarten von diesem Pointer, dass er innerhalb des Dateisystems zeigt – somit können wir gleich das Dateisystem ändern, wenn wir die Inode-Felder verändern. Als nächstes, verwenden wir die Dateigröße um die Liste der Blocknummer in dem Inode zu setzen. Das ist Aufgabe einer Funktion namens `fill_inode_blocks`, die uns die Anzahl der allokierten Blöcke zurückgeben soll.

Nachdem der neue Inode wie beschrieben vollständig initialisiert wurde, iterieren wir über alle seine Blocknummern und kopieren jeweils ein Block aus der Datei in den entsprechenden Block in

das Dateisystem. Dafür verwenden wir die noch nicht implementierte Funktion `copy_datablock`. Als letztes setzen wir einen Eintrag auf die neue Datei im root-Verzeichniss.

Die verwendeten Funktionen werden in den folgenden Abschnitten vorgestellt.

4.3.1 Funktionen für Diskregionen

Bevor wir andere Funktionen implementieren, möchten wir ein paar Hilfsfunktionen schreiben, die unsere Arbeit später einfacher machen und die zugleich den syntaktischen Vorteil von „memory mapping“ zeigen.

4.3.1.1 Zugriff auf den Superblock

Eine erste Funktion nimmt einen Pointer auf das Dateisystem und gibt einen Pointer auf den Superblock des Dateisystems zurück.

58a ⟨*Funktionsdeklarationen* 43a⟩+≡ (42) ◁56a 58c▷
```
static superblock* get_superblock(void *filesystem);
```

Die Implementierung beschränkt sich auf einfache Pointer-Arithmetik.

58b ⟨*Programmfunktionen* 43c⟩+≡ (42) ◁57 58d▷
```
superblock* get_superblock(void *filesystem)
{
    return
    (superblock *)
    (filesystem + (SUPER_BLOCK * BLOCK_SIZE));
}
```

Die verwendeten Makros und Typen sind in der Headerdatei `ulixfs.h` Definiert.

Die nächste Funktion gibt den Superblock aus:

58c ⟨*Funktionsdeklarationen* 43a⟩+≡ (42) ◁58a 59a▷
```
static void print_superblock(superblock *super);
```

58d ⟨*Programmfunktionen* 43c⟩+≡ (42) ◁58b 59b▷
```
void print_superblock(superblock *super)
{
    printf("Number of inodes:             %lu\n", super->inodes);
    printf("Start blocks of disk regions:  \n");
    printf("Inode bitmap:                 %lu\n", super->imap);
    printf("Block bitmap:                 %lu\n", super->bmap);
    printf("Inode table:                  %lu\n", super->itable);
    printf("Data region:                  %lu\n", super->data);
    printf("Total number of blocks:       %lu\n", super->nblocks);
}
```

4.3.1.2 Bitmaps setzen

Zwei weitere Funtionen werden wir in den folgenden Abschnitten benutzen. Die erste setzt ein Bit in der Inode-Bitmap, die zweite in der Block-Bitmap. Das jeweilige Bit wird auf 1 gesetzt. Ihre Signaturen sind sehr ähnlich:

59a ⟨*Funktionsdeklarationen* 43a⟩+≡ (42) ◁58c 60a▷

```
void set_imap_bit(void *filesystem, ino_t inodeno);
void set_bmap_bit(void *filesystem, block_t blockno);
```

Hier ist die erste Funktion, die das Bit mit Index `inodeno` in der Inode-Bitmap setzt. Sie geht davon aus, dass die zu markierende Inodenummer `inodeno` die maximale Inodenummer nicht überschreitet.

59b ⟨*Programmfunktionen* 43c⟩+≡ (42) ◁58d 59c▷

```
void set_imap_bit(void *filesystem, ino_t inodeno)
{
    superblock *super = get_superblock(filesystem);
    int imap_block = super->imap;
    char *imap = (char *) (filesystem + (imap_block * BLOCK_SIZE));

    int byteno = inodeno / 8;
    int bitno  = inodeno % 8;
    imap[byteno] |= (0x80 >> bitno);
}
```

Sie bedarft vielleicht eine Erklärung. Zuerst wird der Superblock des Dateisystems eingelesen und von dort die Blocknummer der Inode-Bitmap gelesen (Variable `imap_block`). Danach wird durch Pointer-Arithmetik der Pointer `imap` so eingerichtet, dass er auf den Anfang der Inode-Bitmap zeigt. Die Variable `byteno` ist der Index desjenigen Bytes, der das Bit mit Index `inodeno` enthält. Die Variable `bitno` ist der Bit-Index im Byte (von `0` bis `7`). Die letzte Zeile verschiebt nach rechts eine 1 entsprechend dem Bit-Index `bitno` und setzt das entsprechende Bit in der Inode-Bitmap.

Die zweite Funktion, die eine Blocknummer in der Block-Bitmap markiert, funktioniert wie die Funktion `set_imap_bit` mit dem Unterschied, dass sie die zu markierende Blocknummer nicht unverändert übernimmt. Eine Blocknummer i wird ja nicht auf das Bit i, sondern auf das Bit $i-n$ abgebildet, mit n die erste Blocknummer in der Datenregion. Siehe dazu auch den Abschnitt 2.2.3.2 (Seite 12), wo die Block-Bitmap vorgestellt wird. Das zweite Argument der Funktion ist also eine gültige Blocknummer und nicht ein Bit-Index.

59c ⟨*Programmfunktionen* 43c⟩+≡ (42) ◁59b 60b▷

```
void set_bmap_bit(void *filesystem, block_t blockno)
{
    superblock *super = get_superblock(filesystem);
    int bmap_block = super->bmap;
    char *bmap = (char *) (filesystem + (bmap_block * BLOCK_SIZE));

    blockno = blockno - super->data;

    int byteno = blockno / 8;
    int bitno  = blockno % 8;
```

```
    bmap[byteno] |= (0x80 >> bitno);
}
```

4.3.1.3 Zugriff auf die Inodetabelle

Die folgende Funktion gibt einen Pointer auf die Inodetabelle des angegebenen Dateisystems zurück:

60a ⟨*Funktionsdeklarationen* 43a⟩+≡ (42) ◁59a 60c▷

```
static inode* get_inodetable(void *filesystem);
```

Sie kann ebenfalls durch einfache Pointer-Arithmetik implementiert werden.

60b ⟨*Programmfunktionen* 43c⟩+≡ (42) ◁59c 61a▷

```
inode* get_inodetable(void *filesystem)
{
    superblock *super = get_superblock(filesystem);
    int blockno = super->itable;

    return
    (inode *)
    (filesystem + (blockno * BLOCK_SIZE));
}
```

4.3.2 Inode-Funktionen

In diesem Abschnitt implementieren wir diejenigen Funktionen, die mit Inodes zu tun haben und die in der Funktion `insert_file` verwendet werden. Deren Signaturen entnehmen wir aus deren Verwendung in `insert_file`:

60c ⟨*Funktionsdeklarationen* 43a⟩+≡ (42) ◁60a 61b▷

```
static ino_t  allocate_inode(void *filesystem);
static inode* get_inode(void *filesystem, ino_t ino);
```

4.3.2.1 Inode lesen

Die zweite Funktion, die zum Auslesen eines Inodes dient, ist einfacher zu implementieren, also fangen wir damit an. Sie hat zwei Parameter: ein Pointer auf das Dateisystem und eine Inodenummer. Die Funktion verwendet den Superblock des Dateisystems um die Inodetabelle zu lokalisieren und gibt einen Pointer auf den Inode mit der spezifizierten Inodenummer zurück. Die Funktion

gibt einen Null-Pointer zurück, falls die übergebene Inodenummer größer als die maximale Anzahl von Inodes ist.

61a ⟨*Programmfunktionen* 43c⟩+≡ (42) ◁60b 61c▷
```
inode* get_inode(void *filesystem, ino_t ino)
{
    superblock *super = get_superblock(filesystem);
    if (ino > (super->inodes)) {
        return NULL;
    }
    inode *inodes = get_inodetable(filesystem);

    return &inodes[ino];
}
```

4.3.2.2 Inode initialisieren

Eine andere nützliche Funktion, die schreiben möchten, ist die folgende:

61b ⟨*Funktionsdeklarationen* 43a⟩+≡ (42) ◁60c 62b▷
```
static void init_inode(inode *ino);
```

Sie initialisiert einen Inode mit Startwerten und ist analog zur Funktion `make_root`, die wir oben geschrieben haben. Die zwei Funktionen setzen unterschiedliche Werte in einem Inode und so ist es schwierig, sie in einer einzigen Funktion zu verallgemeinern. Die Funktion setzt zuerst alle Inodefelder auf Null. Dann setzt sie die Zugriffsrechte so, dass nur der Inhaber Lese- und Schreiberechte hat, sonst keine Rechte für die Gruppe oder alle anderen. Als Inhaber wird der root-Benutzer markiert. Für die Zeitstempel wird die aktuelle Zeit auf dem Testsystem verwendet.

61c ⟨*Programmfunktionen* 43c⟩+≡ (42) ◁61a 62a▷
```
void init_inode(inode *ino)
{
    // empty inode
    inode empty = { 0 };

    // nullify the inode
    *ino = empty;
    /* mode: -rw---- */
    ino->mode = S_IFREG | S_IRUSR | S_IWUSR;
    ino->uid = ROOT_UID;
    ino->gid = ROOT_GID;

    time_t t = time(NULL);
    ino->atime = t;
    ino->ctime = t;
    ino->mtime = t;
}
```

4.3.2.3 Inode allokieren

Nun kommen wir endlich zur der Funktion, die in `insert_file` aufgerufen wird, um einen neuen Inode zu allokieren: `allocate_inode`. Ihr Verlauf ist der folgende: zuerst findet die Funktion eine neue Inodenummer. Dafür sucht sie nicht die Inode-Bitmap nach einem freien Bit, sondern verwendet einen internen Zähler. Man kann sich nämlich die Suche nach freien Bits sparen, wenn man weiß, dass wir Dateien in ein neues Dateisystem einfügen, in dem nur der root-Inode verwendet wird. Starten wir den Zähler bei `ROOT_INO` (Inodenummer des root-Inodes), so können wird davon ausgegen, dass alle folgenden Inodenummern frei sind. Der Zustand dieses Zählers bleibt zwischen Funktionsaufrufen erhalten, da er mit `static` deklariert wird.

Danach wird der neue Inode wie oben beschrieben initialisiert, die Inodenummer in der Inode-Bitmap markiert und anschließend die Inodenummer zurückgegeben.

62a ⟨*Programmfunktionen* 43c⟩+≡ (42) ◁61c 63a▷

```
ino_t allocate_inode(void *filesystem)
{
    static ino_t inode_no = ROOT_INO;
    inode_no ++;

    inode *ino = get_inode(filesystem, inode_no);
    if (! ino) {
        inode_no = inode_no - 1;
        return 0;
    }

    init_inode(ino);
    set_imap_bit(filesystem, inode_no);

    return inode_no;
}
```

4.3.3 Block-Funktionen

Zuletzt schreiben wir diejenigen Funktionen, die in `insert_file` mit Blöcken arbeiten. Die Signaturen (Argumente und Rückgabewerte) dieser Funktionen können wir aus deren Verwendung in `insert_file` ablesen („Programming by Intention“).

62b ⟨*Funktionsdeklarationen* 43a⟩+≡ (42) ◁61b 62c▷

```
static int  fill_inode_blocks(void *filesystem, inode *ino, off_t fsize);
static int  copy_datablock
            (int fd, int blockno, void *filesystem, block_t inoblock);
static void insert_dir_entry(void *filesystem, ino_t inodeno, char *fname);
```

4.3.3.1 Block allokieren

Zu Unterstützung der Block-Funktionen brauchen wir eine Hilfsfunktion, die analog zu der Funktion `allocate_inode` im Abschnitt 4.3.2.3 (Seite 62) funktioniert.

62c ⟨*Funktionsdeklarationen* 43a⟩+≡ (42) ◁62b

```
static block_t allocate_block(void *filesystem);
```

Wie die Funktion `allocate_inode`, sucht sie nicht nach einer freien Blocknummern in der Block-Bitmap, sonder nutzt die Tatsache, dass wir mit einem neuen Dateisystem zu tun haben, wo nur der erste Datenblock benutzt wird. Siehe dazu auch den Abschnitt 4.2.2.4 auf Seite 50. Sie verwendet also einen internen Zähler, der mit jedem Funktionsaufruf inkrementiert wird. Dieser Zähler (Variable `offset`) wird zum ersten Datenblock des Dateisystems hinzuaddiert, um den nächsten freien Datenblock zu berechnen. Danach prüft sie, ob die neue Blocknummer die maximale Blocknummer überschreitet und falls ja, dann gibt sie `0` zurück. Falls nicht, verwendet sie die Funktion `set_bmap_bit` (Abschnitt 4.3.1.2, Seite 59), um die neue Blocknummer in der Block-Bitmap zu markieren und gibt diese Nummer zurück.

63a ⟨*Programmfunktionen* 43c⟩+≡ (42) ◁62a 63b▷

```
block_t allocate_block(void *filesystem)
{
    static block_t offset = 0;
    offset++;

    superblock *super = get_superblock(filesystem);
    block_t block = super->data + offset;

    if (block > (super->nblocks)) {
        offset = offset - 1;
        return 0;
    }

    set_bmap_bit(filesystem, block);

    return block;
}
```

4.3.3.2 Blocknummern im Inode füllen

Die nächste Funktion wird dazu verwendet, die Blockliste eines Inodes zu füllen. Dafür wird die Dateigröße benutzt, um die Anzahl der Blöcke zu bestimmen. Für jeden benötigten Datenblock wird die Funktion `allocate_block` verwendet, um eine neue Blocknummer zu allokieren. Diese Nummer wird in der Blockliste des Inodes eingetragen. In dieser Funktion wird auch die Größe der neuen Datei im Inode eingetragen. Die Funktion gibt -1 zurück, falls sie einen Fehler entdeckt, sonst gibt sie die Anzahl der im Inode eingetragenen Blöcken zurück. Fehler sind: Datei ist zu groß oder keine neue Blocknummer konnte allokiert werden.

63b ⟨*Programmfunktionen* 43c⟩+≡ (42) ◁63a 64▷

```
int fill_inode_blocks(void *filesystem, inode *ino, off_t fsize)
{
    int blocks = fsize / BLOCK_SIZE;
    if (blocks * BLOCK_SIZE < fsize) {
        blocks++;
    }
    if (blocks * BLOCK_SIZE > MAX_FILE_SIZE) {
        return -1;
    }
```

```
    superblock *super = get_superblock(filesystem);
    int i;
    block_t block = 0;
    for (i = 0; i < blocks; i++) {
        block = allocate_block(filesystem);
        if (block < (super->data)) {
            return -1;
        }
        ino->block[i] = block;
    }

    ino->fsize = fsize;
    return blocks;
}
```

4.3.3.3 Datenblock kopieren

Die nächste Funktion dürfte eine der wichtigsten sein, auch wenn sie kurz ist. Sie kopiert einen Datenblock aus einer geöffneten Eingabedatei in das Dateisystem. Die Funktion hat vier Parameter:

`fd` Der Dateideskriptor der geöffneten Datei. Aus dieser Datei sollen wir einen Datenblock lesen.

`blockno` Der wievielte Block sollen wir aus der Datei lesen.

`filesystem` Pointer auf das Dateisystem. Hinter diesem Pointer steckt die Ausgabedatei.

`fsblock` Blocknummer auf dem Dateisystem, wo die Daten hinkopiert werden sollen.

Die Funktion gibt die Anzahl von kopierten Bytes zurück oder -1 falls keine Bytes kopiert werden konnten.

64 ⟨*Programmfunktionen* 43c⟩+≡ (42) ◁63b 65▷

```
int copy_datablock
(int fd, int blockno, void *filesystem, block_t fsblock)
{
    if (lseek(fd, blockno * BLOCK_SIZE, SEEK_SET) < 0) {
        return -1;
    }

    char *data_block = filesystem + (fsblock * BLOCK_SIZE);

    return read(fd, data_block, BLOCK_SIZE);
}
```

Die Funktion tut nichts anderes als den Lesezeiger der Eingabedatei auf den spezifizierten Block zu setzen (`lseek` Aufruf), den Zeiger `data_block` so einzurichten, dass er auf den gewünschten Block im Dateisystem zeigt und dann direkt aus der Datei in das Dateisystem zu lesen. Obwohl sie kurz ist, ist diese Funktion der Kern des zweiten Teils von `main`. Hier werden die Dateien auf

unserem Linux-System auf das gerade neu erstellte ULIXFS-Dateisystem kopiert. Alles was wir bisher geschrieben haben ist eine Vorbereitung für diesen kleinen Schritt.

4.3.3.4 Verzeichniseintrag einfügen

Die letzte Funktion, die wir für das Programm `mkfs.ulixfs` implementieren, ist diejenige, die einen Verzeichniseintrag in das root-Verzeichnis hinzufügt. Sie wird in `insert_file` aufgerufen (Seite 57), nachdem der Inhalt einer Datei mit `copy_datablock` (Seite 64) in das Dateisystem kopiert wurde. Ihre Argumente sind ein Zeiger auf das Dateisystem, die Inodenummer der Datei, die hinzugefügt wird, und ihren Dateinamen. Ist der Dateiname eine Pfadangabe, so wird der Verzeichnisanteil des Pfades ignoriert und nur der Dateiname eingefügt. Zum Beispiel, die Datei `sys/types.h` wird als `types.h` eingefügt und das Verzeichnis `sys/` wird ignoriert.

65 ⟨*Programmfunktionen* 43c⟩+≡ (42) ◁64

```
void insert_dir_entry(void *filesystem, ino_t inodeno, char *fname)
{
    superblock *super = get_superblock(filesystem);
    inode  *root = get_inode(filesystem, ROOT_INO);

    // index of the last block entry in inode's blocklist
    block_t bidx = root->fsize / BLOCK_SIZE;

    // full block?
    if (root->fsize % BLOCK_SIZE == 0) {
        block_t nblockno = allocate_block(filesystem);
        if (nblockno < (super->data)){
            return;
        }
        root->block[bidx] = nblockno;
    }

    // zeroed out entry
    dir_entry entry = { 0 };
    entry.ino = inodeno;
    strncpy(entry.name, basename(fname), FNAME_SIZE - 1);//-1 for '\0'

    // last data block as array of dir_entry
    dir_entry *dirs = filesystem + (root->block[bidx] * BLOCK_SIZE);

    // index of the next entry after last one
    int didx = (root->fsize / DIR_ENTRY_SIZE) % (BLOCK_SIZE / DIR_ENTRY_SIZE);
    // copy entry to disk
    dirs[didx] = entry;
    // increase size of root directory
    root->fsize += DIR_ENTRY_SIZE;

    inode *ino = get_inode(filesystem, inodeno);
    // root points to this inode, so increase its nlinks
    ino->nlinks++;
}
```

Diese Funktion düfter die komplizierteste in diesem Programm sein, daher erläutern wir einige Zeilen davon. Ihre Grundidee ist einfach: im letzten verwendeten Block des root-Inodes wird eine zusätzliche Verzeichniss-Struktur angelegt, die die Inodenummer und den Dateinamen der neuen Datei beinhaltet. Die einzelnen Schritte sind:

1. Inode des root-Verzeichnisses lesen (Varieble `root`).
2. Index der letzten verwendeten Blocknummer im root-Inode berechnen (Variable `bidx`). In diesem Block müsste man einen neuen Eintrag schreiben.
3. Fall notwendig, wird eine neue Blocknummer allokiert und in den Inode eingetragen.
4. Eine leere Verzeichnis-Struktur anlegen (Variable `entry`) und dort die Inodenummer und Dateinamen der neuen Datei eintragen.
5. Einen Zeiger auf den letzten Datenblock des root-Verzeichnisses berechnen und ihn als Array von Verzeichnis-Strukturen casten, damit wir den Block per Index adressieren können (Variable `dirs`).
6. Index in diesem Array berechnen, wo der neue Eintrag hingeschrieben werden muss (Variable `didx`) und den Eintrag schreiben.
7. Größe des root-Verzeichnisses erhöhen.
8. Das Feld `nlinks` des gerade eingefügten Inodes erhöhen (das root-Verzeichnis zeigt ja auf diese Datei).

Die letze verwendete Blocknummer im `root`-Inode wird dadurch berechnet, dass die Dateigröße des root-Verzeichnisses durch die Blockgröße geteilt wird. Diese Berechnung stimmt auch dann, wenn diese Dateigröße ein vielfaches der Blockgröße ist, denn dann wird derjenige Index ausgerechnet, wo eine neue Blocknummer gespeichert wird. Ist diese Größe z. B. 640, so ergibt die Berechnung bei einer Blockgröße von 1024 den Wert `0`. Das heißt, am Index Null in der Blockliste des root-Inodes befindet sich die Blocknummer des letzten Datenblocks (siehe dazu den Abschnitt 2.3, ab Seite 13). Bei einer Dateigröße von 2048 ergibt sich der Index 2. Das ist in disem Fall der Index, wo eine neue Blocknummer geschrieben wird (die ersten 2 Blöcke, mit Indizes `0` und 1 sind voll).

Ob eine neue Blocknummer allokiert werden muss kann man daran erkennen, dass die Dateigröße von root ein Vielfaches der Blockgröße ist (die `if`-Abfrage). Das beruht auf der Tatsache, dass wir die Größe einer Verzeichnis-Struktur so ausgewählt haben, dass immer eine ganzzahlige Anzahl von solchen Strukturen in einem Block passen. Es wird also immer ein voller Block gefüllt, ohne Restbytes. Das gilt natürlich nicht mehr für normale Datenblöcke, die keine vorgesehene Struktur haben.

Ein Wort noch zur Berechnung der Variablen `didx`. Der Ausdruck

```
root->fsize / DIR_ENTRY_SIZE
```

ergibt die gesamte Anzahl der Verzeichniseinträge im root-Inode. Der Ausdruck

```
BLOCK_SIZE / DIR_ENTRY_SIZE
```

ergibt die Anzahl der Verzeichniseinträge in einem Block. Der gesamte Ausdruck ergibt also die Anzahl der Einträge in dem letzen Block und somit der Index, wo der neue Eintrag geschrieben wird.

5 Implementierung

In diesem Kapitel werden wir die Implementierung des Dateisystems ULIXFS, die wir im Kapitel 2 angefangen haben, fortfahren. Dabei werden wir die Implementierung stark modular gestalten und eine Anzahl von Modulen erstellen, die jeweils eine Gruppe von Funktionalitäten implementieren. Die folgende Tabelle gibt eine übersicht über alle Modulen, die wir in diesem Kapitel implementieren.

Modul	Dateien	Funktionen
`block`	block.h, block.c	Blöcke und Bitmaps lesen und schreiben
`inode`	inode.h, inode.c	Inodes allokieren, lesen, schreiben
`filetab`	filetab.h, filetab.c	Dateitabelle und root-Verzeichniss
`mount`	mount.h, mount.c	Einhängen von Disks
`path`	path.h, path.c	Dateipfaden auflösen
`open`	open.h, open.c	Dateien öffnen und schließen
`read`	read.h, read.c	Dateien lesen und schreiben

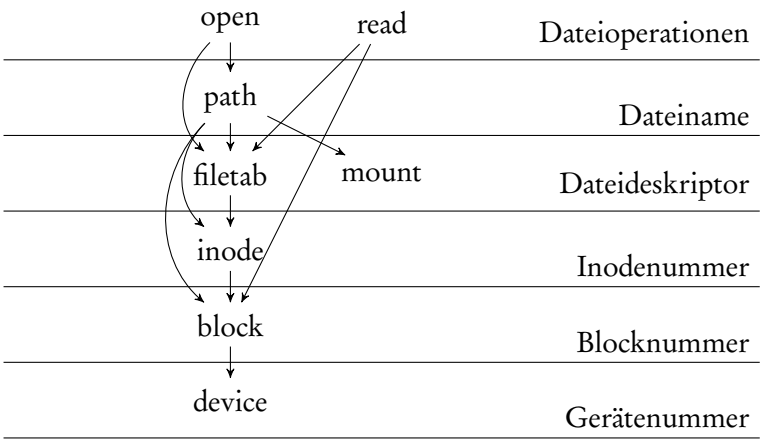

Abbildung 5.1: Hierarchische Organisation der Module, die für die Implementierung von ULIXFS geschrieben werden. Ein Pfeil $A \rightarrow B$ bedeutet, dass das Modul A das Modul B verwendet.

Die Module sind, wie in Abbildung 5.1 dargestellt, hierarchisch organisiert. Sie bauen in ihrer Gesamtheit auf die Schicht der Gerätetreiber auf, die im Kapitel 3 vorgestellt wurde (Modul `device`). Auf jeder hierarchischen Ebene werden die unteren Schichten verwendet, um neue Konzepte zu entwickeln. Auf der rechten Seite werden die Konzepten genannt, die in der jeweiligen Schicht verwendet werden. Ab dem Modul `block`, mit dem die Implementierung anfängt, sind die einzelnen Gerätetreiber nicht mehr sichtbar, sondern nur die allgemeine Geräteschnittstelle, die in nur in diesem Modul verwendet wird. Siehe auch die Abbildung 3.4 auf der Seite 23.

5.1 Blöcke und Bitmaps

Zwei Operationen sind von zentraler Bedeutung für die Implementierung von ULIXFS: einen Block lesen und einen Block schreiben. Fast alle anderen Funktionen verwenden diese Operationen, die wir in einem Modul namens `block` verlagern. In diesem Modul implementieren wir auch alle Bit-Operationen, die sich auf die Bitmaps des Dateisystems beziehen, wie z. B. ein Bit in der Inode-Bitmap setzen. Siehe Abschnitt 2.2.3, Seite 11.

Die öffentliche Schnittstelle des Moduls deklarieren wir in der folgenden Headerdatei:

68a ⟨*block.h* 68a⟩≡

```
#ifndef BLOCK_H
#define BLOCK_H

#include "ulixfs.h"

#define IMAP 1
#define BMAP 2

ssize_t read_block       (dev_t device, block_t blockno, char *buffer);
ssize_t write_block      (dev_t device, block_t blockno, char *buffer);
dev_t   read_superblock  (dev_t device, superblock *super);
int     get_bit          (dev_t device, int map, int bitno);

off_t   find_and_set_bit (dev_t device, int map);
off_t   unset_bit        (dev_t device, int map, off_t bitno);

#endif
```

Das Modul wird in der Datei `block.c` implementiert, die den folgenden Aufbau hat:

68b ⟨*block.c* 68b⟩≡

```
#include "block.h"
#include "device.h"

⟨Blockoperationen 69a⟩
⟨Superblock lesen 69b⟩
⟨Bitwert abfragen 70⟩
⟨Bitmap-Bit setzen 71⟩
⟨Bit zurücksetzen 73⟩
```

Der Minix-Quellcode enthält eine vergleichbare Datei namens „servers/fs/cache.c“. Diese Datei befindet sich in [14, S. 933].

5.1.1 Blöcke lesen und schreiben

Wir verwenden in ULIXFS zwei Blockoperationen: Lesen und Schreiben. Im Unterschied zum Dateisystem von Minix (siehe [14, S. 557]), verwendet ULIXFS der Einfachheit halber und um diese Ausarbeitung nicht noch länger zu machen, kein Caching der Blöcke. Die Daten werden aus der

Disk direkt gelesen und auf die Disk direkt geschrieben. Somit beschränkt sich die Implementierung der Blockoperationen (Lesen und Schreiben) auf eine Delegation an die Geräteschnittstelle, die wir im Kapitel 3.2 (Seite 22) beschrieben haben. Alles was diese Operatione machen ist die Blocknummer in Bytenummer zu übersetzen.

Die Operationen haben beide die selben Parameter:

- `device`: Gerätenummer der Disk, worauf sich die Operation bezieht.
- `blockno`: Blocknummer, die gelesen oder geschrieben werden soll.
- `buffer`: Puffer der Daten.

Ihr Rückgabewert ist die Anzahl der übertragenen Bytes oder -1 im Fehlerfall.

Die Funktion `read_block` implementiert die Leseoperation, `write_block` die Schreiboperation.

69a ⟨*Blockoperationen* 69a⟩≡ (68b)

```
ssize_t read_block(dev_t device, block_t blockno, char *buffer)
{
    return dev_read(device, blockno * BLOCK_SIZE, buffer, BLOCK_SIZE);
}

ssize_t write_block (dev_t device, block_t blockno, char *buffer)
{
    return dev_write(device, blockno * BLOCK_SIZE, buffer, BLOCK_SIZE);
}
```

5.1.2 Superblock lesen

Die erste und wichtigste Diskregion ist der Superblock. Wir implementieren also im Modul `block` eine Funktion, die den Superblock einer Disk einliest. Argumente sind

- `device`: Die Gerätenummer der Disk, die den Superblock enthält.
- `super`: Ein Pointer auf eine `superblock`-Struktur, deren Felder ausgefüllt werden sollen. Diese Struktur enthält nach dem Aufruf der Funktion eine Kopie des Superblocks mit Gerätenummer `device`.

Rückgabewert ist die angegebene Gerätenummer oder `NO_DEV`, der in der Headerdatei `device.h` definiert ist.

69b ⟨*Superblock lesen* 69b⟩≡ (68b)

```
dev_t read_superblock(dev_t device, superblock *super)
{
    char block[BLOCK_SIZE] = { 0 };
    int read = read_block(device, SUPER_BLOCK, block);
    int need = sizeof(superblock);
    if (read < need) {
        return NO_DEV;
    }
```

```
    *super = *((superblock *) block);

    return device;
}
```

Die vorletzte Zeile könnte vielleicht merkwürdig erscheinen. Sie benutzt die Tatsache, dass in C die Zuweisung einer Struktur wie eine reguläre Wertzuweisung funktioniert: es wird der ganze Strukturinhalt kopiert. Die Ausdruck `block` alleine ist äquivalent zu einem Pointer auf das erste Element des Arrays `block`, also äquivalent zu `&block[0]`. Die letzte Zeile „castet“ das `char`-Array zu einem Array von `superblock` und kopiert das erste Element in das Argument `super`.

5.1.3 Bitmap-Operationen

5.1.3.1 Bitwerte abfragen

Das Modul `block` bietet die Funktion `get_bit` an zum abfragen von Bitwerten der Inode-Bitmap und Block-Bitmap. Die Funktion hat die folgenden Parameter:

- `device`: Die Gerätenummer der Disk, die das Bitmap enhält.
- `map`: Integer, der die Bitmap spezifiziert. Für die Inode-Bitmap wird `IMAP` verwendet, für die Block-Bitmap, wird `BMAP` verwendet. Beide Werte sind in `block.h` definiert.
- `bitno`: Die Bitnummer. Diese Nummer entspricht einer Inodenummer oder Blocknummer. Für den Fall einer Blocknummer, wird sie so geändert, dass sie einem gültigen Bit-Index in der Block-Bitmap entspricht (siehe Abschnitt 2.2.3.2, Seite 12).

Die Funktion gibt die folgenden Werte zurück:

- 0, falls das Bit nicht gesetzt ist.
- 1, falls das Bit gesetzt ist.
- −1, im Fehlerfall. Fehler sind: Die Gerätenummer `device` entspricht nicht einer initialisierten Disk (Superblock kann nicht gelesen werden), Argument `map` ist falsch, der entsprechende Block konnte aus der Disk nicht gelesen werden, Bitnummer ist ungültig.

70 ⟨*Bitwert abfragen* 70⟩≡ (68b)

```
int get_bit(dev_t device, int map, int bitno)
{
    superblock super = { 0 };
    if (read_superblock(device, &super) == NO_DEV) {
        return -1;
    }

    block_t start_block = 0;
    int map_blocks = 0;
    switch (map) {
        case IMAP:
            start_block = super.imap;
            map_blocks = super.bmap - super.imap;
            break;
```

```
        case BMAP:
            start_block = super.bmap;
            map_blocks = super.itable - super.bmap;
            bitno = bitno - super.data;
            break;
        default:
            return -1;
    }

    if (bitno < 0 || bitno > (map_blocks * BITS_PER_BLOCK)) {
        return -1;
    }

    int block_offset = bitno / (BITS_PER_BLOCK);

    char block[BLOCK_SIZE] = { 0 };
    if (read_block(device, start_block + block_offset, block) < BLOCK_SIZE) {
        return -1;
    }

    // byte number in block
    int byteno = (bitno / 8) % BLOCK_SIZE;
    // bit number in byte
    bitno = bitno % 8;

    return (block[byteno] & (0x80 >> bitno))? 1 : 0;
}
```

5.1.3.2 Bit setzen

Die nächste Funktion `find_and_set_bit` wird dazu verwendet, ein Bit in einer Bitmap zu finden und zu setzen. Sie hat die folgenden Parameter:

- `device`: Die Gerätenummer der Disk, worauf das Bitmap liegt.
- `map`: Integer, der die Bitmap angibt: `IMAP` für die Inode-Bitmap und `BMAP` für die Block-Bitmap.

Rückgabewert ist entweder die Bitnummer die gesetzt wurde, oder `0`, falls ein Fehler aufgetreten ist. Diese Bitnummer entspricht einer Inodenummer, falls `map` den Wert `IMAP` hat, oder eine Blocknummer, falls `map` den Wert `BMAP` hat.

71 ⟨*Bitmap-Bit setzen 71*⟩≡ (68b)

```
off_t find_and_set_bit(dev_t device, int map)
{
    superblock super = { 0 };
    if (read_superblock(device, &super) == NO_DEV) {
        return 0;
    }

    block_t blockno = 0;
    int map_blocks = 0;
```

```
    switch (map) {
        case IMAP:
            blockno = super.imap;
            map_blocks = super.bmap - super.imap;
            break;
        case BMAP:
            blockno = super.bmap;
            map_blocks = super.itable - super.bmap;
            break;
        default:
            return 0;
    }

    block_t limit = blockno + map_blocks;
    block_t block_offset = 0;

    char          block[BLOCK_SIZE] = { 0 };
    int           byteno = 0;
    unsigned char byte   = 0;
    off_t         bitno  = 0;
    while (blockno < limit) {
        read_block(device, blockno, block);
        // check every byte in block
        for (byteno = 0; byteno < BLOCK_SIZE; byteno++) {
            byte = block[byteno];
            if (byte == 0xFF)
                continue;
            // we have found a byte with at least one 0
            // search every bit
            while (byte & (0x80 >> bitno))
                bitno++;

            block[byteno] = byte | (0x80 >> bitno);
            write_block(device, blockno, block);

            bitno = (block_offset * BITS_PER_BLOCK) + ((byteno * 8) + bitno);
            if (map == BMAP) {
                bitno += super.data;
            }

            return bitno;
        }

        blockno++;
        block_offset++;
    }

    return 0;
}
```

Die Suche nach einem freien Bit und das Setzen dieses Bits wurden in einer einzigen Funktion aufgenommen, um die Anzahl der Leseoperationen niedriger zu halten. Würde diese Implemen-

tierung von ULIXFS Caching verwenden, so würden wir die Suche vom Setzen trennen.

5.1.3.3 Bit zurücksetzen

Eine andere Bit-Operation auf die Bitmaps ist das zurücksetzen (freigeben) eines Bits in einer Bitmap. Hierzu implementieren wir die Funktion `unset_bit`. Ihre Parameter sind die folgenden:

- `device`: Die Gerätenummer der Disk, wo sich die Bitmap befindet.
- `map`: Integer, der die Bitmap spezifiziert (`IMAP` oder `BMAP`).
- `bitno`: Bitnummer, die freigegeben werden soll. Die entspricht einer Inodenummer oder einer Blocknummer.

Rückgabewert ist die Bitnummer `bitno`, falls das Bit zurückgesetzt wurde, oder `0`, falls ein Fehler aufgetreten ist (Null ist als Inodenummer oder Datenblocknummer nicht gültig).

73 ⟨*Bit zurücksetzen 73*⟩≡ (68b)

```
off_t unset_bit(dev_t device, int map, off_t bitno)
{
    superblock super = { 0 };
    if (read_superblock(device, &super) == NO_DEV) {
        return 0;
    }

    block_t blockno = 0;
    off_t bit = bitno;
    switch (map) {
        case IMAP:
            if (bit <= ROOT_INO || bit > super.inodes) {
                return 0;
            }
            blockno = super.imap;
            break;
        case BMAP:
            bit = bit - super.data;
            if (bit < super.data || bit > super.nblocks) {
                return 0;
            }
            blockno = super.bmap;
            break;
        default:
            return 0;

    }

    // block we must read
    blockno =  blockno + (bit / BITS_PER_BLOCK);
    // byte number inside block
    int byteno  = (bit / 8) % BLOCK_SIZE;

    char block[BLOCK_SIZE] = { 0 };
    if (read_block(device, blockno, block) < BLOCK_SIZE) {
        return 0;
```

```
    }
    block[byteno] &= ~(0x80 >> (bit % 8));
    if (write_block(device, blockno, block) < BLOCK_SIZE) {
        return 0;
    }

    return bitno;
}
```

5.2 Inodes

Das Modul `inode` enthält Funktionen, die zum lesen, schreiben, allokieren und löschen eines Inodes verwendet werden. Das Modul hat die folgende öffentliche Schnittstelle:

74a ⟨*inode.h* 74a⟩≡

```
#ifndef INODE_H
#define INODE_H

#include "ulixfs.h"

ino_t allocate_inode (dev_t device);
ino_t delete_inode   (dev_t device, ino_t inode);

dev_t get_inode     (dev_t device, ino_t inodeno, inode *ino);
dev_t store_inode   (dev_t device, ino_t inodeno, inode *ino);

#endif
```

Bemerkung: die Funktionsnamen `read_inode` und `write_inode` werden zum Zeitpunkt der Erstellung dieser Arbei in dem ULIX-Code benutzt und so müssen wir alternative Namen finden.

Das Modul `inode` wird in der Datei `inode.c` implementiert, die den folgenden Aufbau hat:

74b ⟨*inode.c* 74b⟩≡

```
#include "inode.h"
#include "device.h"
#include "block.h"

⟨Konstanten 75c⟩

⟨Inode allokieren 75a⟩
⟨Inode löschen 75b⟩
⟨Inode lesen und schreiben 76a⟩
```

5.2.1 Inodes allokieren und löschen

Einen Inode zu allokieren bedeutet in ULIXFS ein Bit in der Inode-Bitmap zu finden und als belegt zu markieren. Inode löschen heißt dieses Bit auf Null zu setzen. Dafür kann man die Funktionen aus dem Modul `block` verwenden.

Die Funktion `allocate_inode` allokiert eine neue Inodenummer in der Inode-Bitmap und gibt sie zurück. Falls die Allokation fehlgeschlagen ist, gibt sie `0` zurück. Ihr Parameter ist die Gerätenummer der Disk, die den Inode enthält.

75a ⟨*Inode allokieren* 75a⟩≡ (74b)

```
ino_t allocate_inode(dev_t device)
{
    int i = find_and_set_bit(device, IMAP);
    if (i <= ROOT_INO) {
        return 0;
    } else {
        return i;
    }
}
```

Die Funktion `delete_inode` tut das Gegenteil: Sie gibt das entsprechende Bit in der Inode-Bitmap frei und gibt die Inodenummer zurück. Falls die Operation fehlschlägt, gibt sie `0` zurück. Parameter sind die Gerätenummer der Disk, wo der Inode freigegeben wird und die Inodenummer.

75b ⟨*Inode löschen* 75b⟩≡ (74b)

```
ino_t delete_inode(dev_t device, ino_t inode)
{
    int i = unset_bit(device, IMAP, inode);
    if (i <= ROOT_INO) {
        return 0;
    } else {
        return inode;
    }
}
```

5.2.2 Inodes lesen und schreiben

Um das Lesen und Schreiben von Inodes zu implementieren, verwenden wird die gleiche Technik wie im Abschnitt 3.4.6.4 (Seite 36): Wir schreiben zuerst eine generische Funktion, die beide Operationen implementiert und anhand eines zusätzlichen Parameters zum richtigen Zeitpunkt entscheidet, welche Operation auszuführen ist. Wir legen daher die folgenden Werte fest, um die zwei Operationen zu kennzeichnen:

75c ⟨*Konstanten* 75c⟩≡ (74b)

```
#define INODE_READ  1
#define INODE_WRITE 2
```

Die Funktion `get_store_inode` hat die folgenden Parameter:

- `device`: Die Gerätenummer der Disk, die den Inode enthält.
- `ino`: Inodenummer des Inodes.
- `i`: Pointer zu einer `inode`-Struktur. Hier wird der Inode gelesen bzw. dieser Inode wird geschrieben.
- `op`: Kennzeichnung der Operation: Schreiben oder Lesen. Muss einer der oben genannten Werte sein.

Die Funktion gibt die Gerätenummer `device` oder `NO_DEV` zurück, falls ein Fehler auftritt. Sie hat den folgenden Aufbau:

76a ⟨*Inode lesen und schreiben* 76a⟩≡ (74b) 77c▷

```
static
dev_t get_store_inode(dev_t device, ino_t ino, inode *i, int op)
{
    ⟨Inodenummer überprüfen 76b⟩
    ⟨Berechne die Position in der Inodetabelle 76c⟩
    ⟨Lade den Block, wo sich der Inode befindet 77a⟩
    ⟨Übertrage die Inodedaten 77b⟩

    return device;
}
```

Bevor wir den Inode lesen oder schreiben, überprüfen wir die Parameter und geben `NO_DEV` (kein Gerät) zurück, falls wir einen Fehler finden. Fehler sind: Die Inodenummer ist zu groß oder der Inode ist in der Inode-Bitmap nicht als belegt markiert. Dafür müssen wir zuerst den Superblock laden.

76b ⟨*Inodenummer überprüfen* 76b⟩≡ (76a)

```
superblock super = { 0 };
if (read_superblock(device, &super) == NO_DEV) {
    return NO_DEV;
}
if (ino > super.inodes) {
    return NO_DEV;
}
if (get_bit(device, IMAP, ino) != 1) {
    return NO_DEV;
}
```

Als nächstes berechnen wir die Position des Inodes in der Inodetabelle, d.h. im wievielten Block der Inodetabelle befindet sich der Inode (Variable `offset`) und, in diesem Block, an welchem Index (Variable `index`). Die Variable `ino` ist die Inodenummer.

76c ⟨*Berechne die Position in der Inodetabelle* 76c⟩≡ (76a)

```
block_t offset = ino / INODES_PER_BLOCK;
int     index  = ino % INODES_PER_BLOCK;
```

Nun können wir den ausgerechneten Block laden. Falls wir dabei weniger als ein ganzer Block laden konnten, geben wir `NO_DEV` als Fehlersignal zurück.

77a ⟨*Lade den Block, wo sich der Inode befindet* 77a⟩≡ (76a)
```
char block[BLOCK_SIZE] = { 0 };
if (read_block(device, super.itable + offset, block) < BLOCK_SIZE) {
    return NO_DEV;
}
```

Als letztes entscheiden wir anhand des Funktionparameters op die Schreibrichtung.

77b ⟨*Übertrage die Inodedaten* 77b⟩≡ (76a)
```
if (op == INODE_READ) {
    *i = ((inode *) block)[index];
} else {
    ((inode *) block)[index] = *i;
    write_block(device, super.itable + offset, block);
}
```

Nun kann man die Lese- und Schreiboperationen als Delegation an die oben implementierte Funktion angeben. Beide Funktionen haben dieselben Parameter und Rückgabewerte wie die generische Funktion `get_store_inode`, außer dem Parameter op.

77c ⟨*Inode lesen und schreiben* 76a⟩+≡ (74b) ◁76a
```
dev_t get_inode(dev_t device, ino_t ino, inode *i)
{
    return get_store_inode(device, ino, i, INODE_READ);
}

dev_t store_inode(dev_t device, ino_t ino, inode *i)
{
    return get_store_inode(device, ino, i, INODE_WRITE);
}
```

5.3 Dateitabelle

In diesem Abschnitt kommen wir zu einer der zentralsten Themen eines Dateisystems: die Datei. Im Abschnitt 2.1.3 (Seite 8) haben wir eine Datei als ein Aggregat von zwei Komponenten vorgestellt: der Inode, der sich in der Inodetabelle befindet, und die Daten, die sich in der Datenregion der Disk befinden. Unsere Implementierung bezieht sich hauptsächlich auf die Inodes. Dabei sind alle bisherigen Funktionen geräteabhängig („device aware“): Sie haben alle als Parameter eine Gerätenummer, auch wenn sie diese Nummer nicht weiter untersuchen, sondern an das Modul `device` weitergeben. In diesem Abschnitt machen wir einen weiteren Abstraktionsschritt und implementieren die Grundlage für geräteunabhängige Dateistrukturen: die Verwendung von *Dateideskriptoren*. Es handelt sich dabei um globale Dateideskriptoren, die nicht prozessbezogen sind. Die weitere Entwicklung von ULIXFS müsste die lokalen Dateideskriptoren eines Prozesses zu den globalen Deskriptoren zuordnen.

Für die Implementierung schreiben wir ein Modul namens `filetab` (von „file table“), das sich auf zwei Dateien erstreckt: `filetab.h` und `filetab.c`. Die Headerdatei hat den folgenden Aufbau:

78a ⟨*filetab.h* 78a⟩≡

```
#ifndef FILETAB_H
#define FILETAB_H

#include "inode.h"
#include "device.h" // NO_DEV

⟨Öffentliche Konstanten 79a⟩

⟨Datenstruktur der Dateitabelle 79b⟩

int   get_root    (file_entry *root);
dev_t change_root (dev_t device, int mode);
int   load_inode  (dev_t device, ino_t inodeno, int mode);
int   unload_inode(int fd);

int   read_file_entry  (int fd, file_entry *entry);
int   write_file_entry (int fd, file_entry *entry);
off_t lseek_file_entry (int fd, off_t offset, int whence);
int   save_file_entry  (int fd);

#endif
```

Die C-Datei hat den folgenden Aufbau:

78b ⟨*filetab.c* 78b⟩≡

```
#include "filetab.h"

⟨Private Konstanten 80b⟩
⟨Dateitabelle 80a⟩

⟨Freien Tabelleneintrag finden 81a⟩
⟨Inode laden 81b⟩
⟨Eintrag löschen 82a⟩
⟨Eintrag lesen oder schreiben 83a⟩
⟨Eintrag lesen 83b⟩
⟨Eintrag schreiben 83c⟩
⟨Zugriffsposition setzen 84b⟩
⟨Dateieintrag sichern 85⟩
⟨root-Inode lesen 86a⟩
⟨root-Inode setzen 86b⟩
```

5.3.1 Struktur der Dateitabelle

Um die Abstraktion von den Inodes und Gerätenummern zu realisieren, führen wir einen neuen Konzept ein: der *Dateideskriptor*. Der Dateideskriptor ist eine natürliche Zahl, die eine Datenstruktur in einer internen *Dateitabelle* indiziert. Eine geöffnete Datei wird unabhängig von Inodes und Gerätenummern durch diese Zahl referenziert. Die referenzierte Datenstruktur beinhaltet die

Inodenummer und die Gerätenummer, die notwendig sind, um die Datei lesen und schreiben zu können.

Wie für jede Tabelle, müssen wir eine feste Größe definieren. Für die Festlegung dieser Größe orientieren wir uns wieder an Minix, das eine ähnliche aber kompliziertere Tabelle verwendet: die „filp table“ (siehe [14, S. 561ff, 578]). Die Minix-Quelldatei `servers/fs/const.h` setzt diese Größe auf 128 (siehe den Wert von `NR_FILPS` in [14, S. 919]). Wir übernehmen diese Größe für unsere (einfachere) Tabelle:

79a ⟨*Öffentliche Konstanten* 79a⟩≡ (78a) 84a▷
```
#define MAX_FILES 128
```

Die von einem Dateideskriptor referenzierte Datenstruktur deklarieren wir ebenfalls öffentlich in der Headerdatei:

79b ⟨*Datenstruktur der Dateitabelle* 79b⟩≡ (78a)
```
typedef
struct file_entry
{
    inode inode;

    ino_t inodeno;
    dev_t device;
    off_t fpos;
    int   mode;
} file_entry;
```

Wir besprechen nun die einzelnen Felder einer `file_entry` Struktur.

`inode` Hier steht eine Kopie des Inodes worauf zugeriffen wird. Die Kopie verwenden wird als Cache für Inodes, damit wir sie nicht jedes mal von der Disk lesen müssen.

`inodeno` Die Inodenummer des Inodes. Falls dieses Feld den Wert `0` hat, ist der Tabelleneintrag frei (kein Inode hat die Nummer `0`).

`device` Gerätenummer der Disk, worauf sich der Inode befindet.

`fpos` Lese- oder Schreibposition in der Datei bzw. Index des nächsten Bytes, das aus der Datei gelesen oder geschrieben werden kann. Dieses Feld wird mit jedem Schreiben oder Lesen der Datei verändert und nimmt Werte zwischen `0` (erstes Byte) und `inode.fsize` (nach dem letzten Byte). Der Wert in diesem Feld ist unabhängig von der Blockstruktur der Datenregion. Er gibt den nächsten Index an, als würde die Datei aus eine kontinuierlichen Reihe von Bytes bestehen.

`mode` Dateimodus. Hier steht einer der Werte, die wir weiter unten definieren.

Das Feld `mode` in der Tabelle beschreibt den Operationsmodus, wie die Datei geöffnet wurde: zum lesen, zum schreiben oder beide. Der POSIX Standard spezifiziert für die Angabe dieser Operationsmodi bestimmte Werte, die in der Datei `fcntl.h` definiert werden sollen. Siehe den Anhang A.4 (Seite 112), wo diese Datei vorgestellt ist.

Die Dateitabelle definieren wir als `static` in `filetab.c`, damit sie von Manipulationen von außen geschütz ist. Sie besteht aus einem Array von `file_entry` Strukturen, wobei jedes Strukturelement für eine Spalte der Tabelle steht.

80a ⟨*Dateitabelle* 80a⟩≡ (78b)
```
static file_entry filetab [MAX_FILES];
```

`inode`	`inodeno`	`device`	`fpos`	`mode`
Inode	Inodenummer	Gerätenummer	Leseposition	Dateimodus
{...}	1	256	192	0
{...}	2	256	0	0
{...}	0	0	0	0
{...}	4	256	0	1
{...}	1	257	320	0

Abbildung 5.2: Dateitabelle. Jede Zeile entspricht einer geöffneten Datei. Die Zeilen werden mit dem Dateideskriptor indiziert. Der Dateideskriptor 2 ist frei, da das Feld `inode` in der dritten Zeile auf Null gesetzt ist.

Die Abbildung 5.2 zeigt ein Beispiel für eine solche Dateitabelle. Wie man sehen kann, sind in der Tabelle die Inodenummern nicht eindeutig, da mehrere Dateisysteme verwendet werden können, die alle separate Inodetabellen haben.

5.3.1.1 Reservierte Dateideskriptoren

Bestimmte Indizes in der Tabelle wollen wir für bestimmte Zwecke reservieren. Den Index `0` verwenden wir für den Root-Verzeichnis des ganzen Dateibaums. Auf diesen Index kann durch keine Funktion zugeriffen werden, außer den zu diesem Zweck geschriebenen Funktionen `get_root` und `change_root`.

80b ⟨*Private Konstanten* 80b⟩≡ (78b) 82b▹
```
#define ROOT_FILE 0
```

Für die Manipulation der Dateitabelle stellen wir ein paar Operationen zur Verfügung, die in der Headerdatei `filetab.h` öffentlich gemacht und im Folgenden vorstellen werden.

5.3.2 Freien Eintrag finden

Die erste Operation auf die Dateitabelle ist den ersten freien Eintrag zu finden. Diese Operation implementieren wir in einer Hilfsfunktion, die wir „privat“ zu diesem Modul halten. Ein Eintrag ist frei, wenn das entsprechende Feld `inodeno` den Wert Null hat. Die Suche ist eine einfache lineare Suche. Die Funktion hat keine Parameter. Sie gibt den ersten freien Index zurück oder -1, falls die Tabelle voll ist. Die Suche ignoriert den Eintrag für das root-Verzeichnis, um diesen

Eintrag zu schützen. Um auf das root-Verzeichnis zuzugreifen, muss man die Funktion `get_root` und `change_root` verwenden.

81a ⟨*Freien Tabelleneintrag finden* 81a⟩≡ (78b)

```
static int find_free(void)
{
    int i;
    for (i = ROOT_FILE + 1; i < MAX_FILES; i++) {
        if (filetab[i].inodeno == 0)
            return i;
    }

    return -1;
}
```

5.3.3 Inode laden und Dateideskriptor erzeugen

Die Nächste Funktion macht die Überbrückung zwischen Gerätenummern und Inodenummern auf der einen Seite und Dateideskriptoren auf der anderen Seite. Sie wird dazu verwendet, eine Kopie eines Inodes in die Dateitabelle zu laden. Der Index des Tabelleneintrag ist der zurückgegebene Dateideskriptor.

Die Parameter der Funktion sind die folgenden:

- `device`: Die Gerätenummer der Disk, worauf sich der Inode befindet.
- `inodeno`: Inodenummer des Inodes.
- `mode`: Dateimodus, muss `O_RDONLY` (nur lesen), `O_WRONLY` (nur schreiben) oder `O_RDWR` (lesen und schreiben) sein. Diese Werte sind in der Datei `fcntl.h` definert (Anhang A.4, Seite 112).

Der Rückgabewert der Funktion ist der Dateideskriptor. Mit diesem Deskriptor kann weiter auf die Datei zugeriffen werden. Ist dieser Deskriptor -1, so konnte der Inode nicht geladen werden.

Die Funktion verwendet `find_free` um einen neuen Index in der Dateitabelle zu finden. Falls sie einen freien Index findet, ruft sie die Funktion `get_inode` aus dem Modul `inode` auf, um den Inode zu laden.

81b ⟨*Inode laden* 81b⟩≡ (78b)

```
int load_inode(dev_t device, ino_t inodeno, int mode)
{
    int fd = find_free();
    if (fd <= ROOT_FILE) {
        return -1;
    }

    if (get_inode(device, inodeno, &filetab[fd].inode) == NO_DEV) {
        return -1;
    }
    filetab[fd].inodeno = inodeno;
    filetab[fd].device  = device;
    filetab[fd].fpos    = 0;
    filetab[fd].mode    = mode;
```

```
    return fd;
}
```

5.3.4 Eintrag löschen

Nachdem man einen Inode in die Dateitabelle geladen hat, kann man durch die folgende Funktion `unload_inode` den Eintrag wieder löschen. Dazu wird der Funktion der Dateideskriptor übergeben, der von der Funktion `load_inode` zurückgegeben wurde. Falls der Dateideskriptor `fd` keinen gültigen Index darstellt oder falls an dem Index keine Datei geladen wurde, wird -1, sonst wird der Dateideskriptor zurückgegeben.

Die Funktion tut nicht anderes als der entsprechende Tabelleneintrag mit Nullbytes zu füllen.

82a ⟨*Eintrag löschen* 82a⟩≡ (78b)

```
int unload_inode(int fd) {
    if (fd <= ROOT_FILE || fd >= MAX_FILES) {
        return -1;
    }

    if (filetab[fd].inodeno == 0) {
        return -1;
    }

    file_entry empty = {{0}, 0, 0, 0, 0};
    filetab[fd] = empty;

    return fd;
}
```

5.3.5 Eintrag lesen und schreiben

Die nächste Operationen auf die Dateitabelle ist einen Eintrag zu lesen oder zu schreiben (modifizieren). Da diese beide Operationen vom Verlauf her sehr ähnlich sind, schreiben wir eine allgemeine Funktion mit denselben Parametern und zusätzlich einen Operationsparameter, der die Operation spezifiziert. Zuerst definieren wir in `filetab.c` die Werte für diese Operationen:

82b ⟨*Private Konstanten* 80b⟩+≡ (78b) ◁80b

```
#define READ_ENTRY   1   // read entry
#define WRITE_ENTRY  2   // write entry
```

Die Parameter dieser Funktion sind die folgenden:

- `fd`: Dateideskriptor, der von der Funktion `load_inode` zurückgegeben wurde.
- `entry`: Pointer zu einer `file_entry` Struktur. Hier wird der Tabelleneintrag kopiert bzw. der Inhalt dieser Struktur wird in die Tabelle kopiert.
- op: `READ_ENTRY`, falls der Tabelleneintrag in `entry` gelesen wird, oder `WRITE_ENTRY`, falls er geschrieben wird.

Rückgabewert ist `fd` oder -1, falls der Parameter `fd` kein gültiger Index in der Dateitabelle ist oder falls an dem Index keine Datei geladen wurde.

Die Funktion in sich ist recht einfach. Sie prüft zuerst, ob der Dateideskriptor `fd` einen gültigen Index ist und falls ja, entscheidet sie anhand des Parameters `op` in welche Richtung sollen Daten kopiert werden. Der Kopiervorgang wird durch einfache Strukturzuweisung realisiert.

83a ⟨*Eintrag lesen oder schreiben* 83a⟩≡ (78b)

```
static
int read_write_entry(int fd, file_entry *entry, int op)
{
    if (fd <= ROOT_FILE || fd >= MAX_FILES) {
        return -1;
    }
    if (filetab[fd].inodeno == 0) {
        return -1;
    }

    if (op == READ_ENTRY) {
        *entry = filetab[fd];
    } else if (op == WRITE_ENTRY) {
        filetab[fd] = *entry;
    } else {
        return -1;
    }

    return fd;
}
```

Um einen Eintrag aus der Dateitabelle zu lesen, verwendet man die Funktion `read_file_entry`. Sie hat dieselben Parameter wie die Funktion `read_write_entry`, außer dem Parameter `op`.

83b ⟨*Eintrag lesen* 83b⟩≡ (78b)

```
int read_file_entry(int fd, file_entry *entry)
{
    return read_write_entry(fd, entry, READ_ENTRY);
}
```

Überschreiben eines Eintrags wird genauso einfach als Delegation implementiert. Hier wird der Tabelleneintrag mit Index `fd` mit dem Inhalte der Struktur, die vom Pointer `entry` gezeigt wird, überschrieben.

83c ⟨*Eintrag schreiben* 83c⟩≡ (78b)

```
int write_file_entry(int fd, file_entry *entry)
{
    return read_write_entry(fd, entry, WRITE_ENTRY);
}
```

5.3.6 Zugriffsposition setzen

Die Zugriffsposition `fpos` in der Dateitabelle bestimmt die Anfangsposition der nächsten Lese- oder Schreiboperation. Dieser Wert kann mit der Funktion `lseek_file_entry` verändert werden. Die Funktion hat die folgenden Parameter:

- `fd`: Dateideskriptor bzw. Index des Tabelleneintrags, der verändert wird.
- `offset`: Wieviel Bytes Abstand vom Dateianfang, aktuellen Position oder Dateiende soll die neue Zugriffsposition gesetzt werden.
- `whence`: Bestimmt ob der Parameter `offset` sich auf den Dateianfang, aktuelle Position oder Dateiende bezieht. Muss einer der Werten `SEEK_SET` (Dateianfang), `SEEK_CUR` (aktuelle Position) oder `SEEK_END` (Dateiende) haben.

Die Funktion gibt entweder den Wert der neuen Zugriffsposition, oder `NO_SEEK`, falls `whence` keinen spezifizierten Wert hat oder falls die neue Leseposition außerhalb der Datei zeigen würde. Die Funktion erlaubt allerdings das Setzen der Zugriffsposition ein Bytes nach dem letzten Byte der Datei wie das folgende Beispiel zeigt:

```
lseek_file_entry(fd, 0, SEEK_END);
```

Dieses Verhalten entspricht nicht genau der POSIX-Funktion `lseek`.

Wir deklarieren zuerst die genannten Werte in `filetab.h` wie folgt:

84a ⟨*Öffentliche Konstanten* 79a⟩+≡ (78a) ◁79a

```
#define SEEK_SET       0  /* Seek from beginning of file.  */
#define SEEK_CUR       1  /* Seek from current position.   */
#define SEEK_END       2  /* Seek from end of file.        */
#define NO_SEEK  ((off_t) -1) /* Error while seeking */
```

Die Funktion besteht haupstächlich aus einem `switch`, wo abhängig von `offset`, die neue Lese- oder Schreibposition berechnet wird. Falls die neue Position größer als die Dateigröße ist, wird `NO_SEEK` zurückgegeben.

84b ⟨*Zugriffsposition setzen* 84b⟩≡ (78b)

```
off_t lseek_file_entry (int fd, off_t offset, int whence)
{
    if (fd < ROOT_FILE || fd >= MAX_FILES)
        return NO_SEEK;
    if (filetab[fd].inodeno == 0)
        return NO_SEEK;

    off_t newoffset = filetab[fd].fpos;
    off_t fsize     = filetab[fd].inode.fsize;

    switch (whence) {
        case SEEK_SET:
            newoffset = offset;
            break;
        case SEEK_CUR:
            newoffset += offset;
            break;
        case SEEK_END:
```

```
            newoffset = fsize - offset;
            break;
        default:
            return NO_SEEK;
    }

    if (newoffset > fsize)
        return NO_SEEK;

    filetab[fd].fpos = newoffset;
    return newoffset;
}
```

5.3.7 Dateieintrag sichern

Die bisher vorgestellten Operationen auf einen Dateieintrag bewirken sich nur auf den Inhalt der Dateitabelle. Wird ein Inode modifiziert, so wird er nur in der Tabelle modifiziert. Um den Inode auf die Disk zu speichern, wird die Funktion `save_file_entry` verwendet. Ihr Parameter ist ein Dateideskriptor `fd`. Sie gibt entweder `fd` zurück oder -1, falls der Inode nicht gespeichert werden konnte.

85 ⟨*Dateieintrag sichern* 85⟩≡ (78b)

```
int save_file_entry(int fd)
{
    if (fd < ROOT_FILE || fd >= MAX_FILES)
        return -1;
    if (filetab[fd].inodeno == 0)
        return -1;

    dev_t device  = filetab[fd].device;
    ino_t inodeno = filetab[fd].inodeno;
    inode *ino    = &filetab[fd].inode;

    if (store_inode(device, inodeno, ino) == NO_DEV)
        return -1;
    else
        return fd;
}
```

5.3.8 root lesen

Das root-Verzeichnis spielt in ULIXFS wie eine besondere Rolle, denn es ist das Wurzelverzeichniss aller Dateipfaden (siehe Abschnitt 2.6, Seite 18). Es liegt daher nahe, den entsprechenden Eintrag in der Dateitabelle besonders zu beschützen: Alle Funktionen, die auf diese Tabelle operieren ignorieren diesen Eintrag. Um das root-Verzeichnis doch zugänglich zu machen, schreiben wir zwei Funktionen: `get_root` und `change_root`.

Die erste Funktion wird dazu verwendet, das root-Verzeichnis zu lesen. Sie gibt keinen direkten Zugang zum root-Eintrag in der Tabelle `filetab`, sondern kopiert diesen Eintrag an die Speicheradresse, die mit dem einzigen Parameter `root` übergeben wird. Falls der root-Eintrag noch nicht gesetzt ist, gibt sie -1, sonst git sie `0` zurück.

86a ⟨*root-Inode lesen* 86a⟩≡ (78b)

```
int get_root(file_entry *root)
{
    if ((! filetab[ROOT_FILE].device) || (! filetab[ROOT_FILE].inodeno)) {
        return -1;
    } else {
        *root = filetab[ROOT_FILE];
        return 0;
    }
}
```

5.3.9 root setzen

Die zweite Funktion wird dazu verwendet, das root-Verzeichnis des Dateisystems zu setzen. Sie hat zwei Parameter:

- `device`: die Gerätenummer der Disk, die als *root-Gerät* verwendet werden soll. Auf dieser root-Disk befindet sich der root-Inode.
- `mode`: Der Operationsmodus des root-Verzeichnisses. Dieser Wert muss `O_RDONLY`, `O_WRONLY` und `O_RDWR` sein (siehe Anhang A.4, Seite 112). Wir geben dadurch dem Benutzer unserer Funktion die Möglichkeit, das root-Verzeichnis z. B. „read only“ zu setzen.

Rückgabewert der Funktion ist entweder `device` oder `NO_DEV`, falls der root-Inode nicht gelesen werden konnte.

86b ⟨*root-Inode setzen* 86b⟩≡ (78b)

```
dev_t change_root(dev_t device, int mode)
{
    if (get_inode(device, ROOT_INO, &filetab[ROOT_FILE].inode) == NO_DEV) {
        return NO_DEV;
    }

    filetab[ROOT_FILE].inodeno = ROOT_INO;
    filetab[ROOT_FILE].device  = device;
    filetab[ROOT_FILE].fpos    = 0;
    filetab[ROOT_FILE].mode    = mode;

    return device;
}
```

5.4 Einhängen

Einhängen („mounting“) ist in ULIXFS die Zuordnung eines Tuppels aus einer Inodenummer i und Gerätenummer g_1 zu einer Gerätenummer g_2.

$$(i, g_1) \mapsto g_2 \qquad g_1 \neq g_2$$

Der Tupel (i, g_1) nennen wir ein *Einhängepunkt* (mount point).

Ein Einhängepunkt muss ein Verzeichnis sein. Ist das Gerät g_2 im Einhängepunkt (i, g_1) eingehängt, so überschreibt während der Auflösung eines Dateinamens das root-Verzeichnis des Dateisystems auf der Disk g_2 den Inhalt des Inodes (i, g_1).

Im Inode i	auf Gerät g_1	wird das Gerät g_2 eingehängt
5	256	257
6	256	258
7	256	259

Abbildung 5.3: Einhängetabelle

Um alle Zuordnungen $(i, g_1) \mapsto g_2$ zu speichern, verwenden wir eine Zuordnungstabelle, auch *Einhängetabelle* genannt (mount table). Ein Beispiel für eine solche Tabelle ist in der Abbildung 5.3 dargestellt (Seite 87).

Das Modul `mount`, das wir in diesem Abschnitt implementieren, stellt Funktionen zur Verfügung, die auf die interne Einhängetabelle operieren. Seine Schnittstelle ist in der folgenden Headerdatei gegeben:

87a ⟨*mount.h* 87a⟩≡

```
#ifndef MOUNT_H
#define MOUNT_H

#include "sys/types.h"

#define MAX_MOUNT_POINTS 16

int   mount      (ino_t inode, dev_t ondevice, dev_t device);
dev_t mounted_on (ino_t inode, dev_t ondevice);
void  unmount    (ino_t inode, dev_t ondevice);
void  unmount_all(dev_t device);

#endif
```

Der Wert `MAX_MOUNT_POINTS` gibt die maximale Anzahl von Zuordnungen an. Die öffentlichen Funktionen werden in der Datei `mount.c` implementiert, die den folgenden Aufbau hat:

87b ⟨*mount.c* 87b⟩≡

```
#include "mount.h"
#include "device.h" // for NO_DEV

⟨Definition der Einhängetabelle 88a⟩
⟨Freien Eintrag finden 88b⟩
```

⟨*Einhängen* 89a⟩
⟨*Einhängepunkt abfragen* 89b⟩
⟨*Einhängepunkt freigeben* 90a⟩
⟨*Disk Aushängen* 90b⟩

Die Einhängetabelle besteht aus einem Array namens `mounttab` von anonymen Strukturen, die der Tabelle aus Abbildung 5.3 (Seite 87) entsprechen:

88a ⟨*Definition der Einhängetabelle* 88a⟩≡ (87b)

```
static
struct {
    ino_t inode;
    dev_t ondevice;
    dev_t device;
} mounttab [MAX_MOUNT_POINTS];
```

Die ersten zwei Felder beschreiben den Einhängepunkt, das letzte Feld gibt das eingehängte Gerät an.

5.4.1 Einhängen

Das Einhägen eines Gerätes besteht daraus, dass wir zuerst einen freien Eintrag in der Einhängetabelle finden und dann die entsprechenden Felder dorthin kopieren. Um einen freien Eintrag zu finden, benutzen wir die folgende Funktion. Sie läuft über alle Tabelleneinträge und gibt den Index des ersten freien Index zurück. Ein Index ist frei, wenn dort das Feld `device` (das eingehängte Gerät) `0` ist. Falls die Tabelle voll ist, gibt die Funktion -1 zurück.

88b ⟨*Freien Eintrag finden* 88b⟩≡ (87b)

```
static
int find_free(void)
{
    int i;
    for (i = 0; i < MAX_MOUNT_POINTS; i++) {
        if (! mounttab[i].device)
            return i;
    }

    return -1;
}
```

Nun kann man die Einhägefunktion selbst schreiben. Ihre Parameter sind

- `inode` Inodenummer des Einhängepunktes.
- `ondevice` Gerätenummer des Einhängepunktes.
- `device` Gerätenummer der Disk, die im Einhängepunkt eingehängt wird.

Die Funktion gibt den Index des neuen Eintrags zurück oder −1, falls die Tabelle voll ist oder falls `ondevice` und `device` denselben Wert haben.

89a ⟨*Einhängen* 89a⟩≡ (87b)

```
int mount(ino_t inode, dev_t ondevice, dev_t device)
{
    if (ondevice == device) {
        return -1;
    }
    int i = find_free();
    if (i < 0) {
        return -1;
    }

    mounttab[i].ondevice = ondevice;
    mounttab[i].inode    = inode;
    mounttab[i].device   = device;

    return i;
}
```

5.4.2 Einhängepunkt abfragen

Ein Einhängepunkt kann durch die folgende Funktion abgefragt werden. Ihre Parameter sind

- `inode` Inodenummer des Einhängepunktes.
- `ondevice` Gerätenummer des Einhängepunktes.

Falls die Funktion einen Einhängepunkt mit diesen Parametern findet, gibt sie die Gerätenummer der eingehängten Disk zurück. Sonst gibt sie `NO_DEV` zurück.

89b ⟨*Einhängepunkt abfragen* 89b⟩≡ (87b)

```
dev_t mounted_on(ino_t inode, dev_t ondevice)
{
    int i;
    for (i = 0; i < MAX_MOUNT_POINTS; i++) {
        if (mounttab[i].inode    == inode    &&
            mounttab[i].ondevice == ondevice)
            return mounttab[i].device;
    }

    return NO_DEV;
}
```

5.4.3 Aushängen

Ein bestimmter Einhängepunkt kann mit der Funktion `unmount` wieder freigegeben werden. Wir sagen, die Disk, die dort eingehängt wurde, wird ausgehängt. Nach dem Aushängen verhält sich der Einhängepunkt wie ein regulärer Inode, der seinen alten Inhalt hat. Die Funktion läuft über

alle Einträge der Einhängetabelle und setzt alle auf Null, die den Parametern entsprechen. Dadurch wird verhindert, dass ein Benutzer mehrmals einen Einhängepunkt verwendet, aber ihn nicht oft genug freigibt.

Die Parameter der Funktion geben die Komponenten eines Einhängepunktes an:

- `inode` Inodenummer des Einhängepunktes.
- `ondevice` Gerätenummer des Einhängepunktes.

90a ⟨*Einhängepunkt freigeben* 90a⟩≡ (87b)

```
void unmount(ino_t inode, dev_t ondevice)
{
    int i;
    for (i = 0; i < MAX_MOUNT_POINTS; i++) {
        if (mounttab[i].ondevice == ondevice &&
            mounttab[i].inode    == inode       )
        {
            mounttab[i].ondevice = 0;
            mounttab[i].inode    = 0;
            mounttab[i].device   = 0;
        }
    }
}
```

Möchte man nicht einen bestimmten Einhängepunkt freigeben, sondern eine eingehängte Disk aus allen Einhängepunkten aushängen, so verwendet man die folgende Funktion. Sie nimmt als einziger Parameter die Gerätenummer, die ausgehängt werden soll.

90b ⟨*Disk Aushängen* 90b⟩≡ (87b)

```
void unmount_all(dev_t device)
{
    int i;
    for (i = 0; i < MAX_MOUNT_POINTS; i++) {
        if (mounttab[i].device == device) {
            mounttab[i].ondevice = 0;
            mounttab[i].inode = 0;
            mounttab[i].device = 0;
        }
    }
}
```

5.5 Dateipfade

Fassen wir unseren bisherigen Stand zusammen. Im Abschnitt 5.3 haben wir die Grundlagen für einen Umgang mit Dateien gelegt, der unabhängig von Inodenummern und Gerätenummern stattfindet. Zu diesem Zweck haben wir eine Dateitabelle beschrieben, wo die Inodenummern und Gerätenummern in einer Struktur hinterlegt werden, die mit einem Dateideskriptor indiziert wird. Der Dateideskriptor ist das Element, das auf einer höheren Ebene die genannte Abstraktion erlaubt.

In diesem Abschnitt machen einen weiteren Abstraktionsschritt, der uns ermöglicht, Dateien als Pfadangaben zu behandeln. Dafür schreiben wir ein weiteres Modul namens `path`, das die Übersetzung zwischen Pfadangaben und Inodenummern übernimmt. Die Funktionen dieses Modules verwenden die Dateitabelle aus dem Modul `filetab` und gehen davon aus, dass in der Dateitabelle das root-Verzeichniss gesetzt wurde bzw. dass die Funktion `change_root` erfolgreich aufgerufen wurde.

Das Modul `path` ist auf zwei Dateien verteilt: die Headerdatei `path.h`, die die öffentliche Modulfunktionen deklariert, und die C-Datei `path.c`, die dieses Funktionen implementiert.

Den Inhalt der Headerdatei ist der folgende:

91a ⟨*path.h* 91a⟩≡

```
#ifndef PATH_H
#define PATH_H

#include "inode.h"

ino_t search_filename (inode *inode, dev_t device, const char *fname);
int   abspath_to_inode(char *path, ino_t *inodeno, dev_t *device);

#endif
```

Sie deklariert die öffentliche Funktionen dieses Moduls und zusätzlich deklariert sie das

Die C-Datei hat den folgenden Layout:

91b ⟨*path.c* 91b⟩≡

```
#include "path.h"
#include "sys/stat.h"
#include "block.h"
#include "stddef.h" // NULL
#include "filetab.h"
#include "mount.h"

⟨Import von Ulix 92a⟩
⟨Hilfsfunktionen 94a⟩
⟨Dateiname im Verzeichnis suchen 92b⟩
⟨Pfad nach Inode übersetzen 94b⟩
```

5.5.1 Dateiname im Verzeichnis suchen

Eine wichtige Operation, die wir brauchen werden, ist nach einem Dateinamen in einem Verzeichnis zu suchen. Dies erfolgt dadurch, dass wir alle Datenblöcke, die in einem Verzeichnis-Inode eingetragen sind, nach einem Eintrag mit dem gesuchten Namen durchsuchen (siehe Abschnitt 2.5, Seite 17). Um den Namen jedes Verzeichniseintrags mit dem gesuchten Namen zu vergleichen, verwenden wir die ULIX Funktion `strncmp`, die in der Headerdatei `ulix.h` deklariert ist. Da wir aber

dieses Datei, wegen Konflikten der Typdefinitionen, nicht inkludieren, deklarieren wir sie selber, bis sich die Headerdateien von ULIX stabilisieren.

92a ⟨*Import von Ulix* 92a⟩≡ (91b)
```
typedef unsigned int uint;
extern int strncmp(const char *str1, const char *str2, uint n);
```

Die Suchefunktion hat drei Parameter:

- `inode`: Inodenummer des Inodes, dessen Datenblöcke durchsucht werden sollen.
- `device`: Gerätenummer der Disk, worauf sich der durchsuchte Inode befindet.
- `fname`: Dateiname, wonach zu suchen ist.

Die Funktion gibt `0` zurück, falls sie einen fehlerhaften Parameter entdeckt, sonst gibt die Inodenummer zurück, die mit dem Dateinamen assoziiert ist. Der entsprechende Inode befindet sich auf derselben Disk mit Gerätenummer `device`. Sie hat den folgenden Aufbau:

92b ⟨*Dateiname im Verzeichnis suchen* 92b⟩≡ (91b)
```
ino_t search_filename(inode *inode, dev_t device, const char *fname)
{
    ⟨Überprüfe, ob der Inode ein Verzeichnis ist 92c⟩
    ⟨Initialisiere ein Block als Array von Verzeichniseinträgen 92d⟩
    ⟨Berechne Anzahl der Verzeichniseinträge 93a⟩
    ⟨Schleifen Variablen für die Suche 93b⟩
    ⟨Suche nach dem Dateinamen 93c⟩

    return 0;
}
```

Für die Überprüfung des Dateityps eines Inodes verwenden wir den Makro `S_ISDIR(m)` (von „stat: is directory“) aus der Headerdatei `sys/stat.h`, die im Anhang A.2 (Seite 108) vorgestellt wird.

92c ⟨*Überprüfe, ob der Inode ein Verzeichnis ist* 92c⟩≡ (92b)
```
if (! S_ISDIR(inode->mode)) {
    return 0;
}
```

Nun können wir ein Array von Bytes allokieren und das Array als Array von Strukturen des Typs `dir_entry` zu „casten“. Somit können wir leicht die einzelnen Verzeichniseinträge per Index adressieren.

92d ⟨*Initialisiere ein Block als Array von Verzeichniseinträgen* 92d⟩≡ (92b)
```
char block[BLOCK_SIZE] = { 0 };
dir_entry *dirs = (dir_entry *) block;
```

Die Anzahl der Verzeichniseinträge kann dadurch berechnet werden, dass man die Dateigröße des Verzeichnisses durch die Größe eines Verzeichniseintrags teilt:

93a ⟨*Berechne Anzahl der Verzeichniseinträge* 93a⟩≡ (92b)
```
int entries = inode->fsize / DIR_ENTRY_SIZE;
```

Bevor wir die Suchschleife starten, setzen ein paar Kontrollvariablen. Die Variable `dir_idx` enthält den Index des Verzeichniseintrags im Array `dirs`, den wir gerade untersuchen, und wird jedesmal auf `0` gesetzt, wenn ein neuer Block geladen wird. Die Variable `blk_idx` hält den Index der Blocknummer in der Blockliste des Inodes; sie wird mit jedem neuladen eines Blocks inkrementiert. `i` ist die Schleifenvariable und zählt die gesamte Anzahl der Verzeichniseinträge. Sie läuft von `0` bis hächstens `entries`.

93b ⟨*Schleifen Variablen für die Suche* 93b⟩≡ (92b)
```
int dir_idx = 0;
int blk_idx = 0;
int i       = 0;
```

Die Suche selbst findet in einer Schleife statt, die für jeden Verzeichniseintrag läuft. Am Anfang der Schleife überprüfen wir, ob wir einen neuen Block laden müssen. Dies ist immer dann der Fall, wenn der Eintragzähler `i` ein Vielfaches der Anzahl von Verzeichniseinträgen in einem Block erreicht hat (Makro `DIR_ENTRIES_PER_BLOCK`, deklariert in `ulixfs.h`). Das gilt auch dann, wenn `i` gleich Null ist. Falls wir einen neuen Block laden müssen, setzen wir zusätzlich den Eintragsindex `dir_idx` auf Null, damit wir wieder am Anfangs des Arrays `dirs` mit der Suche starten.

93c ⟨*Suche nach dem Dateinamen* 93c⟩≡ (92b)
```
while (i < entries) {
    if (i % DIR_ENTRIES_PER_BLOCK == 0) {
        read_block(device, inode->block[blk_idx], block);
        dir_idx = 0;
        blk_idx++;
    }

    if (strncmp(dirs[dir_idx].name, fname, FNAME_SIZE-1) == 0) {
        return dirs[dir_idx].ino;
    }

    i++;
    dir_idx++;
}
```

5.5.2 Pfad nach Inodenummer auflösen

Die nächste und vielleicht wichtigste Funktion des Moduls `path` übersetzt einen absoluten Dateipfad zu einer Inodenummern. Sie bildet die Brücke zwischen den Funktionen, die auf höherer Ebene mit Dateinamen arbeiten, und Funktionen niedrigerer Ebene, die mit Inodenummern arbeiten.

Wir fangen mit einer Hilfsfunktion an. Sie dient dazu, einen Pfad schrittweise in seinen Komponenten zu zerlegen. Dabei sucht sie in dem Pfadparameter nach dem ersten Schrägstrich und ersetzt ihn mit einem Nullbyte. Sie gibt einen Pointer auf das folgende Byte nach dem gelöschten Schrägstrich zurück. Fall sie keinen Schrägstrich findet, gibt sie einen Pointer auf das letzte Nullbyte des Parameters. Die Abbildung 5.4 auf Seite 95 zeigt die Wirkung dieser kleinen Funktion bei wiederholter Ausführung.

94a ⟨*Hilfsfunktionen* 94a⟩≡ (91b)

```
static
char* cut_rest(char *path)
{
    while(*path && *path != PATH_SEP) {
       path++;
    }

    if (*path) {
        *path = '\0';
        path++;
    }

    return path;
}
```

Mit Hilfe der Funktionen `search_filename` und `cut_rest` können wir die Auflösung eines Pfades relativ einfach implementieren. Die Grundidee dabei ist, den Pfad zu zerlegen und für jede Pfadkomponente Ihre Inodenummer und Gerätenummer zu finden. Wir nutzen dabei das Modul `mount`, um eventuelle Einhängepunkte zu finden.

Die Funktionsparameter sind die folgenden:

- `path` Absoluter Pfad, der mit einem Schrägstrich anfangen muss.
- `inodeno` Speicheradresse, wo wir die gefundene Inodenummer schreiben sollen.
- `device` Speicheradresse, wo wir die gefundene Gerätenummer des Inodes schreiben sollen.

Die Funktion gibt im Fehlerfall -1 zurück, sonst `0`. Sie Funktion hat den folgenden Aufbau:

94b ⟨*Pfad nach Inode übersetzen* 94b⟩≡ (91b)

```
int abspath_to_inode(char *path, ino_t *inodeno, dev_t *device)
{
    ⟨Überprüfe, ob der Pfad absolut ist 96a⟩
    ⟨Setze die Variablen für die Suche 96b⟩
    ⟨Komponentenweise auflösen 96c⟩
    ⟨Inodenummer und Gerätenummer setzen 97d⟩

    return 0;
}
```

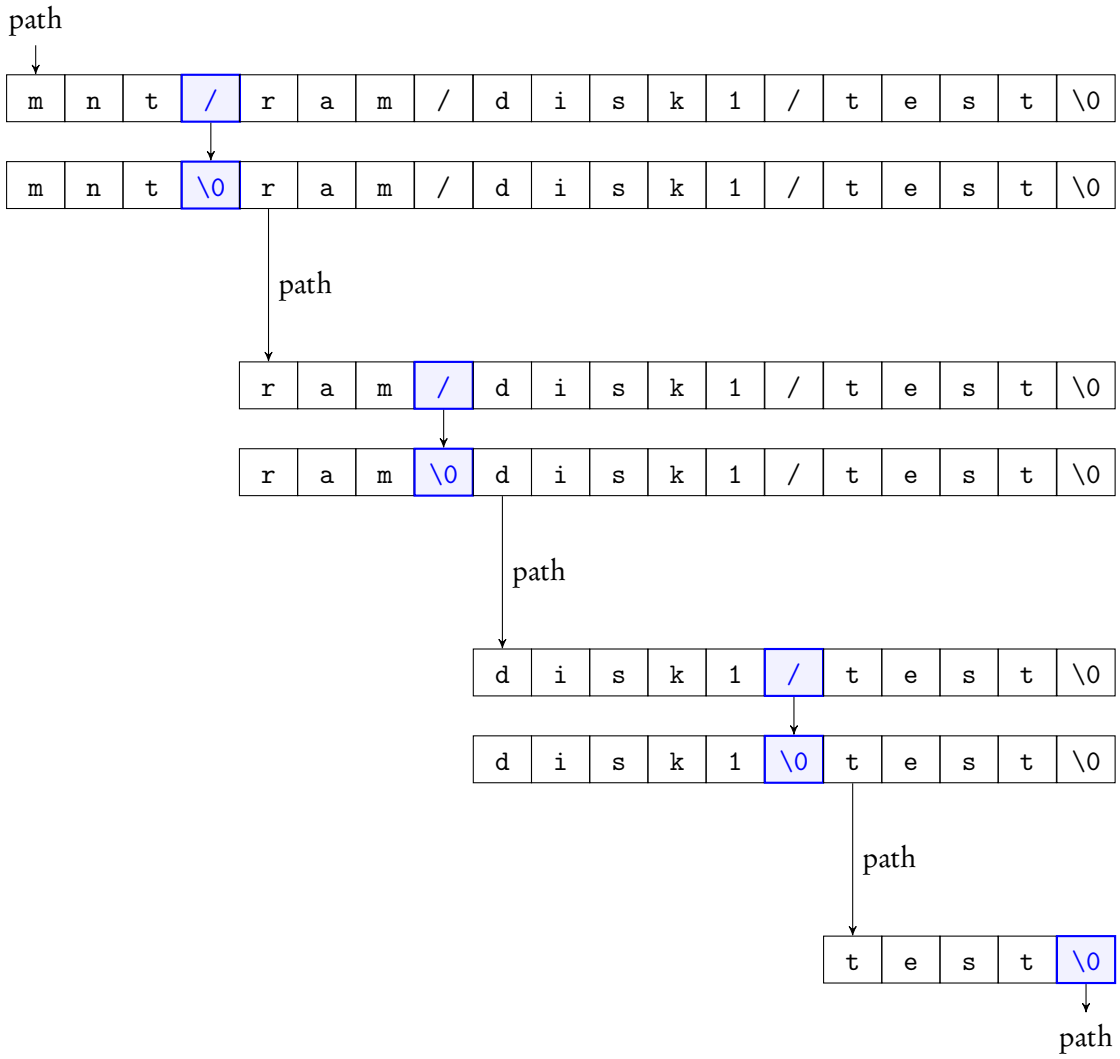

Abbildung 5.4: Schrittweises Zerlegen eines Pfades in Komponenten. „path“ ist ein C-String, der durch wiederholte Anwendung der Funktion `cut_rest` zerlegt wird. Jeweils zwei Zeilen stehen für den Zustand von „path“ vor und nach dem Aufruf der Funktion.

Als erstes überprüfen wir, ob der angegebene Pfad absolut ist. `PATH_SEP` ist in `ulixfs.h` deklariert.

96a ⟨*Überprüfe, ob der Pfad absolut ist* 96a⟩≡ (94b)
```
if (path[0] != PATH_SEP) {
    return -1;
}
```

Während der Suche verwenden wir eine Reihe von Variablen, die den aktuellen Stand der Suche wie folgt speichern. In `root` lesen wir den root-Eintrag aus der Dateitabelle im Modul `filetab`. Besonders wichtig an diesem Eintrag sind die Gerätenummer des root-Dateisystems und der root-Inode selbst, dessen Datenblöcke wir durchsuchen. `curr_inode` speichert den aktuellen Inode, den wir der Funktion `search_filename` übergeben. `curr_inodeno` ist die aktuelle Inodenummer. `curr_device` ist die aktuelle Gerätenummer, sie sich ändern kann, falls wir einen Einhängepunkt finden (siehe Abschnitt 5.4, Seite 87). Die Variable `curr_name` enthält den aktuellen Pfadanteil, der zu suchen ist. Wir initialisieren sie mit dem Pfadnamen ohne den Schrägstrich am Anfang. `rest` enthält den Pfadanteil nach `curr_name`. `tmp_dev` ist eine Hilfsvariable.

96b ⟨*Setze die Variablen für die Suche* 96b⟩≡ (94b)
```
file_entry root;
if (get_root(&root) == -1) {
    return -1;
}

inode curr_inode   = root.inode;
ino_t curr_inodeno = root.inodeno;
dev_t curr_device  = root.device;
dev_t tmp_dev      = NO_DEV;

char *curr_name = path + 1; // ignore leading /
char *rest      = NULL;
```

Nun können wir eine Schleife starten, die solange läuft, bis wir keine Pfadkomponenten mehr haben. Falls der Pfad lediglich aus einem Strich besteht, wird die Schleife gar nicht ausgeführt. Die Schleife fängt mit dem Trennen der aktuellen Pfadkomponenten und endet mit dem Setzen dieser Komponente auf den nachfolgenden Anteil. Dazwischen wird im aktuellen Inode auf der aktuellen Disk nach der Pfadkomponente gesucht und einen eventuellen Einhängepunkt berücksichtigt. Danach wird der aktuelle Inode neugeladen.

96c ⟨*Komponentenweise auflösen* 96c⟩≡ (94b)
```
while (*curr_name) {
    rest = cut_rest(curr_name);

    ⟨Suche nach der aktuellen Pfadkomponente im aktuellen Inode 97a⟩
    ⟨Einhängepunkt behandeln 97b⟩
    ⟨Lade den neuen Inode 97c⟩

    curr_name = rest;
}
```

Die Suche im aktuellen Inode delegieren wir an die Funktion `search_filename`, die wir weiter oben geschrieben haben. Falls die Funktion `0` zurückgibt, konnte sie den aktuellen Dateinamen nicht finden und wir geben als Fehlersignal -1 zurück.

97a ⟨*Suche nach der aktuellen Pfadkomponente im aktuellen Inode* 97a⟩≡ (96c)
```
curr_inodeno = search_filename(&curr_inode, curr_device, curr_name);
if (curr_inodeno < ROOT_INO) {
    return -1; // not found
}
```

An diesem Punkt haben wir die Inodenummer und Gerätenummer der aktuellen Pfadkomponenten `curr_name`. Als nächstes prüfen wir, ob dieser Inode ein *Einhängepunkt* ist und falls ja, dann wechseln wir die Gerätenummer mit der Nummer der eingehängten Disk und setzen die Inodenummer auf die root-Inodenummer, damit wir mit der Suche im root-Inode der eingehängten Disk fortfahren.

97b ⟨*Einhängepunkt behandeln* 97b⟩≡ (96c)
```
tmp_dev = mounted_on(curr_inodeno, curr_device);
if (tmp_dev != NO_DEV) {
    curr_device  = tmp_dev;
    curr_inodeno = ROOT_INO;
}
```

Danach laden wir den gefundenen Inode. Das ist u.U. der root-Inode einer eingehängten Disk.

97c ⟨*Lade den neuen Inode* 97c⟩≡ (96c)
```
if (get_inode(curr_device, curr_inodeno, &curr_inode) == NO_DEV) {
    return -1;
}
```

Als letztes geben wir die letzte Inodenummer und Gerätenummer zurück. Falls die Schleife nicht ausgeführt wurde, geben wir die Startwerte zurück, die auf den root-Inode und root-Gerät zeigen.

97d ⟨*Inodenummer und Gerätenummer setzen* 97d⟩≡ (94b)
```
if (inodeno) *inodeno = curr_inodeno;
if (device)  *device  = curr_device;
```

5.6 Dateien öffnen und schließen

Mit dem Modul `open` kommen wir zu einer zentralen Funktionalität eines Dateisystems: eine Datei öffnen und schließen. Im Gegensatz zu den bisherigen Modulen, die auf Inode- und Bitmap-Ebene arbeiten, benutzen die Funktionen des Moduls `open` Dateinamen und befinden sich somit auf der nächsthöheren Schicht über das Modul `path` und `filetab`, die vom `open` verwendet werden. Mit Hilfe dieser zwei Modulen schaffen wir den Übergang von dem Umgang mit Inodenummern und Gerätenummern zum Umgang mit Dateinamen und Dateideskriptoren.

Die öffentliche Schnittstelle des Moduls `open` befindet sich in der Headerdatei `open.h`, die unten angegeben wird.

98a ⟨*open.h* 98a⟩≡
```
#ifndef OPEN_H
#define OPEN_H

int open_file  (char *filename, int mode);
int close_file (int fd);

#endif
```

Wir haben die Funktionen absichtlich nicht `open` und `close` benannt, damit sie mit den gleichnamigen POSIX Funktionen nicht kollidieren.

Die Implementierung der öffentlichen Funktionen findet in der Datei `open.c` statt:

98b ⟨*open.c* 98b⟩≡
```
#include "path.h"
#include "filetab.h"

⟨Datei öffnen 98c⟩
⟨Datei schließen 99a⟩
```

Eine Datei zu öffnen bedeutet für uns nichts weiteres als den Dateinamen nach Inodenummer und Gerätenummer aufzulösen und diese in die Dateitabelle einzutragen.

Die Funktionsparameter sind

- `filename`: Der Dateipfad, der geöffnet werden soll.
- `mode`: Einer der Werte `O_RDONLY`, `O_WRONLY` oder `O_RDWR` aus der Datei `fcntl.h` (Anhang A.4).

`open_file` gibt einen Dateideskriptor zurück oder -1 in dem Fall, dass die Datei nicht geöffnet werden konnte. Der Dateideskriptor kann für spätere Aufrufe verwendet werden.

98c ⟨*Datei öffnen* 98c⟩≡ (98b)
```
int open_file(char *filename, int mode)
{
    ino_t inodeno = 0;
    dev_t devno = 0;

    if (abspath_to_inode(filename, &inodeno, &devno) == -1) {
        return -1;
    }

    return load_inode(devno, inodeno, mode);
}
```

Noch einfacher ist die Funktion, die eine Datei schließt, denn sie tut nichts anderes als die Funktion `unload_inode` aus dem Modul `filetab` aufzurufen. Ihr Parameter ist der Dateideskriptor, der von `open_file` zurückgegeben wurde. Sie gibt `0` zurück, falls die Datei geschlossen wurde, sonst -1. Siehe auch den Abschnitt 5.3.4 auf der Seite 82.

99a ⟨*Datei schließen* 99a⟩≡ (98b)

```
int close_file(int fd)
{
    if (unload_inode(fd) == -1) {
        return -1;
    } else {
        return 0;
    }
}
```

5.7 Lesen und Schreiben

Kommen wir nun zu den Funktionen, mit denen eine geöffnete Datei gelesen und geschrieben werden kann. Diese dürften die meist verwendeten Funktionen sein, auch wenn sie nicht die einfachsten zu implementieren sind. Das Modul `read`, das diese Funktionen anbietet, hat die folgende öffentliche Schnittstelle:

99b ⟨*read.h* 99b⟩≡

```
#ifndef READ_H
#define READ_H

#include "sys/types.h"

ssize_t read_file  (int fd, void *buffer, size_t nbytes);
ssize_t write_file (int fd, void *buffer, size_t nbytes);

#endif
```

Für die Implementierung der deklarierten Funktionen verwenden wir die folgende C-Datei:

99c ⟨*read.c* 99c⟩≡

```
#include "read.h"
#include "filetab.h"
#include "fcntl.h"
#include "block.h"

⟨Ulix Funktionen 102b⟩

⟨Datei lesen 100a⟩
⟨Datei schreiben 104a⟩
```

5.7.1 Datei lesen

Das Lesen einer Datei bedeutet im Grunde genommen ihre Datenblöcke zu lesen. Die Blocknummern der Datenblöcke sind im Inode eintragen. Was aber das Lesen erschwert ist die Tatsache, dass

die Datenblöcke beliebige Positionen in der Datenregion haben können. Darüberhinaus kann die Anfangsposition des Lesevorgangs beliebig in einer Datei sein. Wir müssen also bei der Implementierung auf Brüche zwischen Blöcke achten.

Die Lesefunktion `read_file` hat die folgenden Parameter:

- `fd`: Dateideskriptor, so wie von der Funktion `open_file` aus dem Modul `open` zurückgegeben wurde (Abschnitt 5.6, Seite 97).
- `buffer`: Lesepuffer, wo die Daten geschrieben werden sollen. Die Funktion geht davon aus, dass der Puffer mindestens die Größe `nbytes` hat.
- `nbytes`: Anzahl der Bytes, die gelesen werden sollen.

Die Funktion gibt die Anzahl der tatsächlich gelesenen Bytes zurück oder -1, falls keine Bytes mehr gelesen werden können oder falls ein Fehler auftritt.

Da die Funktionslogik relativ kompliziert ist, gliedern wir sie in den folgenden logischen Schritte:

100a ⟨*Datei lesen* 100a⟩≡ (99c)
```
ssize_t read_file(int fd, void *buffer, size_t nbytes)
{
    ⟨Lese den Dateieintrag fd und prüfe ihn 100b⟩
    ⟨Anzahl nbytes der lesbaren Bytes anpassen 101a⟩
    ⟨Lese den ersten Datenblock der Datei 101b⟩
    ⟨Lese die anderen Datenblöcke 102c⟩
    ⟨Aktualisiere die Zugriffsposition 103b⟩
}
```

Als erstes Lesen wir aus der Dateitabelle den Eintrag zum Dateideskriptor `fd`. Dann überprüfen wir, ob die Datei gelesen werden kann – dazu darf sie nicht mit dem Modus `O_WRONLY` (write only) geöffnet gewesen sein. Dann prüfen wir, ob die Zugriffsposition sich am Ende der Datei befindet, denn in diesem Fall können wir keine Daten mehr lesen. Falls das der Fall ist, setzen wir die Zugriffsposition auf das Ende der Datei und geben -1 zurück, denn es kann sein, dass wir aus Versehen die Zugriffsposition schon über die Dateigrenzen gesetzt haben.

100b ⟨*Lese den Dateieintrag* `fd` *und prüfe ihn* 100b⟩≡ (100a)
```
file_entry file;
if (read_file_entry(fd, &file) == -1)
    return -1;

if (file.mode == O_WRONLY)
    return -1;

if (file.fpos >= file.inode.fsize) {
    file.fpos = file.inode.fsize;
    write_file_entry(fd, &file);
    return -1;
}
```

Nachdem wir den Dateideskriptor `fd` geprüft haben, überprüfen wir das Parameter `nbytes`. Falls dieser größer als die maximale Anzahl von Bytes, die wir noch lesen können, ist, setzen wir ihn auf diese Anzahl. Wir führen auch zwei Hilfsvariablen ein, die den nachfolgenden Code kürzer machen sollen: `fpos` für die Zugriffsposition und `fsize` für die Dateigröße. Siehe auch die Abbildung 5.5 (Seite 103).

101a ⟨*Anzahl* `nbytes` *der lesbaren Bytes anpassen* 101a⟩≡ (100a)

```
off_t fpos  = file.fpos;
off_t fsize = file.inode.fsize;

if (fpos + nbytes >= fsize)
    nbytes = fsize - fpos;
```

Nun haben wir in der Variable `file` alle Informationen, die wir für das Lesen der Datei benötigen und können damit anfangen, den ersten Block zu lesen. Der erste Block ist besonders, weil er unter Umständen nicht von Anfang gelesen werden muss und auch nicht bis zum Ende. Wir müssen daher berechnen, ab wo bis wo muss dieser erste Block gelesen werden. Diesen Schritt gliedern wir wie folgt:

101b ⟨*Lese den ersten Datenblock der Datei* 101b⟩≡ (100a)

```
⟨Berechne den Index der Blocknummer und Startposition im Block 101c⟩
⟨Berechne die Anzahl der verbliebenen Bytes im ersten Block 101d⟩
⟨Lese die ersten verfügbaren Bytes 102a⟩
```

Den Index `bidx` in der Blockliste des Inodes und die Startposition in dem ersten Block können wir anhand der Zugriffsposition `fpos` ausrechnen.

101c ⟨*Berechne den Index der Blocknummer und Startposition im Block* 101c⟩≡ (101b)

```
int bidx     = fpos / BLOCK_SIZE;  // index of block
int start    = fpos % BLOCK_SIZE;  // index in block
```

Nun wissen wir, dass die Blocknummer, die wir zuerst lesen müssen steht in der Blockliste des Inodes am Index `bidx` und in dem ersten Block müssen wir an der Position `start` anfangen. Es bleibt zu berechnen, wieviel Bytes müssen wir aus dem ersten Block lesen, denn es kann ja sein, dass wir weniger als verfügbar lesen müssen. Die Anzahl der verbliebenen Bytes berechnen wir wie folgt in der Variable `how_much` (siehe auch die Abbildung 5.5).

101d ⟨*Berechne die Anzahl der verbliebenen Bytes im ersten Block* 101d⟩≡ (101b)

```
off_t how_much = BLOCK_SIZE - start; // how much to read
if (nbytes < how_much)
    how_much = nbytes;
```

Jetzt können wir den ersten Block von der Disk laden und hiervon die ersten `how_much` Bytes in den Puffer `buffer` kopieren. Damit haben wir den ersten verfügbaren Block gelesen.

102a ⟨*Lese die ersten verfügbaren Bytes* 102a⟩≡ (101b)
```
char block[BLOCK_SIZE] = { 0 };
read_block(file.device, file.inode.block[bidx], block);
memcpy(buffer, block+start, how_much);
```

Um die Bytes zu kopieren, verwenden wir die ULIX-Funktion `memcpy`. Diese Funktion ist zwar in der Datei `ulix.h` deklariert, aber die Headerdatei inkludieren wir nicht, um Konflikten bei den Typdefinitionen zu vermeiden. Deshalb deklarieren wir diese Funktion direkt in unserer C-Datei.

102b ⟨*Ulix Funktionen* 102b⟩≡ (99c)
```
/* replace this with Ulix-headers */
typedef unsigned int uint;
extern void *memcpy(void *dest, const void *src, size_t count);
```

Nachdem wir den ersten Block abgehandelt haben, können wir die anderen notwendigen Blöcke lesen. Dafür verwenden wir eine Schleife, die solange läuft bis alle übrigen Bytes gelesen wurden. Die Schleifen-Variable `n` initialisieren wir auf `how_much`, die zu diesem Zeitpunkt die Anzahl der bisher gelesenen Bytes enthält.

102c ⟨*Lese die anderen Datenblöcke* 102c⟩≡ (100a) 102d▹
```
size_t n = how_much;
```

In jedem Schleifenlauf inkrementieren wir zuerst den Blockindex `bidx`, damit wir zum nächsten Block kommen und lesen diesen neuen Block.

102d ⟨*Lese die anderen Datenblöcke* 102c⟩+≡ (100a) ◃102c 102e▹
```
while (n < nbytes) {
    bidx++;
    read_block(file.device, file.inode.block[bidx], block);
```

Jetzt müssen wir entscheiden, ob wir den ganzen Block lesen, oder ob nur einen Teil davon, falls wir z. B. zum letzten Block angekommen sind. Würde ein ganzer Blöck dazu führen, dass wir mehr als nötig lesen, so setzen wir die Variable `how_much` auf die restlichen Bytes die noch zu lesen sind, sonst setzen wir sie gleich der Blockgröße.

102e ⟨*Lese die anderen Datenblöcke* 102c⟩+≡ (100a) ◃102d 103a▹
```
    if (n + BLOCK_SIZE >= nbytes)
        how_much = nbytes - n;
    else
        how_much = BLOCK_SIZE;
```

Nun können wir wieder `how_much` viele Bytes aus unserem Blockpuffer `block` in den Parameterpuffer `buffer` kopieren. Anschließend erhöhen wir die Schleifenvariable `n`.

103a ⟨*Lese die anderen Datenblöcke* 102c⟩+≡ (100a) ◁102e

```
        memcpy((char *)buffer + n, block, how_much);
        n += how_much;
    }
```

Wenn die Schleife verlassen wird, haben wir `nbytes` Bytes gelesen. Jetzt können wir die Zugriffsposition der Datei anpassen und die Anzahl `n` der gelesenen Byte zurückgeben.

103b ⟨*Aktualisiere die Zugriffsposition* 103b⟩≡ (100a)

```
    file.fpos += n;
    write_file_entry(fd, &file);

    return n;
```

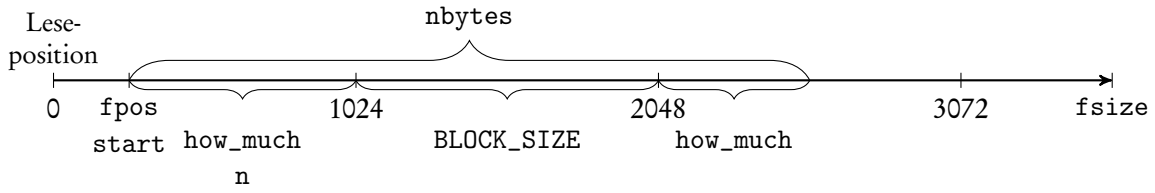

Abbildung 5.5: Überblick der Variablen in der Funktion `read_file`.

5.7.2 Datei schreiben

Die nächste Funktion ist die längste und schwierigste in unserem ganzen Dateisystem. Sie wird verwendet, um eine geöffnete Datei zu beschreiben. Ihre Parameter sind die folgenden:

- `fd`: Dateideskriptor der geöffneten Datei.
- `buffer`: Byte-Puffer, der geschrieben werden soll. `nbytes`: Wieviele Bytes solle aus `buffer` geschrieben werden.

Die Funktion gibt die Anzahl der geschriebenen bytes zurück oder -1, falls die Datei nicht geschrieben werden konnte.

Die Daten aus `buffer` werden ab derjenigen Position in der Datei geschrieben, die im Feld `fpos` in der Dateitabelle vermerkt ist (Zugriffsposition in der Datei). Um diese Zugriffsposition zu verändern, muss man vorher die Funktion `lseek_file_entry` aus dem Modul `filetab` verwenden (siehe Abschnitt 5.3.6, Seite 84). Bedarft der Schreibvorgang mehr Datenblöcke als für den der entsprechenden Inode allokiert wurden, so werden neue allokiert. Die Funktion überschreibt unter Umständen den alten Inhalt, falls die Zugriffsposition `fpos` innerhalb der Datei zeigt.

Da diese Funktion recht kompliziert ist, werden wir sie schrittweise beschreiben. Ihr Aufbau ist der folgende:

104a ⟨*Datei schreiben* 104a⟩≡ (99c)
```
ssize_t write_file(int fd, void *buffer, size_t nbytes)
{
    ⟨Dateieintrag aus der Dateitabelle auslesen 104b⟩
    ⟨Zugriffsposition anpassen 104c⟩
    ⟨Anpassen von nbytes 104d⟩
    ⟨Blockindex berechnen 105a⟩
    ⟨Startposition im Block berechnen 105c⟩
    ⟨Schreibe den ersten Datenblock 105d⟩
    ⟨Schreibe die anderen Blöcke 106a⟩
    ⟨Inode aktualisieren 106b⟩
}
```

Zuerst wird der entsprechende Dateieintrag aus der `filetab` Tabelle gelesen und geprüft, ob die Datei im Scheibgeschützt-Modus geöffnet wurde (siehe Abschnitt 5.3).

104b ⟨*Dateieintrag aus der Dateitabelle auslesen* 104b⟩≡ (104a)
```
file_entry file;
if (read_file_entry(fd, &file) == -1)
    return -1;

if (file.mode == O_RDONLY)
    return -1; // read only, cannot write
```

Dann prüfen wir, ob die Zugriffsposition außerhalb des Dateibereichs zeigt und falls ja, setzen wir sie an das Ende der Datei. Es könnte sein, dass wir durch einen Programmierfehler die Zugriffsposition falsch gesetzt haben. Die Zugriffsposition setzen wir nicht direkt in der Dateitabelle, sondern wir kopieren sie lokal und arbeiten mit der lokalen Kopie. Am Ende der Funktion schreiben wir sie zurück in die Dateitabelle.

104c ⟨*Zugriffsposition anpassen* 104c⟩≡ (104a)
```
off_t fpos  = file.fpos;
off_t fsize = file.inode.fsize;
if (fpos > fsize)
    fpos = fsize;
```

Der Nächste Schritt ist das Parameter `nbytes` anzupassen, damit wir eventuelle Überläufe vermeiden. Falls wir ab der Position `fpos` `nbytes` Bytes schreiben würden und dabei die maximale Dateigröße überschreiten würden, so setzen wir `nbytes` auf die maximale Anzahl von Bytes, die wir noch schreiben können.

104d ⟨*Anpassen von* `nbytes` 104d⟩≡ (104a)
```
if (fpos + nbytes >= MAX_FILE_SIZE)
    nbytes = MAX_FILE_SIZE - fpos;

if (nbytes == 0)
    return 0;
```

Nun können wir den Index `bidx` in der Blockliste des Inodes berechnen, der der aktuellen Zugriffsposition entspricht. Falls notwendig, allokieren wir einen neuen Datenblock, damit am Index `bidx` eine gültige Blocknummer steht.

105a ⟨*Blockindex berechnen* 105a⟩≡ (104a)
```
off_t bidx = fpos / BLOCK_SIZE;
if (bidx >= INODE_BLOCKS)
    return -1;

⟨Allokiere neuen Block falls nötig 105b⟩
```

Ob wir einen neuen Block allokieren müssen, können wir dadurch prüfen, dass wir den Blockindex `bidx` mit der Blockgröße multiplizieren. Ergebniss ist wie viele Bytes passen in `bidx` Blöcken. Falls wir dabei die aktuelle Dateigröße überschreiten, müssen wir am Index `bidx` einen neue Blocknummer allokieren. Für die Allokation verwenden wir die Funktion `find_and_set_bit` aus dem Modul `block` (siehe Abschnitt 5.1.3.2, Seite 71).

105b ⟨*Allokiere neuen Block falls nötig* 105b⟩≡ (105a 106a)
```
if (bidx * BLOCK_SIZE >= fsize) {
    off_t i = find_and_set_bit(file.device, BMAP);
    if (i == 0)
        return -1;
    else
        file.inode.block[bidx] = i;
}
```

Zu diesem Punkt haben wir am Index `bidx` die erste Blocknummer, die wir mit Daten aus `buffer` beschreiben müssen. Als nächstes berechnen wir, ab welchem Index `start` und wie viele Bytes `to_write` müssen wir in diesem Block schreiben. Es kann ja sein, dass wir weniger als den ganzen restlichen Block schreiben müssen.

105c ⟨*Startposition im Block berechnen* 105c⟩≡ (104a)
```
off_t start = fpos % BLOCK_SIZE;
off_t to_write = BLOCK_SIZE - start;
if (to_write > nbytes)
    to_write = nbytes;
```

Als nächstes können wir einen Null-Block deklarieren, den Block aus der Disk lesen (falls wir ihn nicht komplett überschreiben müssen), `to_write` viele Bytes in den Block kopieren und dann den Block zurück auf die Disk schreiben. Wir verwenden hier alle Variablen, die wir bisher berechnet haben.

105d ⟨*Schreibe den ersten Datenblock* 105d⟩≡ (104a)
```
char block[BLOCK_SIZE] = { 0 };
if ((start != 0) || (to_write != BLOCK_SIZE))
    read_block(file.device, file.inode.block[bidx], block);

memcpy(block+start, buffer, to_write);
write_block(file.device, file.inode.block[bidx], block);
```

Nachdem wir den ersten Datenblock geschrieben haben, der gemäß der Zugriffsposition `fpos` zu schreiben war, können wir leichter mit den anderen Blöcken vorgehen, denn dafür müsen wir nicht mehr den Startindex im jedem Block berechnen: Er ist immer Null.

Wir starten eine Schleife, die solange läuft, bis wir alle Bytes geschrieben haben. Eventuell wird diese Schleife gar nicht ausgeführt, falls wir vorher schon alle Bytes geschrieben haben. In jedem Schleifenlauf inkrementieren wir zuerst den Blockindex in der Blockliste des Inodes. Fall notwendig, allokieren wir an diesem Index eine neue Blocknummer. Danach berechnen wir die Anzahl von Bytes, die wir im aktuellen Block schreiben müssen. Diese Anzahl, die wieder in der Varieble `to_write` steht, ist entweder ein ganzer Block oder weniger, falls wir zum letzen Block angekommen sind. Danach laden wir den Block von der Disk, kopieren die Daten aus dem Parameter `buffer` und erhöhen die Schleifenvariable `n`, die die Anzahl der bisherigen geschriebenen Bytes enthält.

106a ⟨*Schreibe die anderen Blöcke* 106a⟩≡ (104a)

```
size_t n = to_write;
while (n < nbytes) {
    bidx++;
    ⟨Allokiere neuen Block falls nötig 105b⟩

    if (n + BLOCK_SIZE >= nbytes)
        to_write = nbytes - n;
    else
        to_write = BLOCK_SIZE;

    read_block(file.device, file.inode.block[bidx], block);
    memcpy(block, (char *)buffer + n, to_write);
    write_block(file.device, file.inode.block[bidx], block);
    n += to_write;
}
```

Als letztes aktualisieren wir die Informationen aus der Dateitabelle und speichern den eventuell modifizierten Inode auf die Disk.

106b ⟨*Inode aktualisieren* 106b⟩≡ (104a)

```
fpos += n;

if (fpos >= fsize)
    fsize += (fpos - fsize);

file.fpos        = fpos;
file.inode.fsize = fsize;
write_file_entry(fd, &file);
save_file_entry(fd);

return n;
```

A POSIX Hearderdateien

A.1 Datentypen (`types.h`)

In diesem Abschnitt werden wir die Datei `sys/types.h` erstellen, die Bestandteil aller Unix-artigen Betriebssysteme ist, bzw. die vom POSIX-Standard spezifiziert wird (siehe [15]). Sie enthält Typdefinitionen, die im ganzen System verwendet werden, wie z. B. `size_t` und `mode_t`. Die meisten Typen dürften aus anderen Unix Versionen bekannt sein. Eine ähnliche Datei wird in [14, S. 135, 656] vorgestellt.

Wir werden nicht alle Datentypen, die vom POSIX spezifiziert sind, in `sys/types.h` implementieren, sondern nur einige davon, die wir gerade brauchen.

107 ⟨*sys/types.h* 107⟩≡

```
#ifndef TYPES_H
#define TYPES_H

#ifndef __size_t_defined
typedef unsigned int       size_t;  // size of objects
#define __size_t_defined
#endif

#ifndef __ssize_t_defined
typedef int                ssize_t; // count of bytes
#define __ssize_t_defined
#endif

#ifndef __off_t_defined
typedef unsigned long int  off_t;   // offset and file size
#define __off_t_defined
#endif

#ifndef __mode_t_defined
typedef unsigned short int mode_t;  // file type, permissions
#define __mode_t_defined
#endif

#ifndef __nlink_t_defined
typedef short int          nlink_t; // number of hard links
#define __nlink_t_defined
#endif

#ifndef __uid_t_defined
typedef short int          uid_t;   // user id
#define __uid_t_defined
#endif
```

```
#ifndef __gid_t_defined
typedef short int          gid_t;   // group id
#define __gid_t_defined
#endif

#ifndef __time_t_defined
typedef long  int          time_t;  // time in seconds
#define __time_t_defined
#endif

#ifndef __block_t_defined
typedef unsigned long int  block_t; // block number
#define __block_t_defined
#endif

#ifndef __ino_t_defined
typedef unsigned long int  ino_t;   // inode number
#define  __ino_t_defined
#endif

#ifndef __dev_t_defined
typedef unsigned short int dev_t;   // device number (major|minor)
#define __dev_t_defined
#endif

#endif
```

Alle Typen werden nur bedingt definiert. Die Bedingungen, als Makros angegeben, orientieren sich an den entsprechenden Makros aus dem Linux-Umfeld. Dies erfolgt aus mehreren Gründen:

- Bestimmte Typen wie `size_t` werden gemäß POSIX mehrfach definiert. So wird `size_t` auch in der Datei `stddef.h` definiert (siehe Anhang A.3).
- Diese Datei wird in `ulixfs.h` inkludiert, die im Kapitel 4 außerhalb von ULIX verwendet wird. In diesem externen Betriebssystem (Linux in unserem Fall) werden die Typen nochmals definiert, wobei ihre Größe eine andere ist. Inkludiert man `ulixfs.h` vor anderen Dateien, so wird durch die Makro-Definitionen verhindert, dass die Typen mehrfach definiert werden.

Der Typ `dev_t` ist dafür gedacht, Gerätenummern zu speichern. Er wird im Abschnitt 3.1.1 auf Seite 21 und in der Abbildung 3.2 auf Seite 22 vorgestellt.

A.2 Status (`stat.h`)

In diesem Abschnitt werden wir wieder eine klassische Unix- bzw. POSIX-Datei beschreiben: `sys/stat.h`. Diese Datei definiert Makros und Größen, die hauptsächlich dazu dienen, die Eigenschaften einer Datei zu erfragen. Wir werden nicht alle spezifizierten Typen und Funktionen implementieren, sondern nur ein Teil davon. Für die komplete POSIX-Definition der Datei siehe [16]. Siehe auch die Minix-Implementierung dieser Datei in [14, S.660].

Da es sich um eine POSIX-Datei handelt, werden wir die Bezeichnungen der einzelnen Typen und Konstanten aus dem Standard übernehmen, auch wenn sie nicht gerade aussagekräftig sind. Es ist zu bemerken, dass der POSIX-Standard nur die Namen der Typen und Hinweise zu deren Verwendung angibt, nicht aber deren genaue Implementierung. Für diese Implementierung orientieren wir uns an der Minix-Implementierung in [14, S.660], da sie uns sinnvoll erscheinen und zu unserem Modell aus dem Abschnitt 2.4 (Seite 16) passt. Das sind übrigens dieselben Definitionen wie in dem Linux-Kernel (siehe [2]). Wir machen nur kleine Änderungen dazu, indem wir manche Werte eindeutig nach `mode_t` casten.

Die Datei hat den folgenden Aufbau:

109a ⟨*sys/stat.h* 109a⟩≡
```
#ifndef STAT_H
#define STAT_H

#include "types.h"

⟨Bitmasken fuer Dateitypen 109b⟩
⟨Tests auf Dateitypen 110a⟩
⟨Bitmasken für die Zugriffsrechte 110b⟩

#endif
```

Alle folgenden Bitmasken sind dafür gedacht, in Verbindung mit dem Wert des Inode-Feldes `mode` benutzt zu werden. Siehe dazu die Abbildung 2.5 auf Seite 16.

Zuerst werden der Reihe nach Oktalwerte definiert, die verschiedenen Dateitypen entsprechen. Ihre Bitmuster sind so konstruiert, dass sie mit den Bitpositionen aus Abbildung 2.5 übereinstimmen. Der erste Wert (`S_IFMT`) extrahiert durch Anwendung der bitweisen UND-Verknüpfung mit dem `mode`-Feld diejenigen Bits, die dem Dateityp entsprechen. Siehe Abbildung A.1 (Seite 110) für ein Beispiel, wie man diese Werte verwenden kann. Dort wird anhand des `mode`-Feldes herausgefunden, dass es sich um eine reguläre Datei handelt.

109b ⟨*Bitmasken fuer Dateitypen* 109b⟩≡ (109a)
```
#define S_IFMT   ((mode_t) 0170000) // file type mask
#define S_IFBLK  ((mode_t) 0060000) // file type: block special
#define S_IFCHR  ((mode_t) 0020000) // file type: character special
#define S_IFIFO  ((mode_t) 0010000) // file type: FIFO special
#define S_IFREG  ((mode_t) 0100000) // file type: regular file
#define S_IFDIR  ((mode_t) 0040000) // file type: directory
#define S_IFLNK  ((mode_t) 0120000) // file type: symbolic link
#define S_IFSOCK ((mode_t) 0140000) // file type: socket
#define S_ISUID  ((mode_t) 0004000) // set user id on execution
#define S_ISGID  ((mode_t) 0002000) // set group id on execution
```

Als nächstes werden Funktionsmakros definiert, mit deren Hilfe der Dateityp geprüft oder getestet werden kann. Argument für jedes Funktionsmakro ist der Wert des `mode` Feldes eines Inodes. Ergebnis ist 1 (wahr) oder `0` (falsch), je nachdem, ob der Test erfolgreich ist oder nicht. Es werden

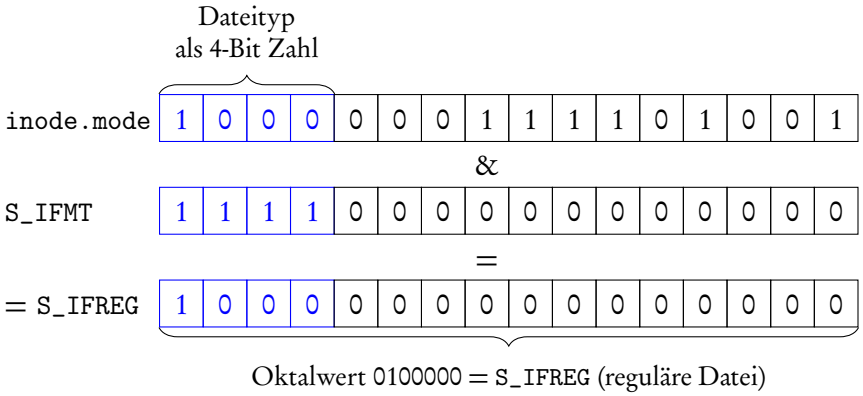

Abbildung A.1: Überprüfung des Dateityps anhand des `mode` Feldes. Dieses Feld wird mit der Dateityp-Bitmaske bitweise verundet. Ergebnis ist der oktale Wert 0100000, der für eine reguläre Datei steht.

der Reihe nach alle Bitmasken von oben geprüft. Die Abbildung A.1 zeigt anschaulich, wie eine solche Funktion funktioniert.

110a ⟨*Tests auf Dateitypen* 110a⟩≡ (109a)

```
#define S_ISBLK(m)  (((m) & S_IFMT) == S_IFBLK) // is it block special?
#define S_ISCHR(m)  (((m) & S_IFMT) == S_IFCHR) // is it character special?
#define S_ISFIFO(m) (((m) & S_IFMT) == S_IFIFO) // is it fifo?
#define S_ISREG(m)  (((m) & S_IFMT) == S_IFREG) // is it regular file?
#define S_ISDIR(m)  (((m) & S_IFMT) == S_IFDIR) // is it directory?
#define S_ISLNK(m)  (((m) & S_IFMT) == S_IFLNK) // is it symbolic link?
#define S_ISSOCK(m) (((m) & S_IFMT) == S_IFSOCK)// is it socket?
```

Möchte man mit diesen Makros überprüfen, ob ein Inode ein Verzeichnis oder eine reguläre Datei darstellt, so kann man wie folgt schreiben:

```
inode ino;
...
if (S_ISDIR(ino.mode)) {/* ein Verzeichnis */}
else if (S_ISREG(ino.mode)) {/* regulaere Datei */}
else {/* keine */}
```

Alternativ kann man selber die bitweise UND-Verknüpfung durchführen und wie folgt ein `switch` verwenden:

```
switch (ino.mode & S_IFMT) {
    case S_IFREG: /* normale Datei */
    case S_IFDIR: /* Verzeichnis */
}
```

Als nächstes und letztes werden Bitmasken definiert, die den Zugriffsrechten entsprechen (siehe Abschnitt 2.4.2, Seite 16).

110b ⟨*Bitmasken für die Zugriffsrechte* 110b⟩≡ (109a)

```
#define S_IRWXU   00700  /* owner mask        */
```

```
#define S_IRUSR   00400  /* owner can read    */
#define S_IWUSR   00200  /* owner can write   */
#define S_IXUSR   00100  /* owner can execute */

#define S_IRWXG   00070  /* group mask        */
#define S_IRGRP   00040  /* group can read    */
#define S_IWGRP   00020  /* group can write   */
#define S_IXGRP   00010  /* group can execute */

#define S_IRWXO   00007  /* others mask        */
#define S_IROTH   00004  /* others can read    */
#define S_IWOTH   00002  /* others can write   */
#define S_IXOTH   00001  /* others can execute */
```

Diese Werte kann man analog zu den Bitmasken für die Dateitypen verwenden. Die Abbildung A.2 zeigt die Verwendung der Bitmaske S_IRWXO, die die Zugriffsrechte aller anderen Benutzer extrahiert, die nicht als Besitzer oder Gruppe in der uid bzw. gid Felder des Inodes eingetragen sind. In der Abbildung wird geprüft, ob diese „andere“ die Datei lesen können. Die Überprüfung schlägt fehl, da das Ergebnis ungleich S_IROTH ist (Leserechte für alle anderen). Würde man das Ergebnis der UND-Verknüpfung mit S_IXOTH vergleichen (Ausführungsrechte für alle anderen), so würde der Vergleich wahr ergeben.

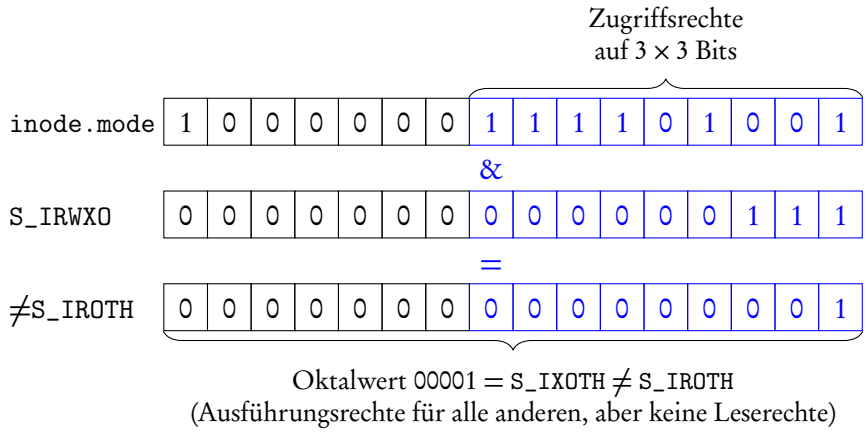

Abbildung A.2: Überprüfung der Zugriffsrechte anhand des mode-Feldes eines Inodes. Hier wird die Maske S_IRWXO verwendet, um alle Zugriffsrechte für „others“ zu extrahieren. Das Ergebnis wird mit dem speziellen Leserecht vergliechen. Durch Anwendung derselben Maske können auch die Schreibe- und Ausführungsrechte geprüft werden.

A.3 Standard Makros (stddef.h)

Eine andere POSIX-Datei, die wir in disem Abschnitt beschreiben werden, ist stddef.h, die oft benutzte Makros und Typen beinhaltet (siehe [18]). Wir werden wieder nicht alle Typen imple-

mentieren, sondern nur diejenigen, die für uns sinnvoll sind.

112a ⟨*stddef.h* 112a⟩≡
```
#ifndef STDDEF_H
#define STDDEF_H

#define NULL   ((void *)0)

#ifndef __size_t_defined
typedef unsigned int       size_t;  // size of objects
#define __size_t_defined
#endif

typedef long ptrdiff_t;  // result of subtracting two pointers
typedef char wchar_t;

#endif
```

Es ist zu bemerken, dass der Typ `size_t` nur dann definiert wird, wenn er noch nicht definiert wurde. Das kommt daher, dass, gemäß POSIX, dieser Typ auch in der Datei `sys/types.h` spezifiziert wird, die wir im Anhang A.1 auf Seite 107 definieren. Inkludiert man beide Headerdateien, so muss dafür gesorgt werden, dass diese Definition nicht zweimal vorkommt.

A.4 Dateikonstanten (`fcntl.h`)

Die Datei `fcntl.h` enthält Makros für den Umgang mit Dateien. Aus dieser Datei definieren nur ein paar der Standardwerte. Siehe [17] für eine Aufzählung aller Werte und Definitionen aus dem Standard.

112b ⟨*fcntl.h* 112b⟩≡
```
/*
 * POSIX Standard: 6.5 File Control Operations <fcntl.h>
 */

#define O_RDONLY   00   // read only
#define O_WRONLY   01   // write only
#define O_RDWR     02   // read and write
```

B Integration in ULIX

B.1 Multiboot Header

Der Bootloader Grub kommuniziert mit dem ULIX-Kernel mittels einem „Multiboot“ Header. Wichtig für uns sind die Einträge `mods_count` und `mods_addr` in diesem Header, die die Anzahl und Startadressen von Grub-Modulen angeben. Grub-Module sind Dateien, die Grub zusätzlich zum ULIX-Kernel in den Speicher laden kann. Die RAM-Disk, die wir mit dem Programm `mkfs.ulixfs` generieren (siehe Kapitel 4), wird als Grub-Modul in den Speicher geladen und von dort von unserem RAM-Disk-Treiber in seine interne Datenstrukturen kopiert. Siehe dazu den Abschnitt 3.4.6.6 und [20].

Die Definition des Multiboot-Headers haben wir aus [10] entnommen. Sie entspricht dem Standard in [20]. Die unten angegeben Headerdatei wird vom Modul `module` im Abschnitt B.2 verwendet.

113 ⟨*multiboot.h* 113⟩≡

```
// multiboot.h - Declares the multiboot info structure.
//               Made for JamesM's tutorials <www.jamesmolloy.co.uk>

#ifndef MULTIBOOT_H
#define MULTIBOOT_H


typedef unsigned int   u32int;
typedef          int   s32int;
typedef unsigned short u16int;
typedef          short s16int;
typedef unsigned char  u8int;
typedef          char  s8int;

#define MULTIBOOT_FLAG_MEM     0x001
#define MULTIBOOT_FLAG_DEVICE  0x002
#define MULTIBOOT_FLAG_CMDLINE 0x004
#define MULTIBOOT_FLAG_MODS    0x008
#define MULTIBOOT_FLAG_AOUT    0x010
#define MULTIBOOT_FLAG_ELF     0x020
#define MULTIBOOT_FLAG_MMAP    0x040
#define MULTIBOOT_FLAG_CONFIG  0x080
#define MULTIBOOT_FLAG_LOADER  0x100
#define MULTIBOOT_FLAG_APM     0x200
#define MULTIBOOT_FLAG_VBE     0x400

struct multiboot_header
{
    u32int flags;
```

```
    u32int mem_lower;
    u32int mem_upper;
    u32int boot_device;
    u32int cmdline;
    u32int mods_count; // number of modules
    u32int mods_addr;
    u32int num;
    u32int size;
    u32int addr;
    u32int shndx;
    u32int mmap_length;
    u32int mmap_addr;
    u32int drives_length;
    u32int drives_addr;
    u32int config_table;
    u32int boot_loader_name;
    u32int apm_table;
    u32int vbe_control_info;
    u32int vbe_mode_info;
    u32int vbe_mode;
    u32int vbe_interface_seg;
    u32int vbe_interface_off;
    u32int vbe_interface_len;
} __attribute__((packed));


#endif
```

B.2 Test Modul

In diesem Abschnitt gehen wir darauf ein, wie man die Implementierung unseres Dateisystems in das Betriebssystem ULIX einbindet. Dafür werden wir ein neues Modul namens `module` schreiben, das das Dateisystem zur Laufzeit initialisiert und die wichtigsten Funktionen des Dateisystems demonstriert. Spätere Arbeiten an ULIXFS können sich an diesem Modul orientieren. Wir geben zuerst die Schnittstelle unseres Moduls an:

114 ⟨*module.h* 114⟩≡

```
#ifndef MODULE_H
#define MODULE_H

void initialize_module (void *mboot_ptr);

#endif
```

Die Funktion `initialize_module` wird aus der Funktion `main` des Betriebssystems ULIX heraus aufgerufen. Der Parameter `mboot_ptr` ist derjenige, der der Funktion `main` in `ulix.c` übergeben wird. Er zeigt auf einen „Multiboot“ Header, der alle Informationen enthält, die wir zum Laden der RAM-Disk und initialisieren des Dateisystems benötigen und die vom Bootloader übergeben wird. Siehe Abschnitt B.1.

Das Modul wird in der Datei `module.c` implementiert und zusammen mit allen anderen Modulen und Dateien von ULIX kompiliert. Die Datei hat den folgenden Aufbau:

115a ⟨*module.c* 115a⟩≡
```
⟨Inkludierte Headerdateien 115b⟩
⟨Deklarationen aus dem Ulix Code 115c⟩
⟨Funktionsdeklarationen 115d⟩

⟨Dateisystem initialisieren und testen 119a⟩

⟨Superblock ausgeben 116a⟩
⟨Dateiinformationen ausgeben 116b⟩
⟨Verzeichnis auflisten 117a⟩
⟨Dateiinhalt ausgeben 118c⟩
```

In diesem Modul werden wir fast alle unsere Module verwenden. Wir inkludieren daher die meisten Headerdateien, die wir in dieser Ausarbeitung erstellt haben.

115b ⟨*Inkludierte Headerdateien* 115b⟩≡ (115a)
```
#include "multiboot.h"
#include "ramdisk-driv.h" // rd_load()
#include "device.h"       // NO_DEV
#include "ulixfs.h"       // superblock
#include "block.h"        // read_superblock()
#include "fcntl.h"        // O_RDWR
#include "filetab.h"      // change_root()
#include "inode.h"        // get_inode()
#include "open.h"         // open_file()
#include "read.h"         // read_file()
```

Aus dem Ulix-Code müssen wir ein paar Funktionen und Makros „importieren“. Da diese noch nicht in eigenen Headerdateien deklariert sind, deklarieren wir sie hier in der Datei `modul.c`. Auf den Makro `PHYSICAL` gehen später ein.

115c ⟨*Deklarationen aus dem Ulix Code* 115c⟩≡ (115a)
```
#define PHYSICAL(x) ((x)+0xd0000000)
extern int printf(const char *format, ...);
extern void *memset(void *dest, char val, size_t count);
```

Nun deklarieren wir zusätzlich die Hilfsfunktionen, die wir in dieser Datei implementieren und verwenden werden.

115d ⟨*Funktionsdeklarationen* 115d⟩≡ (115a)
```
static void dump_superblock(superblock *super);
static void dump_file(file_entry *file);
static void ls_dir(ino_t inodeno, dev_t devno);
static void print_file_content(int fd);
```

B.2.1 Hilfsfunktionen

Fangen wir mit ein paar Hilfsfunktionen an. Sie dienen hauptsächlich zur Ausgabe von verschiedenen Inhalten. Wenn das Dateisystem endgültig eingebunden wird (und nicht nur getestet, wie hier), sollte man auf diese Ausgaben natürlich verzichten bzw. sie auf das Wesentliche reduzieren.

B.2.1.1 Superblock ausgeben

Die folgende Funktion verwendet die ULIX-Funktion `printf`, um den Inhalt eines Superblocks auszugeben.

116a ⟨*Superblock ausgeben* 116a⟩≡ (115a)

```
void dump_superblock(superblock *super)
{
    printf("\n++++ SUPERBLOCK DUMP ++++\n");
    printf("\tInodes      : %u\n", super->inodes);
    printf("\tInode bitmap: %u\n", super->imap);
    printf("\tBlock bitmap: %u\n", super->bmap);
    printf("\tInode table : %u\n", super->itable);
    printf("\tData        : %u\n", super->data);
    printf("\tBlocks total: %u\n\n", super->nblocks);
}
```

B.2.1.2 Dateieintrag ausgeben

Die folgende Funktion gibt einen Eintrag aus der Dateitabelle aus (siehe 5.3). Dabei werden auch die Blocknummern des Inodes ausgegeben.

116b ⟨*Dateiinformationen ausgeben* 116b⟩≡ (115a)

```
void dump_file(file_entry *file)
{
    printf("\n++++ FILE DUMP ++++\n");
    printf("\tInode number     : %u\n", file->inodeno);
    printf("\tDevice number    : %u\n", file->device);
    printf("\tFile position    : %u\n", file->fpos);
    printf("  - Inode Information - \n");
    printf("\tFile size        : %u\n", file->inode.fsize);
    printf("\tNLinks           : %u\n", file->inode.nlinks);

    int i;
    block_t b = 0;
    for (i = 0; i < INODE_BLOCKS; i++) {
        b = file->inode.block[i];
        printf("\tblock[%d] = %u\n", i, b);
    }
}
```

B.2.1.3 Verzeichnis auflisten

Die folgende Funktion ist interessanter. Sie gibt den Inhalt eines Verzeichnisses auf. Sie verzichtet auf eine Reihe von Überprüfungen, kann aber als Orientierung dienen, wenn man auf einer höheren Schicht von ULIX die Funktionalität „list directory“ implementiert.

Sie hat zwei Parameter:

- `inodeno`: Inodenummer des Inodes, dessen Inhalt ausgegeben wird. Es muss sich um ein Verzeichnis handeln.
- `devno`: Gerätenummer der Disk, worauf sich der Inode befindet.

Die Funktion hat den folgenden Aufbau:

117a ⟨*Verzeichnis auflisten* 117a⟩≡ (115a)
```
void ls_dir(ino_t inodeno, dev_t devno)
{
    ⟨Lese den Inode 117b⟩
    ⟨Initialisiere das Array der Verzeichniseinträge 117c⟩
    ⟨Berechne die Anzahl der Einträge im Verzeichnis 118a⟩
    ⟨Durchlaufe das Array der Verzeichniseinträge 118b⟩
}
```

Zuerst lesen wir den Inode ein. Dafür verwenden wir die Funktion `get_inode` aus dem Modul `inode` (siehe Abschnitt 5.2, Seite 74).

117b ⟨*Lese den Inode* 117b⟩≡ (117a)
```
if (inodeno < ROOT_INO)
    return;

inode ino = { 0 };
if (get_inode(devno, inodeno, &ino) == NO_DEV)
    return;
```

Danach können wir ein Array von Verzeichniseinträgen vom Typ `dir_entry` initialisieren, das wir immer wieder durchlaufen werden: Hier befinden sich die Einträge, die wir ausgeben. Für die Initialisierung verwenden wir einen Trick der Sprache C: Zuerst deklarieren wir einen Block von `char` Elementen (Array `block`), den wir nach Bedarf mit einem Datenblock von der Disk belegen werden. Dieses Array casten wir dann zu einem Pointer vom Typ `dir_entry` (Variable `entry_array`), das wir später als Array von Verzeichniseinträgen verwenden. Dieser Pointer ist sozusagen die `dir_entry`-Sicht des Datenblocks `block` und erlaubt uns, sehr bequem die einzelnen Einträge per Index zu adressieren. In der Variablen `entry` speichern wir während des Durchlaufs über das Array den jeweiligen Eintrag.

117c ⟨*Initialisiere das Array der Verzeichniseinträge* 117c⟩≡ (117a)
```
char block[BLOCK_SIZE] = { 0 };
dir_entry *entry_array = (dir_entry *) (block);
dir_entry  entry       = { 0 };
```

Eine wichtige Größe ist natürlich die Anzahl der Einträge im Verzeichnis, die wir wie folgt berechnen können:

118a ⟨*Berechne die Anzahl der Einträge im Verzeichnis* 118a⟩≡ (117a)
```
int entries = ino.fsize / DIR_ENTRY_SIZE;
```

Nun können wir das Array der Verzeichniseinträge in einer Schleife durchlaufen und jeden Eintrag ausgeben. Die Kontrollvariable `i` zählt die Anzahl der Einträge, die wir bisher ausgegeben haben. `block_index` speichert den Index in der Blockliste des Inodes. In jedem Lauf der `while`-Schleife prüfen wir zuerst, ob wir einen neuen Datenblock laden müssen. Das ist immer dann der Fall, wenn `i` ein Vielfaches der Anzahl der Einträge in einem Block erreicht hat. Da wir `i` mit `0` initialisieren, wird auch der erste Block auf dieser Weise geladen ($0 \equiv 0 \bmod x$ für alle $x \in \mathbb{N}^+$). Der interessante Effekt beim Laden eines neuen Blocks ist, dass wir gleichzeitig das Array `entry_array` neu belegen, denn dieses ist ja nur eine andere Sicht des Datenblocks `block`. Im zweiten Teil der Schleife belegen wir die Variable `entry` und geben sie aus. Hier haben wir einen weiteren interessanten Effekt (falls man Modulo-Rechnung interessant findet). Nehmen wir die Variable `i` Modulo Anzahl der Einträge pro Block, so erhalten wir den richtigen Index im Array `entry_array`, unabhängig davon, wie groß `i` mittlerweile geworden ist.

118b ⟨*Durchlaufe das Array der Verzeichniseinträge* 118b⟩≡ (117a)
```
int block_index = 0;
int i = 0;
while (i < entries) {
    if (i % DIR_ENTRIES_PER_BLOCK == 0) {
        read_block(devno, ino.block[block_index], block);
        block_index ++;
    }

    entry = entry_array[i % DIR_ENTRIES_PER_BLOCK];
    printf("inode %u, name <%s>\n", entry.ino, entry.name);
    i++;
}
```

Um den Block zu laden verwenden wir die Funktion `read_block` aus dem Modul `block` (siehe 5.1).

B.2.1.4 Dateiinhalt ausgeben

Die nächste Funktion dient als Beispiel für die Implementierung einer Funktionalität, die normalerweise auf höheren Schichten von ULIX bzw. in „user mode“ Programmen implementiert wird: Inhalt einer Datei ausgeben. Der einzige Parameter ist der Dateideskriptor einer Datei, die geöffnet wurde.

118c ⟨*Dateiinhalt ausgeben* 118c⟩≡ (115a)
```
void print_file_content(int fd)
{
    char block[BLOCK_SIZE] = { 0 };

    while (read_file(fd, block, BLOCK_SIZE) > 0) {
        printf("%s", block);
```

```
        memset(block, 0, BLOCK_SIZE);
    }

    printf("-- END OF FILE --\n");
}
```

B.2.2 Dateisystem initialisieren und testen

Nun kommen wir zu der wichtigsten Funktion unseres Moduls, die aus der Funktion `main` des ULIX Codes aufgerufen wird. Ihr Parameter ist die Adresse eines „Multiboot“ Headers. Diese Adresse wird dem ULIX-Kernel vom Bootloader Grub übergeben und ist eine physikalische Adresse im Speicher. Da dieser Parameter als `void`-Pointer in der `main` Funktion deklariert ist, benutzen wir denselben Typ, um die ULIX-Datei `ulix.c` so wenig wie möglich zu verändern. Ein Konvertierung übernehmen wir selber.

Der Aufbau der Funktion ist der folgende:

119a ⟨*Dateisystem initialisieren und testen* 119a⟩≡ (115a)

```
void initialize_module (void *mboot_ptr)
{
    ⟨Sag Hallo 119b⟩
    ⟨Initialisiere den Multiboot Header 119c⟩
    ⟨Prüfe die Anzahl der Module 120b⟩
    ⟨Berechne die Adresse der RAM-Disk 121a⟩

    ⟨Initialisiere eine RAM-Disk 121b⟩
    ⟨Gebe den Superblock der RAM-Disk aus 121c⟩
    ⟨Setze das root-Gerät 121d⟩

    ⟨Gebe den root-Inode aus 122a⟩
    ⟨Inhalt des root-Verzeichnisses auflisten 122b⟩

    ⟨Test-Datei ausgeben 122c⟩

    ⟨Sag Tschüss 122d⟩
}
```

Zuerst begrüßen wir die Welt mit einer Nachricht, die angibt, dass unser Modul startet.

119b ⟨*Sag Hallo* 119b⟩≡ (119a)

```
printf("++++ INITIALIZING RAMDISK ++++\n");
```

Danach konvertieren wir den Parameter `mboot_ptr` zu einem Pointer auf eine Struktur vom Typ `multiboot_header`, der in `multiboot.h` deklariert ist (siehe Anhang B.1). Vorher prüfen wir, dass der Pointer nicht gerade Null ist, sonst werden wir einen Nullpointer dereferenzieren:

119c ⟨*Initialisiere den Multiboot Header* 119c⟩≡ (119a) 120a▸

```
if (! mboot_ptr) {
    printf("++++ Multiboot pointer is null\n");
    return;
}
```

Der übergebene Pointer `mboot_ptr` ist eine physikalische Adresse. Unsere Funktion wird aber nach dem Einschalten des Paging-Mechanismus aufgerufen. Das bedeutet, die Adresse ist mittlerweile nicht mehr gültig. Um die tatsächliche Adresse zu berechnen, verwenden wir den Makro `PHYSICAL`, der in `ulix.c` definiert ist. Dieser Makro gibt uns die virtuelle Adresse, die der ursprünglichen physikalischen Adresse entspricht, denn ULIX blendet in jedem Adressraum ab der virtuellen Adresse `0xd0000000` den physikalischen Speicher ein.

120a ⟨*Initialisiere den Multiboot Header* 119c⟩+≡ (119a) ◁119c

```
char *physical_boot = PHYSICAL((char*) mboot_ptr);

struct multiboot_header* mboot =
  (struct multiboot_header*) physical_boot;
```

Wenn wir ULIX starten, haben wir die Möglichkeit, dem Bootloader Grub die Verwendung von Modulen anzugeben. Ein Modul ist eine Datei, die vom Grub in den Speicher geladen wird. Diese Angabe erfolgt durch einen einfachen Eintrag der Konfigurationsdatei `menu.lst` von Grub. Hier ist der Inhalt dieser Datei auf unserem Testsystem:

```
#timeout 5
default 0
color light-blue/black light-cyan/blue

title Ulix-i386, the literate OS

kernel /boot/ulix.bin
module /boot/initrd

title Shutdown
     halt

title Reboot
     reboot
```

Man merke den Eintrag, der mit `module` anfängt. Dort geben wir den Pfad zur RAM-Disk an. Sie heißt auf unserem Testsystem `initrd` und ist eine Datei, die wir mit dem Programm `mkfs.ulixfs` erzeugt haben (siehe Kapitel 4). Grub lädt diese Datei in den Speicher und setzt entsprechende Werte (Anfang, Ende usw.) in dem Multiboot-Header. Auf anderen Testsystemen muss man diese Einträge in `menu.lst` eventuell anpassen. Der erste Wert, der uns interessiert, ist die Anzahl der Module. Falls die Anzahl Null ist, wurden keine Modulen übergeben. In unserem Fall sollte das nicht der Fall sein, aber wir prüfen sie Anzahl trotzdem:

120b ⟨*Prüfe die Anzahl der Module* 120b⟩≡ (119a)

```
u32int nmodules = mboot->mods_count;
if (nmodules < 1) {
    printf("++++ NO MODULES FROM GRUB! EXITING ++++\n");
    return;
}
```

Nun können wir den Multiboot-Header auslesen, um die Anfangs- und Endadresse der RAM-Disk zu finden. Wir werden hier nicht weiter auf die Spezifikation des Multiboot-Headers eingehen.

Siehe [10] und besonders [20] für mehr Informationen. Um die tatsächlichen Adressen zu berechnen, verwenden wir wieder den Makro `PHYSICAL`, denn alle Adresse im Multiboot-Header sind physikalisch.

121a ⟨*Berechne die Adresse der RAM-Disk* 121a⟩≡ (119a)
```
u32int *module_start = (u32int*) (PHYSICAL(mboot->mods_addr));
u32int *module_end   = (u32int*) (PHYSICAL(mboot->mods_addr+4));

u32int *initrd_start = (u32int*) (PHYSICAL(*module_start));
u32int  initrd_size  = *module_end - *module_start;
```

Die letzten zwei Variablen sind das, was wir eigentlich gebraucht haben, um das Dateisystem mit dem Inhalt der RAM-Disk zu initialisieren: Anfangsadresse der RAM-Disk und ihre Größe. Jetzt können wir zurück zu unseren Modulen kommen. Das erste was wir machen, ist den RAM-Disk-Treiber direkt anzusprechen um eine RAM-Disk mit dem Inhalt aus der Adresse `initrd_start` zu initialisieren. Dabei wird der Inhalt der vom Grub übergebenen Datei in die RAM-Disk kopiert (siehe 3.4.6.6, Seite 38). Nach der Initialisierung der RAM-Disk brauchen wir den Multiboot-Header nicht mehr und seine Daten können aus unserer Sicht überschrieben werden.

121b ⟨*Initialisiere eine RAM-Disk* 121b⟩≡ (119a)
```
dev_t ram_dev = rd_load(initrd_start, initrd_size);
if (ram_dev == NO_DEV) {
    printf("++++ Could not load RAMDISK ++++\n");
    return;
}
printf("++++ RAMDISK LOADED (%u bytes) ++++ \n", dev_size(ram_dev));
```

Nun befindet sich der ganze Inhalt der `initrd` Datei auf der RAM-Disk mit der Gerätenummer `ram_dev`. Zur Sicherheit geben wir den Superblock dieser Disk aus:

121c ⟨*Gebe den Superblock der RAM-Disk aus* 121c⟩≡ (119a)
```
superblock super = { 0 };
if (read_superblock(ram_dev, &super) == NO_DEV) {
    printf("++++ Could NOT read superblock ++++\n");
    rd_delete(ram_dev);
    return;
} else {
    dump_superblock(&super);
}
```

Zu diesem Punkt haben wir den Superblock erfolgreich gelesen. Wir können daher das root-Gerät des Dateisystems auf die RAM-Disk setzen. Dafür verwenden wir die Funktion `change_root` aus dem Modul `filetab` (siehe 5.3, Seite 77).

121d ⟨*Setze das root-Gerät* 121d⟩≡ (119a)
```
if (change_root(ram_dev, O_RDWR) == NO_DEV) {
    printf("++++ Could NOT change root of file system ++++\n");
    rd_delete(ram_dev);
    return;
} else {
```

```
    printf("++++ CHANGED ROOT TO DEVICE %u ++++\n", ram_dev);
}
```

Jetzt haben wir unser Dateisystem initialisiert. Zum Testen geben wir den root-Inode aus. Die Funktion `get_root` ist aus dem Modul `filetab`.

122a ⟨*Gebe den root-Inode aus* 122a⟩≡ (119a)

```
file_entry root;
if (get_root(&root) == -1) {
    printf("++++ Coult not read root inode ++++\n");
} else {
    dump_file(&root);
}
```

Und weil wir am testen sind, geben wir gleich den Inhalt des root-Verzeichnisses aus:

122b ⟨*Inhalt des root-Verzeichnisses auflisten* 122b⟩≡ (119a)

```
printf("++++ CONTENTS OF ROOT ++++\n");
ls_dir(ROOT_INO, ram_dev);
```

Nun können wir weiter gehen und die Datei ausgeben, die wir zum Testen in die RAM-Disk mit dem Programm `mkfs.ulixfs` eingefügt haben. Den Namen der Datei können wir fest angeben, denn er ist uns bekannt. Um die Datei zu öffnen verwenden wir die Funktion `open_file`, die im Modul open definiert ist (siehe 5.6). Die Funktion `print_file_content` haben wir weiter oben definiert. Am Ende schließen wir die Datei.

122c ⟨*Test-Datei ausgeben* 122c⟩≡ (119a)

```
char filename[] = "/data.txt";
int fd = open_file(filename, O_RDONLY);
if (fd != -1) {
    printf("++++ FILE DUMP ++++\n");
    print_file_content(fd);
    close_file(fd);
} else {
    printf("++++ COULD NOT OPEN FILE ++++\n");
}
```

Damit können wir das Modul verlassen und dies auch bekannt geben:

122d ⟨*Sag Tschüss* 122d⟩≡ (119a)

```
printf("\n++++ EXITING MODULE ++++\n");
```

Unten befindet sich der Output von ULIX, wenn wir alle unsere Module mit ULIX zusammenkompilieren und in der virtuellen Maschine QEMU ausführen. Unsere Ausgaben sind gemischt mit den Ausgaben von ULIX.

```
QEMU 1.2.1 monitor - type 'help' for more information
(qemu) QEMU 1.2.1 monitor - type 'help' for more information
(qemu)
xv6...
Serial port active
Serial port 2 active
```

```
Kernel page directory setup.
Ulix-i386 0.06
Paging activated (CR0, CR3 loaded).
Physical RAM (64 MB) mapped to 0xD0000000-0xD3FFFFFF.
FDC: fda is 1.44M, fdb is not installed
++++ INITIALIZING RAMDISK ++++
Difference in free_frames: -164.
++++ RAMDISK LOADED (664576 bytes) ++++

++++ SUPERBLOCK DUMP ++++
       Inodes : 64
       Inode bitmap: 2
       Block bitmap: 3
       Inode table : 4
       Data : 9
       Blocks total: 649

++++ CHANGED ROOT TO DEVICE 256 ++++

++++ FILE DUMP ++++
       Inode number : 1
       Device number : 256
       File position : 0
  -- Inode Information --
       File size : 192
       NLinks : 2
       block[0] = 9
       block[1] = 0
       block[2] = 0
       block[3] = 0
       block[4] = 0
       block[5] = 0
       block[6] = 0
       block[7] = 0
       block[8] = 0
       block[9] = 0
++++ CONTENTS OF ROOT ++++
inode 1, name <.>
inode 1, name <..>
inode 2, name <data.txt>
++++ FILE DUMP ++++
Welcher Lebendige, Sinnbegabte, liebt nicht vor allen
Wundererscheinungen des verbreiteten Raums um ihn das
allerfreuliche Licht - mit seinen Farben, seinen
Strahlen und Wogen; seiner milden Allgegenwart, als
weckender Tag. Wie des Lebens innerste Seele atmet es
der rastlosen Gestirne Riesenwelt, und schwimmt tanzend
in seiner blauen Flut - atmet es der funkelnde, ewigruhende
Stein, die sinnige, saugende Pflanze, und das wilde,
brennende, vielgestaltete Tier - vor allen aber der
herrliche Fremdling mit den sinnvollen Augen, dem
schwebenden Gange, und den zartgeschlossenen, tonreichen
Lippen. Wie ein König der irdischen Natur ruft es jede
Kraft zu zahllosen Verwandlungen, knüpft und löst unendliche
```

```
Bündnisse, hängt sein himmlisches Bild jedem irdischen
Wesen um. - Seine Gegenwart allein offenbart die
Wunderherrlichkeit der Reiche der Welt.
---- END OF FILE ----

++++ EXITING MODULE ++++
Setting Status Line.
initial_stack = 0xc01f0918
Starting Shell. Type exit to quit.
```

Literaturverzeichnis

[1] http://wiki.minix3.org/en/UsersGuide/DoingInstallation#Select_a_block_size.

[2] http://lxr.linux.no/linux+v3.7/include/uapi/linux/stat.h. Online.

[3] Doxygen Homepage. http://www.stack.nl/~dimitri/doxygen. Online.

[4] Homepage von literate programming. http://www.literateprogramming.com. Online.

[5] Literate programs. http://en.literateprograms.org. Online.

[6] Ulix Homepage. http://www1.informatik.uni-erlangen.de/ulix. Online.

[7] A. Cox. Linux allocated devices. http://www.kernel.org/doc/Documentation/devices.txt, 2009.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[9] D. E. Knuth. Literate programming. *The Computer Journal*, 27(2), 1984. Verfügbar als PDF online unter http://www.literateprogramming.com/knuthweb.pdf.

[10] J. Molloy. Roll your own toy UNIX-clone OS. http://www.jamesmolloy.co.uk/tutorial_html/8.-TheVFSandtheinitrd.html. Online.

[11] Oracle. Javadoc Tool Home Page. http://www.oracle.com/technetwork/java/javase/documentation/javadoc-137458.html. Online.

[12] N. Ramsey. Noweb Homepage. http://www.cs.tufts.edu/~nr/noweb. Online.

[13] N. Ramsey. *A One-Page Guide to Using* noweb *with LaTeX*. http://www.cs.tufts.edu/~nr/noweb/onepage.ps.

[14] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems – Design and Implementation*. Prentice Hall of India, 3rd edition, 2012.

[15] The IEEE and The Open Group. The Open Group Base Specifications Issue 6, IEEE Std 1003.1. http://pubs.opengroup.org/onlinepubs/009695299/basedefs/sys/types.h.html, 2004. Online.

[16] The IEEE and The Open Group. The Open Group Base Specifications Issue 6, IEEE Std 1003.1. http://pubs.opengroup.org/onlinepubs/009695299/basedefs/sys/stat.h.html, 2004. Online.

[17] The IEEE and The Open Group. The Open Group Base Specifications Issue 6, IEEE Std 1003.1. http://pubs.opengroup.org/onlinepubs/009695399/basedefs/fcntl.h.html, 2004. Online.

[18] The IEEE and The Open Group. The Open Group Base Specifications Issue 7, IEEE Std 1003.1-2008. http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/stddef.h.html, 2008. Online.

[19] Wikipedia. http://de.wikipedia.org/wiki/Tabelle_virtueller_Methoden. Online.

[20] www.gnu.org. Multiboot Specification version 0.6.96. http://www.gnu.org/software/grub/manual/multiboot/multiboot.html#Boot-information-format. Online.

Index

. (Verzeichniseintrag), 17
.. (Verzeichniseintrag), 17

Aushängen, 89

Bitmap, 11
    Bits setzen, 59
Block, 7
    Allokieren, 62
    Größe, 8
    Liste im Inode, 15
Block-Bitmap, 12, 46
    Schreiben, 50
Blockgerät, 8
`block_t`, 107
Bootblock, 9
Bootloader, 9

`change_root`, 86, 121
CWEB, 3

Datei, 8, 77
    Inhalt ausgeben, 118
    Lesen, 99
    Öffnen, 98
    Schreiben, 103
Datei Modus, 16
Dateideskriptor, 77, 78, 90
    Erzeugen, 81
    Reservierte, 80
Dateigröße, 14
Dateiname, 17
    Größe, 17
    Im Verzeichnis suchen, 91
    in Inode übersetzen, 93
Dateisystem
    Aufbau, 8
    Datentypen, 107
    Regionen, 9
Dateitabelle, 78
    Datenstruktur, 79
    Eintrag überschreiben, 83
    Eintrag löschen, 82
    Eintrag lesen, 83
    Eintrag sichern, 85
    freien Eintrag finden, 80
    Größe, 79
    root lesen, 86
    root setzen, 86
    Zugriffsposition setzen, 84
Dateityp, 16
Datenblock
    Kopieren, 64
Datenregion, 13, 47
    Schreiben, 53
Datentypen, 107
`dev_t`, 22, 107
`dir_entry`, 17
Disk, 8
    Aufbau, 8
    Regionen, 9
`dmap`, 28
`driver`, 24
    `create`, 24
    `delete`, 24
    `init`, 24
    `read`, 25
    `size`, 25
    `write`, 25

Einhängen, **87**, 88
Einhängepunkt, 87, 97
    Abfragen, 89
    Freigeben, 89
Einhängetabelle, 87, 88
Einheit, 21
Essayist, 2

`fcntl.h`, 112

`file_entry`, 79
`filetab`, 80
formatieren, 41

Gerät, 21
    Operationen, 24
    root-Gerät, 86
    Treiber, 24
Gerätenummer, 22
`gid_t`, 107
Grub, 113

Hard Link, 15
Hoare, 1

Inode, 13
    Allokieren, 62
    `atime`, 15
    `ctime`, 15
    Felder, 14
    `fsize`, 15
    `gid`, 15
    Initialisieren, 61
    Lesen, 60
    `mode`, 15
    `mtime`, 15
    `nlinks`, 15
    `uid`, 15
Inode-Bitmap, 12, 46
    Schreiben, 49
Inodetabelle, 13, 46, 47
    Schreiben, 51
`ino_t`, 17, 107

Knuth, 1

Literate Programming, 1–5
    Eigenschaften, 2
    Große Projekte, 4

Major-Nummer, 21
memory map, 54
memory mapping, 54
Minix, 5, 7, 11, 14
Minor-Nummer, 21, 36
`mode`, 16, 109
`mode_t`, 107
mounting, *siehe* Einhängen
`mounttab`, 88

Multiboot, 113

`nlink_t`, 107
`notangle`, 4
`noweave`, 3
noweb, 3
`NULL`, 112

`off_t`, 107
`O_RDONLY`, 112
ordwr`O_RDWR`, 112
owronly`O_WRONLY`, 112

`PATH_SEP`, 19
Pfad, 18
    in Inode übersetzen, 93
    Trennzeichen, 18
POSIX, 107, 108
Programming by Intention, 42, 56
`ptrdiff_t`, 112

RAM-Disk, 31
    Anzahl, 33
    Erzeugen, 36
    Inhalt laden, 38
    Initialisieren, 35
    Löschen, 36
    Lesen, 36
    Schreiben, 36
    Tabelle, 33
    Verwendung, 31
    Vom Grub laden, 121
Ramsey, 3
root, 12, 18, 50
    Berechnen, 51
    gid, 15
    Setzen, 86
    uid, 15
root-Gerät, 86
`ROOT_GID`, 15
`ROOT_UID`, 15

`size_t`, 34, 107, 112
`ssize_t`, 107
Superblock, 10, 45
    Ausgeben, 116
    Berechnen, 45
    `bmap`, 11
    `datadata`, 11

`imap`, 11
`inodes`, 10
`itable`, 11
Lesen, 69
`magic`, 10
nblocks`nblocks`, 11
Schreiben, 49
`sys/stat.h`, 108
`sys/types.h`, 107

`tangle`, 2
TEX, 1
`time_t`, 107
Treiber, 23, 24
Neue Eintragen, 28
Schnittstelle, 22
Treiberschnittstelle, 22
Treibertabelle, 25, 28

`uid_t`, 107
Ulix, 5, 33, 91
`unmount`, 89

Vaterverzeichnis, 17
Vererbung, 23
Verzeichnis, 17
Auflisten, 117
Verzeichnisbaum, 18
Pfad, 18
Verzeichnisdatei, 18
Verzeichniseintrag, 17

`wchar_t`, 112
`weave`, 2
WEB, 2

Zugriffsrechte, 15, 16