Analyse und Simulation von Fragmentierungseffekten beim "ReiserFS" Dateisystem

von
Constantin Loizides

im Dezember 2001

Diplomarbeit in Informatik

vorgelegt dem
Fachbereich Informatik
der
Johann Wolfgang von Goethe Universität
Frankfurt am Main

Hiermit erkläre ich, daß ich diese Arbeit selbständig und ohne fremde Hilfe angefertigt habe; außer der angegebenen Literatur habe ich keine weiteren Hilfmittel benutzt.

Inhaltsverzeichnis

Abbildungsverzeichnis

Vorwort

Die vorliegende Ausarbeitung beschreibt zum großen Teil die Untersuchungen, die ich im Rahmen meines Forschungsprojekts bei der Innovative Software AG in den zurückliegenden sechs Monaten unternommen habe.

Die Innovative Software AG setzt das fortschrittliche ReiserFS bereits seit längerem für ihre stark beanspruchten Hochverfügbarkeitssysteme ein und hat dazu die Portierung von ReiserFS auf die Alpha-Plattform finanziell gefördert. Dementsprechend besteht ein besonderes Interesse daran, daß die hohe Performanz von ReiserFS auch unter starker Belastung und nach langer Laufzeit nicht oder zumindest nur wenig nachläßt. Der Mangel an gesicherten Erkenntnissen über dieses Verhalten machte die Erstellung meiner Studie in der Forschungsabteilung der Innovativen Software AG möglich.

Die Arbeit ist durchgehend in der ersten Person Plural formuliert.[1] Ich möchte so den Leser in die Entwicklung und Ausarbeitung von Grundlagen und der von mir entwickelten Simulationen miteinbeziehen. In anderen Teilen der Arbeit, insbesondere bei Auswertungs- und Diskussionsteilen, ist die erste Person Plural als ein Ausdruck meiner Überlegungen zu verstehen, ähnlich der Ausdrucksweise einer Veröffentlichung von mehreren Personen. In rein informellen und deskriptiven Abschnitten verwende ich die dritte Person Singular.

Ungewöhnliche englische und auch deutsche Fachbegriffe oder Bezeichnungen setze ich beim ersten Erscheinen in Anführungszeichen. Englische Fachbegriffe benutze ich nur, wenn ich nach längerer Überlegung keine adäquate deutsche Übersetzung gefunden habe. Im Anschluß behandele ich sie dann wie Wörter der deutschen Sprache.

Frankfurt am Main, im Dezember 2001

Constantin Loizides

[1] Die erste Person Singular benutze ich nur hier und in der Danksagung.

1 Einleitung

Dateisysteme sind eine wichtige Softwarekomponente von Betriebssystemen. Sie stellen insbesondere im Kontext von UNIX-Betriebssystemen den Namensraum zur Benennung von Betriebssystemobjekten zur Verfügung. Gleichzeitig realisieren sie die Verwaltung des persistenten Datenspeichers. Bei der Entwicklung eines Dateisystems kommt es daher nicht nur auf Performanz und Flexibilität, sondern auch auf Zuverlässigkeit an.

Das Reiser-Dateisystem ist ein sehr modernes, unter LINUX entwickeltes Dateisystem, das jedes der genannten Kriterien auf innovative Weise zu lösen versucht. Im Unterschied zu den klassischen Dateisystemen verwaltet es seine Datenstruktur in Form eines einzigen balancierten Baums. Damit löst es das Problem konventioneller Dateisysteme, kleine Dateien gemeinsam mit großen effektiv zu verwalten. Zur Protokollierung von Metadatenänderungen reserviert es einen speziellen Bereich auf der Platte, indem es nach dem Konzept von Datenbanken eine effiziente Strategie zur Konsistenzerhaltung seiner Metadaten umsetzt.

Mit seinen Eigenschaften ist es dem unter LINUX sehr verbreiteten Ext2-Dateisystem in jeder Hinsicht überlegen. Es ist daher kein Wunder, daß es seit dem Erreichen einer stabil lauffähigen Version und der standardmäßigen Auslieferung mit dem LINUX-Kernel der Version 2.4 zahlreich Verwendung findet.

Bedingt durch die junge Entwicklungsgeschichte gibt es noch keine zuverlässigen Erkenntnisse über das Performanzverhalten des Reiser-Dateisystems nach langer und intensiver Benutzung. Praktisch jedes Dateisystem unterliegt alterungsbedingten Performanzverlusten, die sich je nach Dateisystem und Benutzungsweise unterschiedlich stark bemerkbar machen. Diese Alterungseffekte treten in der Regel in Form von externer Fragmentierung auf.

Das Ziel dieser Arbeit ist daher, eine Analyse und Simulation von Fragmentierungseffekten beim Reiser-Dateisystem vorzunehmen. Dazu stellen wir eine Methode vor, eine objektive Bemaßung der Fragmentierung eines Dateisystems vorzunehmen, die wir mit Performanzmessungen an gealterten Systemen vergleichen. Um die normalerweise erst nach intensiver Benutzung auftretenden Effekte festzustellen, implementieren wir zwei verschiedene Simulationstypen zur künstlichen Alterung eines Dateisystems. Die Ergebnisse eines der beiden Typen verwenden wir, um das Langzeitverhalten von Reiser-Dateisystemen im Vergleich zu dem konventionellen Ext2-Dateisystem zu studieren.

Alterung in Form von Fragmentierungseffekten tritt auf, wenn die Datenblöcke einer Datei weit verstreut auf dem Datenträger verteilt liegen, so daß insbesondere die Übertragungszeit beim sequentiellen Lesen sinkt. Dabei spielt der Umstand eine große Rolle, daß einem persistenten Speichermedium in der Regel ein mechanisches Gerät zugrundeliegt. Dieses Gerät ermöglicht eine Zugriffszeit in der Größenordnung

von Millisekunden, während beispielsweise die Zugriffszeit auf den Hauptspeicher in der Größenordnung von Nanosekunden liegt. Daher ist die Anordnung der Blöcke auf der Festplatte, die den effektiven Zugriff auf Daten und Metadaten dauerhaft gewährleistet, trotz der immer besseren Festplattentechnik ein überaus wichtiges Designziel eines Dateisystems. Wenn man typische Festplattendatenblätter von heute mit solchen von 1975 vergleicht, stellt man fest, daß sich seitdem die Speicherplatzgröße um den Faktor 10000, die Zugriffszeiten konstruktionsbedingt aber nur um den Faktor 10 verbessert haben [HR99]. Das bedeutet strenggenommen sogar, daß heutzutage eine geschickte Anordnung der Daten auf der Platte im Verhältnis noch wichtiger geworden ist als früher. Dateisysteme werden zudem durch ein anderes Nutzungs- und Erwartungsverhalten des Anwenders mit viel größeren Datenmengen und echtzeitkritischen Anwendungen stärkerer Belastungen ausgesetzt. Natürlich verbergen zugriffsoptimierte Anwendungen sowie Zwischenspeicherstrategien der an dem Zugriff beteiligten Betriebssystemsoftware- und Hardwarekomponenten auf unterschiedlichste Weise den Sachverhalt, daß auch heutzutage Festplatten ineffizient im Zugriff und performant im kontinuierlichen Datentransfer sind.

Die Menge der an dem Zugriff und Transfer beteiligten Komponenten spiegelt sich in der Gliederung der Arbeit wider. Wir untersuchen in Kapitel 2 zunächst die Hardwaretechniken von Festplatten und Blockgeräten im allgemeinen. Insbesondere sind dabei die Abschnitte 2.2.2 und 2.2.6 über die charakteristischen Eigenschaften einer Festplatte sowie der Abschnitt 2.2.8 über das Modell einer SCSI-Festplatte zu erwähnen. In Kapitel 3 wenden wir uns den Softwarekomponenten zu und betrachten zunächst allgemeine Betriebssystem- und Dateisystemkonzepte von UNIX mit dem Schwerpunkt auf LINUX, bevor wir in Abschnitt 3.3 detailliert auf den Pufferspeicher und das virtuelle Dateisystem von LINUX in der Version 2.4 eingehen. In Abschnitt 3.4 stellen wir knapp das SCSI-Gerätetreibermodell von LINUX vor, damit wir die Festplatte mit den Informationen aus 2.2.8 direkt ansteuern können. In den darauffolgenden beiden Abschnitten 3.5 und 3.6 untersuchen wir die Techniken der klassischen und modernen Dateisysteme im Hinblick auf ihre Datenstrukturen und arbeiten dabei die Unterschiede heraus. Zu guter Letzt gehen wir in 3.6.4 und 3.6.5 detailliert auf das Reiser-Dateisystem ein und stellen seine Datenstrukturen und Konzepte bei der Implementierung vor. In Kapitel 4 gehen wir auf das Zusammenspiel der Komponenten im Hinblick auf die zu untersuchende Fragestellung ein und definieren zunächst in Abschnitt 4.1 objektive Maße zur Bewertung eines Dateisystemzustands. In dem darauffolgenden Abschnitt 4.2 referieren wir verschiedene, veröffentlichte Untersuchungen zu Alterungseffekten bei FFS-Dateisystemen. Danach beschreiben wir in Abschnitt 4.3 zwei von uns entwickelte Simulationsverfahren. In Abschnitt 4.4 diskutieren wir schließlich die Meßergebnisse, die wir mithilfe eines der beiden Verfahren erzeugt haben. Im letzten Abschnitt 4.4.5 erläutern wir eine Methode zur Berechnung der Leseperformanzmeßwerte. Zuletzt geben wir in Kapitel 5 eine knappe Zusammenfassung der wichtigsten Ergebnisse an.

2 Festplattenhardware

In diesem Kapitel gehen wir auf Festplatten- und Schnittstellentechniken ein. Wir werden versuchen, die verschiedenen bestehenden Systeme in ihrer historischen Entwicklung zu verstehen und gleichzeitig eine gewissenhafte und logische Modellierung vorzunehmen. Im Vordergrund stehen die Zusammenhänge und nicht einzelne Details. Für diese wird an entsprechender Stelle jeweils auf Originalliteratur verwiesen. Eine Ausnahme stellt das SCSI-Interface dar, da dieses im Rahmen dieser Arbeit eine gewisse Rolle spielt.

2.1 Computer, Peripherie und Schnittstellen

Ein Computer besteht aus einer Anzahl von unabhängigen funktionalen Einheiten (siehe Abb. 2.1). Die wichtigsten darunter sind der Hauptprozessor (CPU), der Arbeitspeicher (RAM), Eingabe- und Ausgabeeinheiten und der Massenspeicher. Die CPU führt die Instruktionen eines Programms aus, welches dazu zur Ausführungszeit zusammen mit den zu verarbeitenden Daten im Hauptspeicher vorhanden sein muß. Das Programm wird vor der Ausführung vom Massenspeicher in den Hauptspeicher geladen. Die zu verarbeitenden Daten können von Eingabegeräten –beispielsweise einer Tastatur– oder auch vom Massenspeicher stammen und werden während und nach der Verarbeitung auf Ausgabegeräten –beispielsweise einem Monitor– dargestellt und eventuell nach Beendigung auf dem Massenspeicher persistent abgelegt. Während der gesamten Verarbeitungsphase greift die CPU in der Regel mehrmals auf den Hauptspeicher zu, um Instruktionen zu laden und Daten zu lesen und zu schreiben. Aus diesem Grund ist die CPU und der Hauptspeicher sehr eng gekoppelt: Der Zugriff ist relativ unkompliziert und vor allem besonders schnell. In Kontrast zum Hauptspeicher sind die Eingabe- und Ausgabegeräte sowie der Massenspeicher nicht so eng mit der CPU gekoppelt; daher gehören sie zur Peripherie: Der Zugriff auf diese Geräte ist komplizierter und in der Regel um Größenordnungen langsamer.

2.1.1 Massenspeicher und andere Peripheriegeräte

Ein Massenspeichergerät ist in der Lage, ein vielfaches der Hauptspeichergröße an Daten aufzunehmen. Darüber hinaus speichert es die Daten nicht flüchtig, so daß sie auch nach Ausschalten des Computers bzw. der Stromversorgung des Laufwerks erhalten bleiben und nach dem Einschalten wieder abrufbar sind.

Festplatten

Scheibenlaufwerke oder Festplatten (engl. „hard disk“) speichern Daten auf rotierenden Scheiben. Die Daten werden dazu in Blöcke fester Länge eingeteilt, von denen jeder einzelne relativ schnell –typischerweise im Bereich zwischen 10 bis 100

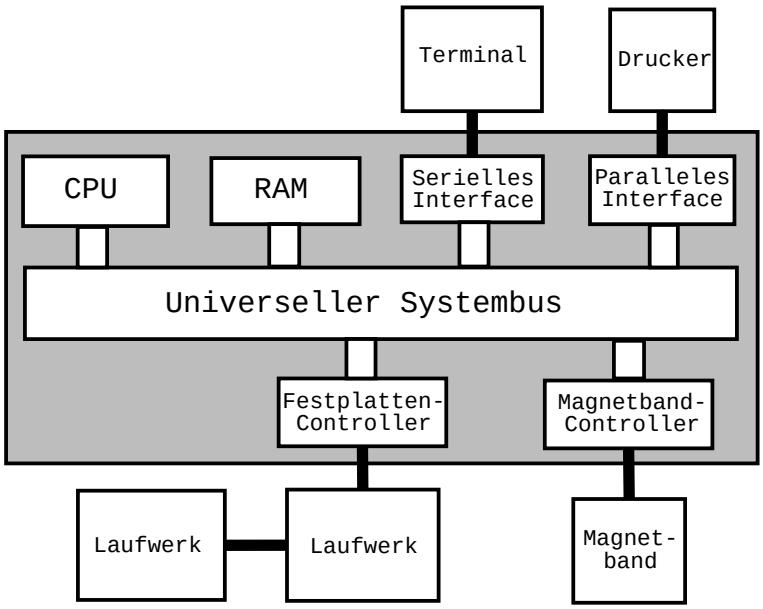

Abbildung 2.1: Computersystem mit Peripheriegeräten

Millisekunden (ms)– angesprochen werden kann. Daher nennt man derartige Massenspeichergeräte auch „beliebig adressierbar“ (engl. „random access mass storage device“). Unter den verschiedenen Massenspeichergeräten diesen Typs sind vor allem Festplatten, Wechselplatten, Disketten, magneto-optische Medien und die CD-ROM bzw. DVD-ROM zu nennen.

Bandlaufwerke

Im Gegensatz zu Festplatten schreiben Bandlaufwerke Daten sequentiell auf ein Magnetband. Die Zugriffszeit auf einen bestimmten Block an Daten hängt von der Position ab, an der sich der Lese- und Schreibkopf gerade befindet. Wenn es nötig ist, weiter vor- oder zurückzuspulen, dann kann die Zugriffszeit im Bereich von einigen Minuten liegen. Bandlaufwerke stellen typische sequentielle Massenspeichergeräte (engl. „sequential mass storage device“) dar. Von ihnen sind vor allem Streamer, Videokassetten, Audiokassetten und digitale Audiokassetten (engl. „digital audio tape“, DAT) zu nennen.

Eingabe-/Ausgabegeräte

Die wohl gängigsten Eingabe- und Ausgabegeräte sind die Tastatur und der Monitor zur Interaktion mit dem Benutzer. Weitere übliche Eingabegeräte sind Mäuse, Scanner und Mikrophone, weitere Ausgabegeräte beispielsweise Drucker und Lautsprecher. Aber auch Netzwerkverbindungen fallen unter diese Kategorie. Es gibt unzählige weitere Geräte, die Daten mit Computern austauschen. Sie gehören alle zur Peripherie und kommunizieren mit der CPU über verschiedenste, meist aber standardisierte Eingabe- und Ausgabeschnittstellen.

2.1.2 Peripherale Schnittstellen

Peripheriegeräte werden mit einem Computersystem über Schnittstellen (engl. „interface“) verbunden. Das abstrakte Modell einer peripheralen Schnittstelle besteht aus verschiedenen (Definitions-)Schichten, deren Abgrenzungen meist –insbesondere für ältere Typen– nicht besonders offensichtlich sind. Bei bestimmten Schnittstellen werden in der Praxis manche Schichten auch einfach weggelassen. In der Regel aber sind sie von unten nach oben aufgebaut, so daß die unteren Schichten notwendig für die Implementierung eines Interfaces sind. Wir halten uns im folgenden an ein Vierschichtenmodell, wie es zum ersten Mal für die Definition der SCSI-Schnittstelle vom ANSI-Komitee für SCSI-3 definiert wurde [Com01].

In der untersten Schnittstellendefinitionsschicht, der physikalischen Ebene, werden Kabel und Kabelanschlußtypen, Signalspannungen und Anforderungen an den Stromfluß für den Gerätetreiber definiert. Desweiteren werden logische und zeitliche Koordination der verschiedenen Signale des Busses beschrieben.

Direkt über der physikalischen Ebene befindet sich die Protokollschicht. Diese Protokolldefiniton einer Schnittstelle enthält beispielsweise Informationen über den Unterschied zwischen Benutzerdaten und Befehlsdaten und über den Austausch von Nachrichten zwischen den Geräten. Wenn korrupte Daten zu erkennen oder korrigieren sind, dann wird das ebenfalls in der Protokollebene definiert.

Über der Protokollschicht befindet sich die Modellebene des Peripheriegeräts. Hier wird das Verhalten der Geräte beschrieben, die über die Schnittstelle verbunden sind. Diese Beschreibung kann sehr detailliert und präzise sein. Der SCSI-Bus ist ein Beispiel für ein solches detailliertes Modell, bei dem zusätzlich zur Charakterisierung eines generellen SCSI-Gerätes auch bestimmte Geräte wie Festplatten, Bandlaufwerke, Drucker und ähnliche genau definiert werden.

Zuletzt gehen manche Schnittstellendefinitionen so weit, daß sie genaustens vorgeben, welche Befehle von den teilnehmenden Geräten verstanden werden müssen. Diese Befehlsebene bildet die oberste Schicht des gerade beschriebenen allgemeinen Schnittstellenmodells.

Der Begriff „Schnittstelle“ bezeichnet immer alle implementierten Schichten in ihrer Gesamtheit. Es gibt verschiedene peripherale Schnittstellen, die dieselbe physikalische Ebene, aber unterschiedliche Protokolle definieren. Es ist auch möglich, daß eine bestimmte Schnittstelle verschiedene Optionen in der physikalischen Ebene zuläßt.[1]

Ein illustratives Beispiel für ein solches Vierschichtenmodell ist eine Druckerschnittstellendefinition, wie sie in Abb. 2.2 zusammen mit der allgemeinen Definition eines Interfaces gezeigt ist. Die unteren zwei Schichten sind durch die Centronics-Schnittstellenbeschreibung definiert.[2] Sie stellt die physikalische Ebene und das Pro-

[1] Letztenendes ist diese modellhafte Beschreibung einer Schnittstelle ähnlich der des OSI-Referenzmodells bei Netzwerkschnittstellen und bringt dieselben Probleme bei der Interpretation und Anwendung mit sich.

[2] Diese parallele Schnittstelle war lange Zeit die Standardschnittstelle zum Anschließen von Druckern und einigen anderen Geräten. Mittlerweile ist sie durch den IEEE-1284 Standard für parallele Schnittstellen ersetzt (siehe http://www.fapo.com/ieee1284.htm).

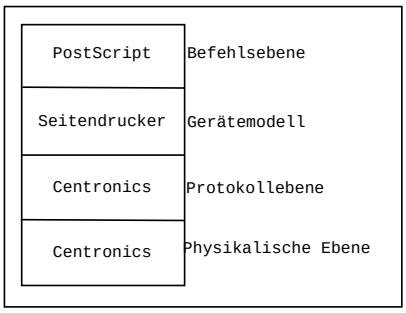

Abbildung 2.2: Vierschichtenmodell eines Druckerinterfaces

tokoll der Verbindung dar. Das konkrete Modell des Druckers beschreibt einen Seitendrucker, der im Unterschied zu einem Zeilendrucker aus den Daten im Hauptspeicher zunächst eine gesamte Seite generiert, bevor er sie an einem Stück ausdruckt. Als Seitenbeschreibungssprache wählen wir „PostScript“, ein Beispiel für einen umfangreichen und komplexen Befehlssatz.[3] Wie wir sehen, ist dieses Druckermodell vollständig, da es alle vier Schichten implementiert. Wenn wir einen Drucker mit einem derartigen Interface kaufen, macht es keinen Unterschied, welche Marke wir wählen, er wird mit jedem Computer zusammenarbeiten, der auch dieses Interface besitzt.

Ebenso wie die eben beschriebene Druckerschnittstelle stellen die IDE und die SCSI-Schnittstelle ein vollständiges Interface dar, auf die wir im folgenden noch genauer eingehen wollen. Beispiele für nicht vollständige Schnittstellen sind die bekannten seriellen RS-232-Schnittstellen[4] und die erwähnte parallele Centronics-Schnittstelle.

2.1.3 Busschnittstellen

Im Unterschied zu peripheralen Schnittstellen verbindet ein Computerbus verschiedenste Geräte und Komponenten innerhalb eines Computers miteinander. Jeder Computer benutzt eine gewisse Anzahl von internen Bussen. Diese transportieren Daten zwischen den Systemkomponenten vergleichbar mit dem Nervensystem eines Menschen. Die Grenze zwischen einer Busschnittstelle und einem peripheralen Interface ist nicht offensichtlich. Ein wichtiges Unterscheidungskriterium ist aber, daß ein Bus verschiedene Geräte mit gleicher Zugriffsberechtigung verbindet. [5]

Das Vierschichtenmodell aus dem letzten Abschnitt läßt sich ohne Änderung auf Busschnittstellen anwenden. Ein Computerbus ist definiert durch die Angabe der physikalischen Ebene, des Busprotokolls, des Gerätemodells und des Befehlssatzes.

Eine Charakterisierung der wichtigsten Eigenschaften eines Busses wie Durchsatz, Adreßraum, Echtzeitperformanz, elektrische und mechanische Spezifikationen und

[3] Siehe beispielsweise http://www.cs.indiana.edu/docproject/programming/postscript/postscript.html für eine Einführung.

[4] Siehe http://www.uwsg.iu.edu/usail/peripherals/serial/rs232/ für eine Beschreibung.

[5] Diese Definition unterscheidet beispielsweise die IDE-Schnittstelle von der SCSI-Schnittstelle. Darüber hinaus wären danach alle Speicherbusse keine Busse. Allerdings ist eine Diskussion derartiger Ausnahmen und Randfälle nicht Ziel dieser Arbeit.

Produktionskosten wird in [Sch95] gegeben. Ohne detailliert auf diese Begriffe einzugehen, ist es anhand der Begriffsgebung offensichtlich, daß es den optimalen Bus nicht gegen kann, so daß in der Praxis eine große Anzahl an spezialisierten Systemen wie Speicherbussen, I/O-Bussen und diversen Universalbussen gibt. Für eine Beschreibung konkreter Bussysteme –wie beispielsweise der bekannten ISA- und PCI-Busschnittstellen– sei auf [Dem01] verwiesen.

2.2 Festplatten und ihre Schnittstellen

Im Laufe der Zeit haben sich für Festplatten vier Schnittstellen zum Anschluß an den Computer bewährt, anhand derer sich auch die dazugehörigen Festplatten identifizieren lassen: die ST506/412-Schnittstelle, die ESDI-Schnittstelle, die SCSI-Schnittstelle und die IDE-Schnittstelle. Auf diese verschiedenen Schnittstellendefinitionen und ihre Modelle sowie die dazugehörenden Festplattentechniken werden wir nach einer kurzen geschichtlichen Einführung in die Entwicklung der Schnittstellen im folgenden detaillierter eingehen.

2.2.1 Historie der Festplattenschnittstellen

Festplattenschnittstellen waren vergleichsweise relativ früh standardisiert. Beginnend im Jahr 1975 wurden von der Firma CDC Festplattenlaufwerke mit einem Durchmesser von 14 Zoll und später mit 8 Zoll geliefert, die über die SMD-Schnittstelle mit dem Computer verbunden waren.

CDC verkaufte später ihre Laufwerkproduktion an Seagate. Mitte der achtziger Jahre wurde SMD als ein Ergebnis stetiger Verbesserungen das Standardinterface für hochperformante Laufwerke mit 8 Zoll Durchmesser. Die endgültige Version, SMD-E, hatte eine Transferrate von ca. 3 MB/s. Die Schnittstelle überlebte aber wegen ihrer breiten Kabel nicht den Sprung auf 5.25 Zoll Laufwerke und starb zusammen mit den 8 Zoll Laufwerken Anfang 1990 aus.

Fünf Jahre nach der Einführung von SMD stellte Seagate ein 5.25 Zoll Laufwerk mit einer Kapazität von 5 MB vor. Dieses ekonomische Laufwerk am unteren Ende der damaligen Leistungsskala benutzte ein neues Interface, die ST506/412-Schnittstelle.[6] ST506/412 wurde nicht neu entwickelt, sondern war eine Weiterentwicklung der Diskettenlaufwerksschnittstelle. Die Übertragungsrate wurde auf ca. 625 KB/s gesteigert, aber die Methode, den Kopf mit einem Schrittmotor zu steuern, blieb dieselbe. In den darauffolgenden Jahren wurde die Übertragungsrate verdoppelt. Trotzdem haben die Anforderungen modernerer Personalcomputer die Schnittstelle letztendlich überfordert. ST506/412 hat ab 1991 gegenüber SCSI und IDE stetig an Boden verloren.

Es war schon relativ früh offensichtlich, daß 5 1/4-Zoll-Laufwerke die Leistungsfähigkeit der ST506/412-Schnittstelle übersteigen würden. SMD hätte diese Lücke schließen können, war aber zu groß und zu teuer. 1983 brachte der Hersteller Maxtor die

[6] Eigentlich bedeutete ST506 die Modellbezeichnung der Festplatte. Das Nachfolgemodell war die ST412 mit 10 MB Kapazität und derselben Schnittstelle. Daher bürgerte sich der häufig gebrauchte Begriff ST506/412 für die Schnittstelle ein.

ESDI-Schnittstelle auf den Markt. ESDI benutzte dieselben Kabel wie ST506/412, hatte aber eine Übertragungsrate von 2.4 MB/s und darüber hinaus einen erweiterten Befehlssatz. Doch auch ESDI wurde letztendlich schnell von der flexibleren und leistungsfähigeren SCSI-Schnittstelle ersetzt.

Bereits 1984 begann die Entwicklung der IDE-Schnittstelle durch Compaq mit der Idee, den Festplattencontroller eines IBM kompatiblen Heimcomputers (IBM-PC/AT) auf dem Laufwerk zu integrieren. Gemeinsam mit Western Digital wurde ein ST506-Controller direkt auf einem Laufwerk befestigt und mit einem 40 poligen Kabel an den Systembus angeschlossen. Andere Festplattenhersteller erkannten schnell den Vorteil von IDE. Die Festplatten waren durch die Integration des Controllers nur unwesentlich teurer in der Herstellung; der externe Controller konnte dadurch aber gänzlich eingespart werden. Daher wurden Schritt für Schritt weitere IDE-Implementierungen entwickelt und damit entstanden vielfältige Abweichungen vom Industriestandard. Diese führte dazu, daß 1988 die ersten ANSI-Standardisierungvorschläge begannen. Als erstes wurde der Name in ATA geändert. Schließlich wurde 1994 der erste Standard ATA-1 verabschiedet. 1998 wurden dann durch die ATA-ATAPI-4 Standardisierung ATA- und ATAPI-Geräte vereinheitlicht. Momentan wird an der Spezifikation ATA-ATAPI-6 (NCITS Projekt 1410) und an ATA-ATAPI-7 gearbeitet. Die höheren Standards sind abwärtskompatibel, erlauben aber in der Regel eine deutlich höhere Übertragungsrate. Einen genauen Überblick über die Entstehung und Entwicklung des ATA/ATAPI-Standards gibt [Lan01].[7]

Die SCSI-Schnittstelle hat ihren Ursprung im Jahr 1979, als der Hersteller Shugart unter dem Namen „Shugart Associates Systems Interface“ ein neuartiges Interface zu entwickeln begann, daß logische Adressierung anstelle von physischer unterstützen sollte. Ein früher ANSI-Standardisierungsversuch dieser Schnittstelle schlug fehl. Nach einem weiteren Versuch und Umbenennung nach SCSI wurde 1986 schließlich SCSI-1 offiziell durch die ANSI standardisiert, lange nachdem es bereits Industriestandard geworden war. Die wachsende Zahl von SCSI-Produkten zeigte aber noch vor der ersten Standardisierung Schwachpunkte im Design, so daß noch vor 1986 mit der Arbeit an SCSI-2 begonnen wurde. SCSI-2 wurde schließlich 1994 zum ANSI-Standard und hat unter anderem eine Erweiterung des Busses von 8 auf 16 bzw. 32 Bit vollzogen. Mittlerweile wird an SCSI-3 gearbeitet, wobei vor allem eine Verfeinerung der Architektur für verschiedenste Geräte und Übertragungsarten vorgesehen ist. Einen guten Einstiegspunkt in das Architekturmodell von SCSI-3 findet man unter [Com01].

2.2.2 Das grundlegende Festplattenmodell

Eine Festplatte speichert Informationen auf einer Menge von rotierenden Scheiben. Diese Informationen können nahezu beliebig oft gelesen oder geschrieben werden und bleiben erhalten, wenn das Laufwerk ausgeschaltet wird.

[7] Siehe dazu auch die Projektseite des technischen Komitees http://www.t13.org/.

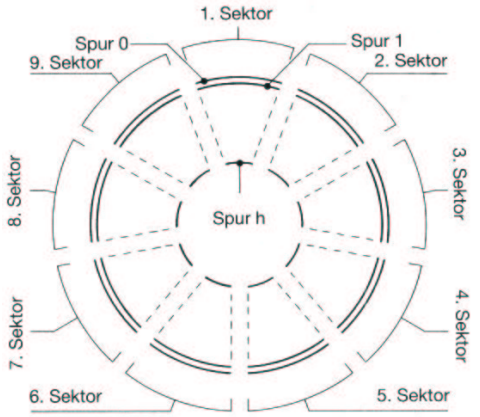

Abbildung 2.3: Die Spuren eines Mediums eingeteilt in Sektoren

Organisation des Mediums

Dazu besteht ein Laufwerk üblicherweise aus einer Anzahl lesbarer und beschreibbarer Oberflächen, auf denen die Daten in konzentrischen Kreisen gespeichert werden (siehe Abb. 2.3). Einen solchen Kreis nennt man „Spur“ (engl. „track“). Die Spuren werden weiter in sogenannte „Sektoren“ (engl. „sector“) unterteilt, die die kleinste beliebig adressierbare Einheit eines Mediums darstellen. Das Laufwerk greift auf einen Sektor zu, indem es zunächst den Lese- und Schreibkopf über der richtigen Spur positioniert und anschließend darauf wartet, daß der gewünschte Sektor unter dem Kopf vorbeirotiert, so daß es lesen oder schreiben kann. Der Lese- und Schreibvorgang erfolgt dabei seriell, Bit für Bit.

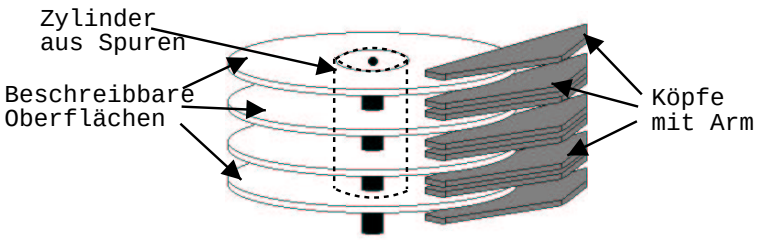

Abbildung 2.4: Ein Plattenstapel mit Köpfen geführt durch einen Arm

Gewöhnlicherweise beinhaltet ein Laufwerk zwischen zwei und acht Scheiben, und jede Seite einer Scheibe kann zur Speicherung genutzt werden (siehe Abb. 2.4). Dazu besitzt jede Oberfläche ihren eigenen Lese- und Schreibkopf, obwohl bei den meisten Modellen zu einem Zeitpunkt nur eine einzige Spur ausgelesen oder geschrieben werden kann. Die Köpfe werden gemeinsam –verbunden durch den sogenannten „Aktuator“– über die Spuren geführt. Die Menge der Spuren, auf die die Köpfe ohne weitere mechanische Justierung zugreifen können wird „Zylinder“ (engl. „cylinder“) genannt.

Eine Folge dieser Organisation ist, daß jeder Sektor auf der Festplatte eindeutig durch die Angabe der Zylinder-, Kopf- und Sektornummer adressiert werden kann. Nach den englischen Bezeichnungen „cylinder“, „head“ und „sector“ werden diese Zahlen „CHS-Koordinaten“ genannt. Die maximalen CHS-Koordinaten sind Kenngrößen der Platte und werden in der Literatur gewöhnlich als „Geometriedaten des Laufwerks“ (engl. „drive geometry“ oder „disk geometry“) bezeichnet.

Sektorformat

Um den Anfang einer Spur zu erkennen, gibt es in der Regel ein Interfacesignal „INDEX“, über welches in dem Moment ein kurzer Stromimpuls fließt, in dem der Kopf den ersten Sektor der Spur erreicht hat. Den Anfang der anderen Sektoren einer Spur signalisiert ein anderes Signal „SECTOR“. Ein Laufwerk wird als „hart gesteuert“ (engl. „hard sectored“) oder „weich gesteuert“ (engl. „soft sectored“) bezeichnet, je nachdem, ob zusätzliche Elektronik die relative Drehung der Scheiben bezüglich der Köpfe mißt, oder ob der Beginn der Sektoren durch die Köpfe vom Medium selbst gelesen wird.

Typischerweise verarbeitet ein Computer Daten parallel, gruppiert in Vielfachen von Bytes, nicht in Bit. Die Formatiereinheit ist ein Chip, der einerseits Sektoren anhand ihrer Sektornummer identifiziert und andererseits den Übergang zwischen den seriell arbeitenden Köpfen und dem parallel arbeitenden Computer darstellt, also Bits in Bytes gruppiert oder Bytes in Bitströme umwandelt. Der Datenseparator liegt zwischen den Köpfen und der Formatiereinheit und wandelt die digitalen Bitströme in analoge oder umgekehrt um. Zwischen ihm und den Köpfen befindet sich in der Regel noch ein Analogverstärker, der das gelesene oder zu schreibende Signal verstärkt. Die Kontrolleinheit (engl. „controller“) steuert den gesamten Ablauf und entscheidet über die auszuführenden Aktionen.

Ein Sektor besteht aus einer Anzahl verschiedener Felder, die zusammen als das „Sektorformat“ (engl. „sector format“) bezeichnet werden. Natürlich unterscheiden sich Sektoren von Interface zu Interface[8], aber ein typisches Format ist das folgende: Zuerst kommt ein Feld mit einem bestimmten Bitmuster, das den Datenseparator synchronisiert. Es wird gefolgt vom Adreßfeld des Sektors, das die CHS-Koordinaten angibt. Mit diesen Informationen kann der Controller bzw. die Formatiereinheit verifizieren, das sie den richtigen Sektor liest oder schreibt. Es folgt ein Feld mit einer CRC-Prüfsumme zum Sicherstellen, daß die Sektoradresse tatsächlich richtig gelesen wurde. Bis zu dieser Stelle gehören die Felder zum Vorspann (engl. „header“) des Sektors. Nach einem weiteren Synchronisationsfeld folgen schließlich die Nutzdaten. Darauf folgen in der Regel mehrere ECC-Felder zur Fehlerkorrektur. Dadurch kann der Controller testen, ob das Laufwerk die Daten korrekt geschrieben oder gelesen hat und je nach Typ und Länge des ECC einzelne Bits korrigieren. Ein Sektor endet gewöhnlich mit einer Lücke (engl. „gap“), um Geschwindigkeitsschwankungen auszugleichen. Die Größenangabe eines Sektors in Bytes bezieht sich immer auf die Nutzdaten im gerade beschriebenen niedrigformatierten Zustand (engl. „low level format“). Typische Werte sind 512, 1024 oder 2048 Byte. Je nach Sektorformat benötigen die restlichen Felder zwischen 40 und 100 Byte auf dem Medium. Es ist

[8] Bei integrierten Controllern ist das Sektorformat im Prinzip sogar frei definierbar.

nicht nötig, daß zwei Sektoren mit aufeinanderfolgenden Adressen auch wirklich auf dem Medium aufeinanderfolgen. Insbesondere der begrenzte Durchsatz der ersten Controller haben spezielle Techniken bei der Anordnung der Sektoren auf der Platte notwendig gemacht.

Sektorversatz

Die ersten Controller besaßen nur einen sehr kleinen lokalen Speicher, der gerade die Daten eines Sektors aufnehmen konnte. Dies führt dazu, daß sie die Daten an den Computer weitergeben muß, bevor er neue einlesen kann. Wenn diese Verarbeitungszeit länger als die Zeitdauer ist, die die Köpfe zum Überstreichen der Lücke zwischen den Sektoren benötigen, dann muß der Controller einen gesamten Umlauf (damals ca. 17 ms) abwarten, um den gewünschten Sektor zu bearbeiten. Um diese Verzögerung zu verhindern, ordnet man bei der Low-Level-Formatierung die Sektoren mit aufeinanderfolgenden Nummern zweckmäßigerweise nicht aufeinanderfolgend an, sondern versetzt sie um eine konstante Anzahl (engl. „interleave“). Ein Interleave von $1 : N + 1$ oder von $N + 1$ bedeutet, daß zwischen aufeinanderfolgenden logischen Sektornummern auf dem Medium N andere Sektoren liegen. Dies ist in Abb. 2.5 für einen Sektorversatz von zwei Sektoren gezeigt. Auf diese

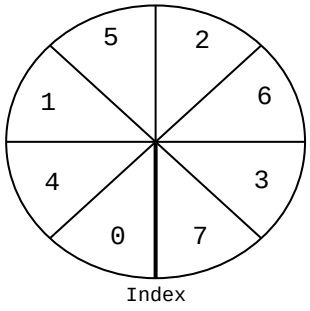

Abbildung 2.5: Sektorversatz von zwei Sektoren

Weise ist es auch bei langsamer Datenverarbeitungszeit des Controllers und ohne Zwischenspeicher (Cache) möglich, kontinuierlich logisch aufeinanderfolgende Sektoren einer Spur zu verarbeiten. Heutige Laufwerke haben mittlerweile meist einen Interleave von 1 : 1 und speichern normalerweise auch mindestens eine gesamte Spur an Sektoren im Datenspeicher bzw. Sektorspeicher auf dem Controller.

Spur- und Zylinderoffset

Um den höchsten Durchsatz für große Datenblöcke zu erreichen, plaziert der Controller oder das Betriebssystem normalerweise Daten auf einer Spur. Wenn diese nicht ausreicht, wird auf den nächsten Kopf im selben Zylinder gewechselt, solange bis der gesamte Zylinder voll ist. Der Grund für dieses Verhalten liegt darin, daß die Zeit für einen Kopfwechsel (engl. „head switch“) kürzer als die eines Spurwechsels (engl. „track switch“) ist, weil ersterer elektronisch und nicht wie letzterer mechanisch erfolgt.[9]

[9] Bei neueren Zonenplatten ist das anders (siehe unten).

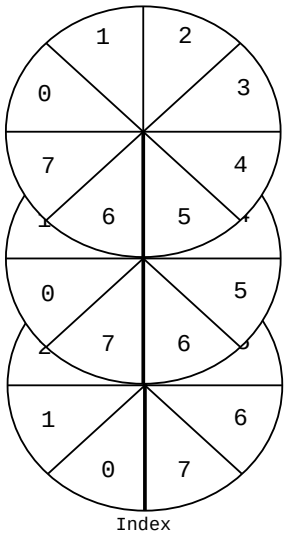

Abbildung 2.6: Spuroffset von einem Sektor

Allerdings kann je nach Rotationsgeschwindigkeit sogar die Verzögerung bei einem Kopfwechsel dazu führen, daß der logisch folgende Sektor verpaßt wird. Um das zu verhindern, kann man beim Sektorlayout die Sektoren von Kopf zu Kopf (engl. „track skew“) und von Zylinder zu Zylinder (engl. „cylinder skew“) um einen bestimmten Offset verschieben, wie in Abb. 2.6 für einen Spuroffset von einem Sektor gezeigt ist.

Technische Spezifikationen

Zwei verschiedene Kapazitätsangaben werden für ein Laufwerk benutzt. Die unformatierte Kapazität ist das Produkt aus Anzahl der Bits per Spur, Anzahl der Zylinder und Anzahl der Köpfe. Die formatierte Kapazität hängt vom Sektorformat und Layout ab und ist das Produkt aus Sektorgröße, Anzahl Sektoren pro Spur und Anzahl der Köpfe.

Die Transferrate bezieht sich auf die Geschwindigkeit, mit der die Köpfe die Bits seriell lesen und schreiben. Es ist einfach das Produkt aus Anzahl der Bits pro Spur und der Anzahl der Scheibenumdrehungen pro Sekunde. Der Durchsatz dagegen bezieht sich auf Nutzdaten und ergibt sich aus der Transferrate in Bytes geteilt durch den Interleave abzüglich ca. 10 Prozent für die Headerinformationen.

Die mittlere Zugriffszeit (engl. „average access time“) besteht vorallem aus zwei Komponenten. Die erste ist die durchschnittliche Suchzeit (engl. „average seek time“), um die Köpfe auf einen spezifischen Zylinder zu justieren. Die zweite ist die Latenzzeit, die die Köpfe nach der Justierung im Mittel auf den gewünschten Sektor warten müssen (engl. „mean rotational latency“). Sie ist demnach die Zeitspanne, die der Festplattenstapel für eine halbe Umdrehung benötigt. Die Latenzzeit ist eine sehr signifikante Größe, denn bei heute üblichen 5000 Umdrehungen pro Minute macht sie bereits 5.5 ms aus, was bei einer durchschnittlichen Suchzeit von 8-10 ms mehr als die Hälfte beträgt.

Die in diesem Abschnitt vorgestellte abstrakte Modellierung eines Festplattenlaufwerks sagt nichts über die tatsächliche physikalische Methode aus, mit der das Laufwerk lesend oder schreibend auf das Medium zugreift. Es steht damit im Prinzip gleichermaßen für Festplatten, magneto-optische Laufwerke, Disketten und Wechselrahmenplatten. CD-ROMs und DVD-ROMs gehören jedoch nur mit Einschränkung dazu, da sie Daten nur lesen, aber nicht schreiben können. Wir werden dieses Modell als Basismodell für eine Festplatte bei der Beschreibung der einzelnen Schnittstellentypen benötigen. Es entspricht beispielsweise völlig dem Modell einer ST506/412-Festplatte.

2.2.3 Plazierung der Schnittstelle

Funktional betrachtet besteht ein Festplattenlaufwerk mit zugehöriger Schnittstelle aus mehreren Subsystemen mit diversen Komponenten, die in Abb. 2.7 dargestellt sind. Zunächst gibt es das mechanische Laufwerk (engl. „head disk assembly“, HDA), das aus dem Medium, den Köpfen samt Aktuator, der analogen Datenelektronik und der Kopfpositionierungselektronik besteht. Die nächste Einheit bildet der Datenseparator, der das analoge Signal digitalisiert. Sie wird gefolgt von der Formatiereinheit, die einerseits die seriellen Daten in Form von Bitströmen entgegennimmt, zwischenspeichert und zu Bytes parallelisiert und andererseits Sektoren anhand ihrer Nummer identifiziert und gegebenenfalls weiterleitet. Die Kontrolleinheit (engl. „controller“) ist verantwortlich für die Steuerung der Köpfe und gibt die Lese- und Schreibbefehle. Als letztes folgt die Anbindung an den Systembus des Gastsystems (engl. „host system bus“) über einen Adapter (engl. „host adapter“).

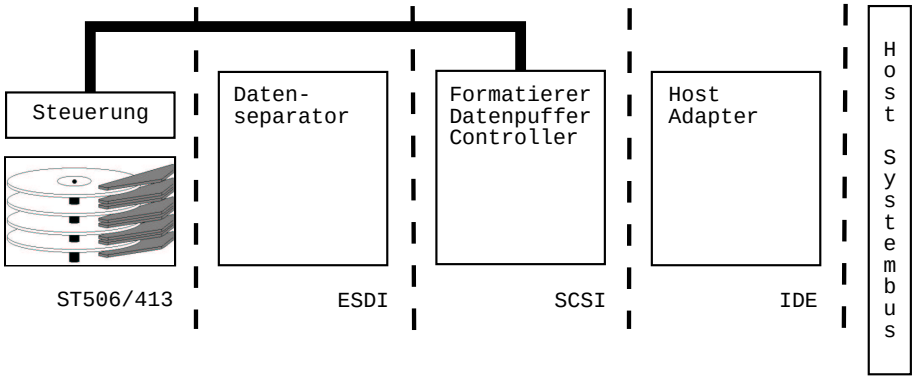

Abbildung 2.7: Verschiedene Schnittstellentypen und ihre Plazierung

Betrachten wir für einen Moment das Interface als das Kabel, das die Einheit, die vom Laufwerkshersteller erzeugt wurde und als Laufwerkseinheit bezeichnet wird, mit dem Computer verbindet. Wie in Abb. 2.7 zu sehen ist, gibt es beim Design des Laufwerks verschiedene Möglichkeiten, das Kabel zu plazieren. In der Entwicklung der Festplattenschnittstellen hat sich durchgesetzt, immer mehr Funktionalität in das Laufwerk selbst zu integrieren. Damit wandert das Kabel –und damit das Interface– immer weiter weg von der mechanischen Einheit der Platte.

Die ST506/412-Schnittstelle liegt zwischen der analogen Datenelektronik und dem Datenseparator. Daher bestimmt der Controller die analoge Methode, mit der die Daten auf das Medium kodiert werden. In der Praxis haben sich zwei Methoden dafür durchgesetzt, MFM und RLL. Die ESDI-Schnittstelle geht einen Schritt weiter und baut den Datenseparator in das Laufwerk ein. SCSI als viel allgemeineres Interface integriert die Formatiereinheit und den Controller selbst auf dem Laufwerk. Zuletzt wird bei IDE sogar fast der gesamte Hostadapter durch das Laufwerk ersetzt[10].

2.2.4 Die ST506/412-Schnittstelle

Wie oben schon durch Abb. 2.7 erläutert, befindet sich die ST506/412-Schnittstelle zwischen dem analogen Lese- und Schreibverstärker und dem Datenseparator, der ein Takt- und ein Datensignal aus den Pulsströmen generiert, die die Köpfe vom Medium einlesen.

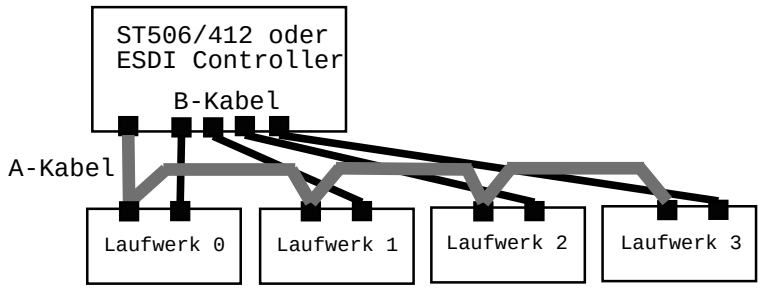

Abbildung 2.8: ST506/412 und ESDI Konfiguration

Physikalische Ebene

ST506/412 kann bis zu vier Laufwerke ansprechen (siehe Abb. 2.8). Zwei Flachbandkabel, bezeichnet mit „A“ und „B“, verbinden den Controller mit den Geräten. Das 34-adrige A-Kabel schließt dabei den Controller an die Laufwerke durchgehend in einer an jedem Ende terminierten Reihe. Es wird ausschließlich zur Signalgebung gebraucht. Das B-Kabel überträgt die analogen Daten über 25 Adern. Jedes Laufwerk wird dabei exklusiv über ein eigenes B-Kabel mit dem Controller verbunden. Um einen bestimmten Sektor zu lesen oder schreiben, muß die richtige Kopf-, Zylinder- und Sektornummer, sowie das Laufwerk selektiert werden. Dies geschieht einerseits durch entsprechende Signale am A-Kabel wie „DRIVE SELECT 1-4“ für das Laufwerk exklusiv und „HEAD SELECT 0-3“ für die bis zu 16 Köpfe[11] 0 bis 15.

[10] Dieser letzte Schritt hat im Prinzip den Nachteil, daß ein solches Laufwerk nur mit einem bestimmten Gastsystem funktioniert bzw. auf dieses angewiesen ist. Im Fall der IDE-Schnittstelle sind das IBM-PC/AT kompatible Computer, die allerdings mittlerweile so verbreitet sind, daß das heutzutage keinen wirklichen Nachteil mehr bedeutet.

[11] In der Originalspezifikation existierte „HEAD SELECT 3“ nicht. Diese Verbindung mit dem ursprünglichen Namen „REDUCED WRITE CURRENT“ wurde statt dessen zur Steuerung des Schreibstrompulses benutzt, da die inneren Spuren beim Schreiben weniger Strom benötigen als die äußeren. Als die Laufwerke dazu übergingen, den Stromfluß selbst zu regulieren, wurde dieses Signal nicht mehr benötigt und zu „HEAD SELECT 3“ umfunktioniert.

Die Zylinder werden wie bei einem Diskettenlaufwerk angesteuert. Ein Impuls am Signal „STEP“ sorgt dafür, daß die Köpfe durch einen Schrittmotor in die Richtung gesteuert werden, die am Signal „DIRECTION IN“ angezeigt wird. Ein Statussignal „SEEK COMPLETE“ zeigt an, daß die Positionierung der Köpfe erfolgt ist. Ein weiteres Statussignal „TRACK 00“ gibt an, ob sich die Köpfe auf dem äußersten Zylinder befinden. Das ST506/412-Interface unterstützt nur weiche Steuerung, daher gibt es nur das Signal „INDEX“ zur Anzeige des ersten Sektors einer Spur, die Sektorinformation befindet sich im Header, welcher bei der Low-Formatierung erzeugt wird. Im Prinzip können die Daten auf beliebig viele Weisen auf dem Medium kodiert werden. Ursprünglich wurde aber MFM verwendet mit einer Datenrate von 5 MHz (entspricht 625 KB/s) bei 17 Sektoren mit einer Größe von 512 Byte pro Spur. Später setzte sich mehr und mehr RLL durch mit einer Rate von 7.5 MHz (entspricht knapp 1 MB/s) und 22 Sektoren bei einer Größe von 512 Byte pro Spur.

Für diese Schnittstelle existiert weder eine ordentlich definierte Protokollschicht noch ein Befehlssatz, so daß die Modellierung gemäß des oben beschriebenen Vierschichtenmodells unvollständig ist. Desweiteren spezifiziert sie auch kein Gerät, so daß sie nur in Verbindung mit (dazupassenden) Festplatten gemäß des einfachen Modells aus Abschnitt 2.2.2 eingesetzt werden kann. Der Controller trägt die gesamte Verantwortung, das Laufwerk zu steuern. Die Methode der Pulssteuerung der Köpfe ist primitiv und sehr langsam. Typische Zugriffszeiten liegen zwischen 30 bis 60 ms. Durch die gepulste Steuerung ist die Anzahl der Zylinder im Prinzip unbeschränkt, hat in der Praxis aber 1024 aus BIOS-Kompatibilitätsgründen nicht überschritten.

Für weitere Details bezüglich der Signalbelegung und Steuerung der Festplatte durch den Controller sowie eine ausführlichere Beschreibung der Schnittstelle siehe [Sch95], Kap. 2.4 und [Mes93], Kap. 9.5.

2.2.5 Die ESDI-Schnittstelle

ESDI ist dazu ausgelegt, die Schwächen von ST506/412 zu überwinden. Die elektrischen und mechanischen Spezifikationen sind die gleichen wie bei der Vorgängerschnittstelle. Der Datenseparator befindet sich nun allerdings auf dem Laufwerk (siehe Abb. 2.7), was die maximale Datenrate auf bis zu 24 MHz erhöht.

Physikalische Ebene

ESDI benutzt als Nachfolger dieselben Kabel und Verbindungsstücke wie ST506/412 (siehe Abb. 2.8). Auch die Laufwerke sind identisch, und es besteht im Anschlußverfahren an den Computer äußerlich kein Unterschied zu den früheren Laufwerken, so daß sie mit altem BIOS einsetzbar sind.[12] Die Signalisierungsanweisungen sind aber deutlich anders. Zusätzlich zu den ST506/412 Signalen werden nun am A-Kabelstrang auch Signale zum Senden von Befehls- und Statusinformationen übertragen. Darüber hinaus gibt es ein Signal „SECTOR/ADDESS MARK FOUND“, was zur harten und weichen Steuerung des Laufwerks benutzt werden kann. Die

[12] Natürlich unterscheiden sich die Controller aber, so daß in der Praxis ein Mischen der beiden Systeme nicht bzw. nur mit speziellen Controllern möglich ist.

Daten am B-Strang gelangen nun digital –kodiert durch NRZ– zum Controller, da der Datenseparator auf dem Laufwerk sitzt. Daher sind die notwendigen Zeitsignale für das Lesen und Schreiben hinzugefügt. Die Daten werden seriell und synchron zu diesen Signalen übertragen. Neu gegenüber der alten Spezifikation ist zudem, daß nun auch am B-Kabel das Indexsignal „INDEX“ vorhanden ist. Das hat den Zweck, daß der Controller für jede angeschlossene Platte die relative Position der Scheiben erfahren kann, nicht nur für die über das A-Kabel selektierte. Falls mehrere I/O-Anfragen für verschiedene Platten anliegen, kann der er durch diese Informationen die Anfragen derart abarbeiten, daß die Zugriffszeit minimiert wird. Diese Optimierungsmethode wird gewöhnlich „Positionsabtastung“ (engl. „rotational position sensing“, RPS) genannt. Befehle und Statusinformationen können durch die Verwendung der zwei physisch getrennten Kabel gleichzeitig mit den Daten durch die Schnittstelle verarbeitet werden.

Protokollebene

ESDI-Befehle bestehen aus 16 Bit mit einem zusätzlichen Paritätsbit und werden durch ein Handshakeverfahren über das A-Kabel übertragen. Der Controller darf einen Befehl absetzen, wenn das Laufwerk am B-Strang „COMMAND COMPLETE“ aktiviert hat. Solange das Laufwerk mit dem Abarbeiten eines Befehls beschäftigt ist, ist „COMMAND COMPLETE“ inaktiviert. Das Laufwerk signalisiert seinerseits dem Controller durch das Setzen von „ATTENTION“ im A-Strang, das es etwas zu melden hat. Der Controller kann als Antwort durch das Senden des Befehls „REQUEST STATUS“ den Grund erfragen.

Modellebene

Das ESDI-Modell einer Festplatte entspricht dem einfachen Modell aus Abschnitt 2.2.2. Eine Festplatte präsentiert sich gegenüber dem Controller durch die Angabe ihrer Geometriedaten. Maximale Werte sind 16 Köpfe, 4095 Zylinder und 256 Sektoren pro Zylinder. Mit erweiterten Adressierungsmethoden kann die Anzahl der Zylinder auf 65535 erhöht werden. Der höchste Zylinder wird immer dazu benutzt, eine Liste defekter Sektoren zu speichern. In der Regel wird durch den Laufwerksherstellers darin eingetragen, welche Sektoren für die Speicherung von Daten unbrauchbar sind.[13]

Es gibt darüber hinaus auch eine ESDI-Spezifikation für Magnetbandlaufwerke. Dies stellt einen ersten Schritt in die Richtung dar, Schnittstellen unabhängig von den sie verwendeten Geräten zu definieren.[14]

Befehlsebene

Ein ESDI-Befehl von insgesamt 16 Bit besteht aus einem Befehlscode von 4 Bit gefolgt entweder Befehlserweiterungen oder Parameterdaten. Durch den Befehl „SET HIGH ORDER VALUE“ können auch 16 Bit an Parameterdaten übergeben werden, indem dieser mit den vier höchst signifikanten Bits, gefolgt von dem eigentlichen Befehl mit den restlichen 12 Bit gesendet wird. Wichtige Befehle sind „SEEK“, um den

[13] Derartige Listen gibt es, seitdem Festplatten hergestellt werden. Ursprünglich wurden sie in gedruckter Form auf Papier dem Laufwerk beigefügt. Bei der Low-Level-Formatierung mußte der Anwender die fehlerhaften Sektoren manuell angeben.

[14] Allerdings ist ein solches Magnetband nie produziert worden.

Kopf zu positionieren und „TRACK OFFSET“, um die gewünschte Spur zu selektieren. Danach ist es Aufgabe des Controllers, die entsprechenden Adreßmarkensignale abzufangen, bis der gewünschte Sektor gefunden ist und die entsprechenden NRZ-Daten zu dekodieren.

Weitere Details bezüglich der Signalbelegung und Steuerung der Festplatten durch den Controller sowie eine vollständige Beschreibung der Schnittstelle entnehme man [Sch95], Kap. 2.4 und [Mes93], Kap. 9.5.

2.2.6 Die IDE-Schnittstelle

Die wesentliche Neuerung der IDE-Schnittstelle ist bereits in Abschnitt 2.2.1 erwähnt und an Abb. 2.2.3 dargelegt worden. Abb. 2.9 zeigt diese fundamentale Verschiebung des Funktionalitätsprinzips von IDE noch einmal. Die Schnittstelle bedient nicht mehr den Gastrechner, sondern die Peripherie. In diesem Sinne ist der gesamte Controller und die Hauptfunktionen des Hostadapters auf dem Laufwerk integriert. Die einzigen Komponenten, die auf dem IBM-PC/AT verbleiben, sind einige Treiber- und Decodiereinheiten. In diesem Sinne stellt nach Abschnitt 2.1 das IDE-Interface eher einen Schnittstellenbus als eine peripherale Schnittstelle dar. Trotzdem wird im allgemeinen IDE nicht als ein I/O-Bus angesehen, weil es keine universale Adressierungsmethode zum Ansprechen der verschiedenen Geräte gibt. IDE kann nur ein oder zwei Festplatten bedienen und erlaubt nur den Zugriff eines Gastrechners auf die Platten. Wie bereits in Abschnitt 2.2.1 erwähnt, heißt der Standard der IDE-Schnittstelle eigentlich ATA. Wir halten uns in der folgenden Schnittstellenbeschreibung an die erste Version aus dem Jahr 1994, die mittlerweile als Standard zurückgezogen worden ist [Com94].

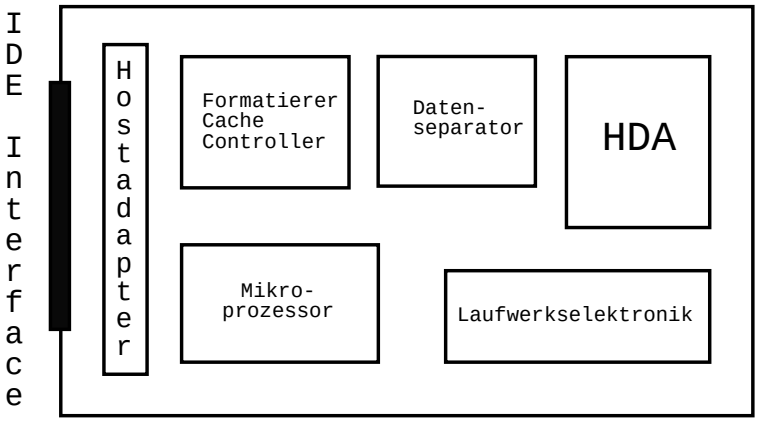

Abbildung 2.9: Blockdiagramm eines IDE-Laufwerks

Physikalische Ebene

IDE kann bis zu zwei Laufwerke unterstützen, die hintereinandergeschaltet mit dem AT-Bus (früher ISA heute PCI) über ein 40-poliges Kabel verbunden sind. Das erste Laufwerk „Drive 0“ wird in der Industrie als „Master“ und das zweite „Drive 1“ als „Slave“ bezeichnet. Mittels der Signale „DRIVE CHIP SELECT 0“ und „DRIVE

CHIP SELECT 1“ bzw. „HOST CHIP SELECT 0“ und „HOST CHIP SELECT 1“ wird der Kontrollregisterblock und der Befehlsregisterblock unterschieden. Ersterer wird bei Adressierung des AT-Busses zwischen 1F0h und 1FFh durch den Hostadapter aktiviert, letzterer durch Zugriff auf den Bereich von 3F0h bis 3FFh. Mittels dreier „DRIVE ADDRESS BUS 0-2“ bzw. „HOST ADDRESS BUS 0-2“ Signale kann der Host bestimmte Kontroll- oder Befehlsregister des Laufwerks adressieren. Die Daten fließen über die 16 Leitungen „DRIVE DATA BUS 0-15“ bzw. „HOST DATA BUS 0-15“.[15] Über „DRIVE I/O READ“ und „DRIVE I/O WRITE“ bzw. „HOST I/O READ“ und „HOST I/O WRITE“ wird der Lese- und Schreibzugriff auf Laufwerksregister per Handshakeverfahren ermöglicht. „DRIVE ACTIVE/DRIVE 1 PRESENT“ ist gemultiplext und erfüllt zwei Aufgaben. Kurz nach Einschalten oder Resets[16] der Laufwerke muß das Laufwerk 1 über dieses Signal anzeigen, daß es im System vorhanden ist.[17] Im Normalfall gibt dieses Signal dann an, welches Laufwerk aktiv ist. Über „DMA REQUEST“ und „DMA ACKNOWLEDGE“ kann über Handshake ein Direktzugriff auf den Hauptspeicher vereinbart werden. Über „I/O CHANNEL READY“ kann der Controller momentan den Zugriff auf die Register durch den Host sperren. Durch „DRIVE INTERRUPT REQUEST“ bzw. „HOST INTERRUPT REQUEST“ wird dem Host außer bei Direktzugriffen mitgeteilt, daß ein Befehl ausgeführt worden ist.

Der ATA-1-Standard definiert damit zwei Arten der Datenübertragung: Programmierbare Eingabe/Ausgabe (PIO) und Hauptspeicherdirektzugriff (DMA) mit verschiedenen Geschwindigkeiten, die „Mode“ genannt werden. Bei ersterem muß jeder I/O-Zugriff auf die Controllerregister extra programmiert werden. Er benötigt daher verhältnismäßig viel CPU-Zeit des Hosts und darüber hinaus viele Interrupts. Bei letzterem initialisiert der Host einen Datendirektzugriff auf den Hauptspeicher und wird erst nach Beendigung des Transfers durch einen Interrupt wieder benachrichtigt.[18]

Protokollebene

Für den Host sieht der IDE-Controller aus wie ein früherer ST506/412-Controller, allerdings mit dem Unterschied, daß der IDE-Controller nun im I/O-Adreßraum statt im Speicheradreßraum liegt. Dort präsentiert er dem Host zwei I/O-Registerblöcke, einen Kontrollblock mit Steuerregistern zur Laufwerkskontrolle und einen Befehlsblock[19] mit Registern zum Absetzen von Befehlen an die Festplatte und zum Datenaustausch.[20] Das Datenregister (1F0h, lesbar/schreibbar) wird im PIO-Modus zum Datenaustausch benutzt. Das Fehlerregister (1F1h, lesbar) beinhaltet den Feh-

[15] Dabei gibt das Signal „DRIVE 16 BIT I/O“ bzw. „HOST 16 BIT I/O“ an, ob tatsächlich ein 16 Bit Zugriff erfolgt.

[16] Dieser ist über Signal „DRIVE RESET“ bzw. „HOST RESET“ durch den Host erzwingbar.

[17] Wenn es vorhanden ist, signalisiert es Laufwerk 0 über „PASSED DIAGNOSTICS“ das Ergebnis seines Selbsttests.

[18] Man unterscheidet zwischen „Single-word DMA“ und „Multiple-word DMA“. Bei letzterem werden im Unterschied zu ersterem gleich mehrere Worte ohne Eingriff der CPU übertragen, so daß der Direktzugriff erst dann einen richtigen Geschwindigkeitsvorteil bringt.

[19] Dieser wird häufig auch als „AT-Aufgabendatei“ (engl. „AT task file“) bezeichnet.

[20] Welcher Block aktiviert ist, wird durch die Signale „DRIVE CHIP SELECT 0“ und „DRIVE CHIP SELECT 1“ wie oben erwähnt automatisch durch die I/O-Adresse gesteuert.

lercode, wenn im Statusregister (1F7h, lesbar) das Fehlerbit gesetzt ist. Das Eigenschaftsregister (1F1h, schreibbar) kann vom Host benutzt werden, um bestimmte Einstellungen am Laufwerk vorzunehmen. Das Sektoranzahlregister (1F2h, lesbar/schreibbar) beinhaltet die Anzahl der zu lesenden bzw. zu schreibenden Sektoren. Die für uns wichtigsten Register sind die Adreßregister bestehend aus dem Sektorzahlregister (1F3h, lesbar/schreibbar), das die Nummer des ersten zu übertragenden Sektors beinhaltet, dem Zylinderzahlregister (1F4h und 1F5h, lesbar/schreibbar), das den adressierten Zylinder angibt, und dem Kopf/Laufwerkregister (1F6h), mit dem ein Kopf auf einem Laufwerk selektiert wird. Das Befehlsregister (1F7h, schreibbar) empfängt die auszuführenden IDE-Befehle, auf die wir unten eingehen werden. Vom Kontrollblock werden nur wenige Register benutzt. Das alternative Statusregister (3F6h, lesbar) enthält dieselben Informationen wie das Statusregister.[21] Durch Gerätekontrollregister (3F6h, schreibbar) kann der Host einen Softwarereset auslösen oder Interrupts blockieren. Das Laufwerkadreßregister (3F7h, lesbar) schließlich beinhaltet laufend aktualisierte Informationen über die Ausführung des gerade aktiven Befehls. Es gibt fünf verschiedene Klassen von IDE-Befehlen. Jede Klasse wird durch ein anderes Signalisierungsprotokoll umgesetzt, auf das wir hier nicht eingehen werden. Auch die Initialisierung nach Einschalten der Geräte oder nach Softwarereset wird je nach Konfiguration durch verschiedene Protokolle vollzogen (siehe [Sch95], Kap. 6.2 und 6.3) und den Standard selbst [Com94]).

Modellebene

Das Diskmodell der IDE-Schnittstelle entspricht weitestgehend dem der ST506/412-Schnittstelle aus 2.2.2. Allerdings enthält das durch den ATA-Standard definierte IDE-Modell einige signifikante Verbesserungen.

Ein Medium ist weiterhin organisiert durch Köpfe, Zylinder und Sektoren. Der Standard erlaubt 16 Köpfe, 65636 Zylinder und 255 Sektoren, die normalerweise 512 Byte an nutzbaren Daten enthalten. Es gibt zwei verschiedene Adressierungsarten:

der CHS-Modus, mit dem die meisten älteren Laufwerke arbeiten und der jüngere LBA-Modus, bei der von der Geometrie der Platte abstrahiert wird. Sie werden auf unterschiedliche Weise in den Adreßregistern kodiert und somit durch das Laufwerk erkannt. Im physikalischen Adressierungsmodus bzw. im CHS-Modus hat das Laufwerk zwei Operationsmöglichkeiten. Im „natürlichen Modus“ stellt das Laufwerk seine Geometriedaten dem Host so dar, wie sie tatsächlich vorliegen. Im „Translationsmodus“ übersetzt es aus Kompatibilitätsgründen zu ST506/412 die physikalischen Adressen in logische mit bis zu 255 Köpfen und dafür nur 17 Zylindern bei gleicher Kapazität. In diesem Modus muß dem Host die Abbildung zwischen den physikalischen Geometriedaten und der verschobenen Geometrie, also das sogenannte „Mapping“, bekannt sein, da er laut ATA-Spezifikation auch mehrere Sektoren gleichzeitig ansprechen kann. Der erste logische Sektor im Translationsmodus ist immer dem ersten Sektor auf dem nullten Zylinder und nulltem Kopf zugeordnet.

Es gibt nach [Bög96] drei bekannte Mappingarten. Beim „vertikalen Mapping“ wird ein physikalischer Zylinder gänzlich benutzt, bevor auf einen neuen physikalischen

[21] Ein Auslesen dieses Registers hat im Unterschied zum Auslesen des Statusregisters keine Auswirkung auf wartende Interruptanfragen.

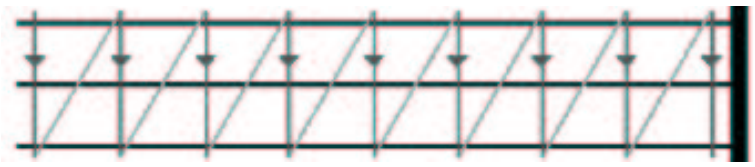

Abbildung 2.10: Vertikales Mapping

Zylinder gewechselt wird (siehe Abb. 2.10). Dieses Mapping wird hauptsächlich von älteren Festplatten verwendet, bei denen die Köpfe von Schrittmotoren von einem Zylinder zum nächsten bewegt werden. Die Lage der Spuren auf der Festplatte ist dabei durch die Schrittweite und die Justage der Schrittmotoren vorgegeben. Etwaiges Spiel in den Schrittmotoren versucht man dabei dadurch auszugleichen, daß zwei oder mehr Motorschritte notwendig sind, um von einer Spur zur nächsten zu gelangen. In jedem Fall sind bei diesen Platten daher rein logische Umschaltungen von einem Kopf zum nächsten schneller als Bewegungen des Kopfes, die den Einsatz von träger Mechanik notwendig machen. Beim „horizontalen Mapping“ werden zu-

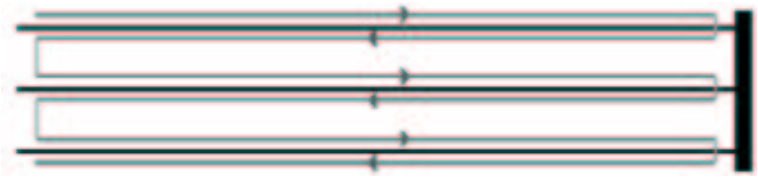

Abbildung 2.11: Horizontales Mapping

nächst alle physikalischen Spuren einer Plattenoberfläche bzw. eines Kopfes benutzt, bevor ein Kopfwechsel vollzogen wird (siehe Abb. 2.11). Dieses Verfahren ist bei modernen Festplatten schneller. Diese haben keine Schrittmotoren mehr, sondern verwenden schnellere Linearmotoren, die außerdem eine höhere Spurdichte zulassen. Die Erkennung von Spuren auf der Platte erfolgt durch Messung der Signalstärke am Lesekopf und dynamisches, stufenloses Nachregeln der Spurlage. Die Originalspuren, nach denen sich die Regelelektronik richtet, werden dabei vom Hersteller bei der Produktion der Rohlinge vor der Montage der Plattenstapel aufgebracht. Bedingt durch Produktionstoleranzen kann es dabei vorkommen, daß die Spurlage zwischen übereinanderliegenden Spuren eines Zylinders nicht ganz paßgenau ist, da die einzelnen Scheiben getrennt hergestellt werden. Ein Wechsel zwischen verschiedenen Köpfen eines Zylinders würde bei solchen Festplatten also ein Nachregeln der Spurlage erfordern und ist langsamer als ein horizontaler Wechsel der Spur.

Im Unterschied zum CHS-Modus wird im LBA-Modus eine lineare Abbildung der physikalischen Geometrie auf logische Blocknummern zwischen Null und einer Maximalzahl vorgenommen. Die Maximalzahl hängt von der Geometrie der Platte ab. In diesem Modus präsentiert sich das Laufwerk demnach als kontinuierliche Folge von Sektoren bzw. Blöcken, die durch Angabe ihrer LBA-Zahl adressiert werden. Nach [Com94] und [Com96] berechnet sich das Mapping als Funktion vom aktuellen

Zylinder „C“, aktuellen Kopf „H“ und aktuellem Sektor „S“ durch

$$\mathrm{LBA} = \mathrm{LBA}(C, H, S) = [(C \cdot N_H + H) \cdot N_S] + S - 1\,, \tag{2.1}$$

wobei N_H die Anzahl der Köpfe und N_S die Anzahl Sektoren pro Spur im gewählten CHS-Modus bzw. Translationsmodus darstellen. Der erste logische Block ist dabei immer dem ersten logischen CHS-Sektor im gewählten Modus zugeordnet. Dieses Mapping impliziert, daß die Zugriffszeit von LBA n auf LBA $n+1$ kürzer ist als von LBA n auf LBA $n+2$. Anders formuliert, die logischen Blöcke sind auch sequentiell angeordnet bezüglich ihrer Zugriffszeit. Das ist insofern wichtig für den Host, als daß lange Dateien demnach in kürzest möglicher Zeit gelesen oder geschrieben werden können, wenn sie aus aufeinanderfolgenden logischen Blöcken bestehen.[22]

Durch Mapping, sei es in Form von Translation oder LBA, ist es möglich, Laufwerke zu konstruieren, die nicht dieselbe konstante Anzahl von Sektoren pro Zylinder über die gesamte Oberfläche verteilen müssen, sondern außen mehr Sektoren pro Zylinder unterbringen als weiter innen. Bei dieser Technik, „Zone Bit Recording“ (ZBR) genannt, wird die Festplatte in mehrere Zonen mit jeweils konstanter Anzahl Sektoren pro Zylinder eingeteilt. Da die Rotationsgeschwindigkeit der Festplatte konstant ist, der Umfang des Mediums auf den äußeren Spuren jedoch bis zu zweimal größer als auf den inneren Spuren ist, sinkt bei konstanter Sektorzahl pro Spur die Aufzeichnungsdichte der Daten auf den äußeren Spuren: Die Bits werden gewissermaßen „breiter“ aufgezeichnet. Erhöht man den Takt des Schreibsignals am Schreib-/Lesekopf und damit die Datenübertragungsrate der Festplatte auf diesen Spuren, kann man bei gleicher Aufzeichnungsdichte mehr Sektoren auf der Platte unterbringen. In der Praxis unterteilen Hersteller ihre Festplatten in zehn bis zwanzig Zonen. Innerhalb einer Zone werden alle Spuren mit jeweils derselben Bitrate und Sektorzahl angelegt, wobei die Anzahl der Sektoren pro Spur in der äußersten Zone am größten ist. Entsprechend ist in dieser Zone in den meisten Fällen auch die Datenrate der Festplatte am größten.

Abbildung 2.12: Zonenweises horizontales Mapping

Horizontales Mapping verträgt sich schlecht mit diesem Zonenverfahren und macht Geschwindigkeitsprognosen bzw. verbindliche Bandbreitenzusagen sehr schwer. Daher sind einige Hersteller, die ein horizontales Mapping einsetzten, dazu übergegangen, statt dessen „zonenweises horizontales Mapping“ zu verwenden (siehe Abb. 2.12). Dieses gemischte Mapping verwendet innerhalb einer Zone horizontales Mapping.

[22] Auf diesen Sachverhalt werden wir in den nächsten Kapiteln öfter zurückgreifen, da er für das Thema der vorliegenden Arbeit äußerst wichtig ist.

Darin können die Geschwindigkeitsvorteile des horizontalen Mappings für Platten mit Linearmotor genutzt werden, ohne daß dies Rückwirkungen auf die Datenrate hat. Beim Wechsel von einer Zone zur nächsten wird jedoch ein vertikales Mapping verwendet. Beim Einsatz von ZBR im Zusammenhang mit CHS-Koordinaten ist zu beachten, daß die maximale Sektornummer eine Funktion der Zylindernummer ist. Für die meisten Betriebssysteme, die CHS-Koordinaten verwenden, ist dies nicht darstellbar. Sie gehen in der Regel davon aus, daß die Festplatte geometrisch ein Quader ist, d.h. daß Zylinder-, Head- und Sektorkomponente der Koordinatenangabe voneinander unabhängig sind. Um trotzdem noch mit CHS-Ansteuerung nutzbar zu sein, geben viele Festplatten, wenn sie nach ihrer Geometrie befragt werden, bei Abfrage ihrer Geometriedaten eine quaderförmige Phantasiegeometrie an, bei der letztlich nur das Produkt $C \cdot H \cdot S$ Verlaß ist. Die an die Festplatte übermittelten CHS-Koordinaten werden von der Platte jedoch nicht direkt verwendet, sondern zunächst wieder in lineare Blockadressen umgewandelt, bevor die Platte sie in ihre internen, physikalischen CHS-Koordinaten umwandelt.

Ein IDE-Gerät besitzt einen Sektorpuffer, als temporären Speicherplatz für jede Lese- und Schreiboperation. Dadurch wird die Rate, bei der Daten zwischen dem Host und der Platte ausgetauscht werden, voneinander entkoppelt. Je nach konkreter Umsetzung kann dadurch auf Interleave und ähnliche Techniken verzichtet werden. Darüber hinaus haben heutige Laufwerke auch Lese- und Schreibcaches, die den Zugriff auf das Medium erheblich steigern können.

Der ATA-Standard spezifiziert zwei rudimentäre Verfahren für die Behandlung defekter Sektoren. Das erste überläßt es dem Hersteller, wie von der Low-Level-Formatierung defekt markierte Sektoren zu behandeln sind. Das zweite Verfahren lagert defekte Sektoren einfach um und funktioniert daher nur mit physisch logischer Translation oder im LBA-Modus.

Befehlsebene

Von den 256 maximal möglichen Befehlen definiert der ATA-Standard ca. 50. Davon muß ein IDE-Laufwerk 15 notwendigerweise unterstützen, während die restlichen optional sind. Darüber hinaus implementieren manche Hersteller properitäre, optionale Befehle. Die wichtigsten notwendigen Befehle sind diverse „READ“ und „WRITE“ Befehle zum Transfer von einem oder mehreren Sektoren und einige Befehle zur Wartung und Statusabfrage der Laufwerke. Zusammengefaßt kann man sagen, daß die IDE-Schnittstelle im Unterschied zu den vorherigen Schnittstellen relativ abstrakt und losgelöst von den eigentlichen Laufwerken definiert ist. Eine Tabelle der standardisierten Befehle findet man in [Com94], Seite 40.

2.2.7 Die SCSI-Schnittstelle

Die SCSI-Schnittstelle ist ein geräteunabhängiger I/O-Bus, der es ermöglicht, eine Vielzahl an unterschiedlichen Geräten über einen gemeinsamen Bus an ein Computersystem anzuschließen. Die Einordnung der Schnittstellenbeschreibung in unser Modell haben wir in 2.2.3 anhand von Abb. 2.7 schon vorgenommen. Als echte Busschnittstelle befindet sie sich logisch gesehen zwischen dem Gastsystem und den Geräten. Sie bedient alle angeschlossenen Geräte –auch den Host über den

Hostadapter– gleichberechtigt. Die elektrische Spezifikation und das Protokoll des Busses sind dazu ausgelegt, verschiedenste Peripheriegeräte anschließen zu können. SCSI stellt dazu eine Menge an Befehlen zur Verfügung, mit denen ein bestimmtes Gerät nach notwendigen Parametern abgefragt werden kann. Dadurch wird es im Prinzip möglich, einen Gerätetreiber zu schreiben, ohne die gerätespezifischen Details zu kennen. Darüber hinaus bietet SCSI mehr Funktionalität als die bisher beschriebenen Schnittstellen. Besonders Festplatten profitieren davon. Da Daten prinzipiell rein logisch –statt physisch– adressiert werden, braucht sich der Host nicht mit der Organisation der Daten auf dem Medium zu beschäftigen.

Bei der folgenden Beschreibung halten wir uns meist an den etwas älteren SCSI-1-Standard [Ins85] oder an den aktuellen, abwärtskompatiblen SCSI-2-Standard [Ins90]. Wir lassen dabei den SCSI-3-Standardisierungsvorschlag, an dem momentan gearbeitet wird, außen vor. Dieser ist eigentlich eine Familie von Standardisierungen mit dem Ziel, die Modularität von SCSI-2 zu erhöhen und verschiedene physikalische Schichten bei gleichzeitiger Abwärtskompatibilität zu integrieren [Com01].

Physikalische Ebene

Der SCSI-Bus ist je nach Typ 8, 16 oder 32 Bit breit. Für den älteren, 8 Bit breiten SCSI-1-Bus –auch „Narrow-SCSI“ bezeichnet– wird ein einfaches 50-poliges Flachkabel bei einer Taktrate von 5 MHz benutzt. Die 16 und 32 Bit breiten Varianten des SCSI-2-Standards heißen „Wide-SCSI“ und benötigen laut Standard ein zusätzliches 68-poliges Kabel. Natürlich benötigt jedes Gerät, daß Wide-SCSI unterstützt, damit einen zusätzlichen Anschluß. Neben der Erweiterung auf höhere Busbreiten ist auch die Taktrate auf 10 MHz („Fast-SCSI“), 20 MHz („Ultra-SCSI“) und 40 MHz („Ultra2-SCSI“) gesteigert worden.[23] Die angegebenen Übertragungsraten[24] beziehen sich auf den synchronen Übertragungsmodus, bei dem die beteiligten Geräte nicht warten, ob die Datenpakete ihren Empfänger erreichen. Sie wissen ohnehin, daß sie einige Bustakte später darüber informiert werden. Allerdings müssen alle angeschlossenen Geräte diesen Modus auch unterstützen. Im asynchronen Modus arbeiten Sender und Empfänger nach dem Handshakeverfahren und quittieren empfangene Pakete. Die Datenrate sinkt dabei für Narrow-SCSI auf 3 MB/s. Die beiden Modi sind nicht kompatibel. An einem Bus müssen alle Geräte auf einen Modus eingestellt werden.

Man unterscheidet drei verschiedene Bustypen, die untereinander nicht kompatibel sind. Beim „Single-Ended-Bus“ werden die Signalpegel „High“ und „Low“ bezüglich derselben Masse gemessen. Die maximale Buslänge beträgt für die niedrigste Geschwindigkeit 6 Meter. Beim „differentiellen Bus“[25] besitzt jede Leitung eine eigene Rückführung, so daß Spannungsdifferenzen zwischen diesen beiden bestimmen, ob ein „High“ oder „Low“ vorliegt. Bei allen definierten Geschwindigkeiten beträgt die maximale Kabellänge 25 Meter. Diese Busform ist äußerst teuer, eine billigere

[23] Im Prinzip kann man alle Fast- und Widevarianten, die heutzutage kaufbar sind, eigentlich als SCSI-3 betrachten, weil sie einige Abweichungen zu SCSI-2 implementieren, die strenggenommen erst in SCSI-3 definiert werden (siehe [Kro00], Kap. 2.1.5).

[24] Die Übertragungsrate ist die Taktrate geteilt durch 8 multipliziert mit der Busbreite.

[25] Manchmal auch mit „High Voltage Differential“ bezeichnet.

Variante, „Low Voltage Differential“ genannt, erlaubt bei allen Geschwindigkeiten immerhin bis zu 12 Metern Buslänge.

Es können bis zu 8 Geräte auf einem Bus adressiert werden.[26] Ein Computer hat Zugriff auf den Bus durch den SCSI-Hostadapter, der selbst ein SCSI-Gerät darstellt. Die eindeutige Identifikation eines Gerätes geschieht durch die in der Regel pro Gerät wählbare „SCSI-Identifikationsnummer“ (ID). Diese Zahl zwischen 0 und 7 darf pro Gerät und Bus nur einmal vergeben werden.[27] Dabei sollte aus verschiedensten Gründen der Hostadapter die ID 7 erhalten[28]. Zu jeder Zeit dürfen nur zwei Geräte miteinander kommunizieren. Im Falle des gleichzeitigen Zugriffs auf den Bus erhält das Gerät mit der höheren ID den Vortritt.[29] Das Gerät, das einen Auftrag über den Bus absetzt, wird „Initiator“ genannt. Das den Auftrag ausführende heißt „Target“ (siehe Abb. 2.13). Ein Initiator beginnt eine Transaktion über den SCSI-Bus, indem er ein Target auswählt. Sobald diese Auswahl beendet ist, übernimmt das Target die Kontrolle des Busprotokolls und entscheidet darüber, den Bus freizugeben und wann es sich wieder mit dem Initiator verbindet.[30]

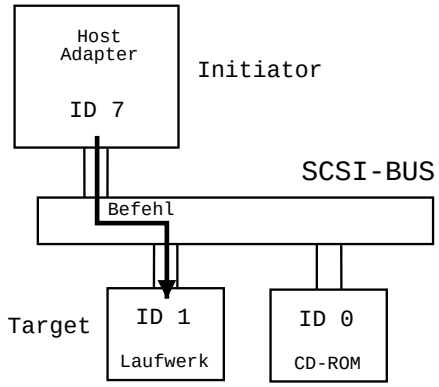

Abbildung 2.13: Einfache SCSI-Konfiguration

Der Narrow-Bus besteht aus 18 Signalen, die über ein 50-adriges Kabel, auch „A-Kabel“ genannt, übertragen werden. Der Wide-Bus benötigt weitere 29 Signale, die über ein zusätzlich 68-poliges „B-Kabel“ transportiert werden. Mit „BUSY“ wird angezeigt, daß der Bus gerade benutzt wird. „SELECT“ signalisiert eine Selektions- oder Reselektionsphase. „COMMAND/DATA“ wird vom Target benutzt, um anzuzeigen, welcher Natur der Datentransfer ist. „INPUT/OUTPUT“ gibt die Richtung des Datenflusses in Bezug auf den Initiator wider. Ist es aktiv, dann empfängt der Initiator Daten.[31] „MESSAGE“ leitet eine Nachrichtenphase durch das Target ein.

[26] Bei SCSI-3 sind es je nach physikalischer Übertragungsart bis zu 64 Geräte.
[27] Heutige Wide-Geräte können meist nicht SCSI-2 konform bis zu 16 IDs adressieren.
[28] Insbesondere bei der Kombination von Narrow- und Wide-Geräten ist zu beachten, daß Narrow-Geräte keine ID oberhalb von 7 ansprechen können.
[29] Damit stellt die Wahl der ID auch eine gewisse Zugriffspriorisierung dar.
[30] Ein Zitat aus [Sch95] drückt diesen Sachverhalt treffend aus: „The initiator is the master in function and the slave in protocol; the target is the slave in function and the master in protocol.“
[31] Außerdem wird es zur Unterscheidung einer Selektion von einer Reselektion benutzt.

„REQUEST“ wird vom Target für das Handshake benötigt. Mit „ACKNOWLEDGE„ bestätigt der Initiator ein Handshake. „ATTENTION“ wird vom Initiator benötigt, um dem Target mitzuteilen, daß er eine Nachricht absetzen will. Die einzige Möglichkeit, alle Geräte gleichzeitig anzusprechen, ist durch einen SCSI-Busreset über das Signal „RESET“. Der Datenbus besteht aus „DATA BUS 0-7“ und einem Paritätsbit bzw. aus „DATA BUS 8-31“ und drei weiteren Paritätsleitungen auf dem B-Kabel.[32]

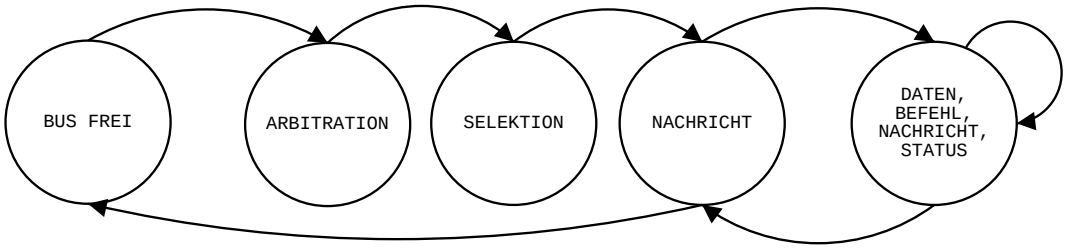

Abbildung 2.14: Vereinfachtes SCSI-1-Zustandsdiagramm

Es gibt 8 verschiedene Buszustände, deren vereinfachtes SCSI-1-Zustandsdiagramm in Abb. 2.14 zu sehen ist. Ein vollständiges SCSI-2-Diagramm findet man in [Sch95], Kap. 10.6. Nach einem Busreset oder wenn kein Gerät den Bus kontrolliert, befindet sich dieser in dem freien Anfangszustand. In dieser Phase sind „BUSY“ und „SELECT“ inaktiv. Wenn ein Gerät die Steuerung des Busses als Initiator oder Target übernehmen will, kann es „BUSY“ aktivieren und seine SCSI-ID auf dem Datenbus aktivieren, indem es die Datenleitung mit dem entsprechendem Datenbit setzt. Der Bus befindet sich dann in der Arbitrationsphase. Die Einheit muß eine definierte Zeitspanne warten und prüft danach die Datenleitung auf höher gesetzte Datenbits. Gibt es keine höheren, hat das Gerät die Arbitration gewonnen und übernimmt nach einer weiteren kurzen Verzögerung die Kontrolle des Busses. Es leitet durch Aktivierung von „SELECT“ eine Selektions- oder Reselektionsphase ein. Für eine Reselektion setzt das Gerät „INPUT/OUTPUT“ und identifiziert sich als Target. Für eine Selektion läßt es „INPUT/OUTPUT“ inaktiv und ist damit Initiator. Es wählt die ID des Kommunikationspartners und setzt das korrespondierende Datenbit und deaktiviert kurz darauf „BUSY“. In diesem Moment schauen alle Geräte, ob sie auf dem Datenbus ihre eigene ID finden. Das selektierte Gerät findet die ID des Senders anhand der anderen aktiven Datenleitung und muß innerhalb von 200 ms „BUSY“ aktivieren, da ansonsten ein Busreset eintritt. Nach einer erfolgreichen Selektion wird „MESSAGE“ durch das aktive Gerät gesetzt. Der Bus befindet sich in diesem Moment in Nachrichtenphase, von der aus er in die vier Informationsphasen eintritt. In diesen werden Daten- und Steuerinformationen über den Datenbus transportiert, wobei die Signale „MESSAGE“, „INPUT/OUTPUT“ und „COMMAND/DATA“ benutzt werden, um diese Phasen zu unterscheiden. Jeder Datenaustausch in einer der vier Phasen läuft über ein Handshake ab. Der Sender legt Daten auf den Datenbus,

[32] „ACKNOWLEDGE“ und „REQUEST“ gibt es zusätzlich auch auf dem B-Kabel.

setzt gegebenenfalls die Paritätbits entsprechend[33] und aktiviert „REQUEST“. Der Empfänger liest die Daten und signalisiert durch „ACKNOWLEDGE“, daß er die Daten übernommen hat. Daraufhin deaktiviert zuerst der Sender „REQUEST“ und dann negiert der Empfänger „ACKNOWLEDGE“. Eine erneute Übertragung per Handshake kann beginnen. Wichtig ist, daß das Target die Steuersignale „MESSAGE“, „INPUT/OUTPUT“ und „COMMAND/DATA“ setzt. Durch „ATTENTION“ kann der Initiator aber auch eine Nachrichtenphase anfordern. Im synchronen Modus muß der Sender nicht auf einzelne Bestätigungen durch den Empfänger warten, sondern kann eine definierte Menge von Informationen senden und die Bestätigungen dazu später einsammeln. In der Befehlsphase kann das adressierte Target Befehlsdaten vom Initiator anfordern. Dazu aktiviert es „COMMAND/DATA“ und deaktiviert „MESSAGE“ sowie „INPUT/OUTPUT“. In der Datenphase deaktiviert das Target „COMMAND/DATA“ sowie „MESSAGE“ und weist entweder mit Deaktivierung von „INPUT/OUTPUT“ den Initiator an, Daten zu übermitteln („Data-Out“), oder sendet unter Aktivierung von „INPUT/OUTPUT“ selbst Daten an den Initiator („Data-In“). In der Nachrichtenphase aktiviert das Target „MESSAGE“ sowie „COMMAND/DATA“ und weist entweder mit Deaktivierung von „INPUT/OUTPUT“ den Initiator an, Nachrichten zu übermitteln („Message-Out“), oder sendet unter Aktivierung von „INPUT/OUTPUT“ selbst Nachrichten an den Initiator („Message-In“). In der Statusphase deaktiviert das Target „MESSAGE“ und aktiviert „INPUT/OUTPUT“ sowie „COMMAND/DATA“ und sendet daraufhin Statusinformationen an den Initiator.

Protokollebene

Beim Betrachten des Ablaufs eines normalen SCSI-Befehls in Abb. 2.14 fällt auf, daß zweimal eine Nachricht abgesetzt werden muß: einmal direkt nach der Selektion oder Reselektion und einmal vor dem Freilassen des Busses. Nachrichten sind der wichtigste Bestandteil des SCSI-Busprotokolls. Sie bestehen aus ein, zwei oder beliebig vielen Bytes. Das erste Byte, der Nachrichtencode, bestimmt das Nachrichtenformat. Jedes Gerät muß die Nachricht „COMMAND COMPLETE“ implementieren. Mit dieser teilt das Target dem Initiator am Ende eines Befehls oder Befehlskette das Erfolgsergebnis mit und ob in einer Statusphase weitere Informationen abgesetzt worden sind. Danach leitet es einen Busreset ein. Unterstützt das aktive Gerät, der Initiator bei der Selektion oder das Target bei der Reselektion, weitere Nachrichtentypen, signalisiert es dies durch Setzen von „ATTENTION“ in der Selektionsphase, bevor es „BUSY“ deaktiviert. Die erste Nachricht des Initiators nach der Selektion bzw. des Targets nach der Reselektion ist dann „IDENTIFY“ zur Identifikation der dem Gerät eigener Implementierungen von Targetroutinen oder LUNs. Weitere wichtige Nachrichten sind „SAVE DATA POINTERS“ und „RESTORE DATA POINTERS“, welche dem Initiator mitteilen, Status-, Befehls- und Datenregister zu sichern oder zu laden. Dazu hat jeder Initiator einen Pufferbereich für jeden Registertyp und zwei Sätze von drei Zeigern. Ein Satz zeigt auf die aktuellen Registerwerte und einer auf die gespeicherten. Desweiteren gibt es Nachrichtentypen zur Einstellung von Transferoptionen, Verwalten von Befehlswarteschlangen, Beenden von Prozessen, Fehlerbehandlung und asynchrone Ereignisverwaltung. Eine

[33] Per Definition sind die Datenbits zusammen mit den Paritätbits ungerade.

Auflistung der verschiedenen Formate und Nachrichtentypen sowie detaillierte Erläuterungen zu den einzelnen Typen findet man in [Sch95], Kap. 11.

Modellebene

Ein SCSI-Target wird durch seine SCSI-ID adressiert. Jedes Target kann bis zu acht logische Einheiten, genannt „Logische Gerätenummer“, und acht Targetroutinen definieren. Ein Target muß mindestens eine LUN implementieren. Die Targetroutinen sind gänzlich optional. Jeder SCSI-Befehl wird von der eigens im Befehl adressierbaren logischen Einheit oder Routine ausgeführt. Die meisten Targets definieren nur eine LUN, aber im Prinzip könnte jedes Target bis zu acht physikalische Geräte unterstützen, die durch eine eigene LUN adressiert werden. Targetroutinen sind herstellerspezifisch und spielen eine untergeordnete Rolle. Jede LUN enthält eine Menge an Konfigurationsparametern. Diese können mit „MODE SELECT“ geschrieben und „MODE SENSE“ gelesen werden. Die Parameter werden über den Bus in Blöcken, in sogenannten „Parameterseiten“, transportiert. Eine LUN verwaltet drei Kopien eines jeden Parametersatzes und hat Zugriff auf eine Seite des Herstellers, die beschreibt, welche Parameter in welcher Weise verändert werden dürfen. SCSI kennt zehn Geräteklassen für die verschiedenen unterstützten Gerätetypen. Zu jeder Klasse definiert SCSI ein Gerätemodell, einen Befehlssatz und Parametersätze, um das Gerät zu konfigurieren. Darüber hinaus gibt es gemeinsame Befehle und Parametersätze für alle Geräte.

Von den zehn definierten Klassen werden wir nach der allgemeinen Beschreibung der Befehlsebene nur auf das Modell einer SCSI-Festplatte eingehen, weil diese Klasse für diese Arbeit die relevante ist. Die anderen Klassen sind ausführlich in [Sch95], Kap. 13-21 erläutert.

Befehlsebene

SCSI-Befehle werden über den Bus als Folge von Bytes, in sogenannten „Befehlsbeschreibungsblöcken“ (engl. „command descriptor block“), geschickt. Manche Befehlsblöcke beinhalten auch eine Liste von zusätzlichen Parametern oder Befehlsdaten. Diese werden im Unterschied zu den Befehlsblöcken nicht in der Befehlsphase, sondern in der Datenphase geschickt.
Ein Beschreibungsblock ist 6, 10 oder 12 Byte lang. Das erste Byte beinhaltet den Befehlsoperationscode. Er besteht aus der Befehlsgruppe, die die Beschreibungsblockgröße angibt, und dem eigentlichen Befehlscode. Im nächsten Feld wird die Nummer der logischen Einheit angegeben, an die der Befehl adressiert ist. Befehle, die auf logischen Blöcken operieren, geben die logische Blocknummer im dem darauffolgenden Feld an. Je nach Beschreibungsblockgröße stehen dabei 21 Bit für den sechs Byte großen Block oder 32 Bit für die beiden anderen Befehlsblockgrößen zur Verfügung.[34] Falls ein Befehl weitere Daten übertragen möchte, wird die Länge der Daten in Byte danach angegeben. Je nach Befehlsbeschreibungsblockgröße stehen dazu 8, 16 oder 32 Bit zur Verfügung. Das letzte Feld ist das Kontrollfeld zum Einstellen befehlsspezifischer Optionen. In Abb. 2.15 ist als Beispiel ein sechs Byte langer Befehlsblock gezeigt.

[34] Damit können bei einer logischen Blockgröße von 512 Byte bis zu 2 TB verwaltet werden.

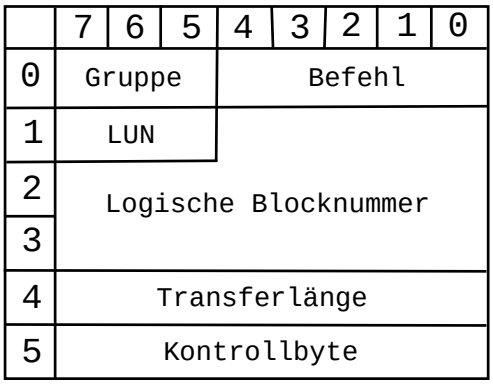

Abbildung 2.15: Generischer Befehlsbeschreibungsblock von sechs Byte

Nicht alle Befehle beziehen sich auf Nutzdaten in logischen Blöcken, manche Befehle benutzen nur Parameterlisten, um Parameter abzufragen oder zu setzen. Es gibt auch Befehle, die gar keine Information übertragen. Jeder Befehl wird durch eine Statusphase abgeschlossen, wenn er nicht vorher schon abgebrochen worden ist. Die häufigsten Statusmeldungen sind „GOOD“, „BUSY“ und „CHECK CONDITION“. „GOOD“ benachrichtigt über erfolgreiche Durchführung des Befehls. „BUSY“ sagt aus, daß das Target gerade beschäftigt ist. „CHECK CONDITION“ zeigt an, daß ein Fehler bei der Bearbeitung aufgetreten ist, den man mit dem Befehl „REQUEST SENSE“ abfragen kann.

Es gibt acht Befehle, die jede Befehlsklasse laut Standard implementieren muß. Einer der wichtigsten Befehle ist „INQUIRY“ (12h). Er liefert detaillierte Informationen über eine LUN. Er wird benutzt, um unter anderem herauszufinden, welche SCSI-Optionen implementiert sind, und liefert darüber hinaus die Versionsnummer, den Namen und den Gerätetyp der LUN. „INQUIRY“ benachrichtigt auch darüber, daß eine nicht existierende LUN adressiert worden ist. Mit „TEST UNIT READY“ (00h) kann man abfragen, ob eine LUN bereit ist, einen Befehl zu akzeptieren. Wenn „CHECK CONDITION“ als Status zurückgeliefert wird, benutzt man in der Regel „REQUEST SENSE“ (03h) zur Anfrage, warum ein Befehl nicht erfolgreich verlaufen ist, . Mit „RESERVE UNIT“ (16h) bzw. „RELEASE UNIT“ (17h) kann eine bestimmte LUN für einen bestimmten Initiator reserviert bzw. für andere wieder freigegeben werden. Für Festplatten gibt es spezielle Implementierungen dieser Befehle. Mit „SEND DIAGNOSTIC“ (1Dh) wird eine LUN veranlaßt, bestimmte wählbare Test- und Diagnoseroutinen ablaufen zu lassen. „MODE SELECT“ (15h) bzw. „MODE SENSE“ (1Ah) haben wir bereits oben erwähnt. Sie werden dazu benutzt, um eine LUN zu konfigurieren bzw. ihre Parameterseiten abzufragen. Beide Befehle sind komplex und erfordern Geräteklassen spezifische Parameterseiten. Optionale Befehle sind beispielsweise „COPY“ (18h), „COPY AND VERIFY“ (3Ah) und „COMPARE“ (39h) zum Kopieren, Kopieren mit Vergleich und Vergleichen von Daten zweier Geräte. Eine detaillierte Beschreibung der erwähnten Befehle sowie aller optionaler Befehle findet man in [Sch95], Kap. 12 und [Ins90], Kap. 7.

2.2.8 Das Modell einer SCSI-Festplatte

Eine SCSI-Festplatte fällt in die SCSI-Klasse der „Direktzugriffsgeräte“ (engl. „direct access device“). Diese Geräte verwalten Daten in logischen Blöcken für späteren Zugriff. Jeder Datenblock ist dabei über eine eindeutige logische Blocknummer (LBN), adressierbar. Ein Initiator kann mit verschiedenen „WRITE-Befehlen“ Datenblöcke schreiben und mit verschiedenen „READ-Befehlen“ Datenblöcke lesen. Diese Klasse schließt alle Geräte ein, die direkten Lese- und Schreibzugriff auf jeden logischen Block des Mediums erlauben. Festplatten, magneto-optische Laufwerke, Disketten und RAM-Disks sind die bekanntesten Implementierungen dieser Klasse.

Das Basismodell und die Organisation einer SCSI-Festplatte ist bereits durch das in Abschnitt 2.2.2 eingeführte Grundmodell bekannt. Im Sinne der Abstraktion der Geräte präsentiert sich eine SCSI-Platte dem Benutzer bzw. Initiator als sequentielle Folge von logischen Blöcken zum Speichern von Benutzerdaten (siehe Abb. 2.16). Diese Blöcke können beliebig oft gelesen oder geschrieben werden. Sie werden durch ihre LBN eindeutig identifiziert. Der erste logische Block hat die Nummer Null. Die Adresse des letzten Blocks ist $n-1$, wobei n die Anzahl der Speicherblöcke auf dem Medium insgesamt ist. Mit „READ CAPACITY“ (25h) kann der Wert von $n-1$ erfragt werden. Wenn ein Befehl einen logischen Block adressiert, der nicht innerhalb der Kapazitätsgrenze des Mediums liegt, dann wird dieser Befehl mit „CHECK CONDITION“ terminiert. Im Gegensatz zu einem Magnetband beispielsweise kann jeder Block auf dem Medium direkt abgefragt werden. Dieser Vorgang geschieht völlig transparent. Gewöhnlich hat der Host keine Vorstellung davon, wo die Daten tatsächlich auf dem Medium liegen.

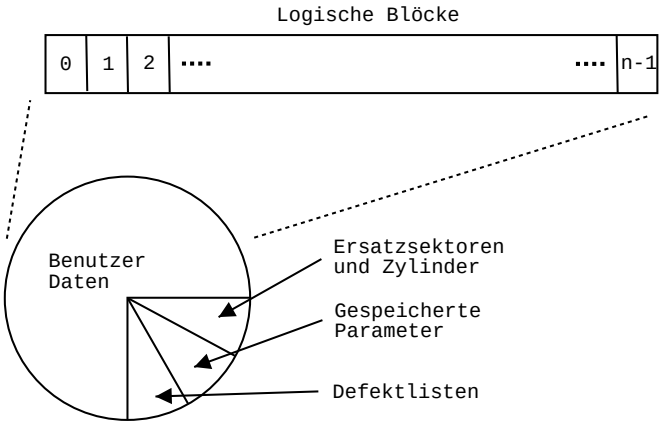

Abbildung 2.16: Die Organisation der SCSI-Festplatte

Extents und Notches

Die Blocklänge eines logischen Blocks gibt die tatsächliche Anzahl von Bytes an, die er an Benutzerdaten fassen kann. Jeder Block speichert eine solche Angabe zusammen mit weiteren Zusatzdaten, die der Controller zum Verwalten der Blöcke benötigt. Theoretisch kann damit jeder logische Block auf dem Medium eine eigene Länge haben. Typischerweise sind jedoch alle Blöcke gleich lang. Oft benutzte Grö-

ßen sind 512, 2048 oder 4096 Byte. Eine kontinuierliche Folge von Blöcken gleicher Länge heißt „Extent“. Mit „MODE SENSE“ und „MODE SELECT“ kann man diese Extents abfragen oder setzen. Somit besteht eine Festplatte typischerweise nur aus einem Extent. Extents sind nicht mit Zonen zu verwechseln. Eine Zone wird im SCSI-Sprachgebrauch „Notch“ genannt. Eine Notch ist ähnlich einer Zone dadurch gekennzeichnet, daß in ihrem Gültigkeitsbereich die Anzahl der Sektoren pro Spur konstant ist. Gleichzeitig ist die Sektorgröße nicht mit logischer Blocklänge zu verwechseln. Zusammengefaßt sind Extents Gruppierungen logischer Blöcke gleicher Länge, und Notches Gruppierungen von Spuren mit gleicher Anzahl von Sektoren.

Mapping

Das Mapping von logischen Blöcken zu physikalischen Sektoren ist nicht spezifiziert. Der physikalische Ort eines logischen Blocks auf dem Medium hat keine Beziehung zu dem Ort eines anderen logischen Blocks. Sie sollten aber derart angeordnet werden, daß die Zugriffszeit auf benachbarte logische Blöcke minimal ist. Dazu verwenden die meisten Geräte lineares Mapping, so daß benachbarte logische Blöcke benachbarten physikalischen Sektoren zugeordnet werden.[35] Allerdings wird in [Ins90] ausdrücklich betont, daß die Zugriffszeit auf einen logischen Block x und dann $x+1$ nicht geringer sein muß als die Zugriffszeit auf x und dann $x+100$.

Defektlisten

SCSI ermöglicht einem Target, dem Initiator ein virtuell fehlerfreies Medium zu präsentieren. Dies wird dadurch erreicht, daß defekte logische Blöcke durch für diesen Zweck reservierte Blöcke ersetzt werden können (siehe Abb. 2.16). Es gibt zwei Modi: Entweder das Laufwerk ersetzt defekte Blöcke automatisch, oder der Host hat die Verantwortung und kann das Ersetzen durch „REASSIGN BLOCKS“ (07h) erwirken. Letzteres ist meistens erwünscht, da dadurch das Betriebssystem volle Kontrolle über diesen kritischen Vorgang hat. Erfolgt das Entdecken eines fehlerhaften Blocks im automatischen Modus beim Schreiben, kann das Laufwerk stillschweigend den Block ersetzen, erfolgt es beim Lesen, muß es den Host durch eine Nachricht darüber informieren, da korrupte Daten zu befürchten sind. Wie das Laufwerk sich in verschiedenen Situationen verhalten soll, läßt sich detailliert einstellen. Es gibt drei verschiedene Defektlisten. In der primären Defektliste (engl. „primary defect list“) trägt der Hersteller die bei der Herstellung des Mediums entdeckten fehlerhaften Blöcke ein. In der wachsenden Liste (engl. „grown defect list“) werden die Fehler vermerkt, die während normaler Operation oder bei der Low-Level-Formatierung des Laufwerks entdeckt werden. In der Zertifikationsliste (engl. „certification defect list“) werden die Fehler notiert, die nach dem Formatieren beim Verifizieren auffallen.

Prefetching

Jeder SCSI-Controller hat einen Sektorpuffer, der groß genug ist, um mindestens einen Datensektor solange speichern zu können, bis er auf das Medium geschrieben ist. Wenn dieser Puffer sogar eine ganze Spur an Sektordaten aufnehmen kann, wird die Lesegeschwindigkeit optimiert, indem man Daten im Voraus liest (engl.

[35] Wie schon beim Modell der IDE-Platte ist diese Aussage von zentraler Bedeutung und wird im folgenden noch eine wichtige Rolle spielen.

„prefetching“). Die Annahme dabei ist, daß wenn immer der Host einen Datenblock x anfordert, er danach $x + 1$ anfordert. Die Gültigkeit dieser Annahme hängt stark vom Betriebssystem des Hosts ab. Nichtsdestotrotz, es kostet den Controller kaum Zeit, den ganzen Sektor in den Puffer einzulesen, um eventuelle weitere Anfragen schnell beantworten zu können. Genauso kann er bei einer langen Anfrage von kontinuierlichen Blöcken bereits mit dem Lesen beginnen, ohne den ersten Block der Anfrage gelesen zu haben. Bevor er die Daten dann dem Host sendet, ordnet er sie in der richtigen Reihenfolge an. Die genaue Implementierung beider Verfahren bleibt dem Hersteller der Platte überlassen.

Schreiboptimierung

Natürlich kann man auch den Schreibvorgang beschleunigen, wenn der Datenpuffer groß genug ist. Normalerweise führt der Controller beim Schreiben eines Blockes ein „SEEK“ (0Bh/2Bh) an die entsprechende Stelle auf dem Medium durch und wartet darauf, daß die Daten geschrieben worden sind. Erst danach sendet er dem Host ein „COMMAND COMPLETE“. Statt dessen kann er zur Optimierung des Schreibvorgangs auch sofort „GOOD“ als Status an den Host senden und erst bei Gelegenheit die Daten schreiben, so daß die Zugriffszeit fiktiv minimiert wird. Gleichzeitig erhöht das natürlich das Risiko, daß Daten verloren gehen können. Das Betriebssystem nimmt an, daß alle Daten, die auf die Platte geschrieben worden sind, sicher geschrieben sind. Je nach Wichtigkeit und Funktion der Daten kann dadurch im Fehlerfall die Integrität des Hosts nicht gewährleistet werden. Letztendlich entscheidet ein Abwägen zwischen Risiko und Performanz. Daher ist die Schreiboptimierung ein Merkmal, welches durch entsprechende Parameterseiten mit „MODE SELECT“ (15h/55h) konfigurierbar ist.

Zwischenspeicher (Cache)

SCSI-2-Cache-Techniken gehen einen Schritt weiter als die bisher beschriebenen Optimierungen. Generell ist ein Cache ein Speicher, der Kopien von Daten eines meist größeren Speichers enthält. Auf den Cache kann man in der Regel mindestens um eine Größenordnung schneller zugreifen, dafür kann er aber auch nur Ausschnitte des langsameren Speichers fassen. Bei Zugriff auf den langsameren Speicher gibt ein Cache-Verzeichnis an, ob die gesuchten Daten im Cache sind. Wenn sie vorhanden sind, spricht man von einem Treffer (engl. „cache hit“), sonst von einem Verfehlen (engl. „cache miss“). Bei Cache-Strategien für den Hauptspeicher mit First-Level-Cache und Second-Level-Cache kann man wegen der Lokalität von Prozessorinstruktionen trotz eines kleinen Verhältnisses von Cache-Kapazität zu RAM-Kapazität eine Trefferrate von über 90% erreichen. Für Massenspeicherzugriffe sieht die Situation allerdings komplett anders aus. Die Effektivität eines Zwischenspeichers hängt hier stark vom verwendeten Betriebssystem und den laufenden Applikationen ab. Zumindest in Mehrbenutzersystemen werden Zugriffe auf den Massenspeicher auf dem ganzen Medium verteilt sein. Es gibt allerdings trotzdem Bereiche, auf die häufig zugegriffen wird, wie beispielsweise Verzeichnisse und allgemeine Tabellen, die das Betriebssystem verwaltet. Die Trefferrate liegt hier meist unter 50%. Trotzdem rentiert es sich, Zwischenspeicherstrategien zu implementieren, da aus einem 10 ms Festplattenzugriff ein nur 100 ns dauernder Hauptspeicherzugriff oder ein

unwesentlich längerer Zugriff auf den Cache des Controllers werden kann.

Die Effektivität eines Festplattenzwischenspeichers hängt sehr stark von der Konfiguration der Platte ab. Der Cache wird gefüllt, während Daten gelesen werden, ohne das die Leseperformanz darunter leidet. Wenn Schreibdaten erst in den Cache geschrieben werden und dann auf die Platte[36], bestehen potentiell dieselben Gefahren wie bei der Zwischenspeicherung im Sektorpuffer. Wartet das Gerät mit dem Senden von „COMMAND COMPLETE“ bis die Daten wirklich physikalisch auf dem Medium vorhanden sind, hat man keinen Geschwindigkeitsvorteil beim Schreiben. Wenn der „GOOD“ Status zurückgeliefert wird, bevor die Daten wirklich geschrieben sind, geht man das Risiko des Datenverlusts ein. In beiden Fällen sind die Daten im Cache und falls sie daraufhin wieder gelesen werden sollen, ist je nach Konfiguration zumindest die Lesegeschwindigkeit stark optimiert. Wie sich das Laufwerk verhält, ob der Schreibcache aktiviert sein soll, und wann die Statusantwort gesendet werden soll, kann man durch „MODE SELECT“ (15h/55h) unter Verwendung der Cacheparameterseite einstellen. Zusätzlich kann man noch festsetzen, wieviele Blöcke beim Lesen im Voraus in den Cache gelesen werden sollen. Ein weiterer Punkt ist das Verhalten des Zwischenspeichers in der Situation, daß er bereits gefüllt ist und neue Daten ältere ersetzen müssen. Meistens werden dabei die Blöcke überschrieben, die die längste Zeit nicht benutzt worden sind (engl. „last recently used“). Optional kann man angegeben, daß vorausgelesene Daten immer zuerst überschrieben werden sollen.

Befehle

In Abb. 2.17 sind übersichtshalber alle SCSI-Befehle einer Festplatte wiedergegeben. Es ist eine Kopie der Tabelle 8.1 aus dem SCSI-2-Entwurf [Ins90]. Die mit „M“ bezeichneten Befehle müssen von jeder SCSI-2-Festplatte unterstützt werden, die mit „O“ bezeichneten sind optional. Die angegebene Abschnittsnummer bezieht sich auf den Abschnitt im Entwurf, an dem der jeweilige Befehl erläutert ist. Die Befehlsnamen sind eigentlich selbsterklärend, so daß wir nur auf einige wenige knapp eingehen werden.

Die „READ“ (08h/28h) Befehle fordern vom Target eine bestimmte Anzahl logischer Blöcke an. Die „WRITE“ (0Ah/2Ah) Befehle übergeben eine bestimmte Anzahl von Datenblöcken an das Target, das diese auf das Medium schreiben soll. Die Struktur beider Befehlsblöcke ist bereits in Abb. 2.15 gezeigt. Die Transferlänge gibt dabei immer die Anzahl von Blöcken an.[37] Die 10 Byte langen Befehle adressieren dabei höhere Blocknummern und haben zusätzliche Kontrollmöglichkeiten für den Cache und für Positionierangaben bezogen auf den letzten Befehl. Mit „READ CAPACITY“ (25h) kann man die Kapazität des Mediums abfragen. Dieser Befehl gibt die letzte LBN des Mediums und die benutzte Blocklänge zurück. „FORMAT UNIT“ (04h) instruiert das Target, das Medium zu formatieren. In der einfachsten Form sind dazu keine weiteren Angaben nötig. Das Target benutzt die voreingestellten Werte des Herstellers. Es gibt aber auch die Möglichkeit, die Low-Level-Formatierung detailliert zu steuern (siehe [Sch95], Kap. 13.2). Mit „MODE

[36] Man nennt diese Strategie des Caches „write-through“.
[37] Bei SCSI-1 steht Null für das Maximum von 256 Blöcken.

Table 8-1: Commands for Direct-Access Devices

Command Name	Operation Code	Type	Section
CHANGE DEFINITION	40h	O	7.2.1
COMPARE	39h	O	7.2.2
COPY	18h	O	7.2.3
COPY AND VERIFY	3Ah	O	7.2.4
FORMAT UNIT	04h	M	8.2.1
INQUIRY	12h	M	7.2.5
LOCK-UNLOCK CACHE	36h	O	8.2.2
LOG SELECT	4Ch	O	7.2.6
LOG SENSE	4Dh	O	7.2.7
MODE SELECT(6)	15h	O	7.2.8
MODE SELECT(10)	55h	O	7.2.9
MODE SENSE(6)	1Ah	O	7.2.10
MODE SENSE(10)	5Ah	O	7.2.11
PRE-FETCH	34h	O	8.2.3
PREVENT-ALLOW MEDIUM REMOVAL	1Eh	O	8.2.4
READ(6)	08h	M	8.2.5
READ(10)	28h	M	8.2.6
READ BUFFER	3Ch	O	7.2.12
READ CAPACITY	25h	M	8.2.7
READ DEFECT DATA	37h	O	8.2.8
READ LONG	3Eh	O	8.2.9
REASSIGN BLOCKS	07h	O	8.2.10
RECEIVE DIAGNOSTIC RESULTS	1Ch	O	7.2.13
RELEASE	17h	M	8.2.11
REQUEST SENSE	03h	M	7.2.14
RESERVE	16h	M	8.2.12
REZERO UNIT	01h	O	8.2.13
SEARCH DATA EQUAL	31h	O	8.2.14.1
SEARCH DATA HIGH	30h	O	8.2.14.2
SEARCH DATA LOW	32h	O	8.2.14.3
SEEK(6)	0Bh	O	8.2.15
SEEK(10)	2Bh	O	8.2.15
SEND DIAGNOSTIC	1Dh	M	7.2.15
SET LIMITS	33h	O	8.2.16
START STOP UNIT	1Bh	O	8.2.17
SYNCHRONIZE CACHE	35h	O	8.2.18
TEST UNIT READY	00h	M	7.2.16
VERIFY	2Fh	O	8.2.19
WRITE(6)	0Ah	M	8.2.20
WRITE(10)	2Ah	M	8.2.21
WRITE AND VERIFY	2Eh	O	8.2.22
WRITE BUFFER	3Bh	O	7.2.17
WRITE LONG	3Fh	O	8.2.23
WRITE SAME	41h	O	8.2.24

Key: M = Command implementation is mandatory.
O = Command implementation is optional.

Abbildung 2.17: Die Befehle für Direktzugriffsgeräte aus [Ins90], Tab. 8.1

SENSE“ (1Ah/5Ah) und „Mode Select“ (15h/55h) können Parameterseiten gelesen und gesetzt werden. Die wichtigsten Parameterseiten sind „Format page“, „Disk drive geometry page“ und „Cache page“ (siehe [Sch95], Tabelle 13.9).

	7	6	5	4	3	2	1	0
0	READ CAPACITY (25h)							
1	LUN			Reserviert				Rel
2	Logische Blocknummer							
3								
4								
5								
6	Reserviert							
7								
8								PMI
9	Kontrollbyte							

	7	6	5	4	3	2	1	0
0	SEEK10 (2Bh)							
1	LUN			Reserviert				
2	Logische Blocknummer							
3								
4								
5								
6	Reserviert							
7								
8								
9	Kontrollbyte							

Abbildung 2.18: Die Befehlsblöcke für „READ_CAPACITY“ und „SEEK10“

Auf die folgenden beiden Befehle gehen wir etwas detaillierter ein, weil sie zur Untersuchung der in dieser Arbeit benutzten SCSI-Festplatte besonders wichtig sind. Den ersten interessanten Befehl „READ_CAPACITY“ (25h) haben wir oben bereits erwähnt. Sein Beschreibungsblock ist in links in Abb. 2.18 dargestellt. Er liefert bei Erfolg acht Byte zurück. Die ersten vier davon geben die höchste logische Blocknummer an, die das Medium zum Speichern von Daten zur Verfügung stellt. Die letzten vier bestimmen die dazugehörende Blocklänge. Wenn das „Partial Medium Indicator“ Bit (PMI-Bit) gesetzt ist, bedeutet der Befehl etwas gänzlich anderes. Das Gerät sucht dann ab der angegebenen LBN absteigend bis zur nächsten LBN, bei der eine für das Gerät signifikante Verzögerung im Zugriff erfolgt. Eine solche Verzögerung kann beispielsweise durch einen Zylinder- oder Kopfwechsel erfolgen. Die die Verzögerung verursachende LBN liefert er dann zusammen mit ihrer Blockgröße zurück.

Der Befehlsblock des zweiten Befehls „SEEK10“ (2Bh) ist rechts in Abb. 2.18 dargestellt. Er fordert die SCSI-Festplatte lediglich dazu auf, die angegebene LBN auf dem Medium anzusteuern. Beide Befehle werden wir in Abschnitt 4.4.5 benutzen, um gerätespezifische Kenngrößen zu berechnen. Beispielsweise werden wir „SEEK10“ (2Bh) dazu benutzen, eine Suchkurve zu erstellen, aus der wir für die Platte charakteristische Größen ableiten können.

3 Dateisysteme

In diesem Kapitel untersuchen wir die Softwareschichten, die den sinnvollen Betrieb von an einem Computersystem angeschlossenen Festplatten überhaupt erst ermöglichen. Natürlich liegt der Schwerpunkt dabei auf der Analyse der Datenstrukturen verschiedenener Dateisysteme. Da aber diese in der Regel Teil des Betriebssystems sind, geben wir zunächst einen Überblick über UNIX im allgemeinen und LINUX im speziellen. Dann skizzieren wir allgemeine Dateisystemkonzepte unter UNIX, bevor wir auf das virtuelle Dateisystem unter LINUX und die Implementierungen von konkreten Dateisystemen detailliert eingehen werden.

3.1 Computer, Peripherie und Betriebssysteme

Die Meinungen, was genau ein Betriebssystem ausmacht, gehen weit auseinander. Wir zitieren hier [Bra01]: „Das Betriebssystem ist die Software, die für den Betrieb eines Rechners anwendungsunabhängig notwendig ist“. Natürlich ist damit die Definitionsfrage nicht gelöst, da die Begriffe „anwendungsunabhängig“ und „notwendig“ fast beliebig interpretierbar sind. Trotzdem kann man daraus ableiten, daß ein Betriebssystem einen Computer mit Peripherie derart kapseln soll, daß eine breite Palette von Applikationen unterstützt wird („anwendungsunabhängig“), die heutigen Anforderungen und Ansprüchen an sichere, schnelle und komfortable Software genügen („notwendig“). Aus der Sicht eines Benutzers läßt sich eine Abhängigkeitsbeziehung wie folgt angeben: Ein Anwender des Systems benutzt eine Applikation. Die Applikation ihrerseits verwendet einige Bibliotheken, Systemroutinen und Systemressourcen, die ihr das Betriebssystem zur Verfügung stellt. Das Betriebssystem wiederum greift auf die verschiedenen Hardwarekomponenten wie den Computer und seine Peripheriegeräte zu, benutzt also die Hardware des Systems direkt. In diesem Sinn kann man auch von einem Betriebssystem als „Systemsoftware“ sprechen, die Anwendungen den komfortablen Zugang zu den Hardwareressourcen des Computers und seiner Peripherie ermöglicht [NS01].

In Abb. 3.1 ist dieses Zusammenspiel für ein Mehrbenutzersystem gezeigt. M Benutzer steuern über die Benutzeroberfläche (engl. „user interface management system“) N Anwendungen. Diese Oberfläche ermöglicht beispielsweise textuelle und graphische Interaktion mit dem System. Die Anwendungen benutzen einerseits Systemfunktionen in Form von Systembibliotheken, die das Betriebssystem zur Verfügung stellt oder in Form von direkten Systemaufrufen (engl. „system call“), die bestimmte Betriebssystemfunktionen aufrufen, ohne deren Eintrittspunkt zu kennen. Darüber hinaus gibt es meist noch vom Betriebssystem unabhängige und oft standardisierte Bibliotheken, die das Schreiben von Programmen erleichtern und vereinheitlichen. Vom Betriebssystem werden in der Regel nur die Funktionen in den Hauptspeicher geladen, die immer präsent sein müssen. Dieser Betriebssystemkern umfaßt oft

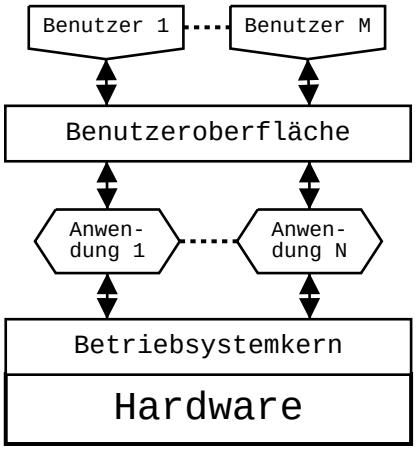

Abbildung 3.1: Überblick über die Betriebssystemstruktur

nur die Prozessor-, Speicher- und Geräteverwaltung, sowie die wichtigsten Teile der Netzwerkverwaltung.

Ein Systemaufruf folgt wie ein normaler Funktionsaufruf einem festen Format, das die Zahl und Typen der Parameter definiert. Da die Lage der Systemfunktionen im Hauptspeicher nicht festgelegt ist, gibt es einen speziellen Aufrufmechanismus. Die Parameter werden auf dem Stack gespeichert und ein spezielles Hardwaresignal, der Softwareinterrupt, wird ausgelöst. Dieses Signal veranlaßt die CPU –wie bei einer durch Geräte ausgelöste Unterbrechung– die augenblickliche Instruktionsadresse und das Statuswort auf dem Stack zu sichern. Danach entnimmt sie das Statuswort und die Adresse der nächsten Instruktion aus einer festen, dem speziellen Interrupt zugeordneten Speicheradresse. Beim Initialisieren des Computersystems (engl. „bootstrap“) muß dazu an diese feste Adresse die Einstiegsadresse des Betriebssystemkerns geschrieben werden . Der Softwareinterrupt wird als „synchrone Unterbrechung mit Falltür“ (engl. „trap door“) bezeichnet, weil die Befehlsfolge durch den Eintritt in den Kern plötzlich unterbrochen wird. Meistens wird zusätzlich bei dem Wechsel zwischen dem Programmcode und Betriebssystemcode in einen anderen Modus des Prozessors geschaltet, der Zugriffsrechte und Zugriffsmöglichkeiten des Codes dramatisch erweitert. Dieses Umschalten vom sogenannten „Benutzermodus“ (engl. „user mode“) in den „Kernelmodus“ (engl. „kernel mode“) geschieht dabei völlig transparent. Im Benutzermodus werden weite Teile des Systems vor Fehlfunktionen durch Benutzerapplikationen geschützt und verhindert, daß diese die Systemintegrität verletzen.

Für eine vollständige Diskussion allgemeiner Betriebssystemeigenschaften und Architekturprinzipien sei auf die bereits erwähnten Bücher [Bra01] und [NS01] verwiesen.

3.1.1 UNIX

Das Betriebssystem „UNIX“ mit all seinen Varianten blickt mittlerweile auf eine lange Vergangenheit von ca. 30 Jahren zurück. Es entstand ursprünglich aus der Notwendigkeit, für die komfortable Programmerstellung ein Rechnerbetriebssystem zu entwickeln. [1] Denn zu der damaligen Zeit waren die meisten der vorhandenen Systeme einfache Batch-Systeme. Der Programmierer gab seine Lochkarten oder Lochstreifen bei einem Operator ab. Diese wurden dann in den Rechner eingelesen und ein Rechenauftrag nach dem anderen abgearbeitet. Der Programmierer konnte dann nach einiger Zeit seine Ergebnisse abholen. Daher begann 1969 Ken Thompson bei den Bell Laboratories die Entwicklung eines neuen Betriebssystems. Ziel seiner Entwicklung war es, ein System zu schaffen, auf dem mehrere Programmierer im Team und im Dialog mit dem Rechner arbeiten, Programme entwickeln, korrigieren und dokumentieren konnten, ohne von den Restriktionen eines Großrechners abhängig zu sein. Im Vordergrund der Entwicklung standen Funktionalität, strukturelle Einfachheit und Transparenz sowie leichte Bedienbarkeit. Dieses erste System mit dem Namen „UNICS“ im Unterschied zu dem damals üblichen „MULTICS“ lief auf einer DEC PDP-7 [Tec00]. Die erste Version war in der Assemblersprache der PDP-7 geschrieben. Um bei künftigen Implementierungen die Maschinenabhängigkeit durch eine maschinennahe Sprache zu umgehen, entwarf Thompson die Programmiersprache B, aus der dann Dennis Ritchie später die Hochsprache C entwickelte [KR78]. UNIX wurde 1971 in C umgeschrieben und auf die PDP-11 übertragen [RT74]. Von nun an erfolgte die Weiterentwicklung des Systemkerns sowie der meisten Dienstprogramme in dieser Sprache. Die Kompaktheit und strukturelle Einfachheit des Systems ermunterte viele Benutzer zur eigenen Aktivität und Weiterentwicklung des Systems, so daß UNIX recht schnell einen relativ hohen Reife- und Bekanntheitsgrad erreichte. Eine ähhnliche Entwicklung zeigt sich seit einigen Jahren bei den freien UNIX-Varianten. Im Laufe der Zeit sind zwei Entwicklungszweige entstanden, da UNIX sowohl bei den Bell Laboratories (AT&T) als „System V“ als auch an der Universität von Berkley als „BSD“ weiterentwickelt wurde. Daneben gibt es zahlreiche weitere UNIX-Derivate, beispielsweise die frei erhältlichen Systeme „Free BSD“ und „LINUX“ für IBM-PC/AT kompatible Computer und andere Architekturen. Für einen historischen Überblick sei auf [Vah95], Kap. 1.1 verwiesen.

In „UNIX“ Betriebssystemen spielt traditionellerweise ein Befehlsinterpreter –die sogenannte „Shell“– eine große Rolle. Dieser Interpreter ist in der Regel mit einem Terminalemulator verbunden, über den der Benutzer mit dem System agiert und interaktiv Befehle, Befehlsketten, kleinere Programme in Form vom Skripten und Anwendungen starten und verwalten kann. Daneben gehören zu einem UNIX-System eine Menge an kleinen Dienstprogrammen, ohne die die tägliche Arbeit nicht vorzustellen wäre (siehe Abb. 3.2). Die Befehlsinterpreter[2] und kleinen Werkzeuge stellen die oben erwähnte Benutzeroberfläche von UNIX dar.[3] Die sinnvolle und fruchtba-

[1] Siehe auch http://www.melbournelinux.com/unix_history.html für eine Beschreibung der Entwicklung.

[2] Mittlerweile gibt es eine ganze Reihe von Shells, die im großen und ganzen denselben Funktionsumfang bieten, aber nicht notwendigerweise kompatibel sind.

[3] Heutzutage ist diese Aussage nicht ganz offensichtlich, da auf jedem System in der Regel ein

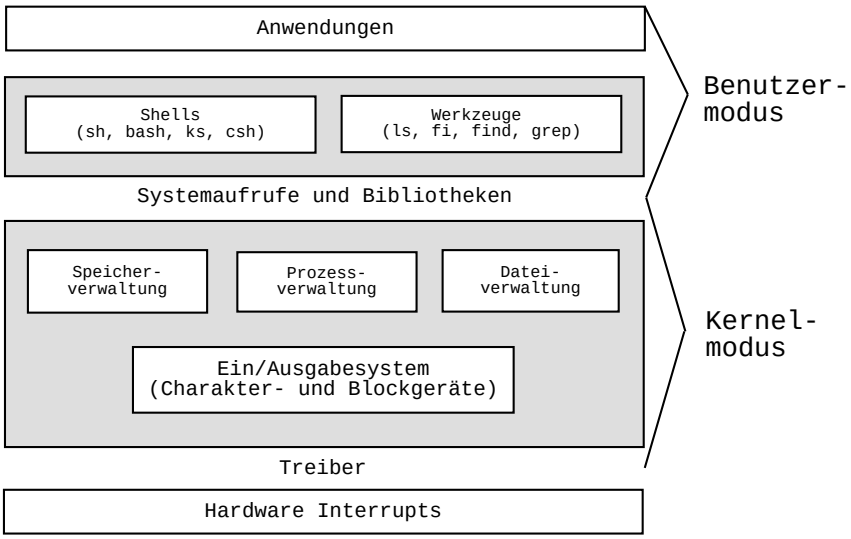

Abbildung 3.2: Das Schalenmodell von UNIX

re Kombination dieser Werkzeuge wird in einem der Leitsätze von UNIX deutlich: „Small is beautiful“. Der Betriebssystemkern wird bei UNIX-Derivaten „Kernel“ genannt, weil er meist relativ klein ist. Heutzutage ist er zusätzlich auch noch modular, weil bestimmte Teile –sogenannte „Module“– zur Laufzeit eingebunden werden können. Zu den wichtigsten Aufgaben der Kernels gehören die Prozeßkommunikation und Prozeßverwaltung, die Dateisystemverwaltung, die Ein-/Ausgabesteuerung, die Gerätesteuerung, sowie die Zugangs- und Rechtekontrolle. Natürlich stellt er den Anwendungen auch eine vollständige Schnittstelle in Form von Systemaufrufen zur Verfügung. Darüber hinaus werden alle Interrupts und I/O-Operationen im geschützten Kernelmodus abgewickelt.

Die Philosophie von UNIX wird treffend durch „In UNIX everything is a file“ ausgedrückt und war von Beginn an ein Merkmal des Systems. Operationen wie „Öffnen“, “Schließen“, „Lesen“ oder „Schreiben“ können unter UNIX gleichermaßen auf Dateien, Verzeichnisse, Geräte, Prozeßverbindungen –sogenannte „benannte Pipes“– oder Netzwerkverbindungen –sogenannte „Sockets“– angewendet werden (siehe [Bac86], [Tan92] und [Vah95]). Diese Vereinheitlichung des Zugriffs auf Systemressourcen erleichtert wiederum die Kombination verschiedener Werkzeuge ungemein und dient damit dem „Small is beautiful“ Konzept.

3.1.2 LINUX

Das frei verfügbare Betriebssystem „LINUX“ in der aktuellen Kernelversion 2.4 hat seinen Ursprung in dem von Andrew Tanenbaum zu akademischen Zwecken geschrie-

hardwarenaher X-Server läuft, der den Zugriff von verschiedenen Clienten über das X-Protokoll auf das System –auch über das Netzwerk– ermöglicht. Trotzdem geschieht die eigentliche Steuerung und Interaktion über Terminalemulatoren, die in diesem Fall ebenfalls X-Clienten darstellen (siehe http://www.tu-chemnitz.de/urz/kurse/unterlagen/unix-ben-umgebung/ofl_grundl.html für eine Einführung in das X-Windows-System).

benen, UNIX ähnlichen „MINIX“ [Tan87]. Linus Torvalds entwickelte 1991 noch als Student aus Unzufriedenheit mit den Fähigkeiten von MINIX und in Ermangelung eines anderen UNIX für die i386-Architektur sein eigenes UNIX, das er auf den Namen „LINUX“, einer Kontraktion aus seinem Vornamen und UNIX, taufte. In der Version 0.02 stellte er LINUX unter der GPL der Internetgemeinde zur Verfügung. Durch die Offenlegung des Quellcodes fanden sich zahlreiche Entwickler, so daß bereits 1994 eine für die Allgemeinheit funktionsfähige Version 1.0 zur Verfügung stand. Mit der fast zeitgleichen Einführung einiger LINUX-Distributionen erlebte LINUX einen wahrhaften Boom, der bis heute anhält. Die Distributionen ergänzen den reinen LINUX-Kernel um die notwendige Benutzersoftware wie Kommandointerpreter, Werkzeuge und wichtige Bibliotheken und bringen darüber hinaus eine Menge weiterer Software mit. Einen detaillierten Überblick über die Entwicklung des Kernels findet man in [Ron01], Kap. 1 und [Hal01].

Wie bei jedem UNIX gibt es in LINUX mehrere konkurrierende Prozesse, die verschiedene Aufgaben zu erledigen haben. Jeder Prozeß benötigt dazu den Zugriff auf Systemressourcen in Form von CPU-Zeit, Speicherplatz im RAM oder ähnlichem. Der Kernel läuft im sicheren Kernelmodus und hat die Aufgabe, diese Zugriffe zu koordinieren und voreinander zu schützen. Er erfüllt alle Anforderungen, die heutzutage an ein modernes UNIX ähnliches Betriebssystem gestellt werden. In Abb. 3.3 sind die wichtigsten Kernelsubsysteme gezeigt, die dazu nötig sind.

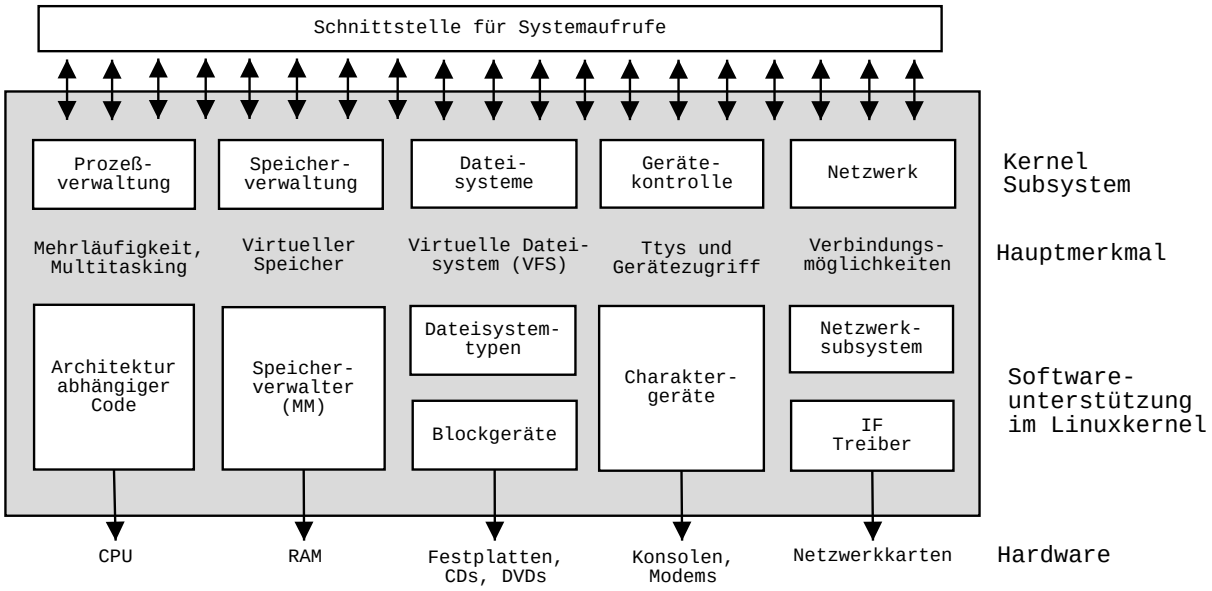

Abbildung 3.3: Überblick über den LINUX-Kernel

Prozeßverwaltung

Der Kernel erzeugt und vernichtet Prozesse und verwaltet ihre Verbindung zur Außenwelt durch die sogenannte „Prozeßumgebung“ sowie durch Eingabe- und Ausgabemöglichkeiten. Er unterstützt echten Mehrprogrammbetrieb, sogenanntes „präemptives Multitasking“, so daß die Prozesse völlig unabhängig voneinander laufen

können und sich nicht um das Abgeben von Rechenzeit kümmern müssen. Der Zugriff auf die CPU wird statt dessen vom „Scheduler“ erledigt. Er stellt einen wichtigen Teil der Prozeßverwaltung dar. Seit der Version 2.0 des Kernels kann dieser nicht nur die Abstraktion von mehreren Prozessen auf einen Prozessor vornehmen, sondern auch auf mehrere, und somit echte parallele Verarbeitung, sogenanntes „Multiprozessing“, ermöglichen. Der Kernel unterstützt darüber hinaus die Kommunikation von Prozessen untereinander in Form von Signalen, Pipes oder IPC-Direktiven. Prozesse können Benutzern des Systems eindeutig zugeordnet werden, so daß auch mehrere Benutzer gleichzeitig an dem System arbeiten können, sogenannte „Multiuser-Fähigkeit“.

Ein Benutzerprozeß kann sich im Nutzermodus oder im Kernelmodus befinden. Wenn er auf Systemressourcen zugreifen will, kann er das nur über die vom Kernel zur Verfügung gestellte Systemaufrufschnittstelle tun. Für die Dauer des Systemrufes befindet sich der Prozeß dann im höher privilegierten Kernelmodus. Ein Kernelprozeß ist ein Prozeß, der innerhalb des Kernels gestartet wird, und dann immer im Kernelmodus läuft.

Speicherverwaltung

Der Hauptspeicher des Computers ist eine der wichtigsten Ressourcen des Systems. Es ist klar, daß der Speicherverwalter (engl. „memory manager“, MM) diesen äußerst kritischen Bereich sicher und effizient zuteilen muß. Der physische Speicher wird in Speicherseiten mit fester, von der MMU abhängigen Größe eingeteilt, und jedem Prozeß wird sein eigener virtueller Adreßraum darüber zugewiesen. Die linearen virtuellen Speicheradressen werden durch einen meist dreistufigen Übersetzungsvorgang mithilfe der MMU in physische Seiten umgesetzt. Daten und Programmcode eines Prozesses sind in seinem Adreßraum verteilt. Die Größe des Adreßraums hängt von der Anzahl der Bits ab, mit der physische Speicherseiten adressiert werden, bei 32 Bit also maximal 4 GB. Der physische Speicher ist im allgemeinen ab einem architekturabhängigen Offset in den virtuellen Speicher eingeblendet. Zugriff auf diesen Bereich, das sogenannte „Kernelsegment“, ist aber nur im Kernelmodus erlaubt. Der tatsächlich nutzbare virtuelle Speicher, das sogenannte „Nutzersegment“, ist demnach immer kleiner als die theoretische Größe des Adreßraums. Bei Zugriff auf den Nutzerbereich muß der Kernel zumindest immer sicherstellen, ob Zeiger auch wirklich in das Nutzersegment verweisen. Verschiedene Teile des Kernels agieren daher mit dem Speicherverwalter über einen definierten Satz von Funktionen, so daß physischer Speicher niemals direkt allokiert werden kann. Die Sicherheit der physischen Speicherseiten vom Nutzermodus aus wird durch die Schutzmechanismen des Prozessors sichergestellt.

Das Konzept des virtuellen Speichers (engl. „virtual memory“, VM) ermöglicht, daß der Speicherverwalter nur die Teile eines Prozesses physisch allokieren muß, die wirklich zur Ausführungszeit referenziert werden. Bei Zugriff auf eine bereits reservierte Seite wird diese dann vom physischen Speicher eingeblendet. Es fördert auch das schnelle Erzeugen eines neuen Prozesses, da große Teile dessen Daten im Nutzersegment mit dem Vaterprozeß übereinstimmen und erst bei einem Schreibvorgang eines der beiden Prozesse auf gemeinsame Bereiche in eigene Speicherseiten kopiert werden müssen. Diese Technik wird „copy on write“ genannt. In diesem Zusammen-

hang spricht man auch von dem virtuellen Speicherverwalter (engl. „virtual memory manager“, VMM).
Wenn physischer Speicher knapp wird, dann lagert der Speichermanager modifizierte –sogenannte „modifierte“ (engl. „dirty“)– Speicherseiten von Nutzersegmenten in Auslagerungsbereiche auf einem temporären Massenspeicher aus. Unveränderte Speicherseiten, die aus Programmcode oder Daten aus eingeblendeten Dateien bestehen, werden verworfen, da sie bei Bedarf wieder gelesen werden können. Physische Speicherseiten im Kernelsegment, die vom Kernel selbst gebraucht werden, müssen immer im Primärspeicher vorhanden sein und werden daher nicht ausgelagert. Dieses sogenannte „Paging“ wird im wesentlichen durch die MMU vorgenommen und geschieht für Benutzerprozesse völlig transparent. Werden ausgelagerte Seiten wieder referenziert, dann schaut der Speicherverwalter nach, von welchem Ort die Seite einzulesen ist und verfährt ansonsten wie bei normaler Speicherallokation.
Ungenutzter physischer Speicher wird für den Cache von Blockgeräten genutzt, um die Zugriffsgeschwindigkeit auf diese zu erhöhen. Der Speicherverwalter kann dazu dynamisch je nach vorhandenem freien physischen Speicher entscheiden, wieviele Speicherseiten er dem System dazu zur Verfügung stellt. Auf den Cache werden wir in Abschnitt 3.3.1 genauer eingehen. Die Speicherverwaltung der Kernelversion 2.4 im Unterschied zu 2.2 beschreibt der Entwickler Rik v. Riel detailliert in [vR01].[4]

Dateisysteme

Wie UNIX ist auch LINUX stark auf dem erwähnten Dateisystemkonzept mit dem vereinheitlichten Namensraum aufgebaut: Im Prinzip kann fast jede vom Betriebssystem zur Verfügung gestellte Ressource als eine Datei betrachtet werden. Der Kernel setzt ein strukturiertes Dateisystem auf die unstrukturierte Hardware. Die resultierende Dateiabstraktion wird fast im ganzen System benutzt. Darüber hinaus unterstützt der Kernel verschiedene Dateisysteme, die über das sogenannte „virtuelle Dateisystem“ (engl. „virtual file system“, VFS) ins System eingebunden werden können. Dieses führt eine Zwischenschicht ein, die von der konkreten Realisierung eines bestimmten Dateisystems wie beispielsweise dem „Second Extended“ (Ext2FS, Ext2) oder dem „ReiserFS“ (Reiser, RFS) Dateisystem abstrahiert. Diese Schicht werden wir in Abschnitt 3.3 genauer beschreiben. Auf die für diese Arbeit relevanten Dateisysteme gehen wir danach in Abschnitt 3.5 ein.

Gerätekontrolle

Fast jede Systemoperation resultiert in einem Zugriff auf ein physikalisches Peripheriegerät. Im Prinzip werden alle Gerätekontrolloperationen von Kernelcode ausgeführt, der spezifisch für das jeweils adressierte Gerät ist. Ein solcher Teil des Codes wird „Gerätetreiber“ (engl. „device driver“) genannt. Der Kernel benötigt einen Treiber für jeden Bus und für jedes angeschlossene Peripheriegerät, von der Festplatte über die Netzwerkkarte bis zur Tastatur. Auf einen speziellen Treiber, den SCSI-Treiber, werden wir unten in Abschnitt 3.4 noch eingehen.

[4] Es sei an dieser Stelle angemerkt, daß gerade die dort beschriebenen Vorgehensweisen für die Seitenersetzung –die Strategie nach der im Falle von Speicherengpässen vorgegangen wird– sich bei dem Kernel 2.4 von den Anfangsversionen bis zur Version 2.4.16, der Version die zum Zeitpunkt des Verfassens dieser Arbeit die offiziell aktuellste darstellt, dramatisch geändert hat. Für einen Überblick über die jüngsten Änderungen sei auf [Bar01b] verwiesen.

Netzwerk

Netzwerkoperationen müssen vom Betriebssystem verwaltet werden, da sie zunächst nicht spezifisch einem Prozeß zugeordnet sind: Eingehende Pakete sind asynchrone Ereignisse. Diese Pakete werden vom Kernel gesammelt, identifiziert und untersucht, bevor sie einem Prozeß übergeben werden können. Der Kernel übermittelt darüber hinaus Datenpakete zwischen Programm und Netzwerkgeräten. Er kontrolliert dazu die Programmausführung je nach Netzwerkaktivität. Auch die Weiterleitung und Adreßauflösung geschieht innerhalb dieses Subsystems.

Es ist klar, daß wir in den folgenden Abschnitten nur auf die für diese Arbeit relevanten Aspekte des Kernels in der Version 2.4 eingehen werden. Eine detaillierte Beschreibung des Kernels in der Version 2.2 –konkret anhand der Version 2.2.14– findet man in [BC01] und Referenzen. Eine allgemeine Architekturübersicht des Kernels 2.2 liefert [BHB99].[5] Über die Kernelprogrammierung in der aktuellen Version 2.4 informieren [BBD+01] und [RC01], wobei letzteres den Schwerpunkt auf Gerätetreiberprogrammierung legt.[6]

Programmcodeauszüge, die wir im folgenden in diesem Kapitel besprechen werden, beziehen sich immer auf die Kernelversion 2.4.9-ac16. Dabei steht das Kürzel „ac“ für die Kernelversion von Alan Cox, die neben dem Standardkernel von Torvalds eine Reihe von zusätzlichen Features bietet. Für die im folgenden gezeigten Ausschnitte besteht aber kein nennenswerter Unterschied zu dem Standardkernel.[7] Pfade zu Dateien des Quelltextes geben wir dabei immer relativ zu dem Wurzelverzeichnis des Kernelverzeichnisbaums (üblicherweise „/usr/src/linux/“ oder „/usr/src/linux-2.4.9-ac16/“) an.

3.2 Generelle Aspekte eines UNIX-Dateisystems

Ein Dateisystem ist das logische Hilfsmittel für ein Betriebssystem, Daten auf einem Blockgerät, in der Regel einer Festplatte, zu speichern oder zu lesen. Dabei spielt es keine Rolle, ob diese Festplatte an das Computersystem als lokales Laufwerk, Netzlaufwerk oder als verteiltes Laufwerk in einem SAN angeschlossen ist. Ein Dateisystem für ein UNIX ähnliches Betriebssystem implementiert dazu mindestens die folgenden Operationen:

- Erzeugen und Löschen von Dateien.

[5] Siehe dazu auch die konzeptuelle Architektur mit vielen Graphen auf der Webseite http://plg.uwaterloo.ca/~itbowman/CS746G/a1/ und die dazu passende konkrete Umsetzung unter http://plg.uwaterloo.ca/~itbowman/CS746G/a1/.

[6] Eine Zusammenstellung der Änderungen von Version 2.2 auf 2.4 findet sich unter der Seite http://linuxkernel.to/module/port-2.4/eng/lkp-all.html. Hilfreiche weitere Online-Quellen sind der Irc-Kanal „#kernelnewbies“ auf „irc.linux.com“ mit dazugehörender Webseite http://www.kernelnewbies.com, sowie die Webseiten http://kernelbook.sourceforge.net/ und http://www.linuxdoc.org/LDP/lki/lki.html.

[7] Für andere Bereiche gilt das aber ganz und gar nicht. Beispielsweise ist Cox bei den Veränderungen, die die Speicherverwaltung in den letzten Monaten erfahren hat, sehr viel konservativer umgegangen als Torvalds.

- Öffnen von Dateien zum Lesen und Schreiben.
- Suchen innerhalb einer Datei.
- Schließen von Dateien.
- Erzeugen von Verzeichnissen, die Gruppen von Dateien zusammenfassen.
- Anzeigen von Verzeichnisinhalten.
- Löschen von Verzeichnissen.

Datei

Eine Datei ist dabei einfach eine geordnete, endliche Folge von Elementen, wobei ein Element je nach konkreter Implementierung ein Maschinenwort, ein Zeichen oder ein Bit sein kann. Ein Programm bzw. ein Benutzer kann diese Dateien nur durch Dateisystemaufrufe erzeugen, ändern oder löschen. Generell gilt für alle UNIX-Systeme, daß auf Dateisystemebene eine Datei formatlos ist. Jegliche Formatierung wird durch höhere Kernelmodule oder Benutzerprogramme vorgenommen, wenn das erwünscht ist. Sobald es einen Benutzer betrifft, hat eine Datei einen symbolhalften Namen. Dieser Name darf bis zu einer dateisystemspezifischen Länge haben und muß gegebenenfalls bestimmte Kriterien bei der Syntax erfüllen. Ein Benutzer kann ein Element einer Datei ansprechen, indem er den Namen der Datei und den linearen Index des Elementes innerhalb der Datei angibt. Mit höheren Modulen oder Programmen kann ein Benutzer auch entsprechend definierte Sequenzen von Elementen direkt durch ihren Kontext referenzieren.

Verzeichnis

Ein Verzeichnis ist eine spezielle Datei, die vom Dateisystem selbst verwaltet wird und deren Inhalt eine Liste von „Einträgen“ ist. Für den Benutzer sieht ein solcher Eintrag aus wie eine Datei. Ein Eintragsname muß nur innerhalb des Verzeichnisses, in dem er vorkommt, eindeutig sein. Daher bezieht sich die Gültigkeit eines Dateinamens in UNIX und in LINUX nur auf dieses Verzeichnis. In Wirklichkeit ist jeder Eintrag ein Zeiger einer der zwei folgenden Typen: Entweder der Eintrag zeigt direkt auf eine Datei, die ihrerseits natürlich wieder ein Verzeichnis sein kann, oder er zeigt auf einen anderen Eintrag in demselben oder einem anderen Verzeichnis. Wenn ein Eintrag auf einen anderen Eintrag zeigt, dann wird er „Verknüpfung“ (engl. „link“) genannt. Die einzige Information, die mit einer Verknüpfung assoziiert ist, ist der Zeiger auf den Eintrag, auf den die Verknüpfung zeigt. Dieser Zeiger wird durch einen symbolischen Namen angegeben, der innerhalb der jeweiligen Hierarchie eindeutig ist.

Hierarchie der Dateistruktur

Das eben beschriebene Konzept aus Dateien und Verzeichnissen definiert auf natürliche Weise eine Baumstruktur, bei der die Verzeichnisse „Knoten“ und die Dateien „Blätter“ darstellen. Eine Datei, die in einem Verzeichnis direkt als Eintrag aufgelistet wird, ist in dieser Hierarchie ein direkter Nachfolger zu diesem Verzeichnis. In Abb. 3.4 ist ein Beispiel für eine Hierarchie dargestellt. Verzeichnisse werden

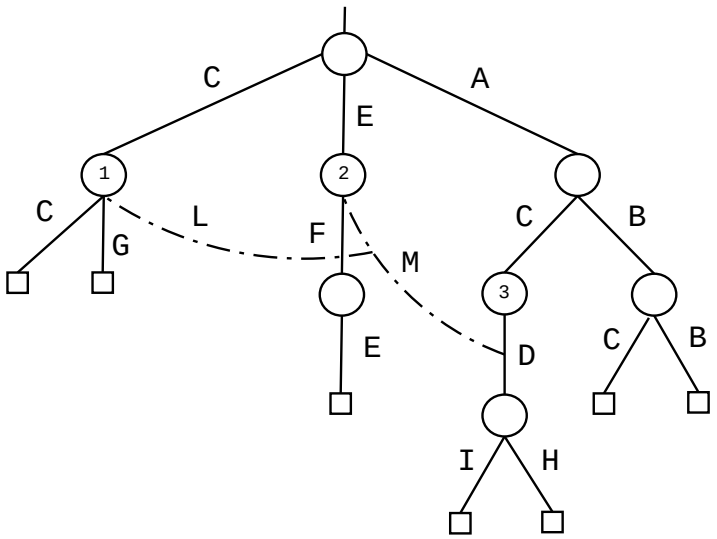

Abbildung 3.4: Beispiel einer Hierarchie

durch Kreise dargestellt, Dateien durch Vierecke. Die durchgezogenen Linien zeigen Einträge eines Verzeichnisses und deren Namen. In einem Verzeichnis dürfen per Definition keine doppelten Dateinamen vorkommen, in der Hierarchie insgesamt können sich Dateinamen aber sehr wohl wiederholen, wie gezeigt ist. Eingezeichnet sind zudem zwei Links, einer mit dem Namen „L“ aus dem Verzeichnis „1“ zeigt auf einen weiteren mit dem Namen „M“ aus dem Verzeichnis „2“. Dieser wiederum zeigt auf den Eintrag „D“ aus dem Verzeichnis „3“. Zu jedem Zeitpunkt befindet sich ein Benutzer bzw. ein Prozeß in einem einzigen Verzeichnis dieser Hierarchie, dem sogenannten „Arbeitsverzeichnis“ (engl. „working directory“). Der Pfadname einer Datei ist der symbolische Name einer Datei, der sie bezogen auf die Wurzel der Hierarchie eindeutig bezeichnet. Er setzt sich aus den Namen der Einträge zusammen, die nötig sind, ihn zu erreichen. Ein Pfadname kann absolut beginnend mit dem Wurzeleintrag oder relativ zum aktuellen Arbeitsverzeichnis angegeben werden. Jedes Verzeichnis beinhaltet normalerweise zwei Verweise mit den Namen „.“ und „..“. Der erste zeigt auf das Verzeichnis selbst, während der zweite das Vaterverzeichnis referenziert. Damit können Pfade auch zur Wurzel hin aufgelöst werden, was insbesondere bei relativen Pfadangaben wichtig ist.

Metadaten

Die Bedeutung eines „guten“ Dateiverwaltungssystems wird leicht unterschätzt. Während ein Mensch sein Gedächtnis dazu benutzen kann, um sich zu merken, wo er welche Daten notiert hat[8], benötigt ein Computer dazu andere Mittel. Er muß zusammen mit den eigentlich abzuspeichernden Daten auch Zugriffsstrukturdaten und Verwaltungsdaten, die sogenannten „Metadaten“, auf dem Datenträger vermerken. Dazu legt ein Dateisystem in der Formatierungsphase entsprechende Vermerke

[8]Gemeint sind Nutzdaten im Unterschied zu den gleich einzuführenden Metadaten.

und Strukturen an bestimmten Stellen auf dem Datenträger an.[9] Eine wichtige Forderung an ein Dateisystem ist daher die sinnvolle Strukturierung von Daten. Bei der Wahl dieser Strukturierung darf aber die Geschwindigkeit des Datenzugriffs sowie die Möglichkeit des wahlfreien Zugriffs auf die Blöcke des Blockgeräts nicht vernachlässigt werden. Die Bedeutung des wahlfreien Zugriffs und die Funktionalität der Blockgeräte haben wir im letzten Kapitel ausführlich untersucht. In LINUX spielt dabei der bereits in Abschnitt 3.1.2 erwähnte Puffercache eine große Rolle, wie wir im nächsten Abschnitt detaillierter sehen werden. Mithilfe seiner Funktionen ist es möglich, auf einen beliebigen der sequentiell durchnumerierten Blöcke eines bestimmten Blockgeräts zuzugreifen. Daher muß das Dateisystem mindestens in der Lage sein, eine eindeutige Zuordnung der Daten –und auch der Metadaten– auf die Geräteblöcke zu gewährleisten. In UNIX und auch in LINUX sind Daten in einem hierarchischen Dateisystem untergebracht, das Dateien unterschiedlichen Typs enthält. Es umfaßt zusätzlich zu den regulären Dateien und Verzeichnissen auch Gerätedateien, benannte Pipes, symbolische Links[10] und Sockets. Da Dateien per Definition strukturlos sind, muß das Dateisystem diese verschiedenen Typen unterscheiden und verwalten können.

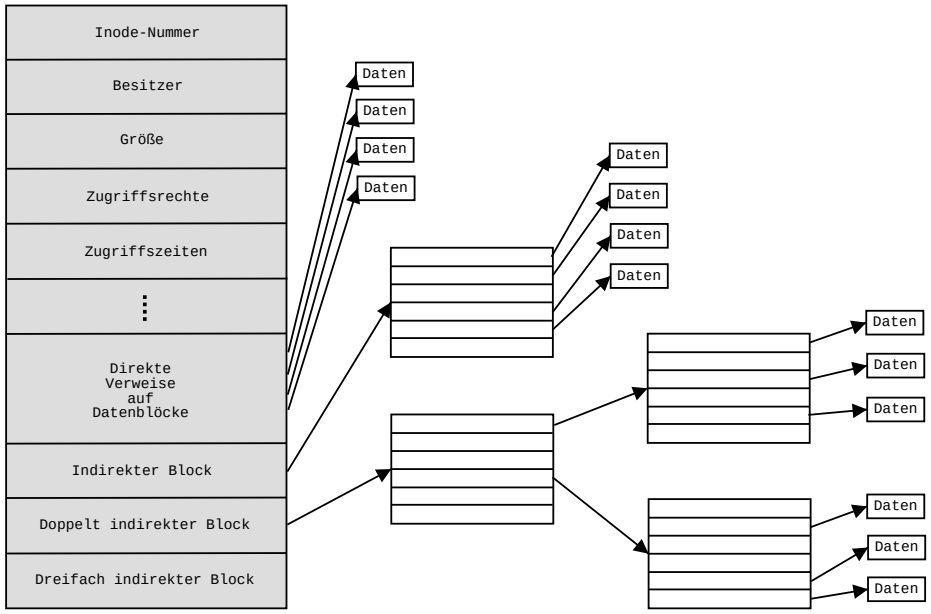

Abbildung 3.5: Prinzipieller Aufbau einer UNIX-Inode

[9] Diese Formatierung ist nicht mit der Low-Level-Formatierung aus dem letzten Kapitel zu verwechseln, erfüllt aber einen ähnlichen Zweck. Sie wird daher auch „High-Level-Formatierung“ (engl. „high level format“) genannt.

[10] Derartige Verweise haben wir oben bereits beschrieben. Im Unterschied zu sogenannten „harten“ Links (engl. „hard link“) funktioniert ein symbolischer Link –auch „soft link“ genannt– zudem über Dateisystemgrenzen hinweg. Ein harter Link erhöht dagegen den Referenzzähler einer Datei und funktioniert nur innerhalb des jeweiligen Dateisystems. Unter LINUX dürfen Verweise auf Verzeichnisse nur symbolische Links sein, um Rekursionen bei der Pfadnamenauflösung zu vermeiden.

Inode

Die zur Verwaltung nötigen Metadaten werden in UNIX streng von den Daten getrennt. Für jede Datei werden diese Informationen in einer separaten Strukur mit dem Namen „Inode“[11] zusammengefaßt. Sie vermerkt die Größe der Datei und unter anderem verschiedene Zugriffszeiten und Zugriffsrechte, sowie die Zuordnung zwischen Daten und Blöcken auf dem physischen Speichermedium. Wie in Abb. 3.5 zu sehen ist, enthält eine Inode einige direkte Verweise auf Datenblöcke, um den Zugriff auf kleine Dateien zu beschleunigen. Der Zugriff auf größere Dateien erfolgt über indirekte Blöcke, die ihrerseits erst auf die eigentlichen Datenblöcke zeigen. Um noch größere Dateien unterstützen zu können, werden zweifach oder sogar dreifach indirekte Verweise benutzt. Wie erwähnt wird jede Datei durch genau eine Inode repräsentiert. Innerhalb eines Dateisystems besitzt diese eine eindeutige Nummer, so daß jede Datei auch durch diese Inode-Nummer identifizierbar ist. Verzeichnisse sorgen –wie oben erläutert– für den hierarchischen Aufbau des Dateisystems und speichern eine Liste von Paaren aus dem Dateinamen und der dazugehörigen Inode-Nummer für alle sich im Verzeichnis befindenden Dateien.

Blockstruktur

Der Aufbau der unterschiedlichen UNIX-Dateisysteme ist von der Struktur her ähnlich [Köh94]. Er ist in Abb. 3.6 dargestellt. Jedes Dateisystem beginnt mit einem „Ladeblock“ (engl. „boot block“). Dieser Block ist reserviert, um den zum Laden des Betriebssystems, das sogenannte „Booten“, notwendigen Code aufzunehmen.

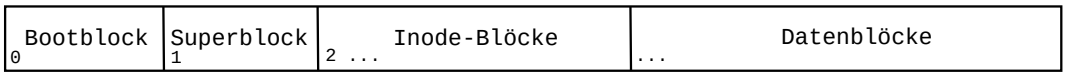

Abbildung 3.6: Schematischer Aufbau eines UNIX-Dateisystems

Aus Gründen der Einheitlichkeit und damit Einfachheit ist dieser Bootblock auch dann vorhanden, wenn das Dateisystem nicht zum Booten verwendet wird.[12] Alle für die globale Verwaltung des gesamten Dateisystems notwendigen Metadaten werden in einem ausgezeichneten Block untergebracht, der daher „Superblock“ (engl. „super block“) genannt wird. Danach folgen mehrere Inode-Blöcke, die die Inode-Strukturen des Dateisystems enthalten. Die verbleibenden Blöcke des Gerätes dienen zur Aufnahme der Daten. Diese Datenblöcke enthalten sowohl den Inhalt regulärer Dateien als auch die Verzeichniseinträge und die indirekten Blöcke. Die Unterschiede bei den klassischen UNIX-Dateisystemen bestehen vor allem in der Anordnung und Verteilung von Inode-Blöcken und Datenblöcken.

Vereinheitlichung

In UNIX werden einzelne Dateisysteme nicht wie bei anderen Betriebssystemen durch einzelne Bezeichner wie beispielsweise „Laufwerksbuchstaben“ angesprochen,

[11] Der Begriff Inode ist eine Zusammenziehung der Worte „index“ und „node“ ([Bac86], Seite 22).

[12] Die Größe des Bootblocks variiert von Architektur zu Architektur und ist nicht festgelegt. Ein Dateisystem sollte soviel Platz lassen, daß verschiedene Ladeprogramme auf verschiedenen Architekturen immer darin Platz finden.

sondern sie sind Teil des hierarchischen Verzeichnisbaums. Dazu muß man ein Dateisystem in ein bestehendes Verzeichnis „einhängen“ (engl. „mount“). Das dadurch überdeckte Verzeichnis wird „Einhängepunkt“ (engl. „mount point“) genannt und stellt die Wurzel des eingehängten Dateisystems dar. Seine Originalinhalte sind für die Dauer des Einhängens nicht mehr zu sehen, statt dessen wird der gesamte Inhalt des eingehängten Dateisystems gezeigt. Dateisysteme fügen sich somit somit nahtlos in den UNIX-Namensraum ein. Eine Zwischenschicht des Kernels, das bereits mehrmals erwähnte virtuelle Dateisystem, stellt dies sicher und ermöglicht auch das Einbinden von nicht UNIX artigen Dateisystemen. Wir werden diese Zwischenschicht im nächsten Abschnitt genauer vorstellen.
Da sich Dateisysteme auf unterschiedlichen Geräten befinden können, müssen die Dateisystemimplementierungen sich auch an unterschiedliche Gerätecharakteristika wie Blöckgrößen oder ähnliches anpassen. Dabei streben in der Regel alle Betriebssysteme nach Geräteunabhängigkeit, so daß es unerheblich ist, auf welchem Blockmedium die Daten abgelegt werden. In LINUX übernimmt die entsprechende konkrete Dateisystemimplementierung zusammen mit dem Gerätetreiber diese Aufgabe, so daß das virtuelle Dateisystem von LINUX mit geräteunabhängigen Strukturen arbeiten kann.

Datensicherheit

Ein bisher nur am Rande erwähnter Aspekt eines Dateisystems ist die Datensicherheit. Dazu gehören einerseits die Möglichkeiten der Konsistenzerhaltung von Metadaten und Daten sowie andererseits Mechanismen zur Gewährleistung des Datenschutzes. Zudem sollte das Dateisystem robust gegenüber Systemfehlern, Verletzungen der Datenintegrität sowie Programm- und Betriebssystemabstürzen sein.

3.3 Das virtuelle Dateisystem von LINUX

Wir stellen in diesem Abschnitt das bereits mehrmals erwähnte virtuelle Dateisystem vor, dem als abstrakte Kernelschicht zwischen Anwendungen und konkreten Dateisystemen eine wichtige Rolle zukommt. Die Implementierung dieser abstrakten Schicht geht auf die Arbeit von Kleiman zurück [Kle86].

Wir betrachten zunächst ein Dateisystem als eine Menge von verschiedenen Objekten. Sie stellen alle logischen Einheiten dar, die dieses Dateisystem für den Benutzer zusammenfaßt und abstrahiert. Eine Datei oder ein Verzeichnis sind offensichtliche Beispiele für Objekte. Ein anderes Beispiel ist ein symbolischer Link. Objekte, die einem Benutzer nicht direkt zugänglich sind, sind beispielsweise ein „Superblock“ oder eine „Inode“. Da die Anzahl der Dateien in einem UNIX-System problemlos die Größenordnung von 100000 oder mehr erreicht, liegt es nahe, die Aufgabe, die Dateisystemstrukturen zu organisieren und zu warten, von den Anwendungen zu isolieren. Strenggenommen ist sie allerdings in LINUX auch nicht Aufgabe des Kernels. Dieser hat keine Kenntnis von einem konkreten Dateisystem, nicht einmal von dem eigenen Ext2- oder Proc-Dateisystem. Alle konkreten Dateisysteme werden gleich behandelt und können sogar zur Laufzeit als Module eingebunden und entbunden werden (siehe Abb. 3.7). Dazu stellt der Kernel das virtuelle Dateisystem zur Verfügung, das

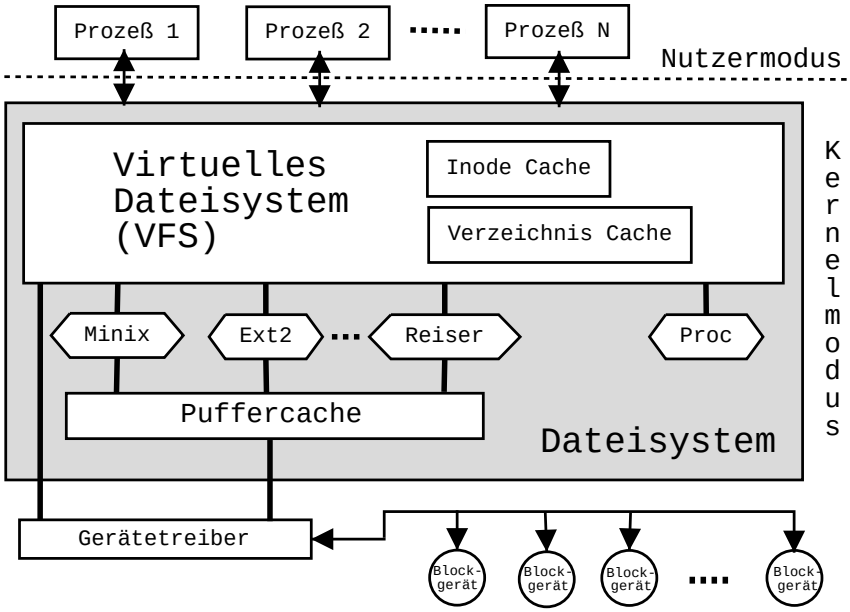

Abbildung 3.7: Die Schichten des Dateisystems

für jedes Dateisystemobjekt einen „Haken“ (engl. „hook“) liefert, an dem sich jede Instantiierung eines konkreten Objekts einhängen kann. Diese liefert gegenüber dem Kernel die eigene Funktionalität, indem es eigene Kontrollstrukturen erzeugt und verwaltet. Den Applikationen und Benutzerprozssen dagegen präsentiert das virtuelle Dateisystem immer einen vollständigen Satz an Systemaufrufen. Es verwaltet dazu eigene interne Strukturen und verschiedene Zwischenspeicher. Darüber hinaus implementiert das VFS Standardaktionen für jedes Objekt. Sie werden ausgeführt, falls das darunterliegende konkrete Dateisystem keine eigene Implementierung dafür liefert. Ein Beispiel dafür ist der Systemaufruf „llseek“, der den Dateizeiger einer Datei an eine bestimmte Stelle positioniert. In der Regel stellt kein Dateisystem eine eigene Implementierung dazu zur Verfügung, da die Standardaktion des VFS vollkommen ausreichende Funktionalität dafür liefert.

3.3.1 Puffercache

Sobald ein Dateisystem durch Benutzerprogramme eingerichtet ist, also seine Funktionen im VFS eingehängt und bekannt gemacht hat, ist es betriebsbereit und in die Verzeichnishierarchie eingebunden. Es wird je nach Auslastung des Systems dem Blockgerät, auf dem seine Daten liegen, eine Menge von Anfragen stellen, bestimmte Blöcke zu lesen oder zu schreiben. Die Blockanfragen werden dem Gerätetreiber in Form von „buffer_head“ Datenstrukturen (siehe Abb. 3.8) durch Standardkernelfunktionsaufrufe übergeben. Diese „Pufferköpfe“ liefern dem Treiber alle notwendigen Informationen, die gewünschten Daten auf dem Blockgerät zu finden. Die Gerätenummer „b_dev“ identifiziert das Gerät systemweit eindeutig, die Blocknummer „b_blocknr“ den Block auf dem Gerät eindeutig. Für den Fall, daß das zugrundeliegende Gerät ein Pseudogerät oder ein logisches Medium ist, gibt es noch

```
struct buffer_head {
  struct buffer_head *b_next; /*Hash queue list*/
  unsigned long b_blocknr;    /*block number*/
  unsigned short b_size;      /*block size*/
  unsigned short b_list;      /*List that this buffer appears*/
  kdev_t b_dev;               /*device (B_FREE = free)*/
  atomic_t b_count;           /*users using this block*/
  kdev_t b_rdev;              /*Real device*/
  unsigned long b_state;      /*buffer state bitmap (see above)*/
  unsigned long b_flushtime;  /*Time when (dirty) buffer should be written*/
  struct buffer_head *b_next_free;  /*lru/free list linkage*/
  struct buffer_head *b_prev_free;  /*doubly linked list of buffers*/
  struct buffer_head *b_this_page;  /*circular list of buffers in one page*/
  struct buffer_head *b_reqnext;    /*request queue*/
  struct buffer_head **b_pprev;     /*doubly linked list of hash-queue*/
  char * b_data;                    /*pointer to data block*/
  struct page *b_page;              /*the page this bh is mapped to*/
  void (*b_end_io)(struct buffer_head *bh, int uptodate); /*I/O completion*/
  void *b_private;                  /*reserved for b_end_io*/
  unsigned long b_rsector;          /*Real buffer location on disk*/
  wait_queue_head_t b_wait;
  struct inode *   b_inode;
  struct list_head b_inode_buffers; /*doubly linked list of inode dirty buffers*/
};
```

Abbildung 3.8: Die Strukur „buffer_head“ aus “include/linux/fs.h“

die Felder „b_rdev“ und „b_rsector“, die den realen Sektor auf dem realen Gerät angeben. Die Blockgröße „b_size“ kann ein Vielfaches von 512 Byte sein, aber die Speicherseitengröße –bei der i386-Architektur 4096 Byte– nicht überschreiten. Der Zeiger „b_data“ zeigt auf die Blockdaten in einem extra reservierten Bereich des physischen Speichers der Größe „b_size“. „b_page“ zeigt auf die Speicherseite, die den Puffer eingebettet hat und „b_this_page“ zeigt auf die Liste aller Puffer, die zu einer Seite gehören. Die Anzahl der den Block benutzenden Prozesse wird in „b_count“ vermerkt. Ein Block wird geladen, wenn er nicht im Puffer vorhanden ist. Er wird geschrieben, wenn der Pufferinhalt des Blocks nicht mehr mit dem externen Medium übereinstimmt. Dazu wird nach einer Schreiboperation der betroffene Block mit dem Flag „BH_Dirty“ als modifiziert gekennzeichnet, was unter anderem im Statusfeld „b_state“ vermerkt wird. Natürlich soll das Schreiben auf das Medium in der Regel verzögert erfolgen, da der gültige Inhalt des Puffers im Cache vorhanden ist.[13] In diesem Fall steht in „b_flushtime“ der Zeitpunkt, ab dem der Blockpuffer auf das Gerät geschrieben werden sollte. Dieser Wert wird jedesmal gesetzt, wenn ein Schreibzugriff erfolgt, so daß sichergestellt ist, daß immer eine gewisse Zeitspanne ohne erfolgte Schreibzugriffe gewartet wird, bevor der Block tatsächlich geschrieben wird. Weiterhin wird in dem Zustandsfeld notiert, ob der Blockinhalt mit dem Medium übereinstimmt („BH_Uptodate“) oder ob der Block exklusiv gesperrt ist

[13] Eine Ausnahme bilden Blöcke von Dateien, die mit dem Flag „O_SYNC“ geöffnet worden sind. Sie werden bei einer Modifikation sofort auf die Platte übertragen.

(„BH_Lock“).[14] Mit „BH_New“ wird angezeigt, daß der Pufferinhalt neu ist und noch nicht auf das Gerät geschrieben worden ist. „BH_Mapped“ dagegen bedeutet, daß dem Puffer auf dem Gerät ein Block zugewiesen ist. „BH_Protect“ schützt einen Puffer vor der Freigabe[15] und „BH_Req“ zeigt, ob der zum Puffer gehörende Block von einem Gerät angefordert worden ist. In „b_reqnext“ werden die zu erledigenden I/O-Anfragen gespeichert. Am Ende einer I/O-Operation wird der entsprechende Handler „b_end_io“ aufgerufen.

Der Puffercache besteht aus zwei funktionalen Teilen. Zum einen gibt es die Ringlisten der freien Blockpuffer: Jede unterstützte Blockgröße hat dazu ihre eigene Liste. Die freien Blöcke werden beim Systemstart oder beim Freiwerden entsprechend eingereiht. Sie werden durch „b_next_free“ und „b_prev_free“ doppelt verkettet und haben die Gerätenummer „B_FREE“. Zum anderen gibt es den eigentlichen Cache. Dieser ist als offene Hashtabelle realisiert, deren Einträge über „b_next“ auf Einträge mit gleichem Index verweisen. Der Index berechnet sich aus der Gerätenummer und der Blocknummer. Wenn Puffer in Benutzung und damit im Cache sind, werden sie zusätzlich in doppelt verketteten Listen gehalten.[16] Es gibt verschiedene Typen dieser sogenannten „LRU-Listen“ (engl. „last recently used“, LRU). Sie werden durch das Flag „b_list“ korrespondierend zu dem Statusflag der verketteten Blöcke unterschieden in „saubere“ („BUF_CLEAN“), modifizierte („BUF_DIRTY“), gesperrte („BUF_LOCKED“) oder geschützte („BUF_PROTECTED“) Listen.

Verwendung des Puffercaches

Um einen Block zu lesen, gibt es die Systemroutine „bread“ mit der Definition

```
struct buffer_head * bread(kdev_t dev, int block, int size)
```

aus „fs/buffer.c“. Sie überprüft zunächst, ob für den gewünschten Block mit der Nummer „block“ auf dem Gerät mit der Nummer „dev“ der entsprechende Eintrag nicht bereits im Cache vorhanden ist. Wird der Block bzw. sein Pufferkopf dort gefunden und ist „BH_Uptodate“ gesetzt, dann kann der gefundene Pufferkopf sofort zurückgeliefert werden. Ansonsten wird die Funktion „ll_rw_block“ aus „driver/block/ll_rw_block.c“ verwendet, die den entsprechenden Gerätetreiber auffordert, den gesuchten Block einzulesen. Nach Absetzen des Befehls an den Treiber muß der aktuelle Prozeß darauf warten, daß die Daten vom Medium geholt werden und liefert den Pufferkopf mit den geladenen Daten zurück. Mit „brelse“ wird der von „bread“ gelieferte Pufferkopf wieder freigegeben.

Für das Lesen oder Schreiben einer ganzen Seite gibt es die Funktion „brw_page“ mit der Definition

```
int brw_page(int rw, struct page *page, kdev_t dev, int b[], int size)
```

[14] Für diesen Fall gibt es die Warteschlange „b_wait“, in die sich die Prozesse einreihen müssen.

[15] Dieses Flag wird eigentlich nur in Zusammenhang mit „Hauptspeicherplatten“ (engl „RAM disk“) benutzt.

[16] Die Ringverkettung geschieht wiederum durch „b_next_free“ und „b_prev_free“. Diese Zeiger haben damit eine doppelte Funktion, die durch ihren Variablennamen nicht verdeutlicht wird.

aus „fs/buffer.c“. In Abhängigkeit von dem Parameter „rw“ schreibt sie entweder aus der Speicherseite „page“ die Blöcke der Größe „size“ mit den Blocknummern aus dem Vektor „b“ auf das Gerät „dev“ oder liest die Blöcke von dem Gerät in die Speicherseite ein.

Ein für diese Arbeit im weiteren wichtiger Systemaufruf ist „sync“. Er liefert immer als Ergebnis Null zurück und hat die Aufgabe, alle modifizierten Pufferblöcke im Puffercache inklusive alle Inodes und Superblöcke mit dem Medium zu synchronisieren. Seine Implementierung stützt sich auf die Funktion „sync_buffers“ mit der Definition

```
static int sync_buffers(kdev_t dev, int wait)
```

aus „fs/buffer.c“. Der Parameter „dev“ gibt das Gerät an, für dessen Blöcke die Synchronisation durchgeführt werden soll. Er kann auf Null gesetzt werden, was bedeutet, daß dann alle Puffer aus dem Cache synchronisiert werden. Der Parameter „wait“ gibt an, ob auf die Ausführung der Schreibanforderungen gewartet werden soll. Wenn nicht gewartet werden soll, wird für die Liste aller modifizierten und nicht gesperrten Blöcke eine Schreibanfrage gestellt. Wenn gewartet werden soll, wird der Puffercache zusätzlich dreimal durchlaufen. Erst wird auf die Beendigung aller gesperrten und modifizierten Blöcke gewartet und dann nochmal alle modifizierten und nicht gesperrten Blöcke geschrieben. Im dritten Schritt wird auf die Beendigung aller blockierten Puffer gewartet. Der erwähnte Synchronisationsaufruf über „sync“ besteht im wesentlichen aus der Routine „fsync_dev“ aus „fs/buffer.c“:

```
int fsync_dev(kdev_t dev)
{
    sync_buffers(dev, 0);
    lock_kernel();
    sync_inodes(dev);
    DQUOT_SYNC(dev);
    sync_supers(dev);
    unlock_kernel();
    return sync_buffers(dev, 1);
}
```

Sie wird in unserem Fall mit „dev=0“ aufgrufen, so daß alle Blockgeräte synchronisiert werden. Zunächst werden nicht wartende Schreibaufträge für alle modifizierten Puffer im System generiert. Danach werden die Inodes und Superblöcke der Dateisysteme synchronisiert. Da dies wieder modifizierte Puffer erzeugen kann, werden abschließend –diesmal aber wartend– alle modifizierten Puffer des Systems geschrieben.

Bdflush und Kupdate

Wie jeder Cache muß der Puffercache sinnvoll gewartet werden, damit er effektiv funktionieren kann. Dafür gibt es zwei Kernelprozesse „bdflush“ und „kupdate“, die modifizierte Puffer auf die Blockgeräte schreiben. Beide Routinen sind in „fs/buffer.c“ definiert. Sie werden bei Systemstart initialisiert und schlafen in der

Regel die meiste Zeit. Über den Systemaufruf „bdflush“ existiert eine Schnittstelle, mit der man das Verhalten beider Prozesse steuern kann. Das Benutzerprogramm „update“ implementiert diesen Befehl.[17] Die Parameter sind zusammen mit ihrer Bedeutung in Abb. 3.9 gezeigt. Die Kombination der beiden Threads hat den Sinn, die Anzahl der modifzierten Puffer im System gering zu halten, ohne das System dabei zu sehr zu belasten.

```
struct bdflush_param {
  int nfract;      /*Percentage of buffer cache dirty to activate bdflush*/
  int ndirty;      /*Maximum number of dirty blocks to write out per wake-cycle*/
  int nrefill;     /*No clean buffers to try to obtain each time we call refill*/
  int dummy1;      /*unused*/
  int interval;    /*jiffies delay between kupdate flushes*/
  int age_buffer;  /*Time for normal buffer to age before we flush it*/
  int nfract_sync; /*Percent buffer cache dirty to activate bdflush synchronously*/
  int dummy2;      /*unused*/
  int dummy3;      /*unused*/
} bdf_prm = {30, 64, 64, 256, 5*HZ, 30*HZ, 60, 0, 0};
```

Abbildung 3.9: Die Parameterstruktur für „bdflush“ aus „fs/buffer.c“

Bdflush läuft in einer Endlosschleife und schreibt standardmäßig, wenn mindestens 30 Prozent modifizierte Puffer im Cache vorhanden sind, eine maximale Anzahl von 64 modifizierten Blöcken.

Kupdate schreibt alle modifizierten Pufferblöcke, auf die seit einem gewissen Zeitraum nicht mehr zugegriffen wurde, sowie alle Superblock- und Inode-Informationen auf die Medien zurück. Standardmäßig wird Kupdate alle fünf Sekunden aufgeweckt. Ein modifizierter Puffer kann bis zu 30 Sekunden altern, bevor er von Kupdate synchronisiert wird. Wenn 60 Prozent der Puffer des Caches modfziert sind, dann wird geschrieben, ohne auf die Aktivierung von Bdflush zu warten.

Kswapd und Kreclaimd

Das Zusammenspiel der Dateisysteme mit dem Cache ist damit relativ einfach. Das Dateisystem bittet das System um eine Seite aus dem Seitenspeicher bzw. einen Puffer aus dem Puffercache.[18] Der Cache nimmt eine Seite von der Liste der freien Speicherseiten, um die Anfrage zu erfüllen. Dieser Vorgang kann natürlich nicht immer so weitergehen, da schlußendlich alle Speicherseiten des Systems für das Zwischenspeichern von Blockgeräten benutzt würden und damit das System keine Programme mehr ausführen könnte.

[17] Man kann alternativ auch „/proc/sys/vm/bdflush“ benutzen.

[18] Der Unterschied ist subtil. Wie wir wissen, ist jeder Puffer einer physikalischen Speicherseite zugeordnet. Speicherseiten, die nicht einer Inode oder einer anderen Strukur „address_space“ (siehe „include/linux/fs.h“) zugeordnet sind, werden zum Puffercache gezählt. Ihr Inhalt sind in der Regel reine Metadaten. Alle anderen Seiten, die sich im Hauptspeicher befinden und benutzt werden, zählen zum Seitenspeicher. Diese Seiten beinhalten nicht nur Nutzdaten von Blockgeräten, sondern beispielsweise auch Programme oder Programmdaten. Die Information über belegte Seiten kann man mit „cat“ aus „/proc/meminfo“ auslesen.

Um diese Möglichkeit zu verhindern, kann man wie folgt vorgehen: Wenn ein Kernelprozeß den Speicherverwalter nach einer freien Seite fragt und diese Anfrage nicht eilig ist[19], dann versucht der Speicherverwalter nach bestimmten Kriterien, zunächst belegte Seiten aus dem Seitenspeicher zurückzuforden, bevor er die Anfrage beantwortet. Diese instantane Strategie ist in Kernelversionen vor 2.4 umgesetzt worden. Sie hat aber den Nachteil, daß gerade in dem Moment in dem Aktivität im System zunimmt, auch noch freie Seiten gesucht werden müssen und das System damit zusätzlich in einem ungünstigen Moment belasten.

Daher hat LINUX 2.4 einen neuen Mechanismus. Der Kernel verwaltet globale Listen von freien, aktiven, modifizierten und inaktiven, aber sauberen Seiten. Auf inaktive Seiten ist seit längerem nicht mehr zugegriffen worden. Saubere inaktive Seiten stimmen mit dem Inhalt auf dem Medium überein. Modifizierte inaktive Seiten können entweder ausgelagert oder auf das Medium geschrieben werden. Werden freie Speicherseiten benötigt, dann sind die inaktiven Seiten als erstes Kandidaten dafür. Ihr Inhalt wird verworfen und die Speicherseite kann somit wiederverwendet werden oder der freien Liste hinzugefügt werden.

Zur Verwaltung der Listen gibt es zwei Kernelprozesse „kswapd“ und „kreclaimd“. Sie sind in „mm/vmscan.c“ definiert und werden bei Systemstart initialisiert. Durch ihre Arbeit im Hintergrund soll das System in der Lage sein, auf eventuelle Speicheranfragen schnell zu reagieren.
Kswapd überprüft in einer Endlosschleife in regelmäßigen Abständen, standardmäßig einmal pro Sekunde, die Speichersituation und führt dabei eine Alterung der Seiten durch. Er untersucht verschiedene Datenstrukturen, einschließlich der Prozeßtabellen und des Seitenspeichers bzw. des Pufferspeichers und überprüft jede Seite auf ihre letzte Zugriffszeit. Nach bestimmten Kriterien führt er dabei eine Alterung der Seiten durch. Seiten, die durch diesen Prozeß ein maximales Alter erreicht haben, macht er inaktiv. Kswapd wird zudem immer geweckt, wenn die Anzahl inaktiver Seiten unter bestimmten kritischen Schwellwerten liegt. Inaktive und modifizierte Seiten kann er nach Bedarf auslagern. Er kann dazu Bdflush wecken, so daß dieser modifizierte Seiten auf das Medium schreibt. Damit erhöht sich auch die Anzahl der inaktiven sauberen Seiten.
Kreclaimd wird nur aktiviert, wenn beim Allokieren von Speicherseiten festgestellt wird, daß zu wenig freie Seiten im System vorhanden sind. Er nimmt dann Speicherseiten von der Liste der inaktiven Seiten und überführt sie den freien Speicherseiten des Systems.

Das genaue Vorgehen und Zusammenspiel der einzelnen Komponenten ist weitaus komplizierter und unterliegt ständigen Detailverbesserungen (siehe [vR01] und die Bemerkungen in Abschnitt 3.1.2 zu diesem Thema).

3.3.2 Dateisystemobjekte

Das virtuelle Dateisystem besteht aus der Interaktion einer Reihe von Objekten, die das Speichern, Zugreifen und Verwalten von Daten über den Puffercache auf Blockgeräten unterstützen. Die wichtigsten Objekte des VFS sind die folgenden:

[19] Es stehen beispielsweise keine unbearbeiteten Interrupts aus.

- Dateien
  Dateien stellen die Objekte dar, von denen gelesen und in die geschrieben wird. Alternativ können sie auch in den Speicher eingeblendet werden. Ein Positionszeiger markiert innerhalb einer Datei die Stelle, an der als nächstes gelesen oder geschrieben wird. Da mehrere Prozesse auf die gleiche physische Datei zugreifen können, werden Dateien innerhalb des Kernels durch die Struktur „file“ dargestellt (siehe Abb. 3.11). Ein Prozeß, der eine Datei geöffnet hat, verwaltet ein Array von Dateiobjektzeigern. Der Index dieses Arrays wird „Dateideskriptor“ (engl. „file descriptor“, fd) genannt. Einem Dateiobjekt des Prozesses können mehrere Dateideskriptoren zugeordnet sein.[20] Die Operationen, die auf einer Datei ausgeführt werden können, werden in der Struktur „file_operations“ angegeben (siehe Abb. 3.12).

- Inodes
  Eine Inode ist das allgemeinste Objekt innerhalb eines Dateisystems (siehe Abb. 3.5). Sie kann eine reguläre Datei, ein Verzeichnis, einen symbolischen Link und weitere Objekte –beispielsweise Gerätedateien– darstellen. Auf der Ebene des VFS wird dabei nicht zwischen diesen Typen unterschieden. Das VFS überläßt es dem eigentlichen Dateisystem, das entsprechende Verhalten zu implementieren und den höheren Schichten des Kernels, mit den verschiedenen Typen unterschiedlich umzugehen. Jede Inode des VFS wird durch eine Struktur „inode“ repräsentiert (siehe Abb. 3.13). Sie korrespondiert in gewisser Weise mit der Inode auf dem darunterliegenden Dateisystem. Die auf die Inode anwendbaren Methoden sind in der Struktur „inode_operations“ (siehe Abb. 3.14) gespeichert. Auf den ersten Blick sehen Dateien und Inodes sehr ähnlich aus. Es macht dennoch Sinn, sie voneinander zu trennen. Es gibt Dateiobjekte, deren Inodes Eigenschaften haben, die Dateien nicht haben. Ein Beispiel ist der symbolische Link. Andererseits gibt es Dateien, die keiner Inode zugeordnet sind, wie Pipes und Sockets.[21] Zudem haben Dateien einen Status, den Inodes nicht haben. Der Positionszeiger zur Angabe, wo in der Datei der nächste Lese- oder Schreibbefehl stattfindet, macht bei einer Inode keinen Sinn.

- Dateisysteme
  Ein Dateisystem besteht aus einer Menge von Inodes, unter denen es eine ausgezeichnete Inode gibt, die als die „Wurzel-Inode“ (engl. „root inode“) bezeichnet wird. Die anderen Inodes werden üblicherweise angesprochen, indem ein Dateiname von der Wurzel beginnend aufgelöst wird. Dazu muß es beim Einhängen wissen, wo auf dem Datenträger die Wurzel-Inode zu finden ist. Ein Dateisystem hat eine bestimmte Anzahl von Charakteristika, welche für alle Inodes gültig sind. Solche sind beispielsweise die Blockgröße oder das „Nur-Lese-Flag“, welches beim Einbinden als Option angegeben werden kann. Es wird durch die Struktur „superblock“ repräsentiert (siehe Abb. 3.15). Die

[20] Anders könnten beispielsweise Ein/Ausgabeumleitungen wie „2>1“ in einer Shell nicht funktionieren. Siehe auch die Systemaufrufe „dup“, „dup2“ und „fcntl“.
[21] Gemeint sind nicht „benannte Pipes“ oder „UNIX Domain Sockets“.

Methoden, die auf einen Superblock anwendbar sind, stehen in der Struktur „super_operations“ (siehe Abb. 3.16). Zwischen Gerätenummern und Superblöcken besteht eine starke Beziehung. Es muß so aussehen, als ob jedes Dateisystem einem eindeutig identifizierbaren Gerät zugeordnet ist. Manche Dateisysteme, beispielsweise das Proc-Dateisystem, benötigen kein echtes Gerät. Für diese wird ein anonymes Gerät mit der Hauptnummer Null automatisch bereitgestellt. Die verschiedenen Dateisystemtypen, die dem VFS bekannt sind, werden in der Struktur „file_system_type“ gespeichert. Diese Struktur enthält nur die Methode bzw. den Funktionszeiger auf die Funktion „read_super“, die den Superblock des entsprechenden Dateisystems instantiiert.

- Namen und Dentrys
  Auf jede Inode innerhalb eines Dateisystems wird durch die Angabe eines Namens zugegriffen. Da die Auflösung der Abbilung „Name zu Inode“ je nach konkretem Dateisystem sehr zeitaufwendig sein kann, verwaltet das VFS einen Cache derzeit gültiger und zuletzt benutzter Namen. Dieser Cache wird „Dcache“ oder „Verzeichniscache“ genannt. Er ist im Speicher strukturiert wie ein Baum. Jeder Knoten in diesem Baum korrespondiert zu einer Inode. Einer Inode können mehrere Knoten in dem Baum zugeordnet sein. Der Baum ist kein vollständiges Abbild der Dateisystemhierarchie. Es ist aber zumindest sichergestellt, daß wenn ein Knoten des Dateibaums im Dcache ist, auch alle Vorgänger zwischengespeichert sind. Ein Knoten wird durch die Struktur „dentry“ repräsentiert (siehe Abb. 3.17). Die darauf anwendbaren Operationen sind in der Struktur „dentry_operations“ gespeichert (siehe Abb. 3.18). Zwischen einer Datei und einer Inode ist ein solcher „Dentry“ als Eintrag im Dcache zwischengeschaltet. Eine geöffnete Datei zeigt auf den Dentry, den sie erzeugt hat, der Dentry auf die Inode, die er referenziert. Dadurch werden für eine offene Datei der Dentry der Datei und die Dentrys aller Vorgänger im Speicher gehalten werden.

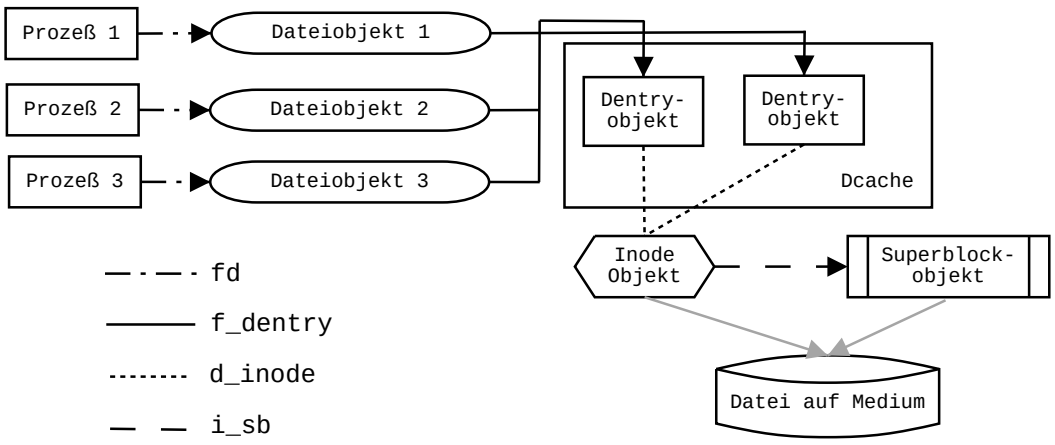

Abbildung 3.10: Das Zusammenspiel der VFS-Objekte

Das Zusammenspiel der verschiedenden Objekte ist in Abb. 3.10 skizziert. Wir wollen im folgenden auf die einzelnen Strukturen und ihre Operationen etwas näher eingehen.

Dateien

Wie oben erwähnt, zeigen Dateideskriptoren zeigen auf Dateiobjekte, so daß ein Prozeß durch sie auf die Daten der Datei zugreifen. Durch die Kapselung in Dateiobjekte ist es möglich, daß mehrere Prozesse eine Datei an verschiedenen Stellen lesen oder schreiben können. Die Struktur „file“ für ein Dateiobjekt ist in „include/linux/fs.h“ definiert und in Abb. 3.11 wiedergegeben. Mit „f_list“ werden mehrere

```
struct file {
  struct list_head  f_list;
  struct dentry          *f_dentry;
  struct vfsmount        *f_vfsmnt;
  struct file_operations *f_op;
  atomic_t      f_count;
  unsigned int  f_flags;
  mode_t        f_mode;
  loff_t        f_pos;
  unsigned long      f_reada, f_ramax, f_raend, f_ralen, f_rawin;
  struct fown_struct f_owner;
  unsigned int       f_uid, f_gid;
  int                f_error;
  unsigned long      f_version;
  void               *private_data; /*needed for tty driver, and maybe others*/
};
```

Abbildung 3.11: Die Struktur „file“ aus „include/linux/fs.h“

Dateistrukturen miteinander zu einer Liste verbunden. Für jedes aktive Dateisystem gibt es eine solche Liste, die an dem Zeiger „s_files“ im Superblock aufgehängt wird (siehe Abb. 3.15). Darüber hinaus gibt es je eine Liste für freie[22] und anonyme Dateistrukturen (in fs/file_table.c.). Wie erwähnt speichert „f_dentry“ den Dentry, der zu der Inode für die Datei zeigt. „f_op“ zeigt auf die Operationen, die mit der Datei möglich sind (siehe Abb. 3.12). Die Anzahl der Referenzen auf die Datei wird in „f_count“ vermerkt. Jeder Dateideskriptor eines Benutzerprozesses inkrementiert diese Zahl um eins. In „f_flags“ werden die Flags für die Datei gespeichert. Diese sind beispielsweise die Zugriffsart (lesend oder schreibend), nicht blockierend („O_NONBLOCK“) oder anhängend („O_APPEND“). Flags, die nur im Moment des Öffnens der Datei relevant sind, werden nicht vermerkt. Diese ausgeschlossenen Flags sind beispielsweise „O_CREAT“ zum Öffnen mit gleichzeitigem Erzeugen der Datei oder „O_TRUNC“ zum Öffnen beim gleichzeitigem Abschneiden der Datei. Die Zugriffsart, lesend oder schreibend, wird nochmal in „f_mode“ redundant gehalten, um sie durch zwei verschiedene Bits leichter erkennbar zu machen. In dem Positionszeiger „f_pos“ wird die aktuelle Position innerhalb der Datei

[22] Die Liste der freien Dateiobjekte wird unter anderem deswegen gepflegt, damit wichtige Prozesse im Fall von Speichermangel trotzdem Dateien öffnen können.

vermerkt. Diese wird für den nächsten Lesezugriff benutzt, und auch für den nächsten Schreibzugriff, wenn die Datei nicht mit „O_APPEND“ geöffnet worden ist. In „f_uid“ und „f_gid“ werden die Benutzeridentifikationsnummer (UID) und Gruppenidentifikationsnummer (GID) des Prozesses gespeichert, der die Datei geöffnet hat. Das Flag „f_read“ gibt an, ob im Voraus gelesen werden soll. „f_ramax“ vermerkt die maximale Anzahl an vorauslesbaren Seiten. „f_raend“ zeigt auf die Position in der Datei, bis zu der vorausgelesen worden ist. „f_ralen“ und „f_rawin“ geben die Anzahl der vorausgelesen Bytes und Seiten an. „f_error“ wird von Netzwerkdateisystemen benutzt, um Fehlercodes bei Schreiboperationen zurückzuliefern. „f_version“ ist eine Versionsnummer, die bei jedem Zugriff auf die Datei automatisch erhöht wird. Die nicht erwähnten Felder sind für uns nicht wichtig, sie dienen meist einem speziellen Zweck.

Die Struktur „file_operations“ ist die allgemeine Schnittstelle für die Arbeit mit Dateien. Die zu der Datei gehörende Inode hält einen Zeiger „i_fop“ auf diese Struktur. Er wird beim Erzeugen des Dateiobjekts auf „f_op“ kopiert. Funktionszeiger, die vom darunterliegenden Dateisystem nicht unterstützt werden, werden auf „NULL“ gesetzt und damit ungültig gemacht. Das VFS überprüft in Systemaufrufen bei Benutzung eines Funktionszeigers immer, ob er gültig ist. Wenn er gültig ist, wird die konkrete Funktion ausgeführt. Ansonsten führt das VFS eine Standardaktion aus oder liefert einen Fehler, meist „EINVAL“ für „Invalid argument“, zurück. Die Liste der unterstützten Dateifunktionen aus „include/linux/fs.h“ ist in Abb. 3.12

```
struct file_operations {
  struct module *owner;
  loff_t (*llseek) (struct file *, loff_t, int);
  ssize_t (*read) (struct file *, char *, size_t, loff_t *);
  ssize_t (*write) (struct file *, const char *, size_t, loff_t *);
  int (*readdir) (struct file *, void *, filldir_t);
  unsigned int (*poll) (struct file *, struct poll_table_struct *);
  int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
  int (*mmap) (struct file *, struct vm_area_struct *);
  int (*open) (struct inode *, struct file *);
  int (*flush) (struct file *);
  int (*release) (struct inode *, struct file *);
  int (*fsync) (struct file *, struct dentry *, int datasync);
  int (*fasync) (int, struct file *, int);
  int (*lock) (struct file *, int, struct file_lock *);
  ssize_t (*readv) (struct file *, const struct iovec *, unsigned long, loff_t *);
  ssize_t (*writev) (struct file *, const struct iovec *, unsigned long, loff_t *);
  ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
  unsigned long (*get_unmapped_area)(struct file *, unsigned long,
                                     unsigned long, unsigned long, unsigned long);
};
```

Abbildung 3.12: Die Struktur „file_operations“ aus „include/linux/fs.h“

wiedergegeben. Im folgenden gehen wir auf die wichtigsten darunter ein:

- Die Funktion „llseek(file, offset, origin)“ implementiert den Systemaufruf „lseek“ bzw. „llseek“. Wenn sie vom darunterliegenden Dateisystem undefiniert gelas-

sen wird, dann wird die Standarddefinition „default_llseek“ aus „fs/read_write.c“ benutzt. Wie erwartet setzt diese den Dateizeiger der Datei auf die neue Position entsprechend der übergebenen Parameter „offset“ und „origin“.

- Die Funktion „read(file, buf, count, ppos)“ wird benutzt, um den Systemaufruf „read“ und einige Variationen davon zu implementieren. Es wird erwartet, daß das sie „count“ Bytes aus der Datei liest und in den Puffer„buf“ aus dem Nutzeradressraum kopiert. Das VFS testet vorher, ob dieser vollständig im Nutzeradressraum liegt und beschrieben werden kann, sowie ob der Dateizeiger gültig ist und die Datei zum Lesen geöffnet ist. Es wird erwartet, daß die Funktion die Dateiposition anpaßt, wenn die Arbeit mit solchen unterstützt wird.[23] Ist keine Funktion „read“ vom darunterliegenden Dateisystem definiert, wird der Fehler „EINVAL“ zurückgegeben.

- Die Funktion „write(file, buf, count, ppos)“ kopiert Daten aus dem Nutzeradressraum in die Datei und verhält sich ansonsten ähnlich wie „read“.

- Die Funktion „readdir(file, buf, callback)“ liest den nächsten Verzeichniseintrag aus der Datei, die ein Verzeichnis darstellen sollte, und liefert das Ergebnis in der Struktur „dirent“ zurück. Der Aufruf über den „callback“ und „buf“ ist etwas undurchsichtig, da diese Dateifunktion von zwei Systemaufrufen mit unterschiedlichen Formaten, „readdir“ und „getdents“, benutzt wird.

- Die Funktion „poll(file, poll_table)“ wird benutzt, um die Systemaufrufe „poll“ und „select“ zu implementieren. Sie überprüft im wesentlichen, ob die Datei aktiv ist.

- Die Funktion „ioctl(inode, file, cmd, arg)“ dient im eigentlichen Sinne zum Einstellen von gerätespezifischen Parametern. Der Systemruf „ioctl“ fragt vor der Weiterleitung an die Funktion einige Standardargumente wie „FIONCLEX“ oder „FIOCLEX“ ab. Befindet sich „cmd“ nicht unter diesen, dann wird überprüft, ob „file“ auf eine reguläre Datei verweist. Falls ja, dann wird die Funktion „file_ioctl“ aufgerufen und der Systemruf damit beendet. Für nicht reguläre Dateien testet das VFS, ob das darunterliegende Dateisystem eine Implementierung der Funktion zur Verfügung stellt. Fehlt sie, wird „ENOTTY“ als Fehler zurückgeliefert. Ansonsten wird die dateisystemspezifischen Funktion benutzt.

  Die Funktion „file_ioctl“ kennt die folgenden Kommandos, von denen die ersten beiden für unsere Zwecke wichtig sind.

  – „FIBMAP“ erwartet im Argument „arg“ einen Zeiger auf eine Blockzahl und liefert die logische Blocknummer dieses Blocks der Datei auf dem Gerät zurück, falls die zur Datei gehörende Inode die Funktion „get_block“ besitzt. Diese logische Nummer wird in „arg“ überliefert. Fehlen die Inode-Operationen wird „EBADF“, fehlt „get_block“ wird „EINVAL“ als Fehler zurückgegeben.

[23] Gerätetreiber ignorieren die Dateiposition meist, so daß diese stets Null ist.

  - „FIGETBSZ“ überliefert in „arg“ die Blockgröße des Dateisystems, in dem sich die Datei befindet. Wenn der Datei kein Superblock zugeordnet ist, dann wird „EBDAF“ als Fehler zurückgegeben.
  - „FIONREAD“ schreibt die Anzahl der innerhalb der Datei noch nicht gelesenen Bytes an die Adresse „arg“.

- Die Funktion „mmap(file, vm_area)“ bildet einen Teil einer Datei in den Nutzeradressraum des aktuellen Prozesses ab. Die Struktur „vm_area_struct“ beschreibt alle Eigenschaften des einzublendenden Speicherbereichs (siehe „include/linux/mm.h“).

- Die Funktion „open(inode, file)“ wird in der Regel nicht benötigt, da die Standardfunktion des VFS alle notwendigen Maßnahmen, wie das Allokieren der Dateistruktur und das Verbinden mit der entsprechenden Inode vornimmt.[24]

- Die Funktion „flush(file)“ wird aufgerufen, wenn ein Dateideskriptor mit dem Systemaufruf „close“ geschlossen wird, also der Referenzzähler der Datei „f_count“ dekrementiert wird. Damit wird gewährleistet, daß eventuell gepufferte Daten geschrieben werden. Fehlt diese Funktion im darunterliegenden Dateisystem, dann unternimmt das VFS keine eigene Aktion.

- Die Funktion „release(inode, file)“ wird bei der Freigabe des Dateiobjekts benutzt, wenn also „f_count“ Null wird. Ein Fehlen wird vom VFS ignoriert, da es die Aktualisierung der Inode selbst vornimmt.

- Die Funktion „fsync(file, dentry)“ wird zur Implementierung der Systembefehle „fsync“ und „fdatasync“ benutzt.[25] Die Funktion muß dafür sorgen, daß alle Puffer der Datei aktualisiert und auf das Gerät zurückgeschrieben sind. Implementiert ein Dateisystem diese Funktion nicht, dann liefert das VFS den Fehler „EINVAL“ zurück.

- Die Funktion „lock(file, op, file_lock)“ wird benutzt, wenn über den Systemaufruf „fcntl“ Dateisperren gesetzt oder gelesen werden. Definiert das darunterliegende Dateisystem diese Funktion nicht, führt das VFS die Standardaktionen „posix_test_lock“ und „posix_test_file“ aus (siehe „fs/locks.c“).

Inodes

Wir wir schon in Abschnitt 3.2 gesehen haben, stellt die Inode eine fundamentale Struktur in einem UNIX artigen Dateisystem dar. Allerdings gilt es zu unterscheiden zwischen der Inode des VFS und der Inode eines konkreten Dateisystems. Die VFS-Inode, manchmal auch „Vnode“ genannt, ist eine Struktur, die nur im Hauptspeicher existiert. Die konkrete Inode eines Dateisystems ist ein abstrakter Begriff für die Kontrollstruktur der Metadaten und Daten eines konkreten Dateisystems auf dem jeweiligen Blockgerät. Diese Struktur auf dem Blockgerät wird bei UNIX artigen Dateisystemen immer „Inode“ genannt. In Abb. 3.13 ist die Strukur „inode“ für die VFS-Inode aus „include/linux/fs.h“ mit eingefügten deutschen Kommentaren

[24] Eine Ausnahme stellt das Öffnen eines Gerätetreibers dar.
[25] Sie sind momentan identisch.

```
struct inode { /*Kommentare in Deutsch stehen nicht im Orginal-Code*/
  struct list_head i_hash;  /*Verschiedene Listen, siehe Text*/
  struct list_head i_list;
  struct list_head i_dentry;
  struct list_head i_dirty_buffers;
  unsigned long  i_ino;     /*Inode-Nummer  */
  atomic_t       i_count;   /*Referenzzähler*/
  kdev_t         i_dev;     /*Gerätenummer  */
  umode_t        i_mode; /*Dateiart und Zugriffsrechte*/
  nlink_t        i_nlink;   /*Anzahl harter Verweise auf die Inode*/
  uid_t          i_uid;     /*Benutzerid des Eigentümers*/
  gid_t          i_gid;     /*Gruppenid des Eigentümers*/
  kdev_t         i_rdev;    /*Echte Gerätenummer bei Gerätedateien*/
  loff_t         i_size;    /*Größe der Datei in Bytes*/
  time_t         i_atime;   /*Zeitpunkt des letzten Zugriffs*/
  time_t         i_mtime;   /*Zeitpunkt der letzten Änderung*/
  time_t         i_ctime;   /*Zeitpunkt der Erzeugung*/
  unsigned long  i_blksize; /*Blockgröße in Bytes*/
  unsigned long  i_blocks;  /*Anzahl der Blöcke*/
  unsigned long  i_version; /*Versionsverwaltung*/
  unsigned short i_bytes;   /*Anzahl Bytes im letzten Block*/
  struct semaphore    i_sem;          /*Semaphoren zur Zugriffsteuerung*/
  struct semaphore    i_zombie;
  struct inode_operations *i_op;      /*Inode-Operationen*/
  struct file_operations  *i_fop;     /*Datei-Operationen*/
  struct super_block      *i_sb;      /*Superblock-Operationen*/
  wait_queue_head_t        i_wait;    /*Warteschlange*/
  struct file_lock        *i_flock;   /*Dateisperren*/
  struct address_space    *i_mapping; /*Speicherseiten*/
  struct address_space     i_data;
  struct dquot            *i_dquot[MAXQUOTAS];
  struct pipe_inode_info  *i_pipe;    /*Diese drei Felder sollten       */
  struct block_device     *i_bdev;    /*eigentlich eine Union sein,     */
  struct char_device      *i_cdev;    /*da eine Inode nur eins davon ist*/
  unsigned long            i_dnotify_mask; /*Directory notify events*/
  struct dnotify_struct   *i_dnotify;      /*for directory notifications*/
  unsigned long  i_state;      /*Statusfeld*/
  unsigned int   i_flags;      /*Flags teilw. korrespondierend dem Superblock*/
  unsigned char  i_sock;       /*Inode ist Socket*/
  atomic_t       i_writecount; /*Zähler für Schreibzugriffe*/
  unsigned int   i_attr_flags;
  __u32          i_generation; /*Zählt die Wiederbenutzung der Inode bei NFS*/

  union { /*zweiter Teil der Inode mit dateisystemspezifischen Informationen*/
    struct ext2_inode_info     ext2_i;
    struct reiserfs_inode_info reiserfs_i;
    struct proc_inode_info     proc_i;
    struct socket              socket_i;
    /*weitere für das dateisystemspezifische Inode-Informationsstrukturen*/
    void  *generic_ip;
  } u;
};
```

Abbildung 3.13: Die modifizierte Struktur „inode“ aus „include/linux/fs.h“

gezeigt. Sie besteht aus zwei Teilen. Der erste beinhaltet alle Informationen, die das VFS für seine Funktion als Zwischenschicht benötigt und bei der Erzeugung der Inode mit Informationen aus der jeweiligen konkreten Inode füllt. Er beschreibt im wesentlichen die Dateiattribute und beinhaltet Verwaltungsinformationen über die Inode. Der zweite Teil, repräsentiert durch die Union „u“, besteht aus spezifischen Informationen, die das jeweilige Dateisystem liefert, zu dem die Inode gehört. Dieser Teil ist meist ein Ausschnitt der eigentlichen Inode auf dem Gerät. Aus Platzgründen sind nur einige Mitglieder der Union gezeigt.
Es ist wichtig, den Referenzzähler „i_count“ im Hauptspeicher, vom Referenzzähler „i_nlink“ im konkreten Dateisystem zu unterscheiden. Ersterer zählt die Anzahl der Referenzen auf die Inode im Hauptspeicher und entscheidet damit, ob die Inode „aktiv“ ist. Letzterer zählt die Referenzen von harten Verweisen im konkreten Dateisystem und zeigt damit an, ob die Inode auf dem Medium benötigt wird oder gelöscht werden kann. Als Zugriff auf aktive Inodes dient eine offene Hashtabelle im Speicher, die alle aktiven Inodes speichert. Die Liste „i_hash“ verbindet alle Inodes miteinander, die denselben Hashindex haben. Dieser berechnet sich aus der Adresse der Superblockstruktur und der Inode-Nummer „i_no“. Es gibt drei weitere globale Ringlisten von Inodes, die Liste der aktiven („i_count“ größer Null), der inaktiven („i_count“ gleich Null) und der modifizierten (Statusflag „I_DIRTY“). Diese Listen zusammen mit der Hashtabelle werden auch als Inode-Cache bezeichnet. Innerhalb einer Liste sind die Inodes durch „i_list“ verbunden. Das Statusfeld „i_state“ zeigt an, ob eine Inode modifiziert („I_DIRTY“), gesperrt („I_LOCK“) oder unbenutzt („I_FREEING“) ist. Wenn eine Inode modifiziert worden ist, muß die zugrundeliegende Inode auf dem Medium beim nächsten Synchronisationsvorgang aktualisiert werden. Diese Liste sowie eine Liste der gesperrten Inodes werden im Superblock eingehängt (siehe Abb. 3.15). Die Liste „i_dentry“ zeigt auf alle Dentrys, die auf die Inode verweisen. Die Zeiger „i_op“, „i_fop“ und „i_sb“ verweisen auf die Funktionszeiger für die Inode-, Datei- und Superblockoperationen. Die Flags „i_flags“ korrespondieren zu den Flags „s_flags“ aus dem Superblock. Viele von ihnen gelten für das ganze Dateisystem. Es gibt aber auch Flags, die pro Inode einstellbar sind. Beispiele sind „MS_NOSUID“, „MS_NODEV“ oder „MS_NOEXEC“ zum Verhindern der Übernahme von Rechten, das Öffnen von Geräten oder das Ausführen der Datei.

Die Funktion „i_get“ aus „fs/inode.c“ mit der Definition

```
struct inode *iget(struct super_block *sb, unsigned long ino)
```

liefert die durch den Superblock und die Inode-Nummer bestimmte Inode zurück. Dazu schaut sie zunächst im Inode-Cache nach. Falls die gesuchte Inode dort zu finden ist, wird ihr Referenzzähler erhöht und die gefundene Inode zurückgeliefert. Ansonsten wird mit „get_new_inode“ eine neue VFS-Inode erzeugt und die Funktion „read_inode“ des Superblocks benutzt, um die Inode-Daten vom Medium zu lesen. Die Funktion „i_put“ wird dazu benutzt, um den Referenzzähler „i_count“ einer Inode runterzuzählen, und wenn nötig, ihre Ressourcen freizugeben. Wenn die Anzahl der harten Verweise „i_nlink“ auf Null dekrementiert worden ist, ruft sie die Funktion „delete_inode“ auf, wenn nur die Anzahl der Referenzen auf Null

reduziert worden ist, „put_inode“. Beides sind Funktionen des Superblocks (siehe Abb. 3.16). Die Struktur „inode_operations“ definiert die Operationen, die auf einer Inode definierbar sind. Sie dienen hauptsächlich zur Verwaltung von Dateien. In der Regel werden sie relativ direkt von den entsprechenden VFS-Systemaufrufen benutzt. Wie bei den Dateiobjektfunktionen gilt die Regel, daß das VFS Standardfunktionen bereithält, falls das darunterliegende Dateisystem eine Inode-Operation nicht definiert. Häufig wird bei Fehlen einer Implementierung aber auch der Fehler „EPERM“ für „Operation not permitted“ zurückgeliefert. Viele Operationen sind auch nur bei bestimmten Inode-Typen sinnvoll, so daß es beispielsweise unterschiedliche Implementierungen für einfache Dateien und Verzeichnisse gibt. Die Liste der unterstützten Funktionen aus „include/linux/fs.h“ ist in Abb. 3.14 wiedergegeben.

```
struct inode_operations {
  int (*create) (struct inode *,struct dentry *,int);
  struct dentry * (*lookup) (struct inode *,struct dentry *);
  int (*link) (struct dentry *,struct inode *,struct dentry *);
  int (*unlink) (struct inode *,struct dentry *);
  int (*symlink) (struct inode *,struct dentry *,const char *);
  int (*mkdir) (struct inode *,struct dentry *,int);
  int (*rmdir) (struct inode *,struct dentry *);
  int (*mknod) (struct inode *,struct dentry *,int,int);
  int (*rename) (struct inode *, struct dentry *, struct inode *, struct dentry *);
  int (*readlink) (struct dentry *, char *,int);
  int (*follow_link) (struct dentry *, struct nameidata *);
  void (*truncate) (struct inode *);
  int (*permission) (struct inode *, int);
  int (*revalidate) (struct dentry *);
  int (*setattr) (struct dentry *, struct iattr *);
  int (*getattr) (struct dentry *, struct iattr *);
};
```

Abbildung 3.14: Die Struktur „inode_operations“ aus „include/linux/fs.h“

Wir geben die wichtigsten darunter im folgenden an:

- Die Funktion „create(inode, dentry, mode)“ ist wie die folgenden sieben beschriebenen Funktionen nur für Verzeichnis-Inodes sinnvoll. Sie wird vom VFS aufgerufen, wenn eine Datei mit einem bestimmten Namen, der durch den Dentry gegeben ist, in einem Verzeichnis erzeugt werden soll. Dazu entnimmt sie mit der Funktion „get_empty_inode“ eine Inode aus der Liste der inaktiven Inodes. Sie füllt diese mit dateisystemspezifischen Informationen und sucht dazu eine freie Inode auf dem Medium. Der Dateiname aus dem Dentry wird anschließend in das Verzeichnis eingetragen und das Zugriffsfeld der neuen Datei auf „mode“ gesetzt. Die neue Inode wird als modifiziert gekennzeichnet, aktiv gemacht und der Hashtabelle mit „insert_inode_hash“ übergeben. Mittels „d_instantiate“ wird sie dem Dentry zugeordnet und damit dem Dcache hinzugefügt. Bei Fehlen der Funktion „create“ liefert das VFS den Fehler „EACCES“ zurück.

- Die Funktion „lookup(inode, dentry)“ sucht in dem Verzeichnis die Inode der Datei, deren Name in dem Dentry steht. Wird die Inode gefunden, wird sie mittels „d_add“ an den Dentry gebunden. Wird sie nicht gefunden, wird ein sogenannter „negativer“ Dentry zurückgeliefert, dessen Inode-Zeiger auf „NULL“ gesetzt ist.

- Die Funktion „link(dentry1, inode, dentry2)“ dient zum Anlegen eines harten Verweises. Im ersten Dentry steht der Name und die Inode der alten Datei, im zweiten Dentry nur der Name für die neue Datei und ein negativer Dentry. Die Inode verwaltet das Verzeichnis für die neue Datei. Bei Erfolg wird die neue Inode dem zweiten Dentry mittels „d_instantiate“ zugeordnet. Fehlt die Funktion „link“, dann liefert das VFS den Fehler „EPERM“ zurück.

- Die Funktion „unlink(inode, dentry)“ löscht die im Dentry angegebene Datei aus dem durch die Inode referenzierten Verzeichnis. Zuvor werden vom VFS die Zugriffsrechte überprüft. Bei Erfolg wird der Dentry aus dem Dcache mittels „d_delete“ entfernt. Existiert keine Implementierung von „unlink“, liefert das VFS den Fehler „EPERM“ zurück.

- Die Funktion „symlink(inode, dentry, name)“ richtet im durch die Inode angegebenen Verzeichnis den symbolischen Link „name“ ein. Zuvor werden vom VFS die Zugriffsrechte geprüft. Bei Erfolg instantiiert es den Dentry mit der neuen Inode durch die Funktion „d_instantiate“. Fehlt die Funktion „symlink“, liefert das VFS den Fehler „EPERM“ zurück.

- Die Funktion „mkdir(inode, dentry, mode)“ erzeugt in dem durch die Inode angebenen Verzeichnis ein neues Verzeichnis mit den Zugriffsrechten „mode“, dessen Name durch den negativen Dentry übergeben wird. Die Zugriffsrechte auf das Vaterverzeichnis werden zuvor bereits durch das VFS geprüft. Die Funktion sucht auf dem Medium nach einer neuen Inode, allokiert mindestens soviel Speicherplatz, daß sie die Standardverzeichnisse „.“ und „..“ anlegen kann und überführt die neue Inode in den Dcache und in die Hashtabelle. Bei Fehlen der Funktion „mkdir“ liefert das VFS den Fehler „EPERM“ zurück.

- Die Funktion „rmdir(inode, dentry)“ löscht das durch den Dentry angebene Unterverzeichnis aus dem durch die Inode referenzierten Verzeichnis. Wenn das zu löschende Verzeichnis leer ist, entfernt sie die dazu korrespondierende Inode und die verwendeten Blöcke auf dem Medium. Danach invalidiert sie den Dentry durch „d_delete“. Wie üblich werden die Zugriffsrechte auf die Inode zuvor durch das VFS überprüft. Bei Fehlen der Funktion „rmdir“ liefert das VFS den Fehler „EPERM“ zurück.

- Die Funktion „rename(inode1, dentry1, inode2, dentry2)“ verschiebt eine Datei bzw. ändert ihren Namen. Die ersten beiden Argumente definieren die Quelle, die letzten beiden das Ziel. Bevor die Änderungen am Dateisystem durchgeführt werden, überprüft das VFS die Zugriffsrechte der beiden Verzeichnisse gegeben durch die Inodes und stellt sicher, daß das Ziel kein Nachfolger der

Quelle ist. Ist die Funktion „rename“ nicht implementiert, liefert das VFS den Fehler „EPERM“ zurück.

- Die Funktion „readlink(dentry, buf, size)“ liest den symbolischen Link aus dem Dentry aus und kopiert den Pfad der Datei in den Puffer aus dem Nutzersegment mit der maximalen Länge von „size“ Bytes. Der Puffer wird zuvor durch das VFS geprüft. Fehlt die Implementierung der Funktion „readlink“, oder ist die zugeordnete Inode kein symbolischer Link, liefert das VFS den Fehler „EINVAL“ zurück.

- Die Funktion „truncate(inode)“ dient zum Abschneiden einer Datei, kann aber auch die Datei auf eine neue Größe verlängern, falls das darunterliegende Dateisystem dies implementiert. Das Feld „i_size“ der Inode muß zuvor auf die entsprechende Größe gesetzt worden sein. Die Funktion allokiert bzw. deallokiert entsprechend Speicherplatz auf dem Medium und aktualisiert die Inode.

Superblöcke

Jedes Dateisystem wird durch die Funktion „register_filesystem“

```
int register_filesystem(struct file_system_type * fs)
```

aus „fs/super.c“ dem VFS bekannt gemacht.[26] Sie hängt die Struktur „file_system_type“

```
struct file_system_type {
  const char *name;
  int fs_flags;
  struct super_block *(*read_super) (struct super_block*, void*, int);
  struct module *owner;
  struct file_system_type * next;
  struct list_head fs_supers;
};
```

aus „include/linux/fs.h“ in die Liste der bekannten Dateisysteme. Dadurch erhält das VFS den Namen des Dateisystems und einen Funktionszeiger „read_super“ auf die Funktion, die in der Lage ist, den Superblock des entsprechenden Dateisystems beim Einhängen des Dateisystems vom Blockgerät einzulesen. Die aktiven Superblöcke werden in der Liste „fs_supers“ gespeichert. Sie enthalten die globalen Informationen über das jeweilige Dateisystem. Die Superblockstruktur „super_block“ ist in Abb. 3.15 mit eingefügten deutschen Kommentaren wiedergegeben. Wie die Inode-Struktur besteht der VFS-Superblock aus zwei Teilen. Der erste Teil beinhaltet die allgemeinen Daten, die das VFS beim Erzeugen des Superblocks initialisiert und teilweise auch aus dem Superblock auf dem Medium einliest. Der zweite Teil, die Union „u“, beinhaltet die dateisystemspezifischen Superblockinformationen.
Die Superblöcke aller eingehängten Dateisysteme sind in der Ringliste „s_list“ verkettet, die desselben Dateisystemtyps in „s_instances“. Die Gerätenummer „s_dev“

[26] Es gibt auch eine Funktion „unregister_filesystem“ zum Abmelden eines angemeldeten Dateisystems. Dies macht Sinn beim dynamischen Laden von Dateisystemen als Kernelmodule.

```
struct super_block { /*Kommentare in Deutsch stehen nicht im Orginal-Code*/
  struct list_head s_list;           /*Liste aller Superblöcke*/
  kdev_t           s_dev;            /*Gerät des Dateisystems*/
  unsigned long    s_blocksize;      /*Blockgröße in Bytes als Potenz von 2*/
  unsigned char    s_blocksize_bits; /*Anzahl der benötigten Bits für Größe*/
  unsigned char    s_dirt;           /*Dirty-Flag*/
  unsigned long long       s_maxbytes; /*Maximale Dateigröße einer Datei*/
  struct file_system_type *s_type;     /*Zeiger auf den Dateisystemtyp /*
  struct super_operations *s_op;       /*Superblock-Operationen*/
  struct dquot_operations *dq_op;      /*Diskquota-Operationen*/
  unsigned long        s_flags;   /*Globale Flags*/
  unsigned long        s_magic;   /*Magischer String des Dateisystems*/
  struct dentry       *s_root;    /*Dentry der Wurzel*/
  struct rw_semaphore  s_umount;  /*Zugriffssteuerung durch Semaphoren*/
  struct semaphore     s_lock;
  int                  s_count;   /*Referenzzähler*/
  atomic_t             s_active;  /*Zugriffsanzeige*/
  struct list_head     s_dirty;         /*Liste der modifizierten Inodes*/
  struct list_head     s_locked_inodes; /*Liste der gesperrten Inodes*/
  struct list_head     s_files;         /*Liste geöffneter Dateien*/
  struct block_device *s_bdev;          /*Blockgerätestruktur*/
  struct list_head     s_instances;     /*Liste gleicher Superblöcke*/
  struct quota_info    s_dquot;         /*Diskquota spezifische Optionen*/

  union { /*2ter Teil des Superblocks mit dateisystemspezifischen Informationen*/
    struct ext2_sb_info ext2_sb;
    struct reiserfs_sb_info reiserfs_sb;
    /*weitere für das dateisystemspezifische Superblock-Informationsstrukturen*/
    void *generic_sbp;
  } u;
  struct semaphore s_vfs_rename_sem;  /*Schutz beim Umbenennen von Verz.*/
  struct semaphore s_nfsd_free_path_sem; /*Schutz beim Zugriff auf Unterverz.*/
};
```

Abbildung 3.15: Die modifizierte Struktur „super_block“ aus „include/linux/fs.h“

und die Gerätestruktur „s_bdev“ identifizieren das Gerät, auf dem sich das Dateisystem befindet. „s_blocksize“ ist die Blockgröße des Dateisystems. Sie muß eine Potenz von zwei sein. „s_maxbytes“ gibt die maximale Größe an, die eine Datei des Dateisystems höchstens haben kann. Die globalen Flags für das Dateisystem werden in „s_flags“ gespeichert. Sie werden mit den Inode-Flags „i_flag“ ver-odert. Ein Beispiel ist „MS_RDONLY“, bei dem das Dateisystem nur lesend eingehängt wird. Im Superblock werden die modifizierten und gesperrten Inode-Listen „s_dirty“ und „s_lock“, sowie die Liste geöffneter Dateien „s_files“ eingehängt. „s_op“ zeigt wie erwartet auf die Liste der vom Dateisystem unterstützten Operationen. Da der dateisystemspezifische Teil meist Informationen wie Bitmaps über verwendete Inodes oder ähnliches aus Effizienzgründen im Speicher hält, gibt es ein Flag „s_dirt“, das anzeigt, ob der Superblock modifiziert worden ist.

Die Superblock-Operationen dienen in der Regel dem Lesen und Schreiben von Inodes, dem Schreiben des Superblocks und dem Auslesen von Dateisystemstatistiken.

Abb. 3.16 zeigt die Struktur „super_operations“ aus „include/linux/fs.h“, die die entsprechenden Funktionszeiger definiert. Sie stellen Funktionen dar, die die konkrete

```
struct super_operations {
  void (*read_inode) (struct inode *);
  void (*read_inode2) (struct inode *, void *) ;
  void (*dirty_inode) (struct inode *);
  void (*write_inode) (struct inode *, int);
  void (*put_inode) (struct inode *);
  void (*delete_inode) (struct inode *);
  void (*put_super) (struct super_block *);
  void (*write_super) (struct super_block *);
  void (*write_super_lockfs) (struct super_block *);
  void (*unlockfs) (struct super_block *);
  int (*statfs) (struct super_block *, struct statfs *);
  int (*remount_fs) (struct super_block *, int *, char *);
  void (*clear_inode) (struct inode *);
  void (*umount_begin) (struct super_block *);
};
```

Abbildung 3.16: Die Struktur „super_operations“ aus „include/linux/fs.h“

Darstellung des Superblocks und der Inodes auf dem Datenträger in die allgemeine VFS-Form im Speicher überführen. Genaugenommen müssen die Inodes und der Superblock nicht einmal existieren. Ein Beispiel dafür ist das Dateisystem von MS-DOS. Bei diesem wird die FAT und die Informationen aus dem Bootblock in die interne Darstellung von Superblock und Inodes übersetzt. Manche Superblock-Operationen sind optional und müssen nicht implementiert werden. Eine Operation ist nicht implementiert, wenn der Zeiger auf „NULL“ gesetzt ist. Es findet in diesem Fall keine weitere Aktion statt. Ein Beispiel, das deutlich macht, daß die Abstraktion des VFS manchmal teuer erkauft wird, ist das folgende. ReiserFS benötigt zum Auffinden einer Inode 64 Bit an Informationen, statt 32 Bit wie eigentlich in der VFS-Inode vorgesehen. Daher wird als Trick beim Erzeugen und Einlesen einer Inode in der Funktion „get_new_inode“ aus „fs/inode.c“ zuerst geprüft, ob die optionale Funktion „read_inode2“ implementiert ist. Ist sie es, wird sie benutzt. Ist der Funktionszeiger aber „NULL“ wird die eigentlich zu implementierende Funktion „read_inode“ benutzt. Die zu „read_inode“ komplementäre Funktion ist die optionale Funktion „write_inode“. Sie muß nur von Dateisystemen implementiert werden, die auch beschreibbar sind. Die optionale Funktion „put_inode“ wird von „iput“ aufgerufen, wenn eine Inode nicht mehr gebraucht wird, etwa weil die zugehörige Datei geschlossen wurde. Die Funktion „clear_inode“ ist vom VFS implementiert und gibt in diesem Fall durch die VFS-Inode belegten Ressourcen frei, wenn die Inode nicht mehr referenziert wird, also „i_count“ durch das Schließen auf Null gesetzt wurde. Die optionale Funktion „delete_inode“ muß von beschreibbaren Dateisystemen implementiert werden. Sie wird von „iput“ aufgerufen, wenn „i_nlink“ gleich Null ist und die Inode damit auf dem Medium freigegeben werden muß. Die optionale Funktion „put_super“ wird beim Aushängen (engl. „unmount“) eines Dateisystems aufgerufen. Sie sollte durch den Superblock allokierte Ressourcen wieder freigeben

und bei Bedarf den Superblock mit dem Medium synchronisieren. Die optionalen Funktionen „write_super“ und „write_super_lockfs“ dienen zum Synchronisieren des Superblocks mit dem Medium. Letztere wird nur von Journaling-Dateisystemen benutzt. Diese synchronisieren mit „write_super“ alle bestätigten Änderungen an dem Superblock aus dem Journal. Mit „write_super_lockfs“ führen sie dagegen alle Änderungen durch, die im Journal vermerkt sind. Die Funktion „statfs“ wird von den Systemaufrufen „statfs“ und „fstatfs“ benutzt, um globale Informationen aus dem Superblock in die Struktur „statfs“ einzulesen. Beim Fehlen dieser Funktion liefert das VFS den Fehler „ENODEV“ mit der Bedeutung „No such device“ zurück.

Namen und Dentrys

Das VFS verwaltet die Pfadnamen der Dateien im Hierarchiebaum und versteckt diesen vor den eigentlichen Dateisystemen. Es übersetzt dazu einen Namen in einen Dentry, den es dann dem jeweils darunterliegenden Dateisystem übergibt. Die einzige Ausnahme stellt das Ziel eines symbolischen Links dar, welches einem Dateisystem durch direkte Namensinformation übergeben wird (siehe Abb. 3.14). Wie das Dateiobjekt stellt ein Dentry eine reine Speicherstruktur dar, die kein Pendant auf dem Medium besitzt. Ihr Sinn ist die Kapselung von Pfadnamen und zum Namen gehöriger Inode, sowie die Beschleunigung der Auflösung von Pfadnamen im Hierarchiebaum. Die dazugehörende Struktur „dentry“ aus „include/linux/dcache.h“ ist in Abb. 3.17 wiedergegeben.

```
struct dentry {
  atomic_t d_count;
  unsigned int d_flags;
  struct inode  * d_inode;    /*Where the name belongs to - NULL is negative*/
  struct dentry * d_parent;   /*parent directory*/
  struct list_head d_hash;    /*lookup hash list*/
  struct list_head d_lru;     /*d_count = 0 LRU list*/
  struct list_head d_child;   /*child of parent list*/
  struct list_head d_subdirs; /*our children*/
  struct list_head d_alias;   /*inode alias list*/
  int d_mounted;
  struct qstr d_name;
  unsigned long d_time;           /*used by d_revalidate*/
  struct dentry_operations *d_op;
  struct super_block * d_sb;      /*The root of the dentry tree*/
  unsigned long d_vfs_flags;
  void * d_fsdata;                /*fs-specific data*/
  unsigned char d_iname[DNAME_INLINE_LEN]; /*small names*/
};
```

Abbildung 3.17: Die Struktur „dentry“ aus „include/linux/dcache.h“

Der Pfadname eines Dentrys steht in der Struktur „d_name“ zusammen mit seinem Hashwert. Kurze Namen bzw. die ersten Buchstaben eines längeren Namens werden in „d_iname“ gespeichert. Der Zeiger „d_inode“ verweist in der Regel auf die zugeordnete Inode. Der Dentry „d_parent“ zeigt auf den Vorfahren im Hierarchiebaum. „d_subdirs“ ist die Liste der Dentrys, die im Hierarchiebaum nachfolgen. „d_child“

ist die Liste der Nachfolger des Vaters, zu denen auch der Dentry gehört. Die Anzahl der Referenzen von Nachfolgern wird in „d_count“ gespeichert. Nur Blätter können daher einen Wert von Null haben. „d_sb“ zeigt auf den Superblock des Dateisystems, zu dem die assoziierte Inode gehört. Dentrys, die auf dieselbe Inode zeigen, sind über „d_alias“ verlinkt. Der letzte Zeitpunkt des Zugriffs auf den Dentry wird in „d_time“ vermerkt. Die Operationen, die auf Dentryobjekte auszuführen sind, werden in der Struktur „d_op“ angegeben.
Ein Dentry befindet sich in einem der vier folgenden Zustände:

- Er ist frei und enthält daher keine gültigen Informationen.
- Er ist unbenutzt. Der Referenzzähler „d_count“ ist Null, aber der Zeiger „d_inode“ zeigt immer noch auf eine gültige Inode.
- Er ist in Benutzung. Der Referenzzähler ist positiv ist, und der Zeiger „d_inode“ verweist auf die zugeordnete Inode.
- Er ist negativ. Der Zeiger „i_node“ zeigt auf „NULL“, aber der Dentry ist im Dcache vorhanden.

Der Verzeichniscache bzw. Dcache ist eine globale Hashtabelle, in der über „d_hash“ doppelt verkettete Listen von Dentrys eingetragen sind. Diese Dentrys sind entweder unbenutzt, benutzt oder negativ. Der Dcache kontrolliert zudem den Inode-Cache. Die Inodes im Speicher, die unbenutzten Dentrys zugeordnet sind, werden nicht aus dem Speicher gelöscht, da der Dentry den Referenzzähler der Inode „i_count“ auf mindestens Eins hält. Alle unbenutzten Dentrys werden in der Liste „d_lru“ doppelt verlinkt. Die Einträge mit dem ältesten Zeitstempel „d_time“, sowie negative Dentrys werden vom VFS zuerst für neue Dentrys wiederverwendet.

Die Namensauflösung geschieht durch die Funktion „d_lookup“, das Eintragen in den Dcache mit „d_add“, das Austragen mit „d_drop“. Die Methoden, die auf Dentrys definierbar sind, werden in der Struktur „dentry_operations“ aus „include/linux/dcache.h“ definiert und sind in Abb. 3.18 abgebildet. Im Unterschied zu den vorangegangenen Strukturen besteht eigentlich kein Grund für ein Dateisystem, die Standard-

```
struct dentry_operations {
  int (*d_revalidate)(struct dentry *, int);
  int (*d_hash) (struct dentry *, struct qstr *);
  int (*d_compare) (struct dentry *, struct qstr *, struct qstr *);
  int (*d_delete)(struct dentry *);
  void (*d_release)(struct dentry *);
  void (*d_iput)(struct dentry *, struct inode *);
};
```

Abbildung 3.18: Die Struktur „dentry_operations“ aus „include/linux/dcache.h“

implementierung des VFS zu überschreiben. Daher sind alle Funktionen optional. „d_revalidate“ ist dazu gedacht, zu überprüfen, ob ein Dentryobjekt noch gültig ist, bevor es einem Dateisystem übergeben wird. Die Standarddefinition des VFS unternimmt nichts, aber Netzwerkdateisysteme könnten diese Funktion implementieren.

„d_hash“ berechnet aus dem Hashwert des Dentrys und seiner Adresse die Position innerhalb der Liste in der Hashtabelle. „d_compare“ vergleicht zwei Namenseinträge und muß bei speziellen Feinheiten wie der Unterscheidung zwischen Groß- und Kleinbuchstaben vom darunterliegenden Dateisystem implementiert werden. „d_delete“ entfernt einen Dentry aus dem Dcache und ruft dazu „d_iput“ und „d_drop“ auf. „d_iput“ kann implementiert werden, und wird dann anstelle von „i_put“ benutzt, um eine Inode aus dem Inode-Cache zu löschen.

Für die Originalarbeit zu dem Vnode-Konzept siehe [Kle86]. Für verschiedene Implementierungen unter UNIX siehe [Vah95], Kap. 8. Weitere Details zur Implementierung des VFS unter LINUX[27] findet man in [Bra98], [Bro99] und [Bar01a], Kap. 3. In [BC01], Kap. 12 und [BBD+01], Kap. 6 wird unter anderem genauer auf den Ablauf bestimmter Systemaufrufe wie beispielsweise „open“, „close“, „read“ oder „write“ eingegangen.

3.4 SCSI-Treiber unter LINUX

Wir haben im Abschnitt 3.3.1 gesehen, daß der Puffercache den Gerätetreiber durch die Funktion

```
void ll_rw_block(int rw, int nr, struct buffer_head * bhs[])
```

aus „fs/buffer.c“ auffordert, einen oder mehrere bestimmte Blöcke von bestimmten Geräten zu lesen oder zu schreiben. Der Gerätetreiber hat dann für jeden übergebenen Pufferkopf die Aufgabe, das entsprechende Gerät zu identifizieren, bei Bedarf die logische Blocknummer in physikalische Koordinaten umzurechnen und die entsprechenden Steuerbefehle abzusetzen.[28] Die Steuerbefehle haben wir im letzten Kapitel für verschiedene Festplattenschnittstelllen besprochen. Den globalen Zusammenhang der beteiligten Systeme haben wir bereits in Abb. 3.7 dargestellt.

Die Architektur des SCSI-Gerätetreibers von LINUX besteht aus drei Schichten, die in Abb. 3.19 dargestellt sind. Die oberste Schicht stellt die Schnittstelle zum Kernel und dadurch zum Benutzerinterface dar. Die mittlere Schicht stellt allgemeine Funktionen zur Verfügung und bildet die integrierende Schnittstelle zu der unteren Schicht, in der die eigentlichen Gerätetreiber implementiert sind.

Die Treiber „SD“ (engl. „scsi disk“) für Festplatten und magneto-optische Laufwerke und „SR“ (engl. „scsi rom“) für CD-ROM- und DVD-ROM-Laufwerke sind Blockgerätetreiber. Die Treiber „ST“ (engl „scsi tape“) für Magnetbandlaufwerke und „SG“ (engl. „scsi generic“) für generische und Pseudo-SCSI-Geräte sind Charaktergerätetreiber. Die Kernelschnittstelle zu beiden Treibertypen wird beispielsweise in [RC01]

[27] Sehr nützlich ist auch die Datei „Documentation/filesystems/vsf.txt“, in der ein Überblick über das VFS gegeben wird.

[28] Er muß zumindest die vom Dateisystem benutzte logische Blocknummer in die logische Blocknummer des Gerätes umwandeln. Da Blockgrößen des Dateisystems immer ein Vielfaches der Sektorgröße bzw. der logischen Blockgröße des Mediums darstellen, geschieht dies in der Regel durch eine Multiplikation mit einer Potenz von Zwei –meistens mit Acht für eine logische Dateisystemblockgröße von 4096 Byte und eine Sektorgröße von 512 Byte.

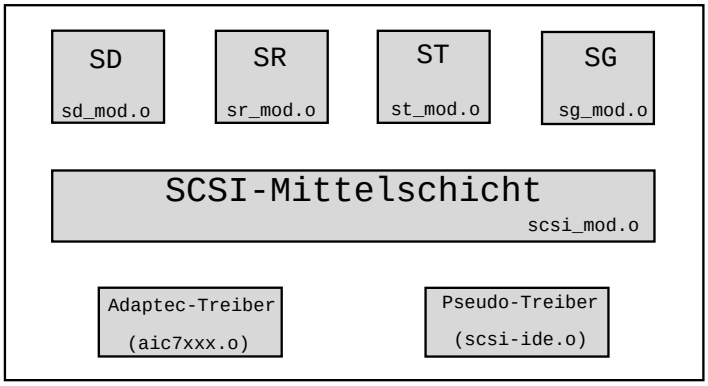

Abbildung 3.19: Überblick über die SCSI-Treiberarchitektur

beschrieben. Im wesentlichen muß dazu die Struktur „file_operations“ aus Abb. 3.12 für die Gerätedatei implementiert werden[29] und mit den Kernelfunktionen „register_blkdev“ und „register_chardev“ im System angemeldet werden.

Jede Operation, die das SCSI-Subsystem beinhaltet, benutzt einen Treiber der drei Schichten. Beispielsweise benötigt das Lesen eines Sektors von einer an den Adaptec-Controller angeschlossenen Festplatte für die oberste Schicht das Modul „sd_mod.o“, für die Mittelschicht das Modul „scsi_mod.o“ und für den eigentlichen Treiber des Hostadapters das Modul „aic7xxx.o“.

Die oberste Schicht implementiert die Befehle für verschiedene SCSI-Geräteklassen und stellt spezifische Befehle zur Gerätekontrolle über „ioctl“ zur Verfügung. Die Mittelschicht definiert interne Schnittstellen für die darüber- und darunterliegende Schicht und sorgt dadurch für die Integration verschiedener Hostadapter oder Pseudogeräte im System. Darüber hinaus stellt sie ihre „ioctl“ Aufrufe allen Dateideskriptoren der verschiedenen Treiber aus der obersten Schicht zur Verfügung. Intern verwaltet die Mittelschicht ein SCSI-Gerät durch die Angabe der Hostadapternummer, die Angabe der Busnummer, der Targetnummer und der LUN. Die Hostadapternummer wird vom Kernel für jeden im System gefundenen SCSI-Adapter um Eins beginnend bei Null erhöht. Jeder Hostadapter kann einen oder mehrere SCSI-Bustypen kontrollieren. Die Bustypen werden durch die Busnummer spezifiziert. Eine Targetnummer samt seiner LUN identifiziert ein Gerät an diesem Bus, wie wir aus dem letzten Kapitel wissen. Die Schnittstelle zum Kernel wird durch den Namensraum des VFS-Dateisystems ermöglicht. Zur Benutzung von außen besitzt ein Gerät eine ihm zugewiesene Gerätedatei mit einer Haupt- und Nebennummer als Modus, die durch das Programm „mknod“ erzeugt werden kann. Für SCSI-Platten sind die Hauptnummern 8, 65, 66, 67, 68, 69, 70 und 71 mit je 256 Nebennummern reserviert. Üblicherweise bezeichnet die Datei „/dev/sda“ mit dem Modus „b,8,0“ die erste SCSI-Platte an dem Hostadapter mit der Nummer Null und „/dev/sdb“

[29] Bei der Kernelversion 2.4 ist dazu mittlerweile eine eigene Struktur „block_device_operations“ für die Blockgeräte vorgesehen. Sie ersetzt die beschriebene Struktur und beinhaltet die Funktionen „open“, „release“, „ioctl“, „check_media_change“ und „revalidate“.

mit dem Modus „b,8,16“ die zweite Platte.[30] Die unterste Schicht implementiert den Zugriff auf den Bus über den jeweiligen Hostadapter und benutzt dazu die Eigenschaften des jeweiligen Adapters. Sie stellt damit den hardwareabhängigen Teil des Gerätetreibers dar. In [Cox99b] und [Cox99a] beschreibt Cox anhand eines fiktiven Adapters, wie ein solcher Treiber zu implementieren ist. Für weitere Informationen bezüglich der Architektur des SCSI-Subsystems im Kernel 2.4 sei auf [Gil01] und die Referenzen darin verwiesen.

3.4.1 Die Gerätebefehle der Mittelschicht

In diesem Abschnitt beschreiben wir einen Befehl der Mittelschicht, der für diese Arbeit in 4.4.5 wichtig werden wird. Wie oben erwähnt stellt die Mittelschicht eine allgemeine Schnittstelle zu den an den Bus angeschlossenen SCSI-Geräten dar. Mittels der von ihr zur Verfügung gestellten „ioctl“ Befehle kann man unter anderen einen beliebigen SCSI-Befehl zusammenstellen und an das Gerät absetzen. Der Name des uns interessierenden Befehls lautet „SCSI_IOCTL_SEND_COMMAND“. Die mit ihm verbundene Datenstruktur

```
struct sdata {
   unsigned int inlen;     /*# Bytes, die das Gerät lesen soll*/
   unsigned int outlen;    /*# Bytes, die das Gerät schreiben soll*/
   unsigned char cmd[x];   /*SCSI-Befehl mit 6 <= x <= 12*/
   unsigned char wdata[y]; /*Daten, die zum Gerät geschickt werden*/
  };
```

aus „drivers/scsi/scsi_ioctl.c“ stellt die dem Kernel zu übergebenden Daten dar. Die Struktur des Befehlsblocks „cmd“ haben wir in Abschnitt 2.2.7 erläutert und für einen generischen Befehlsblock in Abb. 2.15 dargestellt.

Zum Absetzen eines beliebigen SCSI-Befehls erzeugen wir eine Struktur „sdata“, füllen sie mit den relevanten Daten und übergeben sie dem Kernel zusammen mit dem Befehlscode von „SCSI_IOCTL_SEND_COMMAND“ und einem geöffneten SCSI-Gerätedateideskriptor durch den Systemaufruf „ioctl“. Seine Syntax ist durch

```
int ioctl(int fd, int request, char *arg)
```

gegegen. Er wechselt über einen Softwareinterrupt in die Kernelfunktion „sys_ioctl“ des VFS. Wie in Abschnitt 3.3.2 erläutert, ermittelt diese aus dem Dateideskriptor „fd“ die zu der Datei gehörende Inode und überprüft, ob die Inode regulär ist. In unserem Fall zeigt der Dateideskriptor beispielsweise auf die geöffnete Datei „/dev/sda“ einer SCSI-Platte. Die dazugehörende Inode stellt somit eine Gerätedatei dar. Auf diese Weise wird der „ioctl“ Befehl an das SCSI-Subsystem deligiert.[31] Das Subsystem setzt den SCSI-Befehl ab und liefert einen Returnwert. Ist der Returnwert

[30] Die Datei „Documentation/devices.txt“ gibt Aufschluß über die statische Zuweisung von Geräten zu Haupt- und Nebennummern.

[31] Die „ioctl“ Befehle werden vom Subsystem hierarchisch von oben nach unten abgearbeitet. Wenn die oberste Schicht einen Befehl auf einen offenen Dateideskriptor nicht erkennt, leitet sie ihn an die Mittelschicht weiter. Ist in dieser der Befehl auch nicht bekannt, wird er an die unterste Schicht weitergeleitet. Wenn auch diese den Befehl nicht implementiert hat, dann wird „EINVAL“ zurückgeliefert.

Null, ist der Befehl erfolgreich gewesen. Ist er positiv, dann ist der SCSI-Befehl mit einem Fehler beendet worden, dessen Status im niedrigsten Byte des Returnwerts überliefert wird. Ist er negativ, dann ist ein Fehler aufgetreten, der nichts mit dem SCSI-Subsystem zu tun hat.

3.4.2 Partitionen einer Festplatte

Für einige Zwecke kann es nützlich oder sogar notwendig sein, statt einer großen Festplatte mehrere, kleinere Festplatten zur organisierten Datenverwaltung benutzen zu können. So möchte man eventuell auf einem Rechner mehr als ein Betriebssystem installieren und bei Bedarf zwischen den verschiedenen Installationen wechseln können. Auch wenn man nur ein einzelnes Betriebssystem verwendet, kann es nützlich sein, eine Festplatte aufzuteilen, etwa um die Datensicherung zu vereinfachen oder verschiedene Dateisystemtypen voneinander zu isolieren.
Die meisten Betriebssysteme stellen dazu den Mechanismus der sogenannten „Partitionierung von Festplatten“ zur Verfügung. Eine „Partition“ ist dabei ein zusammenhängender Abschnitt von logischen Blöcken einer Festplatte. Die benutzten Partitionen einer Festplatte dürfen sich dabei niemals überlappen. Sie stoßen im Normalfall nahtlos aneinander. Es gibt aber keinen Mechanismus, der dies erzwingt.[32]
Eine Partition einer Festplatte wird vom Betriebssystem wie eine einzelne Festplatte behandelt. Auf einer Partition kann dann genau wie vorher auf einer Festplatte ein Dateisystem eingerichtet werden. Üblicherweise gibt man beim Einschalten des Rechners an, von welcher Partition man den Rechner starten möchte. Dazu muß sich im sogenannten „Master Boot Record“ der im BIOS vermerkten Startfestplatte[33] ein Festplattenverwaltungsprogramm befinden. Alternativ markiert man eine der Partitionen als aktiv, so daß das BIOS von dieser Partition den Bootvorgang des Betriebssystems vornimmt.

Im IBM-PC-Bereich werden Partitionen in der Regel mit dem Programm „fdisk“ erzeugt, das mit dem jeweiligen Betriebssystem mitgeliefert wird. Die Partitionierung einer Festplatte wird im ersten Sektor der Platte in einer Partitionstabelle festgehalten, die ursprünglich genau vier Einträge enthalten konnte. Damit war es möglich, eine physikalische Festplatte in maximal vier verschiedene logische Laufwerke, die sogenannten „primären“ Partitionen, zu unterteilen. Heutzutage kann man eine primäre Partition dazu benutzen, um eine sogenannte „erweiterte“ Partition, die im zweiten Sektor liegt, zu definieren. Diese läßt sich dann in beliebig viele sogenannte „logische“ Partitionen unterteilen. Die meisten Betriebssysteme starten jedoch nur aus einer primären Partition. Für weitere Informationen sei auf [HK01] verwiesen.

Der Umgang mit Partitionen erfolgt in den meisten UNIX-Versionen im Gerätetreiber, der nicht nur für jedes Gerät sondern auch für jede Partition eine eigene Gerätedatei mit Haupt- und Nebennummer zur Verfügung stellt. In LINUX werden dazu pro SCSI-Festplatte 15 Partitionen unterstützt, die sich in ihrer Nebennummer

[32] Normalerweise wird man eine Festplatte vollständig partitionieren, so daß der gesamte Platz der Platte genutzt wird und keine Lücken zwischen den Partitionen sind.
[33] Der „Master Boot Record“ ist ein ausgezeichneter Bereich einer Festplatte, im IBM-PC-Bereich üblicherweise der erste Sektor bzw. die ersten 512 Byte einer Platte.

unterscheiden. Die Partition $1 \leq x \leq 15$ auf der ersten Platte wird dabei meist durch die Datei „/dev/sdax“ adressiert. Sie besitzt den Modus „b,8,x“. Sogenannte „absolute“ Blocknummern, die vom Gerätetreiber als Koordinaten akzeptiert werden, geben lineare Blockadressen relativ zum Anfang der durch das Gerät repräsentierten Partition und nicht zum physikalischen Anfang der Festplatte an. Der Blocktreiber rechnet absolute Nummern in logische oder physikalische Koordinaten um, bevor er sie der Festplatte übergibt.

3.5 Klassische Dateisysteme unter LINUX

Wir wenden uns in diesem Abschnitt der konkreten Implementierung klassischer Dateisysteme und ihren Strukturen auf dem Blockmedium bzw. der Partition einer Festplatte zu.

3.5.1 Das S5FS-Dateisystem

Das Dateisystem von AT&T System V (S5FS) entspricht dem ursprünglichen UNIX-Dateisystem, das in [Bac86] und [RT74] vorgestellt wird. Neben der ursprünglichen Variante mit einer Blockgröße von 512 Byte sind auch noch Abwandlungen mit einer Blockgröße von 1 KB (S51K) und von 2 KB (S52K) gebräuchlich. Den folgenden Größenberechnungen liegt jedoch immer die Originalblockgröße von 512 Byte zugrunde. Die wesentlichen Konzepte von S5FS haben sich ausgezeichnet bewährt. Sie sind auch in den weiter unten zu besprechenden Dateisystemen FFS und Ext2 wiederzufinden.

Daten, Metadaten und Inodes

Wie in Abschnitt 3.2 ausführlich erläutert ist, definiert UNIX eine Datei als eine geordnete, endliche Folge von Bytes mit einer Reihe von weiteren Informationen, den Metadaten, die zur Verwaltung dieser Nutzdaten notwendig sind. Die Metadaten in S5FS bestehen aus dem Dateityp, dem Dateieigentümer, der Dateigruppe, den Basiszugriffsrechten für Eigentümer, Gruppe und alle anderen Benutzer, die Anzahl der harten Verweise, die Größe der Datei sowie die Zeitstempel des letzten Lesezugriffs auf die Daten, des letzten Schreibzugriffs auf die Daten und des letzten Schreibzugriffs auf die Metainformationen. Um es noch einmal zu betonen: Der Name einer Datei gehört in diesem Konzept nicht zu den Metadaten der Datei, sondern zu den Daten eines speziellen Dateityps, genannt „Verzeichnis“. Zu diesen Metadaten sind bis heute kaum weitere Informationen dazugekommen. Eine Datei auf dem Medium besteht aus der Zuordnung von Datenblöcken zu den Metadaten. Diese Zuordnung geschieht über die Inode, die wir in diesem Zusammenhang manchmal auch „Disk-Inode“ nennen, um sie von der VFS-Inode zu unterscheiden. Die Struktur „sysv_inode“ der S5FS-Inode aus „include/linux/sysv_fs.hist“ ist in Abb. 3.20 –für unsere Zwecke geringfügig modifiziert[34]– wiedergegeben. In dieser Form belegt

[34] Die Struktur „sysv_inode“ wird unter LINUX auch für das „Coherent“ Dateisystem benutzt. Dieses unterstützt benannte Pipes. Die für die Blockadressen und die Pipe verwendete Union zusammen mit der Struktur für die Pipe ist hier weggelassen.

```
struct sysv_inode {
  u16 i_mode;
  u16 i_nlink;
  u16 i_uid;
  u16 i_gid;
  u32 i_size;
  unsigned char i_addb[3*(10+1+1+1)+1]; /*zone numbers: max. 10 data blocks,
                                          1 indirection block,
                                          1 double indirection block,
                                          1 triple indirection block,
                                          maybe a "file generation number"??*/
  u32 i_atime;  /*time of last access*/
  u32 i_mtime;  /*time of last modification*/
  u32 i_ctime;  /*time of creation*/
};
```

Abbildung 3.20: Zur Darstellung der originalen S5FS-Inode dient die modifizierte Struktur „sysv_inode“ aus „include/linux/sysv_fs.h“.

sie 64 Byte auf der Festplatte. Bei einer Blockgröße von 512 Byte können demnach acht Inodes in einem Diskblock gespeichert werden.
Die in Abb. 3.20 gezeigte Inode stellt 40 Bytes zur Speicherung der Adressen von Datenblöcken zur Verfügung. Eine Adresse eines Blocks belegt dabei 3 Byte, so daß ein Dateisystem 2^{24} Blocknummern haben und damit bei einer Blockgröße von 512 Byte insgesamt maximal 8 GB groß sein kann. In der Inode werden die Adressen der ersten zehn Datenblöcke direkt (engl. „direct data blocks“) gespeichert. Damit kann bereits eine Datei mit einer Größe von 5 KB verwaltet werden. Größere Dateien werden daher indirekt adressiert. Die elfte Adresse ist die Adresse eines einfach indirekten Blocks (engl. „indirection block“), der die Adressen von 128 weiteren Datenblöcken enthält.[35] Entsprechend ist die zwölfte Adresse die Adresse eines zweifach indirekten Blocks (engl. „double indirection block“), der die Adressen von 128 indirekten Blöcken enthält. Die dreizehnte Adresse ist die Adresse eines dreifach indirekten Blocks (engl. „triple indirection block“), der die Adressen von 128 zweifach indirekten Blöcken enthält. In Abb. 3.5 haben wir eine allgemeine UNIX-Inode bereits dargestellt, die dieses Prinzip der Adressierung über direkte und indirekte Blöcke bereits verdeutlicht. Die abgebildete Inode benutzt allerdings keinen dreifach indirekten Block.

Blockstruktur

Die Zuordnung von Metadaten und Daten zu Dateien und Verzeichnissen ist im Prinzip durch die Disk-Inode gelöst. Wie aber entscheidet das Dateisystem, ob es sich bei einem bestimmten Block um einen Daten- oder Inode-Block handelt?
Dazu werden in S5FS beim Anlegen des Dateisystems die ersten beiden Blöcke der Festplatte für den Bootblock und den Superblock reserviert. Der Superblock enthält globale Metainformationen über das Dateisystem, im wesentlichen den Namen

[35] S5FS speichert in den indirekten Blöcken die Blockadressen in 32 Bit anstelle von 24 Bit. Dabei setzt es die höchsten 8 Bit immer auf Null.

des Dateisystems, seine Größe in Blöcken und die Größe der „Inode-Tabelle“ (engl. „inode table“), also die Anzahl an reservierten Inode-Blöcken. Außerdem speichert der Superblock wichtige dynamische Informationen, auf die wir weiter unten eingehen werden (siehe Abb. 3.21). Auf den Superblock folgen die Inode-Blöcke der erwähnten Inode-Tabelle. Ihre Größe wird statisch bei der Formatierung festgelegt. Sie ist eine wichtige Größe, beschränkt sie doch die maximale Anzahl an verschiedenen Dateisystemobjekten. Auf den letzten Inode-Block folgen im Anschluß direkt die dem Dateisystem zur Verfügung stehenden Dateiblöcke. Diese Blöcke stellen nicht direkt den freien Speicherplatz für die Benutzer des Dateisystems dar, denn sie werden auch für Verzeichniseinträge, indirekte oder mehrfach indirekte Blöcke und für die dynamischen Daten des Superblocks benötigt. Die beschriebene Anordnung der Blöcke haben wir bereits in Abb. 3.6 dargestellt.
Es ist klar, daß durch die statische Inode-Tabelle Datenblöcke von Inode-Blöcken für das Dateisystem einfach unterscheidbar sind. Innerhalb der Inode-Tabelle werden die Inodes beginnend mit Eins durchnumeriert.[36] Das Dateisystem ermittelt durch Angabe der Inode-Nummer die Blocknummer des Inode-Blocks durch

$$n_{\text{Block}} = (n_{\text{Inode}} - 1) \div n_{\text{Inodes pro Block}} + n_{\text{Startblock der Inode-Tabelle}}$$

und den Byteoffset innerhalb dieses Blocks durch

$$n_{\text{Offset}} = [(n_{\text{Inode}} - 1) \mod n_{\text{Inodes pro Block}}] \times \text{Inodegröße in Bytes}\,.$$

Verzeichnisse und Pfadnamen

Mit der Inode-Tabelle hat das Dateisystem bereits alle benötigten Informationen zur Verwaltung von Dateien. Den Inodes und ihren Daten sind aber noch keine Namen zugeordnet. Diese Zuordnung sowie die Erzeugung eines Hierarchiebaums erfolgt in UNIX über Verzeichnisse (siehe Abb. 3.4).
In S5FS besteht ein Verzeichnis aus einer Folge von Datensätzen aus jeweils 16 Byte. Die ersten zwei Byte enthalten die Inode-Nummer einer Datei, die folgenden 14 Byte speichern den Namen der Datei. Ist der Dateiname kürzer als 14 Zeichen, wird er mit Nullbytes aufgefüllt. Längere Dateinamen sind nicht möglich. Der erste Datensatz eines Verzeichnisses ist immer der Name „.“, der die Inode-Nummer des Verzeichnisses selbst enthält und damit auf sich selbst zeigt. Der zweite Eintrag ist immer der Name „..“, der die Inode-Nummer des übergeordneten Verzeichnisses speichert. Eine Ausnahme bildet die Wurzel-Inode mit der Inode-Nummer Zwei.[37] Beim Formatieren legt das Formatierungsprogramm –üblicherweise „mkfs“– dieses Wurzelverzeichnis an und ordnet sowohl „.“ als auch “..“ der Inode-Nummer Zwei zu.
Wie in Abschnitt 3.2 bereits erwähnt, ist es somit möglich, mehr als einen Verzeichniseintrag für eine Datei im Hierarchiebaum zu haben. Es sind keine Inkonsistenzen

[36] Man beachte den englischen Kommentar in Abb. 3.20. Die S5FS-Inode speichert tatsächlich keine eigene Inode-Nummer. Sie benutzt das 40. Adressierungsbyte einfach gar nicht, sondern –wie im folgenden beschrieben– einfach die Position der Inode innerhalb der Inode-Tabelle.

[37] Die Inode mit der Nummer Eins zeigt auf die Datei, die die für das Dateisystem „unbrauchbaren Blöcke“ (engl „bad blocks“) auflistet.

durch doppelte Datenhaltung zu befürchten, da alle Metainformationen der Datei in der Inode gespeichert sind. Weitere Verzeichniseinträge für eine existierende Datei erzeugt man –auch heute noch– mit dem Systemaufruf „link“, der einen alten und einen neuen Pfadnamen für die Datei mitgeteilt bekommen muß. Der Aufruf schlägt die Inode-Nummer der Datei im alten Verzeichnis über den alten Pfadnamen nach und erzeugt im neuen Verzeichnis einen Eintrag für den neuen Namen der Datei mit derselben Inode-Nummer. Dabei inkrementiert er in der Inode die Anzahl der harten Verweise „i_nlink“ um Eins. Beide Verzeichniseinträge verweisen damit auf dieselbe Inode und sind gleichberechtigt. Es ist nachträglich nicht feststellbar, welcher Name der Datei der originale Name war. Mit dem Systemaufruf „unlink“ können Pfadnamen einer Datei entfernt werden. Sobald einer Datei bzw. ihrer Inode kein Name mehr zugeordnet ist, also ihr Referenzzähler „i_nlink“ auf den Wert Null dekrementiert ist, wird der Speicherplatz für die Inode und die von ihr adressierten Datenblöcke wieder freigegeben.

Superblock

Ein Problem, das wir bisher noch nicht angesprochen haben, fällt dabei auf. Woran kann das Dateisystem freie von benutzten Blöcken unterscheiden? Da es keine Möglichkeit gibt, einen Datenblock zu kennzeichnen und diese Kennzeichnung von Nutzdaten zu unterscheiden, muß jedes Dateisystem auf eine gewisse Weise notieren, welche Datenblöcke frei oder belegt sind. Bei der Belegung der Inode-Blöcke ist der Sachverhalt ein wenig anders. Diese könnte man im Prinzip kennzeichnen, da sie eine dem Dateisystem bekannte Struktur haben. Allerdings benutzen die meisten Dateisysteme vor allem aus Effizienzgründen, aber auch aus Konsistenzgründen zusätzlich eine Kontrollstruktur für die Inodes.
S5FS führt sowohl eine Liste der freien Datenblöcke als auch eine Liste der freien Inodes im Superblock. Seine Struktur „sysv4_super_block“ aus „include/linux/sysv_fs.h“ für das „System V Release 4“[38] ist in Abb. 3.21 wiedergegeben. Darin werden die erwähnten statischen Angaben über die Größe des Dateisystems in Blöcken „s_fsize“ und die Größe der Inode-Tabelle gemacht. Letzteres geschieht indirekt durch Angabe des Offsets „s_isize“ auf den ersten Datenblock nach der Inode-Tabelle und dem Wissen, daß der Superblock genau 512 Byte groß ist. Darüber hinaus vermerkt das Dateisystem im Superblock die Anzahl der freien Datenblöcke „s_tfree“ und die Anzahl der freien Inodes „s_tinode“.

Liste freier Datenblöcke

Zur Verwaltung der freien Datenblöcke benutzt S5FS eine Liste freier Datenblöcke, die im Superblock an „s_free“ aufgehängt wird. Diese Liste besteht aus einem statischen Array, das bis zu 50 Blocknummern freier Datenblöcke speichert. Bei Allokation eines Diskblocks benutzt das Dateisystem den Block, dessen Nummer im Array durch „s_nfree“ gegeben ist, und dekrementiert „s_nfree“ um Eins. Ist „s_nfree“ bereits auf Eins reduziert, dann schaut es in dem letzten vorhandenen freien Block nach einem neuen Array mit maximal 50 freien Blöcken, von denen der nullte in der Regel wieder auf einen Block mit einem weiteren solchen Array oder sonst auf Null zeigt. Es kopiert das gefundene Array an die Stelle im Superblock und löscht

[38] Beim „System V Release 2“ sieht der Superblock leicht anders aus.

```
struct sysv4_super_block {
  u16 s_isize;    /*index of first data zone*/
  u16 s_pad0;
  u32 s_fsize;    /*total number of zones of this fs*/
  /*the start of the free block list:*/
  u16 s_nfree;    /*number of free blocks in s_free, <= SYSV_NICFREE*/
  u16 s_pad1;
  u32 s_free[50]; /*first free block list chunk*/
  /*the cache of free inodes:*/
  u16 s_ninode;   /*number of free inodes in s_inode, <= SYSV_NICINOD*/
  u16 s_pad2;
  sysv_ino_t s_inode[100]; /*some free inodes*/
  /*locks, not used by Linux:*/
  char s_flock;    /*lock during free block list manipulation*/
  char s_ilock;    /*lock during inode cache manipulation*/
  char s_fmod;     /*super-block modified flag*/
  char s_ronly;    /*flag whether fs is mounted read-only*/
  u32  s_time;     /*time of last super block update*/
  s16  s_dinfo[4]; /*device information ??*/
  u32  s_tfree;    /*total number of free zones*/
  u16  s_tinode;   /*total number of free inodes*/
  u16  s_pad3;
  char s_fname[6]; /*file system volume name*/
  char s_fpack[6]; /*file system pack name*/
  s32  s_fill[12];
  s32  s_state;    /*file system state: 0x7c269d38-s_time means clean*/
  s32  s_magic;    /*version of file system*/
  s32  s_type;     /*type of file system: 1 for 512 byte blocks
                                         2 for 1024 byte blocks*/
};
```

Abbildung 3.21: Die Struktur des S5FS-Superblocks „sysv4_super_block“ aus „include/linux/sysv_fs.h“ für das „System V Release 4“

den ursprünglichen Datenblock, dessen Array es gerade kopiert hat, mit Nullbytes, um ihn für die eigentlich geplante Allokation zu benutzen. Diese Liste freier Datenblöcke bestehend aus Arrays mit jeweils 50 freien Datenblöcken, von denen der nullte Block auf einen weiteren Datenblock mit einem Array zeigt, ist ursprünglich beim Formatieren durch „mkfs“ erstellt worden.[39] Beim Löschen eines Datenblocks wird dieser in das Array im Superblock an der Stelle „s_nfree“ eingetragen und „s_nfree“ inkrementiert. Ist in diesem aber kein Platz mehr, dann wird das Array aus dem Superblock in den zu löschenden Block kopiert und dessen Blocknummer als das einzige Element im Array an nullter Stelle vermerkt. Für ein Beispiel einer freien Liste mit jeweils zehn Einträgen pro Block siehe Abb. 3.26.

[39] Dabei achtet „mkfs“ darauf, daß benachbarte Blöcke auf der Platte auch in der Liste aufeinanderfolgen. Durch wiederkehrendes Löschen und Erzeugen fragmentiert dieses Dateisystem trotzdem sehr schnell (siehe Abb. 3.26).

Liste freier Inodes

Freie Inodes werden in S5FS einfach dadurch gekennzeichnet, daß das Typfeld „i_mode“ auf Null gesetzt ist. Wie erwähnt, ist es nicht notwendig, zusätzliche Strukturen zum Auffinden von freien Inodes zu benutzen. Das Dateisystem könnte einfach bei jeder Anforderung die Inode-Tabelle durchgehen und nach einer freien Inode suchen. Doch da diese lineare Suche gerade in dem Moment einer Schreibanforderung nicht besonders effizient ist, werden bis zu 100 Nummern freier Inodes in dem Array „s_inode“ im Superblock zwischengespeichert. „s_ninode“ vermerkt dabei die Anzahl freier Inodes in diesem Array und bestimmt die Position aus dem Array, deren Inode-Nummer zur Allokation benutzt wird. An nullter Stelle innerhalb des Arrays steht immer die Inode-Nummer, bis zu der das Dateisystem sichergestellt hat, daß in der Inode-Tabelle keine Inode mit kleinerer Nummer frei ist. Wenn in dem Array nur noch diese Inode-Nummer eingetragen ist, sucht das Dateisystem ab ihrer Position in der Inode-Tabelle nach bis zu 100 weiteren freien Inodes und fügt diese dem Array im Superblock hinzu. Dabei achtet es natürlich darauf, die größte Inode-Nummer an die nullte Stelle im Array zu schreiben. Danach benutzt es die Inode mit der Nummer, die ursprünglich an der nullten Stelle stand, zur eigentlichen Beantwortung der Anfrage. Wenn eine Inode frei wird und ihre Nummer größer als die Inode-Nummer an nullter Stelle im Array ist, versucht das Dateisystem diese im Superblockarray einzutragen, wenn darin noch Platz ist. Ist die freiwerdende Nummer kleiner, dann wird sie auf alle Fälle an die nullte Stelle des Arrays gesetzt und statt dessen versucht, die ursprüngliche nullte Stelle im Array einzusetzen. Wenn kein Platz vorhanden ist, kann das Dateisystem die Inode-Nummer einfach ignorieren, da sie mit Sicherheit bei einem der nächsten linearen Durchläufe der Inode-Tabelle gefunden wird. Durch diese Strategie des Zwischenspeicherns wird die Inode-Tabelle ca. 100 mal weniger durchsucht.

Für weitere Details bezüglich der Verwaltung und des Umgangs der Listen sei auf [Bac86] und [Haw01] verwiesen. Auf die Probleme, die mit der Benutzung dieser beiden Listenstrukturen verbunden sind, gehen wir im nächsten Abschnitt ein.

3.5.2 Das FFS-Dateisystem

Das „Fast File System“ Dateisystem von BSD behebt eine ganze Reihe von Schwächen des klassischen UNIX-Dateisystems und stellt einige neue Features zur Verfügung, die im originalen S5SF-Dateisystem nicht vorhanden sind. Es wurde erstmals in [MJLF84] vorgestellt.
Der Hauptgrund für die Entwicklung von FFS war die Beobachtung, daß die Datentransferleistung eines typischen UNIX-Systems deutlich unter dem berechneten Maximaldurchsatz seiner Festplatten zurückblieb. Das ursprüngliche Dateisystem arbeitete mit einer Blockgröße von 512 Byte. Schon durch eine Vergrößerung der Blockgröße auf 1024 Byte konnte die Transfergeschwindigkeit mehr als verdoppelt werden. Zum einen konnten dadurch größere Blöcke von der Festplatte in den Speicher bewegt werden, wodurch der Verwaltungsaufwand des Kernels geringer wurde. Zum anderen waren durch die größeren Blöcke bei gleichbleibender mittlerer Dateigröße im Schnitt weniger Zugriffe auf indirekte Blöcke notwendig.

Ein weiterer Mangel, der mit den wachsenden Festplattengrößen offensichtlich wurde, ist die Begrenzung von Inode-Nummern auf nur 16 Bit. Da die Inode-Nummern innerhalb eines Dateisystems eindeutig sein müssen, schränkt dies die Anzahl der Dateien in einem S5FS-Dateisystem auf 65536 ein. Je nach Größe der Festplatte und der mittleren Dateigröße wird diese Grenze leicht überschritten. Eine Erweiterung der Inode-Nummer auf 32 Bit erforderte eine Überarbeitung des Verzeichniskonzepts und der anderen Datenstrukturen eines Dateisystems. Bei dieser Gelegenheit erweiterte man zugleich auch viele Datenfelder in der Inode selbst. Die neue Inode-Struktur des FFS ist durch die leicht modifizierte Struktur„ufs_inode“ aus „include/linux/ufs_fs.h“ in Abb. 3.22 dargestellt.[40] Die Anzahl der direkten Blöcke ist auf

```
struct ufs_inode {
  __u16 ui_mode;    /* 0x0*/
  __u16 ui_nlink;   /* 0x2*/
  __u16 ui_suid;    /* 0x4*/
  __u16 ui_sgid;    /* 0x6*/
  __u64 ui_size;    /* 0x8*/
  struct ufs_timeval ui_atime; /*0x10 access*/
  struct ufs_timeval ui_mtime; /*0x18 modification*/
  struct ufs_timeval ui_ctime; /*0x20 creation*/
  union {
    struct {
      __u32 ui_db[12];   /*0x28 data blocks*/
      __u32 ui_ib[3];    /*0x58 indirect blocks*/
    } ui_addr;
    __u8 ui_symlink[60]; /*0x28 fast symlink*/
  } ui_u2;
  __u32 ui_flags;        /*0x64 immutable, append-only...*/
  __u32 ui_blocks;       /*0x68 blocks in use*/
  __u32 ui_gen;          /*0x6c like ext2 i_version, for NFS support*/
  __u32 ui_uid;          /*0x70 File owner*/
  __u32 ui_gid;          /*0x74 File group*/
  __s32 ui_spare[2];     /*0x78 reserved*/
};
```

Abbildung 3.22: Die modifizierte Struktur „ufs_inode“ aus „include/linux/ufs_fs.h“

zwölf erhöht, und Blocknummern werden durchgängig mit 32 Bit angegeben. Durch die Änderungen am Dateisystem ist eine Inode beim FFS nun 128 Byte groß. Weiterhin erlaubt die Verzeichnisstruktur beim FFS die Zuordnung einer Inode zu einem Dateinamen von bis zu 255 Zeichen. Ein anderes Problem, das in [MJLF84] und in [LMKQ89] diagnostiziert wird, ist eine kontinuierlich sinkende Transfergeschwindigkeit bei alternden Dateisystemen. Der Festplattendurchsatz eines typischen UNIX-Systems sinkt nach wenigen Wochen der Benutzung auf ca. ein Fünftel der ursprünglichen Geschwindigkeit. Der Grund dafür liegt in der Organisation der freien Blöcke beim S5FS-Dateisystem. Wie oben erläutert, speichert das alte Dateisystem freie Blöcke im Prinzip in einer LIFO-Liste ab: Freiwerdende Blöcke werden vorne in die Liste eingefügt, benötigte Blöcke werden ebenfalls von vorne her der Liste

[40] Die Bezeichnung „ufs“ für Variablenpräfixe steht „unix file system“.

entnommen (engl „last in, first out“, LIFO). Wenn in einem System häufig Dateien erzeugt und gelöscht werden, ist die Liste freier Blöcke in kurzer Zeit völlig durchmischt. Damit werden die Datenblöcke neu angelegter Dateien über die ganze Platte verstreut. Diesen Vorgang bezeichnet man als „Fragmentierung“, genauer „externe Fragmentierung“, um ihn von der „internen Fragmentierung“ zu unterscheiden.[41] Um Dateien zusammenhängend –und damit kaum fragmentiert– anzulegen, müßte diese Liste immer sortiert gehalten werden. Dies ist mit dieser Darstellungsform der Liste nicht effizient implementierbar (siehe [MJLF84]). Eine ähnliche Situation tritt bei der Verwaltung der Inodes in nur einem Bereich der Festplatte in Form der Inode-Tabelle auf. Dies erfordert beim Zugriff auf die Datei unverhältnismäßig weite Kopfbewegungen über die Oberfläche des Mediums. Es wäre effizienter, wenn die Inodes in der Nähe der Daten, die sie referenzieren, angelegt werden könnten.
Zusammengefaßt ergeben sich die folgenden Verbesserungen von FFS gegenüber S5FS, auf die wir im folgenden teilweise detaillierter eingehen werden:

- Maximal 2^{32} statt 2^{16} Inodes pro Dateisystem
- Maximal 255 statt 14 Zeichen pro Dateiname
- Verbesserte, zusammenhängende Anordnung von Blöcken wegen der Verwendung einer sogenannten „Bitmap“ statt der Verwaltung freier Blöcke in einer Liste
- Speicherung der Inodes näher an den Daten statt in nur einem Bereich am Anfang der Platte
- Geschwindigkeitssteigerung durch Erhöhung der Blockgröße; trotzdem geringe Platzverschwendung durch ungenutzte Blockenden wegen der Verwendung von Blockfragmenten, sogenannten „Fragments“
- Auswertung von Informationen über die Plattengeometrie
- Verweise über Dateisystemgrenzen hinweg (symbolische Links)
- Bessere administrative Kontrolle über den Platzverbrauch einzelner Benutzer durch die Einführung eines Quotasystems

Diese Verbesserungen haben ihren Preis in der Komplexität des Codes, wie man alleine beim Betrachten der Superblockstruktur Abb. 3.23 feststellen kann. Diese Struktur „ufs_super_block“ aus „include/linux/ufs_fs.h“ ist dabei mit sehr kleiner Schriftgröße dargestellt, nur um einen Eindruck von der Anzahl der Variablen und Parameter des Superblocks zu vermitteln. Im Unterschied zu S5FS speichert FFS im Superblock nur statische Informationen der Partition, die bei der Formatierung erfaßt werden oder wählbar sind. Eine wesentliche Angabe darunter ist die Blockgröße „fs_bsize“ in Bytes. Damit ist es mit FFS möglich, verschiedene FFS-Dateisysteme mit unterschiedlichen Blockgrößen im System zu verwalten. Andere, für uns neue

[41] Diese für die vorliegende Arbeit zentralen Begriffe werden im nächsten Kapitel definiert und ausgiebig erläutert.

```
struct ufs_super_block {
  __u32 fs_link;  /*UNUSED*/
  __u32 fs_rlink; /*UNUSED*/
  __u32 fs_sblkno;    /*addr of super-block in filesys*/
  __u32 fs_cblkno;    /*offset of cyl-block in filesys*/
  __u32 fs_iblkno;    /*offset of inode-blocks in filesys*/
  __u32 fs_dblkno;    /*offset of first data after cg*/
  __u32 fs_cgoffset;  /*cylinder group offset in cylinder*/
  __u32 fs_cgmask;    /*used to calc mod fs_ntrak*/
  __u32 fs_time;      /*last time written -- time_t*/
  __u32 fs_size;      /*number of blocks in fs*/
  __u32 fs_dsize;     /*number of data blocks in fs*/
  __u32 fs_ncg;       /*number of cylinder groups*/
  __u32 fs_bsize;     /*size of basic blocks in fs*/
  __u32 fs_fsize;     /*size of frag blocks in fs*/
  __u32 fs_frag;      /*number of frags in a block in fs*/
  /*these are configuration parameters*/
  __u32 fs_minfree;   /*minimum percentage of free blocks*/
  __u32 fs_rotdelay;  /*num of ms for optimal next block*/
  __u32 fs_rps;       /*disk revolutions per second*/
  /*these fields can be computed from the others*/
  __u32 fs_bmask;     /*``blkoff'' calc of blk offsets*/
  __u32 fs_fmask;     /*``fragoff'' calc of frag offsets*/
  __u32 fs_bshift;    /*``lblkno'' calc of logical blkno*/
  __u32 fs_fshift;    /*``numfrags'' calc number of frags*/
  /*these are configuration parameters*/
  __u32 fs_maxcontig; /*max number of contiguous blks*/
  __u32 fs_maxbpg;    /*max number of blks per cyl group*/
  /*these fields can be computed from the others*/
  __u32 fs_fragshift; /*block to frag shift*/
  __u32 fs_fsbtodb;   /*fsbtodb and dbtofsb shift constant*/
  __u32 fs_sbsize;    /*actual size of super block*/
  __u32 fs_csmask;    /*csum block offset*/
  __u32 fs_csshift;   /*csum block number*/
  __u32 fs_nindir;    /*value of NINDIR*/
  __u32 fs_inopb;     /*value of INOPB*/
  __u32 fs_nspf;      /*value of NSPF*/
  /*yet another configuration parameter*/
  __u32 fs_optim;     /*optimization preference, see below*/
  /*these fields are derived from the hardware*/
  __u32 fs_npsect;    /*# sectors/track including spares*/
  __u32 fs_interleave;/*hardware sector interleave*/
  __u32 fs_trackskew; /*sector 0 skew, per track*/
  __u32 fs_id[2];     /*file system id - unused*/
  /*sizes determined by number of cylinder groups and their sizes*/
  __u32 fs_csaddr;    /*blk addr of cyl grp summary area*/
  __u32 fs_cssize;    /*size of cyl grp summary area*/
  __u32 fs_cgsize;    /*cylinder group size*/
  /*these fields are derived from the hardware*/
  __u32 fs_ntrak;     /*tracks per cylinder*/
  __u32 fs_nsect;     /*sectors per track*/
  __u32 fs_spc;       /*sectors per cylinder*/
  /*this comes from the disk driver partitioning*/
  __u32 fs_ncyl;      /*cylinders in file system*/
  /*these fields can be computed from the others*/
  __u32 fs_cpg;       /*cylinders per group*/
  __u32 fs_ipg;       /*inodes per group*/
  __u32 fs_fpg;       /*blocks per group * fs_frag*/
  /*this data must be re-computed after crashes*/
  struct ufs_csum fs_cstotal; /*cylinder summary information*/
  /*these fields are cleared at mount time*/
  __s8 fs_fmod;       /*super block modified flag*/
  __s8 fs_clean;      /*file system is clean flag*/
  __s8 fs_ronly;      /*mounted read-only flag*/
  __s8 fs_flags;      /*currently unused flag*/
  __s8 fs_fsmnt[UFS_MAXMNTLEN]; /*name mounted on*/
  /*these fields retain the current block allocation info*/
  __u32 fs_cgrotor;             /*last cg searched*/
  __u32 fs_csp[UFS_MAXCSBUFS];  /*list of fs_cs info buffers*/
  __u32 fs_maxcluster;
  __u32 fs_cpc;                 /*cyl per cycle in postbl*/
  __u16 fs_opostbl[16][8];      /*old rotation block list head*/
  __s32 fs_sparecon[50];        /*reserved for future constants*/
  __s32 fs_contigsumsize;       /*size of cluster summary array*/
  __s32 fs_maxsymlinklen;       /*max length of an internal symlink*/
  __s32 fs_inodefmt;            /*format of on-disk inodes*/
  __u32 fs_maxfilesize[2];      /*max representable file size*/
  __u32 fs_qbmask[2];           /*~usb_bmask*/
  __u32 fs_qfmask[2];           /*~usb_fmask*/
  __s32 fs_state;               /*file system state time stamp*/
  __s32 fs_postblformat;        /*format of positional layout tables*/
  __s32 fs_nrpos;               /*number of rotational positions*/
  __s32 fs_postbloff;           /*(__s16) rotation block list head*/
  __s32 fs_rotbloff;            /*(__u8) blocks for each rotation*/
  __s32 fs_magic;               /*magic number*/
  __u8 fs_space[1];             /*list of blocks for each rotation*/
};
```

Abbildung 3.23: Die Struktur „ufs_super_block“ aus „include/linux/ufs_fs.h“

Felder im Superblock betreffen die Hardware-Angaben zur Beschreibung der Plattengeometrie, ihre Umdrehungsgeschwindigkeit, Interleave und Spuroffset[42], sowie einstellbare Parameter wie beispielsweise „fs_minfree“ und „fs_maxcontig“. Dabei gibt „fs_minfree“ die prozentuale Anzahl freier Blöcke an, die das Dateisystem freihalten soll, so daß sie dem Administrator im Notfall zur Verfügung stehen. Auf „fs_maxcontig“ gehen wir weiter unten ein.

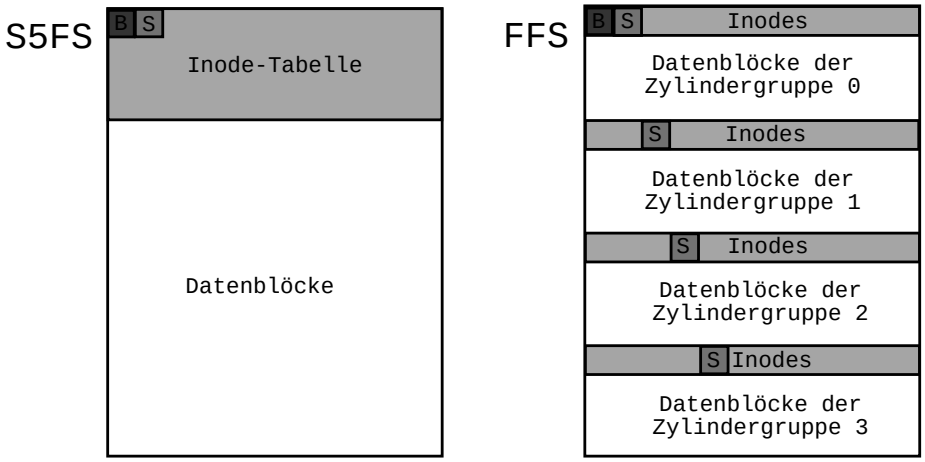

Abbildung 3.24: Blocklayout von S5FSund FFS im Vergleich

Blockstruktur

S5FS unterteilt die Festplatte zur Verwaltung der Daten in zwei Bereiche, einen für die Inode-Tabelle und einen für die Datenblöcke. FFS statt dessen teilt die Platte in Gruppen von benachbarten Zylindern auf. Jede dieser Zylindergruppen enthält eine Kopie des Superblocks, eine Beschreibung der Zylindergruppe mit dynamischen Informationen über die Belegung, einen Anteil an den Inodes des Dateisystems und Datenblöcke (siehe Abb. 3.24). Jede Zylindergruppe bildet praktisch ein kleines Dateisystem für sich alleine, das jedoch mit den anderen Zylindergruppen des Dateisystems gemeinsame Block- und Inode-Adressen teilt. Ihre Struktur „ufs_cylinder_group“ aus „include/linux/ufs_fs.h“ ist in Abb. 3.25 wiedergegeben. Die für uns relevanten Felder sind die Anzahl der freien Inode-Blöcke „cg_ncyl“ und

[42] Bei der Entwicklung von FFS ging man davon aus, daß eine Festplatte eine quaderförmige Geometrie hat. FFS benutzt die angegebenen Informationen, um die Anordnung von Datenblöcken auf der Platte zu optimieren. Während die Köpfe der Festplatte die Spur wechseln, dreht sich das Medium unter den Köpfen weiter. FFS schätzt anhand der Informationen über Spurwechselzeit und Rotationsgeschwindigkeit, wie weit sich das Medium während des Spurwechsels gedreht haben wird, und versucht die folgenden Blöcke der Datei dann so anzuordnen, daß sie an dieser vorausberechneten Stelle stehen. Zur Zeit der Entwicklung von FFS, 1984, war dies eine durchaus sinnvolle Optimierung. Aber mit einer nicht quaderförmigen Geometrie, LVM oder mit einem RAID-Array ist dies im günstigsten Fall wirkungslos. Das Dateisystem oder der Gerätetreiber haben gar keine Möglichkeit, die wahre Geometrie der angeschlossenen Medien zu erfragen. Aber selbst bei bekannter Geometrie ist es nicht garantiert, daß die Vorstellung des Gerätetreibers vom Mapping der Festplatte mit dem tatsächlich von der Platte verwendeten Mapping übereinstimmt, wie wir im letzten Kapitel beschrieben haben.

```
struct ufs_cylinder_group {
  __u32 cg_link;     /*linked list of cyl groups*/
  __u32 cg_magic;    /*magic number*/
  __u32 cg_time;     /*time last written*/
  __u32 cg_cgx;      /*we are the cgx'th cylinder group*/
  __u16 cg_ncyl;     /*number of cyl's this cg*/
  __u16 cg_niblk;    /*number of inode blocks this cg*/
  __u32 cg_ndblk;    /*number of data blocks this cg*/
  struct ufs_csum cg_cs;       /*cylinder summary information*/
  __u32 cg_rotor;              /*position of last used block*/
  __u32 cg_frotor;             /*position of last used frag*/
  __u32 cg_irotor;             /*position of last used inode*/
  __u32 cg_frsum[UFS_MAXFRAG]; /*counts of available frags*/
  __u32 cg_btotoff;            /*(__u32) block totals per cylinder*/
  __u32 cg_boff;               /*(short) free block positions*/
  __u32 cg_iusedoff;           /*(char) used inode map*/
  __u32 cg_freeoff;            /*(u_char) free block map*/
  __u32 cg_nextfreeoff;        /*(u_char) next available space*/
  __u32 cg_clustersumoff;      /*(u_int32) counts of avail clusters*/
  __u32 cg_clusteroff;         /*(u_int8) free cluster map*/
  __u32 cg_nclusterblks;       /*number of clusters this cg*/
  __u32 cg_sparecon[13];       /*reserved for future use*/
  __u8 cg_space[1];           /*space for cylinder group maps*/
};
```

Abbildung 3.25: Die Struktur „ufs_cylinder_group“ aus „include/linux/ufs_fs.h“

der freien Datenblöcke „cg_ndblk“ sowie die wichtigsten Neuerungen „cg_freeoff“ und „cg_iusedoff“. Letztere stellen jeweils eine Bitmap dar, die erste markiert die freien Blöcke und die zweite die benutzten Inodes innerhalb der Gruppe und ersetzen die entsprechenden Listen aus S5FS. Die Verteilung der Superblöcke über die Platte geschieht nur aus Sicherheitsgründen; benutzt wird nur der erste, sogenannte „primäre“, Superblock. Die Kopien können bei Beschädigung des primären Superblocks zur Reparatur herangezogen werden. Da der Superblock nur statische Daten enthält, entsteht kein zusätzlicher Aufwand zur synchronen Aufrechterhaltung dieser Redundanz.

Cluster und Lokalität

Durch das Konzept der Zylindergruppen werden die Inodes über die gesamte Platte verteilt. Beim Anlegen einer neuen Datei oder beim Vergrößern einer bereits bestehenden Datei versucht FFS, freie Datenblöcke in derselben Zylindergruppe zu finden, zu der die Inode für die Datei gehört. Auf diese Weise stehen die Verwaltungsinformationen einer Datei in der Nähe der Daten der Datei, und die Kopfbewegungen zwischen einer Inode und den zugeordneten Datenblöcken, die zum Zugriff auf die Daten notwendig sind, werden minimiert.

Durch diese Strategie entsteht jedoch ein Problem bei großen Dateien. Wenn in einer Zylindergruppe eine sehr große Datei angelegt wird, belegt sie alle oder zumindest überproportional viele der dort vorhandenen Datenblöcke. In dieser Zylindergrup-

pe sind jedoch wahrscheinlich noch viele Inodes frei.[43] Wenn diese jetzt ebenfalls belegt werden, müssen die Datenblöcke zu diesen Inodes in anderen Zylindergruppen gesucht werden, weil alle lokal vorhandenen Blöcke belegt sind. Dadurch wird wiederum die Balance von Inodes und Datenblöcken in diesen Zylindergruppen gestört. Um diesen Effekt zu vermeiden, wird beim Schreiben von langen Dateien nach „fs_maxcontig“ Bytes eine Lücke erzwungen, bei dem die Zylindergruppe gewechselt wird. „fs_maxcontig“ ist eine spezifisch einstellbare Variable des Superblocks, für die erfahrungsgemäß mit einem Wert von ein Achtel der Größe einer Zylindergruppe der gewünschte Effekt erreicht wird.[44] Der Wechsel soll bewirken, daß sich die Daten einer sehr großen Datei über mehrere Zylindergruppen verteilen und das Verhältnis zwischen freien Inodes und freien Datenblöcken in allen Zylindergruppen etwa gleich bleibt.
Auch auf Verzeichnisebene wird versucht, verwandte –sich referenzierende– Daten lokal zu gruppieren. Dateien eines Verzeichnisses werden normalerweise in derselben Zylindergruppe angelegt. Auf der anderen Seite darf die Gruppierung von Daten zu Clustern aber nicht übertrieben werden. Sonst fänden sich im Extremfall alle Dateien in einem Cluster wieder, und man hätte gegenüber S5FS nichts gewonnen. Um zu verhindern, daß die Lokalisierungsmaßnahmen zu große Cluster erzeugen, werden neue Verzeichnisse in Zylindergruppen angelegt, die überdurchschnittlich viele freie Inodes und möglichst wenige Verzeichnisse enthalten [MJLF84].

Blockfragmente

Der Übergang auf eine Blockgröße von 4096 oder sogar 8192 Byte erhöht die Transfergeschwindigkeit sogar bei verstreut angeordneten Dateien deutlich. Jedoch steigt natürlich der Platzverlust durch verschwendete, im Prinzip fast leere, letzte Datenblöcke stark an. Bei kleiner bis mittlerer Dateigröße unterhalb der Blockgröße erhöht dies die interne Fragmentierung des Dateisystems ungemein.
Betroffen von der Verschwendung von Speicherplatz sind natürlich immer nur die jeweils letzten Blöcke einer Datei. Im FFS ist es möglich, einzelne Datenblöcke je nach Konfiguration des Dateisystems in zwei, vier oder acht Fragmente aufzuteilen. Die Anzahl der Fragmente pro Datenblock wird in „fs_frag“ im Superblock gespeichert. Wenn bei der Speicherung einer Datei der letzte Datenblock nicht komplett gefüllt werden kann, bricht das Dateisystem einen Datenblock in die vorgegebene Anzahl Fragmente auf. Es speichert dabei das Dateiende in so wenig Fragmenten wie möglich. Der Rest des Datenblocks steht dann zur Speicherung von Enden anderer Dateien zur Verfügung. Auf diese Weise gelingt es, den Platzverlust durch interne Fragmentierung bei einem FFS mit 8 KB großen Blöcken und 1024 Byte großen Fragmenten in derselben Größenordnung zu halten wie bei einem alten Dateisystem mit 1024 Byte Blockgröße [MJLF84].
Diese Strategie ist jedoch ausgesprochen ungünstig, wenn eine Datei wächst, die ihren letzten Datenblock mit einem anderen Dateiende teilt. Da in FFS teilweise belegte Blöcke nur als Dateienden vorkommen dürfen, kann FFS nicht einfach

[43] Standardmäßig erzeugt FFS beim Formatieren eine Inode pro 2048 Byte an Datenplatz in einer Zylindergruppe.
[44] Eine gute Wahl stellt auch die Dateilänge dar, die maximal durch direkte Blöcke adressiert werden kann.

einen anderen Block belegen und die Datei dort fortsetzen. Statt dessen muß das Dateiende der wachsenden Datei in einen leeren Block umkopiert werden, der dann erweitert werden darf. Wenn keine besonderen Vorkehrungen getroffen werden, kann man leicht den pathologischen Fall konstruieren, bei dem sich die Dateienden zweier langsam wachsender Dateien wieder und wieder gegenseitig behindern, so daß ständiges Umkopieren notwendig ist. Solange ausreichend freier Platz im Dateisystem vorhanden ist, versucht FFS, derartige Grenzfälle zu verhindern, indem es eine Datei blockweise wachsen läßt, sobald sie das zweite Fragment eines Blocks belegt.

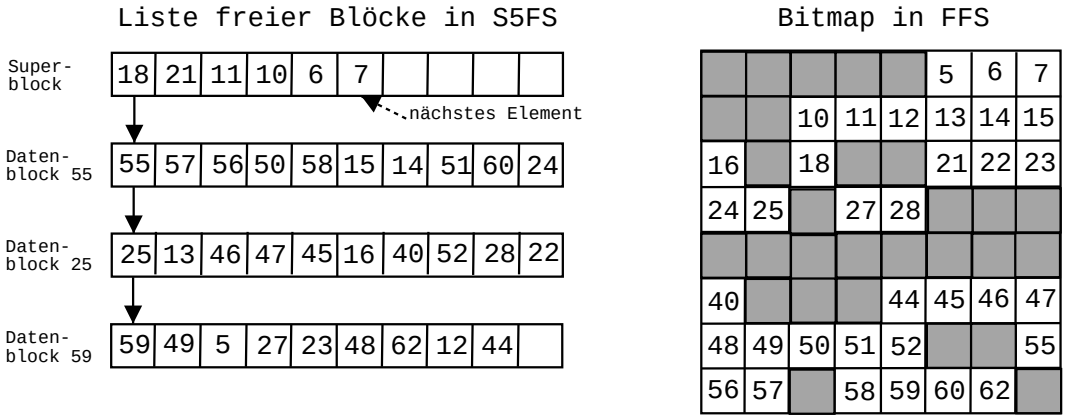

Abbildung 3.26: Die Verwaltung freier Blöcke als Liste bei S5FS und als Bitmap bei FFS im Vergleich

Verwaltung freier Blöcke und Inodes

Die Verwaltung freien Speicherplatzes erfolgt in FFS auf der Ebene von Fragmenten und mithilfe von Bitmaps. Auch die Blockadressen, die in den Inodes gespeichert werden, um die zu einer Datei gehörenden Datenblöcke anzugeben, sind fragmentweise auflösend. Zwei aufeinanderfolgende Datenblöcke unterscheiden sich in ihren Adressen nicht um Eins, sondern um die Anzahl der Fragmente pro Block. Die Bitmap ist ein Array, das für jedes Fragment einer Zylindergruppe ein Bit enthält. Ist dieses Bit Null, ist das Fragment frei. Ist es Eins, ist das Fragment belegt. Komplett freie Blöcke sind in der Bitmap als Nullbytes zu erkennen. Im Gegensatz zu der Liste freier Blöcke bei S5FS sind Bitmaps implizit geordnet. Zusammenhängende freie Bereiche lassen sich im Vergleich zum S5FS sehr viel leichter erkennen. Abb. 3.26 zeigt links eine Liste freier Blöcke von S5FS und rechts die dazu gehörende Bitmap. Zum besseren Vergleich benutzt die dargestellte Bitmap keine Fragmente und arbeitet auf der Ebene von Blöcken. Ein graues Kästchen markiert demnach einen belegten Block, ein weißes einen freien. Es ist deutlich zu erkennen, daß die LIFO-Liste bei S5FS je nach Vorgeschichte des Dateisystems einen anderen Sachverhalt für die gleiche Belegung anzeigt, während die Bitmap zusammengehörende Blöcke unabhängig von ihrer Vorgeschichte immer gleich darstellt. Jede Zylindergruppe speichert für die ihr zugeordneten Datenblöcke und Inodes eine Bitmap für die freien Blöcke „cg_freeoff“, sowie eine Bitmap für die belegten Inodes „cg_iusedoff“ (siehe Abb. 3.25).

Quotas

Die Inode- und Blockverwaltung von FFS wurde so modifiziert, daß sie den Verbrauch von Inodes und Blöcken durch Benutzer und Benutzergruppen registriert. Dieses sogenannte „Quotasystem“ erlaubt dem Systemverwalter einen schnellen Überblick darüber, wie viele Inodes oder Blöcke einem einzelnen Benutzer oder seiner Benutzergruppe zugeordnet sind. Dabei ist es möglich, harte und weiche Limits zu vergeben. Benutzer dürfen ein weiches Limit für eine festzulegende Zeit überschreiten. Nach Ablauf dieser Zeit wird das weiche Limit für den Benutzer so lange wie ein hartes Limit behandelt, bis der betreffende Benutzer in seinem Verbrauch einmal die durch das vormals weiche Limit gesetzte Grenze unterschritten hat. Ein hartes Limit darf in keinen Fall überschritten werden. Wenn das Limit doch überschritten wird, meldet das Dateisystem beim Anlegen von Dateien oder beim Schreiben in eine Datei die neue spezielle Fehlermeldung „EDQUOT“ für „quota exceeded“, die von Anwendungsprogrammen wie „ENOSPC“ für „no space left on device“ behandelt werden sollte.

Verzeichnisse und Pfadnamen

Das FFS läßt Dateinamen von bis zu 255 Zeichen Länge zu. Würde man Namen dieser Länge statisch mit einer ähnlichen Verzeichnisstruktur verwalten wie bei S5FS, wäre die Zuordnung wenig effizient, weil im Schnitt wenige Namen wirklich 255 Zeichen benötigen. FFS verwendet daher eine Struktur, in der Namenseinträge in Verzeichnissen eine variable Länge haben können. Die Struktur „ufs_dir_entry“

```
struct ufs_dir_entry {
  __u32  d_ino;    /*inode number of this entry*/
  __u16  d_reclen; /*length of this entry*/
  __u16  d_namlen; /*actual length of d_name*/
  __u8  *d_name;   /*file name*/
};
```

aus „include/linux/ufs_fs.h“ zeigt das Vorgehen. In den Datenblöcken, die einem Verzeichnis zugeordnet sind, wird eine einfach verkettete Liste bestehend aus der Struktur „ufs_dir_entry“ gebildet. Das Feld „d_reclen“ sorgt für die Verkettung, indem es angibt, wie lang der Verzeichniseintrag insgesamt ist. Das Feld „d_namlen“ beinhaltet die Information über die Länge des im Verzeichniseintrag „d_name“ gespeicherten Namens. Wird ein Verzeichniseintrag gelöscht, wird der von ihm belegte Platz dem „d_reclen“ des vorhergehenden Eintrages dazugerechnet. Wegen „.“ und „..“ gibt es im ersten Verzeichnisdatenblock immer einen Vorgänger. Weitere Verzeichnisblöcke, die vollständig leer geworden sind, werden freigegeben. Dazu wird wie üblich in ihrer Inode das entsprechende Adreßfeld auf Null gesetzt. Verzeichniseinträge werden derart gespeichert, daß sie immer in einen Block passen. Sie überschneiden daher niemals Blockgrenzen.
Zugleich wurde in FFS erstmals der Zugriff auf Verzeichnisse vom konkreten Speicherungsformat abstrahiert. Während in S5FS die übliche Methode zum Lesen eines Verzeichnisses darin bestand, die Verzeichnisdatei zum Lesen zu öffnen und die Verzeichniseinträge selbst zu verarbeiten, wird in FFS und allen danach entwickelten Dateisystemen der Zugriff auf die Verzeichnisse durch einen Satz von Systemaufru-

fen realisiert, der den Systemaufrufen von regulären Dateien ähnelt. Diese Aufrufe sind „opendir“ zum Öffnen, „readdir“ zum Lesen, „scandir“ zum Durchlaufen, „rewinddir“ zum Zurückspulen, „telldir“ zur Angabe der Position innerhalb, „seekdir“ zum Setzen der Position innerhalb und „closedir“ zum Schließen eines Verzeichnisses.

Symbolische Verweise

Wie erwähnt unterstützt FFS im Unterschied zu S5FS symbolische Verweise. Es handelt sich dabei um einen neuen Typ, der im Typfeld „ui_mode“ in der Inode vermerkt wird. Die Dateiblöcke eines symbolischen Links enthalten den Pfadnamen der Datei, auf den der symbolische Link zeigt. Wenn das Dateisystem bei der Auflösung eines Pfadnamens auf einen symbolischen Link trifft, wird sein Inhalt dem noch zu interpretierenden Restpfadnamen vorangestellt und der so erzeugte Name als Pfadname interpretiert. Symbolische Links können damit im Gegensatz zu gewöhnlichen Verweisen auch Dateisystemgrenzen überwinden. Außerdem sind bei einem symbolischen Verweis der originale Dateiname und der Verweis immer klar zu unterscheiden. Heutige FFS-Versionen benutzen eine Technik, die sich „fast symlink“ nennt. Dabei wird der Verweis des symbolischen Links direkt in dem Array „ui_symlink“ der Inode gespeichert, wenn er kürzer als eine bestimmte Länge ist. Diese Länge beträgt 60 Zeichen für die gezeigte Inode aus Abb. 3.22.

Für weitere Details über FFS und seine Implementierung sei auf [MJLF84] und [LMKQ89] und die Referenzen darin verwiesen. Für Untersuchungen über die Auswirkung verschiedener Blockallokationsstrategien auf die kurzfristige und langfristige Performanz von FFS siehe [SS94] und [SS96]. In [GP94] wird auf das Problem eingegangen, daß die Konsistenzerhaltung von Dateisystemmetadaten gerade bei Änderungen an kleinen Dateien einen erheblichen Aufwand für das Dateisystem darstellt. Klassischerweise werden Metadaten bei strukturellen Änderungen an UNIX-Dateisystemen entweder synchron mit Festplattenzugriffszeiten statt mit Hauptspeicherzugriffszeiten im Cache durchgeführt, oder in Form von geordneten Schreibaufträgen dem Festplattentreiber übergeben. Auf beide Weisen wird die Integrität des Dateisystems sichergestellt. Letzteres Vorgehen ist um 30% schneller, kann aber nicht verzögert schreiben, wenn voneinander abhängende Schreibaufträge geordnet werden müssen. Mittels der vorgeschlagenen „Soft Updates“ Technik wird ohne Ändern der gespeicherten Strukturen auf der Festplatte sichergestellt, daß auch verzögertes Schreiben der Metadaten (engl. „delayed meta data write“) immer derart erfolgen kann, daß keine Inkonsistenzen am Dateisystem zu befürchten sind. Dazu werden wartende Abhängigkeiten von zu schreibenden Blöcken beim Schreiben temporär im Hauptspeicher rückgängig gemacht. Bevor im Hauptspeicher auf diese Blöcke wieder zugegriffen werden kann, werden noch nicht gespeicherte Abhängigkeiten wieder hergestellt. Somit ist der Zustand der Metadaten und Daten auf der Festplatte und Hauptspeicher (eventuell jeweils für sich) immer konsistent und sowohl Metadaten als auch Daten können verzögert geschrieben werden [GMSP00].

3.5.3 Das Ext2-Dateisystem

Wie wir in Abschnitt 3.1.2 erwähnt haben, fand die Entwicklung von LINUX anfangs unter MINIX statt. So verwundert es natürlich nicht, daß das erste LINUX-Dateisystem das MINIX Dateisystem war [Tan87].[45] Dem Benutzer gegenüber präsentiert sich MINIX genauso wie S5FS. Beispielsweise erlaubt es Dateinamen von nur 14 Zeichen. Auch die internen Datenstrukturen wie die Inode und der Superblock sind sehr ähnlich. Eine wesentliche Verbesserung besteht immerhin darin, das MINIX statt der Listenstrukturen Bitmaps zur Kennzeichnung freier Blöcke und Inodes verwendet. Die Blockstruktur von MINIX ist in Abb. 3.27 gezeigt. Ein Block

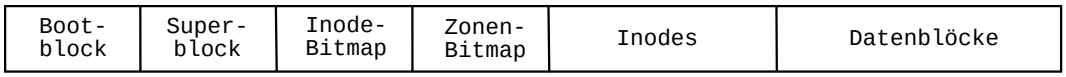

Abbildung 3.27: Die Blockstruktur von MINIX

hat in der Regel eine Größe von 1024 Byte. Standardmäßig verwendet MINIX nur 16 Bit für die Blockadressen, so daß es damit maximal eine Partition von 64 MB verwalten kann. Um diese auch für damalige Verhältnisse kleine Größe zu erhöhen, werden mehrere Diskblöcke zu einer Zone zusammengefaßt, und diese anstelle der Blöcke durch die Inodes adressiert.
Die Entwicklergemeinde von LINUX war von den Fähigkeiten von MINIX nicht gerade begeistert und präsentierte Anfang 1992 ein neues Dateisystem mit dem Namen „Erweitertes Dateisystem“ („Extended File System“). Dieses konnte mit Dateinamen von 255 Zeichen umgehen und unterstützte Partitionen bis zu einer Größe von 2 GB. Allerdings war es langsamer als MINIX und fragmentierte wegen seiner einfachen Listenverwaltung freier Blöcke und freier Inodes außerordentlich stark. 1993 stellte dann Frank Xia eine Erweiterung des MINIX Dateisystems unter dem Namen „XIA“ Dateisystem vor, das Partitionen von 2 GB und Dateinamen bis zu 248 Zeichen unterstützte. Ungefähr zeitgleich stellten Rémy Card, Theodore Tsó und Stephen Tweedie das „Zweite Erweiterte Dateisystem“ („Second Extended File System“, Ext2) vor [CTT94]. Dieses Dateisystem ist unter LINUX bis heute das Standarddateisystem. Es ist strukturell dem FFS sehr ähnlich, basiert aber nicht auf BSD-Quelltexten. Gegenüber dem BSD-Code ist der Ext2-Quelltext weniger umfangreich und daher nicht so komplex. Trotzdem verallgemeinert Ext2 einige Konzepte von FFS und ist schneller als das BSD-Dateisystem.

Das Ext2 unterscheidet sich vom FFS in den folgenden Punkten:

- Ext2 verfügt nicht über den FFS-Code zur kleinräumigen Optimierung der Blockanordnung und benutzt generell eine einfachere Allokationsstrategie.
- Ext2 arbeitet mit Blöcken in der Größenordnung von FFS-Fragmenten und verwendet andere Hilfsmittel als FFS zur Erzeugung von zusammenhängend angeordneten Datenbereichen.

[45] Siehe auch die Informationen und weiteren Verweise von Tanenbaum online unter http://www.cs.vu.nl/~ast/minix.html.

- Ext2 kennt keine Fragmente. Der umständliche Code zur Fragmentverwaltung und das Umkopieren von Fragmenten bei langsam wachsenden Dateien entfällt damit.

- Ext2 kennt besondere Dateiattribute wie unveränderbare Dateien, nicht kürzbare Dateien, und nicht löschbare Dateien.[46]

- Ext2 notiert seinen Status im Superblock und überprüft in einstellbaren Intervallen das Dateisystem auf Konsistenzfehler.

- Die Entwicklung von Ext2 ist nicht abgeschlossen. Zu der Liste geplanter Erweiterungen gehören die Restauration gelöschter Daten, Zugriffskontrollisten (engl. „access control lists“, ACL), automatische Dateikomprimierung und Fragmentunterstützung.[47]

Die Inode-Datenstruktur von Ext2 ist der FFS-Inode sehr ähnlich. Ihre Struktur „ext2_inode“ aus „include/linux/ext2_fs.h“ ist in Abb. 3.28 dargestellt. Sie ist ebenso wie die FFS-Inode 128 Byte groß. Unterschiede bestehen in der Größe einiger Datenfelder. So ist in der Ext2-Inode das Feld „i_size“ nur 32 Bit breit, und „i_uid“ und „i_gid“ belegen nur 16 Bit. Statt dessen werden die überzähligen Bytes der Inode für die ACLs verwendet. Es ist aber noch kein offizieller Code dazu vorhanden.[48] Einen weiteren Unterschied zum FFS stellt das zusätzliche Feld „i_dtime“ dar. Ext2 überschreibt eine Inode nicht, wenn sie gelöscht wird. Statt dessen wird die Inode einer gelöschten Datei durch das Belegen des Feldes „i_dtime“ mit dem Löschzeitpunkt markiert. Der Referenzzähler „i_links_count“ steht natürlich auf Null. Mit geeigneten Werkzeugen ist es dem Systemverwalter dadurch möglich, nach gelöschten Dateien zu suchen und die Datei wiederherzustellen, sofern die durch das Löschen freigegebenen Datenblöcke noch nicht überschrieben sind. Ein weiteres neues Feld in der Inode-Struktur ist „i_flags“. Dieses Flag wird benutzt, um einer Datei eine Reihe von besonderen Eigenschaften und Attributen zu verleihen. Dazu gehören unter anderem Attribute, die eine Datei als unveränderlich oder als Logbuch kennzeichnen. Eine unveränderliche Datei (engl. „immutable file“) kann weder gelöscht noch umbenannt noch auf eine andere Art verändert werden, sobald das System in den Mehrbenutzerbetrieb gewechselt ist. Eine Logbuchdatei (engl. „append only file“) kann ebenfalls weder gelöscht noch umbenannt werden. Als einzige Schreiboperation ist das Anhängen an das Dateiende erlaubt. Andere Attribute erzwingen für bestimmte Dateien synchrone Schreibzugriffe. Jeder normale Aufruf „open“ auf diese Datei verhält sich so, als sei die Datei zusätzlich mit dem Modus „O_SYNC“ geöffnet worden. Ein weiteres Attribut kennzeichnet eine Datei derart, daß sie aus von einem Backup ausgeschlossen wird. Ext2 implementiert eine spezielle Version von „ioctl“, um die Flags einer Datei zu setzen und zu lesen. Die Kommandos „lsattr“ und „chattr“ nutzen dieses Interface und erlauben Benutzern den Zugriff auf die Attribute.

[46] Diese Attribute gibt es teilweise auch bei FFS in BSD Version 4.4.
[47] Siehe http://e2fsprogs.sourceforge.net/ext2.html für die Ext2-Projektseite.
[48] Siehe aber http://acl.bestbits.at/ für diverse Patches.

```
struct ext2_inode {
  __u16 i_mode;  /*File mode*/
  __u16 i_uid;   /*Low 16 bits of Owner Uid*/
  __u32 i_size;  /*Size in bytes*/
  __u32 i_atime; /*Access time*/
  __u32 i_ctime; /*Creation time*/
  __u32 i_mtime; /*Modification time*/
  __u32 i_dtime; /*Deletion Time*/
  __u16 i_gid;   /*Low 16 bits of Group Id*/
  __u16 i_links_count; /*Links count*/
  __u32 i_blocks;      /*Blocks count*/
  __u32 i_flags;       /*File flags*/
  __u32 l_i_reserved1;
  __u32 i_block[15];   /*Pointers to blocks*/
  __u32 i_generation;  /*File version (for NFS)*/
  __u32 i_file_acl;    /*File ACL*/
  __u32 i_dir_acl;     /*Directory ACL*/
  __u32 i_faddr;       /*Fragment address*/
  __u8  l_i_frag;      /*Fragment number*/
  __u8  l_i_fsize;     /*Fragment size*/
  __u16 i_pad1;
  __u16 l_i_uid_high;  /*these 2 fields   */
  __u16 l_i_gid_high;  /*were reserved2[0]*/
  __u32 l_i_reserved2;
};
```

Abbildung 3.28: Die Struktur „ext2_inode“ aus „include/linux/ext2_fs.h“

Der Superblock von Ext2 ist im Vergleich zu FFS deutlich reduziert. Seine Struktur „ext2_super_block“ aus „include/linux/ext2_fs.h“ ist in Abb. 3.29 wiedergegeben. Er enthält statische Informationen über das Dateisystem wie die gesamte Anzahl der Inodes „s_inodes_count“ und Blöcke „s_blocks_count“, sowie die verwendete Blockgröße „s_log_block_size“. Wie FFS reserviert auch Ext2 dem Administrator –bzw. dem Benutzer mit der UID „s_def_resuid“ und GID „s_def_resgid“– eine bestimmte Anzahl an Inodes und Blöcken für den Notfall und zählt die davon benutzten Blöcke in „s_r_blocks_count“. Ein wichtiger Unterschied zu FFS ist das Feld „s_lastcheck“, das angibt, wann das Dateisystem zum letzten Mal auf Konsistenzfehler überprüft worden ist. Nach „s_checkinterval“ vergangenen Sekunden oder Mountvorgängen wird das Dateisystem auf Fehler überprüft und diese automatisch korrigiert. Desweiteren gibt es ein Statusfeld „s_state“ zur Anzeige, ob das Dateisystem ordentlich ein- und ausgehängt worden ist. Nach einem Systemabsturz wird beispielsweise das System beim Mounten gezwungen, einen Konsistenzcheck durchzuführen, weil es den Absturz anhand „s_state“ erkannt hat. In „s_errors“ wird vermerkt, wie sich das Dateisystem in einem derartigen Fall verhalten soll. Der Superblock belegt 1024 Byte und wird durch Füllbytes immer auf diese Größe aufgefüllt, so daß er leicht erweiterbar ist. Eine dieser Erweiterung ist beispielsweise die Fähigkeit, Volumennamen zu vergeben und zu verwalten. Dies ist im ursprünglichen Konzept von Ext2 nicht vorhanden gewesen.

Zum Ext2-Dateisystem gehört neben dem Kernelcode auch eine Benutzerbiblio-

```
struct ext2_super_block {
  __u32 s_inodes_count;      /*Inodes count*/
  __u32 s_blocks_count;      /*Blocks count*/
  __u32 s_r_blocks_count;    /*Reserved blocks count*/
  __u32 s_free_blocks_count; /*Free blocks count*/
  __u32 s_free_inodes_count; /*Free inodes count*/
  __u32 s_first_data_block;  /*First Data Block*/
  __u32 s_log_block_size;    /*Block size*/
  __s32 s_log_frag_size;     /*Fragment size*/
  __u32 s_blocks_per_group;  /*# Blocks per group*/
  __u32 s_frags_per_group;   /*# Fragments per group*/
  __u32 s_inodes_per_group;  /*# Inodes per group*/
  __u32 s_mtime;             /*Mount time*/
  __u32 s_wtime;             /*Write time*/
  __u16 s_mnt_count;         /*Mount count*/
  __s16 s_max_mnt_count;     /*Maximal mount count*/
  __u16 s_magic;             /*Magic signature*/
  __u16 s_state;             /*File system state*/
  __u16 s_errors;            /*Behaviour when detecting errors*/
  __u16 s_minor_rev_level;   /*minor revision level*/
  __u32 s_lastcheck;         /*time of last check*/
  __u32 s_checkinterval;     /*max. time between checks*/
  __u32 s_creator_os;        /*OS*/
  __u32 s_rev_level;         /*Revision level*/
  __u16 s_def_resuid;        /*Default uid for reserved blocks*/
  __u16 s_def_resgid;        /*Default gid for reserved blocks*/
  __u32 s_first_ino;         /*First non-reserved inode*/
  __u16 s_inode_size;        /*size of inode structure*/
  __u16 s_block_group_nr;    /*block group # of this superblock*/
  __u32 s_feature_compat;    /*compatible feature set*/
  __u32 s_feature_incompat;  /*incompatible feature set*/
  __u32 s_feature_ro_compat; /*readonly-compatible feature set*/
  __u8  s_uuid[16];          /*128-bit uuid for volume*/
  char  s_volume_name[16];   /*volume name*/
  char  s_last_mounted[64];  /*directory where last mounted*/
  __u32 s_algorithm_usage_bitmap; /*For compression*/
  __u8  s_prealloc_blocks;        /*Nr of blocks to try to preallocate*/
  __u8  s_prealloc_dir_blocks;    /*Nr to preallocate for dirs*/
  __u16 s_padding1;
  __u32 s_reserved[204];          /*Padding to the end of the block*/
};
```

Abbildung 3.29: Die Struktur „ext2_super_block“ aus „include/linux/ext2_fs.h“

thek „libext2fs“, die privilegierten Programmen den einfachen Zugriff auf die Datenstrukturen des Dateisystems erlaubt. Diese Bibliothek, die auch von den Ext2-Verwaltungsprogrammen benutzt wird, erleichtert die Entwicklung verschiedener Hilfsprogramme. Zu diesen Programmen gehören „tune2fs“ zum Einstellen verschiedener Parameter des Superblocks und „e2fsck“ zum Überprüfen der Konsistenz des Dateisystems.

Blockstruktur

Das Layout von Ext2 auf der Festplatte ist der Blockstruktur von FFS äußerst ähnlich. Ext2 unterteilt die Festplatte in sogenannte „Blockgruppen“ (engl. „block groups“), die konzeptuell den Zylindergruppen von FFS entsprechen. Es wertet allerdings keine Geometrieinformationen des Mediums aus, so daß kein besonderer Zusammenhang zwischen dem Layout des Dateisystems und der physikalischen Geometrie der Festplatte zu erwarten ist. Ext2 kann mit Blöcken von 1, 2 oder 4 KB Größe arbeiten. Je nach Partitionsgröße paßt das Formatierungsprogramm „mke2fs“ die Blockgröße automatisch an, wenn eine Größe nicht explizit gewählt wird. Eine

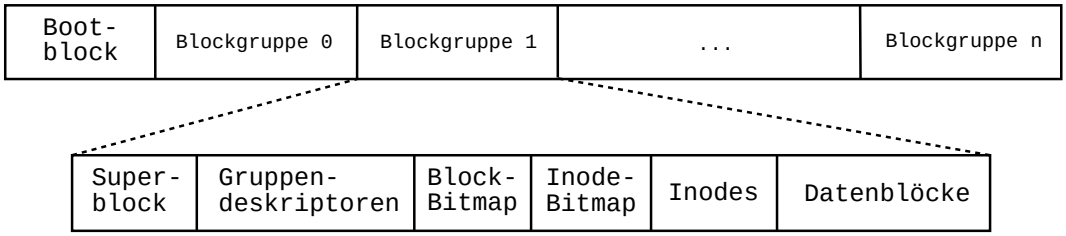

Abbildung 3.30: Das Blocklayout von Ext2

Blockgruppe besteht aus einer Kopie des Superblocks, einer Kopie aller Blockgruppendeskriptoren, einer Blockbelegungsbitmap für die Datenblöcke der Gruppe, einer Inodebelegungsbitmap für die belegten Inodes dieser Blockgruppe, den Inodes und den Datenblöcken, die der Gruppe zugeordnet sind. Abb. 3.30 zeigt das Layout einer Partition und einer Blockgruppe bei Ext2. Die Anzahl der Blöcke pro Gruppe „s_blocks_per_group“ und die Anzahl der Inodes pro Gruppe „s_inodes_per_group“ werden im Superblock vermerkt.

Jede Blockgruppe wird durch ihren Deskriptor beschrieben. Der Deskriptor enthält außer der Blockadresse „bg_block_bitmap“, der Blockbelegungsbitmap und der Blockadresse „bg_inode_bitmap“ der Inodebelegungsbitmap auch die Startadresse der Inodetabelle „bg_inode_table“ für diese Blockgruppe (siehe Abb. 3.31). Zudem sind einige statistische Informationen über die Auslastung der Blockgruppe gespeichert. Neben der Anzahl der freien Blöcke „bg_free_blocks_count“ und der freien Inodes „bg_free_inodes_count“ in der Blockgruppe wird die Anzahl der in dieser Blockgruppe angelegten Verzeichnisse „bg_used_dirs_count“ notiert. Ext2 verwendet ähnlich wie FFS diese Informationen bei der globalen Suche nach freien Blöcken und freien Inodes. Die Anzahl der bereits benutzten Verzeichnisse wird beim Anlegen neuer Verzeichnisse verwendet. Ein Deskriptor belegt insgesamt 32 Byte. Bei einer Blockgröße von 1 KB passen also 32 Deskriptoren in einen Block. Die Größe der Blockgruppen wird durch die Forderung beschränkt, daß die Inode- und

```
struct ext2_group_desc
{
  __u32 bg_block_bitmap;      /*Blocks bitmap block*/
  __u32 bg_inode_bitmap;      /*Inodes bitmap block*/
  __u32 bg_inode_table;       /*Inodes table block*/
  __u16 bg_free_blocks_count; /*Free blocks count*/
  __u16 bg_free_inodes_count; /*Free inodes count*/
  __u16 bg_used_dirs_count;   /*Directories count*/
  __u16 bg_pad;
  __u32 bg_reserved[3];
};
```

Abbildung 3.31: Die Struktur „ext2_group_desc“ aus „include/linux/ext2_fs.h“

Blockbitmaps der Blockgruppe in jeweils einen Datenblock passen müssen. Bei einer Blockgröße von 1024 Byte kann eine Blockgruppe also maximal 8192 Inodes und 8192 Blöcke enthalten. Bei den anderen Blockgrößen liegt das Limit entsprechend bei 16384 bzw. 32768 Objekten pro Bitmapblock. Ein mit Standardparametern angelegtes Dateisystem wird bei einer Blockgröße von 1KB also aus Blockgruppen mit jeweils 8 MB an Datenblöcken bestehen. Beim Anlegen von Dateien und Verzeichnissen verwendet Ext2 eine ähnliche Strategie wie FFS: Es versucht die Inode einer Datei in derselben Blockgruppe einzuordnen wie die Inode des Verzeichnisses, das die Datei referenziert. Datenblöcke von Dateien und Verzeichnissen sollen möglichst in derselben Blockgruppe belegt werden, in der die Inode der jeweiligen Datei liegt. Um zu verhindern, daß sich alle Dateien in derselben Blockgruppe sammeln, wird die Inode eines neuen Verzeichnisses in einer anderen Blockgruppe angelegt. Ext2 wählt für neue Verzeichnisse unter den Blockgruppen mit überdurchschnittlich vielen freien Inodes die Blockgruppe mit den meisten freien Datenblöcken aus. Dazu werden in allen Blockgruppendeskriptoren die Felder „bg_free_blocks_count“ und „bg_free_inodes_count“ ausgewertet.

Extentbasierte Preallokation

Da Ext2 keinen Versuch unternimmt, die Anordnung von Blöcken innerhalb einer Blockgruppe der Geometrie des Mediums anzupassen, entfallen Routinen zur Optimierung der Lage von Blöcken innerhalb einer Spur. Ext2 versucht jedoch, Datenblöcke einer Datei in möglichst großen zusammenhängenden Stücken, sogenannten „Extents“, zu belegen. Dazu verwendet das Dateisystem einen Blockallokierungssstrategie, die mit der Eingabes eines Zielblocks arbeitet (engl. „goal based allocator“). Der implementierte Algorithmus versucht zunächst, den Zielblock zu belegen. Ist dieser nicht verfügbar, werden in der Umgebung des Zielblocks –innerhalb von 64 Blöcken– freie Blöcke gesucht. Ist auch dort kein freier Platz vorhanden, wird zunächst versucht, innerhalb der Gruppe des Zielblocks überhaupt einen freien Datenblock zu finden. Erst danach wird auf eine lineare Suche nach freiem Platz in anderen Gruppen zurückgegriffen. Die eben beschriebenen Vorgänge spielen sich im Hauptspeicher ab und werden nicht in der Disk-Inode gespeichert, sondern in dem spezifischen Teil der VFS-Inode (siehe Abb. 3.13). Dieser für Ext2 spezifische Teil „ext2_inode_info“ der VFS-Inode aus „include/linux/ext2_fs_i.h“ ist in Abb. 3.32

```
struct ext2_inode_info {
  __u32 i_data[15];
  __u32 i_flags;
  __u32 i_faddr;
  __u8  i_frag_no;
  __u8  i_frag_size;
  __u16 i_osync;
  __u32 i_file_acl;
  __u32 i_dir_acl;
  __u32 i_dtime;
  __u32 i_block_group;
  __u32 i_next_alloc_block;
  __u32 i_next_alloc_goal;
  __u32 i_prealloc_block;
  __u32 i_prealloc_count;
  __u32 i_dir_start_lookup;
  int i_new_inode:1; /*Is a freshly allocated inode*/
};
```

Abbildung 3.32: Die Struktur „ext2_inode_info“ aus „include/linux/ext2_fs_i.h“

gezeigt. Der gewünschte Zielblock wird in „i_next_alloc_goal“ gespeichert und das gelieferte Ergebnis in „i_next_alloc_block“. Durch Vorgabe des Zielblocks versucht Ext2, große Extents zu bilden. Für neu angelegte Dateien wird der erste freie Datenblock der Blockgruppe als Ziel vorgegeben. Für alle weiteren Blöcke wird der zuletzt von dieser Datei belegte Block angegeben. Zusätzlich unterstützt Ext2 das Entstehen von Extents durch das Vorbestellung von Blöcken, „Preallokation“ genannt. Wann immer ein neuer Block für eine Datei angefordert wird, versucht das Dateisystem, bis zu acht zusammenhängende Blöcke in Folge zu belegen. Die vorbelegte Anzahl von Blöcken wird in „i_prealloc_count“ gespeichert. Die Blocknummer des ersten der reservierten Blöcke wird in „i_prealloc_block“ vermerkt. Nachfolgende Anforderungen von Blöcken für diese Datei werden dann durch diese vorbestellten Blöcke erfüllt. Die Vorbestellung wird aufgegeben und die vorbestellten Blöcke damit freigegeben, wenn der Zielblock nicht im vorbestellten Bereich liegt. Wenn die Datei geschlossen wird bzw. wenn ihre VFS-Inode aus dem Inode-Cache genommen wird, dann geht die Vorbestellung natürlicherweise verloren, da die Vorbestellung nicht in der Disk-Inode vermerkt werden.

Leseoptimierungen

Dateien, die aus zusammenhängenden Stücken bestehen, werden vom Dateisystem und den Gerätetreibern der Festplatte in LINUX besonders effizient behandelt. Ext2 versucht daher, aufeinanderfolgende Leseaufträge innerhalb einer Datei zu erkennen, und erzeugt in diesem Fall auf Verdacht Leseanforderungen, um sie im voraus in den Puffercache zu laden (engl. „read ahead“). Gleichzeitig versucht LINUX, aufeinanderfolgende Leseanforderungen zusammenzuziehen (engl. „read clustering“), so daß der Gerätetreiber statt vieler kleiner, aufeinanderfolgender Anforderungen nur eine

große Anforderung zu sehen bekommt.[49] Beide Maßnahmen zusammen, Vorauslesen begünstigt durch Preallokation und das Zusammenführen von Aufträgen, haben im Endeffekt dieselbe Beschleunigungswirkung wie der Wechsel auf eine größere Blockgröße beim FFS. Setzt man FFS-Fragmente Ext2-Blöcken gleich und interpretiert man FFS-Blöcke als Ext2-Vorbestellungen, dann ist das Ext2 in gewisser Weise eine Verallgemeinerung von FFS. Zwar kann eine Ext2-Vorbestellung genau wie ein FFS-Block immer nur einen zusammenhängenden Bereich belegen, aber im Gegensatz zu FFS-Blöcken sind Ext2-Vorbestellungen in der Länge zwischen zwei und acht Ext2-Blöcken variabel. Außerdem können sie auf einem beliebigen Ext2-Block beginnen, während FFS-Blöcke nur auf Blockgrenzen beginnen können.

Verzeichnisse

Ebenso wie FFS läßt Ext2 bis zu 255 Zeichen lange Dateinamen zu. Die Organisation der Verzeichnisse erfolgt genau wie in FFS als lineare Liste von Inode-Nummern und Namen. Ein Unterschied besteht darin, daß bei neueren Ext2-Versionen das Namenslängenfeld „name_len“ von 16 Bit auf 8 Bit verkleinert ist, was für 255 Zeichen immer noch ausreichend ist. Dafür ist statt dessen der Typ der Datei zusätzlich im Verzeichnis notiert. Diese Struktur „ext2_dir_entry2“ aus „include/linux/ext2fs_fs.h“ist

```
struct ext2_dir_entry_2 {
  __u32 inode; /*Inode number*/
  __u16 rec_len; /*Directory entry length*/
  __u8  name_len; /*Name length*/
  __u8  file_type;
  char  name[EXT2_NAME_LEN]; /*File name*/
};
```

Abbildung 3.33: Die Struktur „ext2_dir_entry2“ aus „include/linux/ext2fs_fs.h“

in Abb. 3.33 dargestellt. Sie läßt diverse Optimierungen beim Auffinden oder Listen von Dateien in Verzeichnissen zu, weil dem Dateisystem, ohne die Inode gelesen zu haben, der Typ der Datei bekannt ist. Trotzdem ist die Behandlung von Verzeichnissen bei Ext2 im Vergleich zu modernen Dateisystemen sehr primitiv, wie wir weiter unten im Vergleich mit modernen Systemen sehen werden.

Konsistenzüberprüfung

Wie bereits erwähnt, müssen die Metadaten auf Konsistenz überprüft werden, um ein funktionierendes Dateisystem gewährleisten zu können. Das zu den Werkzeugen von Ext2 gehörende Programm „e2fsck“ geht dazu in fünf Phasen vor:

- Phase 1: Überprüfung der Inodes, Blöcke und Dateigrößen
  Zuerst durchläuft „e2fsck“ die Liste aller Inodes und sucht nach ungültigen Einträgen. Typische Fehler sind ungültige Zugriffsrechte, falsche Dateigrößen oder Verweise auf Datenblöcke, die bereits in der Blockliste einer anderen Inode enthalten ist.

[49] Beide Optimierungsversuche sind strenggenommen nicht spezifisch für Ext2, sondern werden auf der Ebene des VFS durchgeführt (siehe „mm/filemap.c“ und den Abschnitt 3.3.2).

- Phase 2: Überprüfung der Verzeichnisstruktur
  Alle Verzeichniseinträge werden nach aus der ersten Phase ungültigen Inode-Einträgen durchsucht.

- Phase 3: Überprüfung der Verbindungen zwischen den Verzeichnissen
  Es wird getestet, das jedes Verzeichnis erreichbar ist, und es in einem Pfadeintrag ausgehend vom obersten Verzeichnis des Dateisystems erscheint. Verzeichnisse, die keine Verbindung aufweisen, werden gemeinsam mit den enthaltenen Dateien in ein spezielles Verzeichnis namens „/lost+found“ verschoben.

- Phase 4: Überprüfung des Verweiszählers
  Die in jeder Inode gespeicherte Anzahl an harten Verweisen wird mit den tatsächlich auf diesen Inode verweisenden festen Links verglichen.

- Phase 5: Überprüfung der Bitmaps
  Die Inode- und Blockbitmaps werden entsprechend der belegten bzw. unbelegten Inodes und Blöcke korrigiert.

Es ist offensichtlich, daß diese Überprüfung trotz aller Optimierungsversuche durch geschicktes Zwischenspeichern von Inodes und Verzeichnisblöcken bei großen Partitionen bis zu mehreren Stunden dauern kann. In dieser Zeit kann auf die zu überprüfende Partition natürlich nicht zugegriffen werden, weil das die bereits erfaßte Statistik durcheinander bringen würde.

Weitere Details zu der Implementierung von Ext2 entnehme man [CTT94], [Die00] [Bar01a], Kap. 7 und [BC01], Kap. 17.

3.6 Moderne Dateisysteme unter LINUX

Wie wir aus Abschnitt 3.2 wissen, hat ein Dateisystem im allgemeinen die Aufgabe, eine logische Organisation des Speicherplatzes auf dem Speichermedium vorzunehmen, so daß auf die Daten und auf die zur Verwaltung der Daten notwendigen Metadaten sicher und effizient zugegriffen werden kann. Die im letzten Abschnitt beschriebenen Dateisysteme stellen klassische Dateisysteme unter UNIX dar und erfüllen die an sie gestellten Ansprüche in dem ihnen zugedachten Kontext. Die Ansprüche an die Sicherheit und Leistungsfähigkeit eines Dateisystems hängen unter anderem von der Wichtigkeit und dem Verwendungszweck der Daten, des insgesamt zu verwaltenden Speicherplatzes auf dem Medium und der durchschnittlichen Größe der Daten ab. Sie unterliegen damit genauso einem Wandel mit dem technischen Fortschritt wie die anderen Komponenten eines modernen Computersystems.

Im LINUX-Umfeld spricht man momentan von den folgenden Journaling-Dateisystemen als den „Dateisystemen der nächsten/neuen Generation“ (engl. „next/new generation file system“):

- „ReiserFS“ Dateisystem
  Dieses innovative Dateisystem von Hans Reiser und seinem Team ist bereits

seit fast zwei Jahren einigermaßen stabil unter LINUX verfügbar[50] und auch speziell dafür implementiert. Die aktuelle Version 3.6.x wird im Kernel ab der Version 2.4 standardmäßig mitgeliefert.

- „Irix File System“ Dateisystem
  Dieses Dateisystem ist eine Portierung des Multimediadateisystems von IRIX der Firma SGI auf LINUX.[51] Es ist mittlerweile in der Version 1.02 veröffentlicht und wird voraussichtlich in den Kernel 2.5 bzw. 2.6 integriert.

- „Journaled File System“ Dateisystem
  Dieses Dateisystem ist eine Portierung des Journaling-Dateisystems von OS/2 der Firma IBM auf LINUX.[52] Es ist seit kurzem in der Version 1.09 verfügbar und auch noch nicht offiziell in den Kernel integriert. Es ist zu erwarten, daß auch dieses in Kürze in den Standardkernel integriert werden wird.

- „Ext3FS“ Dateisystem
  Dieses Dateisystem ist eine Erweiterung von Ext2.[53] Es ist abwärtskompatibel zu Ext2 und erweitert es um Journaling-Fähigkeiten. Es ist bereits in den 2.4 Kerneln von Alan Cox integriert, aber als „experimentell“ gekennzeichnet.

3.6.1 Zusammenfassung der klassischen Techniken

Wir wollen im folgenden die Techniken der neuen Dateisysteme vorstellen und den klassischen gegenüberstellen. Dazu fassen wir diese zunächst einmal zusammen.

Die bisher beschriebenen klassischen Dateisysteme unter UNIX haben die folgenden Eigenschaften:

- Trennung von Daten und Metadaten
  Die Metadaten einer Datei werden in ihrer Inode verwaltet. Die Datenblöcke der Datei werden über bis zu dreifach indirekte Verweise adressiert. Damit sind Daten und Metadaten strikt voneinander getrennt.

- Feste Anzahl von Inodes
  Zur Formatierungszeit muß der Administrator festlegen, wieviele Inodes das System anlegen soll. Somit ist die Anzahl der unterstützten Dateisystemobjekte statisch festgelegt und nur mit großem Aufwand nachträglich auf eine neue Anzahl änderbar.

- Blockbasierte Allokation
  Wenn eine Datei erweitert oder angelegt wird, dann allokiert das Dateisystem in dem Moment die minimale Anzahl der dafür benötigten Blöcke. Es werden dadurch niemals mehr Blöcke allokiert als benötigt.

[50] Informationen, Quellcode und Benchmarks findet man unter http://www.namesys.com/.
[51] Informationen und Quellcode findet man unter http://oss.sgi.com/projects/xfs/.
[52] Quellcode findet man unter http://oss.software.ibm.com/developer/opensource/jfs/.
[53] Quellcode findet man unter http://www.zipworld.com.au/~akpm/linux/ext3/.

- Bitmaps
  Zur Verwaltung von freien Speicherblöcken und freien Inodes werden Bitmaps benutzt. Der Blockallokationsalgorithmus benutzt ihre Information zur Umsetzung seiner Strategie bei der Blockwahl.

- Verzeichnisse als lineare Listen
  Ein Verzeichnis ist eine durch das Dateisystem verwaltete Datei, deren Datenblöcke Verzeichniseinträge der Dateien im Verzeichnis speichern. Die Verzeichniseinträge sind zu einer linearen Liste verkettet. Sie ordnen einer Inode einen Dateinamen innerhalb des Verzeichnisses zu.

- Fragmente
  Zur Verminderung von interner Fragmentierung werden bei manchen Dateisystemen Fragmente benutzt, die die Enden von Dateien speichern können.

- Starre Konsistenzerhaltung
  Dateisystemkonsistenz zwischen Metadaten und Daten wird im Prinzip statisch vollzogen. Das Dateisystem notiert seinen Zustand im Superblock und überprüft seinen Zustand vollständig, wann immer es den Verdacht hat, daß es zu Inkonsistenzen gekommen sein könnte.

3.6.2 Einführung neuer Techniken

Dagegen stellen wir nun einige neuere Ideen und Konzepte, die bei moderneren, kommerziellen UNIX-Systemen in der jüngeren Vergangenheit umgesetzt wurden. Sie haben ihren Ursprung größtenteils im Datenbankdesign und halten seit kurzem verstärkt in LINUX-Dateisystemen Einzug:

Extentbasierte Allokation

Eine Menge von aufeinanderfolgenden logischen Datenblöcken wird „Extent“ genannt. Statt die Datenblöcke einer Datei in Form einer Menge von logischen Blocknummern aufzuzählen, kann man die Menge der von der Datei benutzten Extentbeschreibungen angeben. Eine Extentbeschreibung (engl. „extent descriptor“) besteht dabei aus der logischen Blocknummer des ersten Blocks und der insgesamten Anzahl der Blöcke des Extents sowie einem Offset zur Angabe des Bytes der Datei, ab welcher das Extent zur Speicherung von Daten benutzt wird. Allokierung ganzer Extents statt einzelner Blöcke erhöht automatisch die räumliche Lokalität und vermindert externe Fragmentierung, weil die Datenblöcke innerhalb eines Extents per Definition aufeinanderfolgen müssen. Allerdings erhöhen Extents naturgemäß die interne Fragmentierung.

B^+-Bäume als Zugriffstruktur

Aus dem Datenbankdesign ist bekannt, daß bei index-orientierten Zugriffsverfahren dynamische Mehrwegbäume sehr effiziente Datenstrukturen darstellen [HR99]. Eine in der Praxis oft benutzte Variante balancierter Bäume ist der B^+-Baum:
Seien k, k^* und h^* ganze Zahlen mit $h^* \geq 0$, $k, k^* > 0$. Ein B^+-Baum der Klasse (k, k^*, h^*) ist entweder ein leerer Baum oder ein geordneter Baum mit den Eigenschaften:

1. Jeder Pfad von der Wurzel zu einem Blatt besitzt die gleiche Länge h^*.
2. Jeder Knoten außer der Wurzel und den Blättern hat mindestens $k+1$ Söhne, die Wurzel mindestens zwei Söhne, außer wenn sie ein Blatt ist.
3. Jeder innere Knoten hat höchstens $2k+1$ Söhne.
4. Jeder Blattknoten mit Ausnahme der Wurzel als Blatt hat mindestens k^* und höchstens $2k^*$ Einträge.

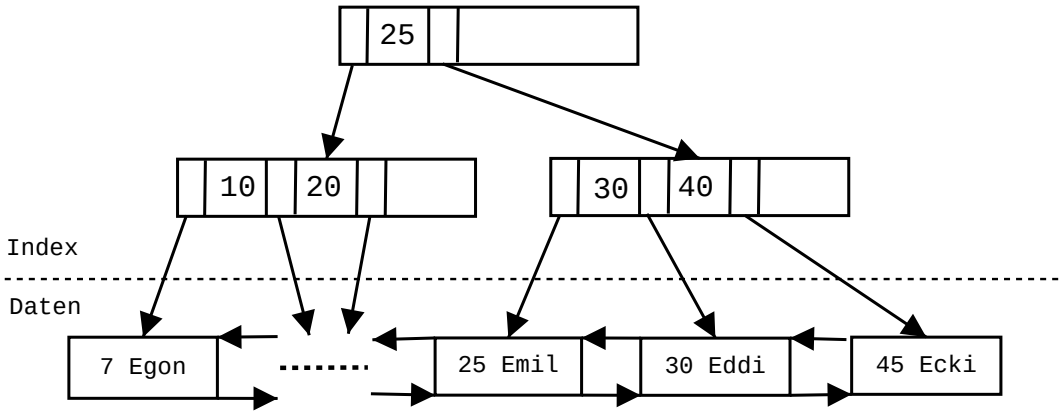

Abbildung 3.34: Ein B^+-Baum als Indexstruktur einer Namensdatei

In Abb. 3.34 ist ein B^+-Baum der Klasse (1,1,2) zur Indizierung einer Namensdatei dargestellt. Anhand des Indexes bzw. Schlüssels kann auf die Namen innerhalb der Datei effizient zugegriffen werden. Pro Name sind dazu genau zwei Vergleiche nötig. Es ist offensichtlich, daß eine Implementierung eines B^+-Baums bei der Verwaltung von freien Speicherblöcken, aber auch bei der Verwaltung von Verzeichniseinträgen effektiver ist als Bitmaps oder lineare Listen. Allerdings hat das seinen Preis: Operationen wie Einfügen und Löschen auf B^+-Bäumen sind komplex in ihrer Implementierung und verbrauchen relativ viel CPU-Zeit. Der Baum muß zudem nach einer Operation meist ausbalanciert werden, damit die oben genannten Bedingungen erfüllt bleiben [OW93]. Moderne Datenbanken benutzen in der Regel verbesserte Erweiterungen des B^+-Baums. Beim sogenannten „B^*-Baum“ teilt man bei einem Überlauf zwei Knoten in drei Knoten auf statt einen in zwei wie beim B^+-Baum und vergrößert dadurch die durchschnittliche Auslastung von 50% auf 66.7% [SH99].

Protokollierung der Transaktionen

Datenbanken benutzen Transaktionen bei der Ausführung von Befehlen auf ihrem Datenbestand. Eine Transaktionen ist eine Menge von einzelnen Operationen. Sie erfüllen das sogenannte „ACID“ Prinzip:

- Atomarität (engl. „atomicity“): Eine Transaktion wird entweder ganz oder gar nicht durchgeführt.

- Konsistenz (engl. „consistency“): Transaktionen führen eine Datenbank aus einem konsistenten Zustand in einen neuen konsistenten Zustand. Alle Integritätsbedingungen müssen erfüllt sein.
- Isolation (engl. „isolation“): Transaktionen laufen isoliert und können sich gegenseitig nicht beeinflussen.
- Dauerhaftigkeit (engl. „durabilty“): Die Wirkung einer erfolgreich beendeten Transaktion ist dauerhaft.

Manche Datenbanken erfüllen diese Serialisierungsanforderungen ohne aufwendige Sperrprotokolle. Bei diesen sogenannten „optimistischen Verfahren“ führen sie beispielsweise jede einzelne Operation einer Transaktion –auch parallel ablaufender Transaktionen– ohne Prüfung durch, protokollieren aber den Zustand der Datenbank vor und nach jeder einzelnen Operation detailliert in ein Logbuch.[54] Eine Transaktion gilt aber erst dann als wirklich durchgeführt bzw. bestätigt (engl. „commit“), wenn die Datenbank sichergestellt hat, daß sie keine Integritätsbedingung verletzt hat. Nach dem sogenannten „WAL-Prinzip“ (engl. „write ahead log“) muß vor jeder Bestätigung einer Transakion der entsprechende Logeintrag physikalisch geschrieben sein. Das gleiche gilt, wenn noch nicht bestätigte Transaktionen bereits physikalisch geschrieben werden sollen. Bei Auftreten eines Fehlers ermöglicht das Logbuch in erstem Fall ein erneutes Durchführen der Transaktion (engl. „redo“) und im zweiten Fall eine Rücknahme der Transaktion (engl. „undo“). In regelmäßigen Abständen oder wann immer notwendig, geht die Datenbank das Logbuch durch und überprüft, welche Transaktionen bereits persistent geschrieben sind. Deren Einträge können dann im Prinzip aus dem Logbuch entfernt werden und die Zeit des letzten Sicherungspunkts (engl „checkpoint“) wird im Journal vermerkt [HR99].

Journaling-Dateisysteme benutzen das eben erläuterte Verfahren, um Dateisystemoperationen zu protokollieren. Im Falle eines Systemabsturzes oder ähnlichen Katastrophen kann mithilfe des Logbuchs (engl. „ journal“) die Konsistenz des Dateisystems in kurzer Zeit hergestellt werden, indem die Logbucheinträge bis zum letzten Sicherungspunkt ausgewertet werden. Im Unterschied zu Datenbanken protokollieren die meisten Dateisysteme nur Änderungen an Metadaten. Es gibt aber auch Journaling-Systeme, die Metadaten und Daten im Logbuch vermerken.

3.6.3 Eigenschaften moderner Systeme

Die klassischen Dateisysteme sind konzipiert worden, als die Speichermedien bei weitem noch nicht die Kapazität heutiger Dimensionen hatten. Die höhere Kapazität hat zur Folge, daß ein Dateisystem heutzutage größere Dateien, größere Verzeichnisse und größere Partitionen verwalten muß. Das kann bei manchen Dateisystemen

[54] Unterschieden wird dabei, ob die Einträge logisch formuliert werden oder ob einfach die Zustände vor und nach der Operation (engl. „before and after image“) gespeichert werden. Bei einer Rekonstruktion müssen im ersten Fall die Operationen nachgespielt oder eventuell invertiert werden, dafür nehmen die Einträge im Protokoll typischerweise kaum Platz in Anspruch. Im zweiten Fall können die Bilder einfach ersetzt werden, allerdings kostet ihre komplette Speicherung auch mehr Speicherplatz im Logbuch.

zu Problemen führen, weil die internen Strukturen und Zugriffspfade nicht dafür vorgesehen sind. Im Prinzip gibt es bei den klassischen Strukturen zwei Probleme:

- Sie können die neuen Größen nicht bewältigen, da beispielsweise in den Inodes manche Felder wie die Dateigröße, die Inode-Nummer oder die Blockadressen zu wenig Bits zur Verfügung stellen.
- Sie können die neuen Dimensionen nicht effektiv verwalten, obwohl sie prinzipiell dazu in der Lagen sind. Ein Beispiel ist die Verwaltung von Verzeichniseinträgen als lineare Liste. Hunderttausend Einträge hatte zur Konzeptionszeit des Dateisystems niemand vorgesehen.

Prinzipielle Skalierung mit der Kapazität

Die internen Dateisystemstrukturen der modernen Dateisysteme sind deutlich erweitert worden, so daß sie auch mit zukünftigen Partionsgrößen umgehen können. Ihre Begrenzungen sind in Abb. 3.35 angegeben. Es gilt zu beachten, daß unter LINUX eine Dateigrößengrenze von 4 GB durch die VFS-Inode besteht.

	Max. Partitionsgröße	Blockgröße	Max. Dateigröße
RFS	Bei 4 KB Blöcken **16 TB**	Bis 64 KB vorgesehen; momentan fest bei **4 KB**	**1024 PB**
XFS	**18000 PB**	512 Byte bis 64 KB; unter LINUX fest bei **4 KB**	**9000 PB**
JFS	Bei 512 Byte Blöcken 4 PB; bei 4 KB Blöcken **32 PB**	512 Byte bis 4 KB unter LINUX fest bei **4 KB**	Bei 512 Byte 512 TB; bei 4 KB Blöcken **4 PB**
Ext3	**4 TB**	**1 bis 4 KB**	**4GB**

Abbildung 3.35: Die Größenbeschränkungen der neuen Dateisysteme

Verwaltung freien Speichers durch Extents und B^+-Bäume

Um den höheren Ansprüchen zu genügen, benutzen die neuen Dateisysteme effektivere Zugriffsstrukturen als Bitmaps und lineare Listen. Die Algorithmen auf den Bitmaps führen im schlechtesten Fall eine sequentielle Suche nach freiem Speicherplatz durch, so daß letztendlich Bitmaps und lineare Listen im schlechtesten Fall mit $O(n)$ skalieren, wobei n die Bitmapgröße oder die Anzahl der Listenelemente darstellt.

Moderne Dateisysteme verwenden Extents und B^+-Bäume zur Organisation von freiem Speicher. Der Ansatz ist sinnvoll, denn durch Extents werden mehrere freie logische Blöcke gleichzeitig mit weniger Beschreibungsaufwand adressiert. Dadurch braucht man selbst bei der Verwendung von Bitmaps nicht mehr für jeden logischen Block ein Bit, sondern nur für jedes Extent. Die Performanz linearer Suchalgorithmen steigt dadurch, skaliert aber immer noch mit der Größe der Bitmap. Einen wesentlichen Performanzunterschied kann man erreichen, wenn die freien Blöcke oder die freien Extents durch einen B^+-Baum organisiert werden. Wenn freie Blöcke benötigt werden, dann sucht das Dateisystem den Baum der freien Blöcke durch, um den entsprechenden Freiraum zu finden. Im Fall der Verwaltung von Extents durch

einen B^+-Baum kann man sogar auf zwei verschiedene Weisen indizieren. Einerseits kann man –wie bei den Blöcken– die Position der Extents indizieren. Andererseits kann man zusätzlich die Größe der Extents als zweiten Index benutzen, so daß man effektiv nach Position und Größe des Freiraums suchen kann.

Verwaltung von Verzeichniseinträgen durch B^+-Bäume

Verzeichnisse, die bei den klassischen Dateisystemen ihre Einträge durch lineare Listen verwalten, profitieren natürlich ebenfalls von einer Organisation durch einen B^+-Baum. Dazu werden sinnvollerweise die Dateinamen eines Verzeichnisses als Index benutzt. Die Suche nach der einem bestimmten Namen zugeordneten Inode ist bei vielen Einträgen weitaus effizienter als eine lineare Liste zu durchlaufen, insbesondere wenn die Verzeichniseinträge sich über mehrere Datenblöcke erstrecken. Je nach Dateisystem gibt es unterschiedliche Herangehensweisen. Manche benutzen einen Baum für die gesamte Dateihierarchie, manche einen Baum pro Verzeichnis.

Verwaltung der Dateiblöcke durch B^+-Bäume

Die klassischen Dateisysteme gehen davon aus, daß sie im Durchschnitt viele kleine Dateien verwalten müssen. Durch das daraus resultierende Konzept mit der Verwaltung der Datenblöcke über wenige direkte und einige indirekte Blöcke sind große Dateien benachteiligt. Bei ihnen erfolgen alle Zugriffe über teilweise mehrfach indirekte Blöcke. Die modernen Systeme dagegen organisieren die Blöcke einer Datei wiederum durch einen B^+-Baum. Die verwendeten Blöcke werden durch ihren Offset innerhalb der Datei indiziert. Anstelle von Blöcken unterstützen manche Dateisysteme Extents, so daß die Anzahl von zu adressierenden Einheiten geringer wird. Gleichzeitig erzeugen sie einen Baum erst ab einer bestimmten Größe und benutzen vorher direkte Verweise. Um auch sehr kleinen Dateien oder Verzeichnissen gerecht zu werden, werden Informationen oder Daten auch direkt in der Inode gespeichert, ähnlich wie wir das für kurze symbolische Links bereits bei den klassischen Dateisystemen gesehen haben. Abb. 3.36 gibt einen Überblick über die verwendeten Verfahren der neueren Dateisysteme.

	Verwalt. freier Blöcke	Extents f. freien Platz	Bäume f. Verzeichnisse	Bäume f. Dateiblöcke	Extents f. Dateiblöcke	Kleine Dateien in Inode	Symb. Links Inode	Kleine Verz. Inode
RFS	basierend auf Bitmap	Geplant	Als Subbaum v. Hauptbaum[55]	Im Hauptbaum[55]	Ab Version 4	Im Hauptbaum[55]	Im Hauptbaum[55]	Im Hauptbaum[55]
XFS	B^+-Bäume	Ja	Ja	Ja	Ja	Ja	Ja	Ja
JFS	B^+-Baum u. Bitmap	Nein	Ja	Ja	Ja	Nein	Ja	Bis zu 8
Ext3	Basiert auf Ext2 und beinhaltet außer dem Journal keine der genannten Techniken[56]							

Abbildung 3.36: Zugriffstechniken der neuen Dateisysteme

Protokollierung von Metadatenänderungen

Jedes der erwähnten neuen Dateisysteme ist ein Journaling-Dateisystem. Diese benutzen –wie im letzten Abschnitt beschrieben– ein Logbuch, um Änderungen an ihrem Datenbestand zu protokollieren. Ext3 unterscheidet sich von den anderen dadurch, daß es auch Änderungen der Daten erfassen kann, was durch eine Option beim Einhängen eines Ext3-Dateisystems einstellbar ist. Dieses Verhalten reduziert natürlich die Schreibperformanz ganz erheblich. Betrachten wir an Abb. 3.37 eine Datei „Test“ mit der zugehörigen Inode-Nummer 854, die durch einen Schreibvor-

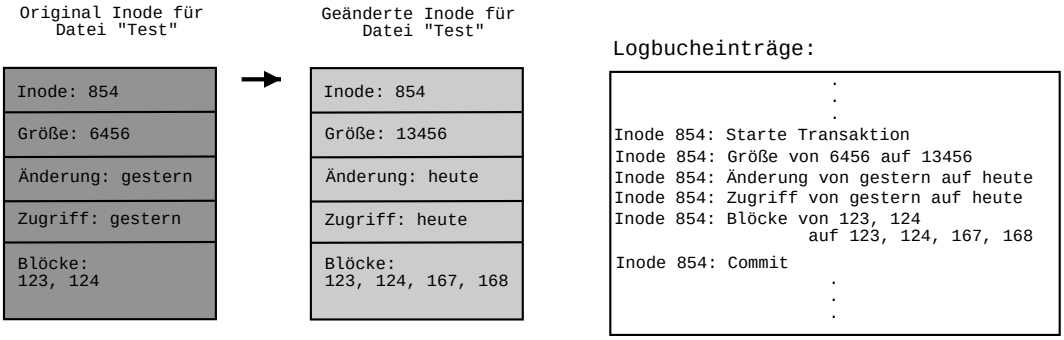

Abbildung 3.37: Änderung einer Inode und dazu passende Journaleinträge

gang um zwei Datenblöcke erweitert wird. Beim Schließen der Datei werden die gezeigten Logbucheinträge mit dem Logbuch auf dem Medium synchronisiert. Die modifizierte Inode und die geänderten Datenblöcke verbleiben in der Regel zunächst aber im Inode- bzw. Puffercache. Findet nun ein Systemabsturz statt, konsultiert das Dateisystem beim Einhängen das Logbuch und überprüft bis zum letzten Sicherungspunkt die Einträge darin. In unserem Beispiel könnten die Änderungen an der Inode übernommen werden. Da die Datenblockänderungen nicht notiert worden sind, ist es wahrscheinlich, daß diese vor dem Absturz nicht aktualisiert worden sind. Sie sind durch die Protokollierung der Metadaten nicht geschützt, nur die Konsistenz des Systems ist durch das Metadatenjournal in kurzer Zeit herstellbar.

Dynamische Inodes

Die beschriebenen klassischen Dateisysteme lassen die Erzeugung von Inodes nur zur Formatierungszeit zu. Modernere Dateisysteme erzeugen eine Inode statt dessen dynamisch in dem Moment, in dem sie benötigt wird. Dazu benutzen sie eine weitere Kontrollstrukur zur Abbildung zwischen Inodes und Blöcken, in denen die Inodes auf dem Medium gespeichert werden. In der Regel werden dazu wieder B^+-Bäume benutzt. Bei den klassischen Dateisystemen war diese weitere Struktur durch die statische Allokierung der Inodes bei gleichzeitig bekanntem Platz auf dem Medium nicht notwendig.

[55] Reiser organisiert jedes Dateisystemobjekt innerhalb eines B^*-Baums für die gesamte Partition, wie wir im nächsten Abschnitt sehen werden.

[56] Immerhin gibt es für Ext2 und damit auch meist für Ext3 viele Erweiterungen. Eine dieser Erweiterungen ist ein Patch von D. Phillips zur Indizierung von Verzeichniseinträgen, so daß auch unter Ext2 eine große Anzahl von Verzeichnisinhalten effizient speicherbar ist. Siehe http://people.nl.linux.org/~phillips/htree/ für diesen „Hash Tree Directory Index“ Patch.

Dateien mit Löchern

Durch die Benutzung von Extents wird die Verwaltung von sogenannten „dünn besetzten“ (engl. „sparse“) Dateien sehr vereinfacht. Derartige Dateien enthalten sogenannte „Löcher“ (engl. „hole“), die die Blöcke markieren, die von der Datei beispielsweise durch Positionierung des Dateizeigers hinter das Ende der Datei übersprungen werden. Beim Auslesen einer solchen Datei werden die Löcher mit entsprechend vielen Nullbytes ausgegeben. Durch die Extentbeschreibung läßt sich ein Loch und seine Größe in Byte genau angeben. Bei klassischen Dateisystemen können nur ganze Blöcke einfach übersprungen werden. Alle Sprünge innerhalb von Blöcken müssen tatsächlich mit Nullen gefüllt werden, und indirekte Blöcke müssen in jedem Fall allokiert werden.

	Journal	Dynamische Inodes	Verwaltung der Inodes	Dünne Dateien
Reiser	Metadaten	Ja	Im Hauptbaum[55]	Im Hauptbaum[55]
XFS	Metadaten	Ja	B^+-Baum	Ja
JFS	Metadaten	Ja	B^+-Baum mit Inode Extents	Ja
Ext3	Meta- und Daten	Nein	Nein	Eingeschränkt

Abbildung 3.38: Weitere Eigenschaften der neuen Dateisysteme

Abb. 3.38 stellt die eben beschriebenen weiteren Eigenschaften der neuen Dateisysteme gegenüber. Wiederum fällt auf, daß das Ext3-Dateisystem eigentlich kein neues Dateisystem ist, sondern lediglich das alte Ext2-Dateisystem um die Journaling-Fähigkeit erweitert. Allerdings ist das auch ein großer Vorteil, denn das Dateisystem ist nicht auf das Journal angewiesen und daher völlig kompatibel zu dem weitverbreiteten Ext2.

Weitere Informationen über die neuen Dateisysteme man findet unter den oben angegebenen Webseiten. Für eine detaillierte Gegenüberstellung der verschiedenen Techniken siehe [Flo01]. Für eine Einführung in allgemeine Eigenschaften von Journaling-Dateisystemen eignet sich [Bar00b]. Konkrete Journaling-Dateisysteme unter LINUX werden in [Bar00a] und [Del00] vorgestellt. Eine Untersuchung über die Implementierung dynamischer Inodes bei klassischen UNIX-Dateisystemen mit konkreter Umsetzung anhand des Ext2-Dateisystems wird in [Köh96] vorgenommen. Darin finden sich auch Referenzen auf andere Dateisysteme mit dynamischer Inode-Generierung, genauso wie Erläuterungen und Referenzen auf die aus Platz- und Zeitgründen in dieser Arbeit nicht erwähnten logbasierten Dateisysteme.[57] Da wir

[57] Ein Beispiel für ein logbasiertes Dateisystem ist das LFS von Sprite [RO92]. Dieses trägt alle Modifikationen am Dateisystem –also Daten und Metadaten– an einem Stück sequentiell in eine Art Logbuch ein. Das Logbuch stellt im wesentlichen die einzige Struktur auf der Platte dar und beinhaltet einen Index zum Lesen der Daten. Logbasierte Dateisysteme sind besonders schnell beim Schreiben und beim Wiederherstellung der Datenintegrität nach einem Systemabsturz. Die Logdatei wird dazu aus Effizienzgründen auf verschiedene Segmente verteilt. Ein sogenannter „Cleaner“ defragmentiert im Hintergrund diese Segmente, so daß immer genügend Platz für

uns in dieser Arbeit auf das Reiser-Dateisystem im Vergleich zu Ext2 konzentrieren, stellen wir hier einige Referenzen bezüglich der anderen neuen Dateisysteme zusammen. In [Joh01] und [Rob01] wird das relativ junge Ext3-Dateisystem vorgestellt. [SDH+96] beschreibt das XFS-Dateisystem, wie es seit der Version 5.3 von IRIX ausgeliefert wird, [MEL+00] beschreibt die Portierung auf LINUX. In einer Serie von drei Artikeln beschreibt Steve Best, einer der Entwickler von JFS für LINUX, das portierte Dateisystem, sein Layout auf der Platte und seine Journaling-Fähigkeiten [Bes00b], [BK00], [Bes00a]. Referenzen bezüglich des Reiser-Dateisystems geben wir am Ende des nächsten Abschnitts an.

3.6.4 Das Reiser-Dateisystem

Das „ReiserFS“ Dateisystem ist ein von Hans Reiser und seinem Programmierteam zunächst als Studie implementiertes Dateisystem. In der LINUX-Gemeinde fand es auch erst größere Beachtung, als es als eines der ersten Dateisysteme unter LINUX ein Journal zum Protokollieren von Metadaten integrierte. Dabei liegt seine Innovation eigentlich darin, daß es einen balancierten Baum für alle Dateisystemobjekte implementiert, wie wir im folgenden sehen werden.

Vereinheitlichte Namensräume

Hans Reiser bezeichnet sich selbst als einen der Betriebssystementwickler, die die Vereinheitlichung der Namensräume innerhalb eines Betriebssystems anzustreben versuchen.[58] Sein Ziel und gleichzeitig seine Vision ist es, die Anzahl der Kommunikationsebenen und die verschiedenen Systeme mit unterschiedlichen Namensräumen eines Betriebssystems zu reduzieren oder zu vereinheitlichen. Dadurch erhöhen sich automatisch die Kommunikationsmöglichkeiten der Objekte eines Systems bei gleichzeitig niedrigerem Kodierungs -und Wartungsaufwand. Dieses Ziel, eine Begründung für seine Rentabilität und seine Definition für einen übergreifenden, relationalen statt hierarchischen Dateisystemnamensraum formuliert er in [Rei01a].

Fokussierung auf kleine Dateien

Das Reiser-Dateisystem ist daher mehr als nur ein neues Dateisystem. Es ist der erste Beitrag von Hans Reiser, die Namensraumfragmentierung der Dateisystemschnittstelle zu vermindern: Sehr kleine Dateisystemobjekte sollen dasselbe Dateisysteminterface genauso effektiv wie die sehr großen Objekte nutzen können. Demnach ist das primäre Designziel des Reiser-Dateisystems die effektive Integration kleiner Dateien in den Namensraum. Dabei bedeutet „klein“ eine Größenordnung von 100 bis 1000 Byte. Bei den bisher beschriebenen und auch bei anderen, nicht erwähnten Dateisystemen stellten sich die Entwickler beim Design auf den Standpunkt, daß die effektive Behandlung derart kleiner Dateien nicht signifikant für die gesamte Performanz des Dateisystems ist. Sie überließen es den Applikationsprogrammierern, sich

das sequentielle Schreiben vorhanden ist [BHS95]. Es stellt sich aber bei Langlaufzeitsimulationen heraus, daß die Performanz des Dateisystems im Vergleich zu FFS gerade durch das ständige Aufräumen der Platte egalisiert wird [SS95].

[58] Für eine klassische Publikation zu dem Thema „Einheitliche Namensräume“ sei auf [PW85] verwiesen. Eine Implementierung eines sehr integrierten Dateisystemnamensraums erfolgte bei dem Betriebssystem „Plan9“, das ursprünglich UNIX ablösen sollte [PPT+00].

hinreichend darum zu kümmern, viele kleine Dateien mangels Performanz niemals als einzelne Entitäten im Dateisystem zu speichern.

Blockgrenzen als natürliche Dateigrenzen

Dateigrenzen an Blockgrenzen auszurichten, zieht eine Reihe von Konsequenzen nach sich:

1. Es minimiert die Anzahl von Blöcken, auf die sich eine Datei verteilt. Dies zahlt sich besonders für größere Dateien aus, wenn die Referenzlokalität nicht gegeben ist.

2. Es verschwendet Speicherplatz und Pufferplatz für jeden Block, der nicht völlig durch die Datei benutzt wird.

3. Es verschwendet I/O-Bandbreite mit jedem Zugriff auf das Medium, wenn der Block nicht zur Gänze durch die Datei benutzt wird. Dies gilt auch, wenn die Lokalitätsreferenz gegeben ist.

4. Es erhöht die durchschnittliche Anzahl an Blöcken, die notwendigerweise gelesen werden müssen, um auf jede Datei in einem Verzeichnis zuzugreifen.

5. Es resultiert in einfachem Programmcode.

Der einfache Programmcode ist dabei einfach eine Folge dessen, daß keine Schicht implementiert werden muß, um die funktionalen Einheiten des Festplattencontrollers und des Puffercaches mit seinen Algorithmen von den funktionalen Einheiten der Speicherplatzallokation zu trennen.

Die einfachste und naheliegenste Lösung, viele kleine Dateien effizient in einem Verzeichnis zu verwalten, ist die Organisation mithilfe eines balancierten Baums. Dadurch führt man anstelle von statischen Konstruktionen, die viele Sonderfälle zu berücksichtigen haben, eine dynamische Ordnung der Dateien innerhalb von Knoten ein: Die Dateien werden je nach Bedarf in den Knoten des Baums zusammengehalten. Der Preis dafür ist jedoch hoch, da das Dateisystem nun die Aufgabe übernehmen muß, Daten in Knoten zu packen, sowie den Baum samt seiner Knoten auf die Blöcken des Mediums abzubilden.

Durch die Aufgabe von Blockgrenzen als natürliche Begrenzer von Dateien unterscheidet sich ReiserFS fundamental von anderen Dateisystemen. Dateien und Dateinamen werden in einem einzigen balancierten Baum gespeichert. Dadurch, daß die Blockgrenzen nicht mehr maßgebend sind, werden kleine Dateien, Verzeichniseinträge, Inodes und die Enden von Dateien –sogenannte „Tails“ (engl. „tail“) in der Sprache von Hans Reiser– sehr viel effektiver zusammengepackt. Darüber hinaus werden Inodes nicht mehr statisch allokiert, so daß kein unnötiger Speicherplatz verloren geht. Durch die Anordnung aller Dateisystemobjekte in einem Baum gibt es zudem die Möglichkeit, die Anordnung der Dateien auf dem Medium durch die Wahl der Ordnung innerhalb des Baums direkt zu beeinflussen. Damit kann man beispielsweise versuchen, maximale Referenzlokalität zu erzeugen. Große Dateien bzw. ihre

Datenblöcke werden ähnlich einem „BLOB“ außerhalb des Baums gespeichert.[59] Sie sind dadurch vom Baum isoliert und werden durch die Balancierungsalgorithmen nicht beeinflußt.

Jedes der bisher erwähnten anderen Dateisysteme hält die Blockgrenzen für Dateien aufrecht. Keines zieht kleine Dateien zur effektiven Speicherung zusammen (siehe beispielsweise [SDH+96] für XFS, [GK97] für eine verbesserte Version des FFS[60] und [Cus94] für das Dateisystem von Microsoft NT). Darüber hinaus speichern sie auch nur die Dateinamen im Baum (siehe bespielsweise [SDH+96] für XFS). Fragmente stellen dabei auch keine wirkliche Ausnahme dar, denn bei ihnen wird auch nur bereits statisch belegter Platz von anderen Dateien benutzt. Der Vorteil des Konzepts beim Reiser-Dateisystem dagegen ist, daß es die Granularitäten jedes an der Dateiverwaltung beteiligten Subsystems soweit wie möglich unbeeinflußt läßt. Semantik in Form von Dateien, Packen und Entpacken von Knoten und Blöcken, Zwischenspeichern und Vorauslesen, Hardware und Speicherseiten, sie alle haben verschiedene optimale Granularitäten. ReiserFS zwängt diesen keine Datei- und Blockgrenzen auf. Darin liegt die eigentliche Innovation des neuen Dateisystems.

Vereinigung von kleinen Dateien

Um kleine Dateien effektiv zu behandeln, gibt es prinzipiell zwei Möglichkeiten. Entweder man schreibt ein Dateisystem, das selbst dazu in der Lage ist, oder man verlagert die Aufgabe an eine darüberliegende Schicht. Das Verlagern hat die folgenden Nachteile:

- Benutzungs- und Programmcodekomplexität
  Ein auch in der Praxis einheitlicher Namensraum für große und kleine Dateisystemobjekte erhöht die Aussagekraft der Komponenten des Betriebssystems und vermindert Programmierkosten. Mehrere Schichten dagegen sind vom Aufwand und den Kosten ineffektiv. ReiserFS ändert zum Erreichen seines Ziels nichts an der Dateisystemschnittstelle des Kernels, sondern überläßt es den Verzeichnissen, kleine Objekte bei Bedarf zusammenzufassen.

- Performanz
  In der Regel kostet die Einführung zusätzlicher Schichten in einem Dateisystem mehr Rechenzeit. ReiserFS vermeidet diesen unnötigen Verlust dadurch, daß es die effektive Behandlung von kleinen Dateien nicht an höhere Schichten weiterdelegiert, sondern selbst vornimmt.

Die Autoren von [GK97] verweisen darauf, daß über 80% der Dateien eines durchschnittlichen UNIX-Systems kleiner als 10 KB sind. Zudem zeigen sie auch, daß kleine Dateien die Performanz des Systems nicht proportional zu ihrer gesamtem

[59] Ein „BLOB“ (engl. „binary large object“) ist eine Datei, die durch einen B+-Baum indiziert wird (siehe [SH99] für eine Definition). Dieser heißt in diesem Zusammenhang auch „Positionsbaum“.

[60] Diese verbesserte Version von FFS erhöht die Lokalitätsreferenz für kleine Dateien dadurch, daß ihre Inodes in den sie referenzierenden Verzeichnissen eingebettet werden, und ihre Datenblöcke explizit in der Nähe der Verzeichnisse in benachbarten Blöcken gruppiert werden. Damit erhöht sich der Durchsatz insbesondere bei kleinen Dateien um bis zu einem Faktor Sieben bei Lese- und Schreiboperationen.

Größe sondern eher zur ihrer gesamten Anzahl bestimmen. Diese 80% der Dateien sind allerdings um einen Faktor 10 bis 100 größer als die Dateien, auf die ReiserFS optimiert ist. Das ist gerade der Beginn des Bereichs für den bespielsweise Ext2 und FFS konstruiert sind und dementsprechend sehr effektiv sind. Es ist gleichzeitig der Bereich, in dem gerade die Schwächen des Reiser-Dateisystems liegen. Aus dieser Statistik kann man aber nach der Meinung von Hans Reiser nicht ablesen, daß es keinen Bedarf an Dateisystemen gibt, die sehr effektiv mit kleinen Dateien umgehen können. Er ist der Auffassung, daß ein Dateisystem, das performant mit kleinen Dateien umgehen kann, das Benutzungverhalten der Applikationsprogrammierer und damit auch der Benutzer des Systems ändern wird.

Abschließend sei bemerkt, daß nach Meinung der Entwickler von ReiserFS die Entwicklung von inkompatiblen Schichten, die Dateien über dem Dateisystem zusammenfassen, teurer, langsamer, weniger ausdruckstark und weniger platzsparend ist, als die Verwendung von balancierten Bäumen im Dateisystem einzuführen. Der Preis dafür ist die deutlich erhöhte Komplexität des Programmcodes. Vergleichbare Programmcodeabschnitte sind gegenüber den entsprechenden Stellen des Ext2-Programmcodes um fast das Zehnfache länger.

Definition des Baums

Wie bereits erwähnt, benutzt ReiserFS einen B*-Baum, um alle Datensystemobjekte darin zu speichern. Eine allgemeine Einführung von balancierten Bäumen als Zugriffstruktur haben wir im Abschnitt 3.6.2 gegeben. Normalerweise benutzt man beim Design eines balancierten Baums eine Menge von abstrakten Schlüsseln als Grundlage, welche bei der Anwendung von der konkreten Applikation geliefert werden muß. Der Zweck und Nutzen des Baums für die Applikation besteht dann darin, optimale Suchalgorithmen für die Schlüsselmenge der Applikation zur Verfügung zu stellen (siehe Abb. 3.34). Bei ReiserFS dient der Baum dazu, die Referenzlokalität der zusammengehörenden Daten auf dem Medium zu erhöhen und den Speicherplatz effizient zu verwalten. Die Schlüssel sind dementsprechend definiert, die Algorithmen dafür zu optimieren, daß sie die verschiedenen Dateisystemobjekte sinnvoll in die Knoten des Baums packen. Aus diesem Grund stellen die Schlüssel im Prinzip die nicht vorhandenen Inode-Nummern im Dateisystem dar. Die klassische Abbildung Inode-Nummer zu Dateiblöcken wird dabei durch die Abbildung Schlüssel zu internen Knoten des Baums ersetzt. Die Datenstruktur der Schlüssel ist größer als die Datenstruktur typischer Inode-Nummern, aber man muß auch weniger davon im Inode-Cache zwischenspeichern, da in einem Knoten mehrere Datensätze gespeichert werden können.

Der Baum selbst besteht aus drei Knotentypen: Interne Knoten, formatierte Knoten und unformatierte Knoten. Ihre Beziehung ist in Abb. 3.39 dargestellt. Wie die Namensgebung der Knoten suggeriert, speichern sie verschiedene Datensatzformen. Der Inhalt der internen und formatierten Knoten wird in der Reihenfolge ihrer Schlüssel gespeichert. Unformatierte Knoten enthalten keine Schlüssel.

Die unterste Ebene des Baums besteht aus den unformatierten Knoten, die Ebene darüber aus den formatierten, alle Ebenen darüber aus internen Knoten. Ganz oben befindet sich natürlich der Wurzelknoten des Baums. Die Tiefe des Baums wird bei

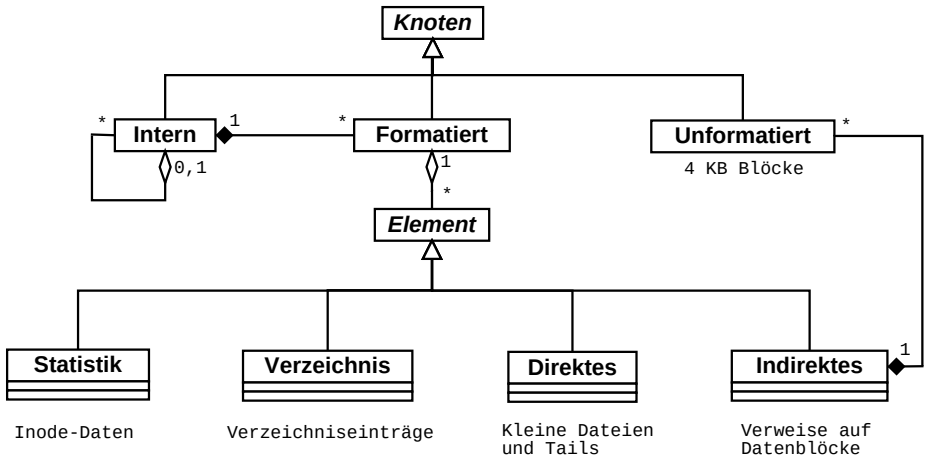

Abbildung 3.39: Vereinfachte Darstellung der Objektbeziehungen von Reiser

Bedarf erhöht, in dem eine neue Wurzel an der Spitze des Baums eingefügt wird. Die Weglänge von der Wurzel zu allen formatierten Knoten ist gleich, genauso sind die Wege zu allen unformatierten gleich lang und nur um einen Schritt länger als zu den formatierten Knoten. Dies ist einer der Gründe dafür, daß diese Zugriffstruktur effizient bei der Suche funktioniert.

Interne Knoten zeigen auf Subbäume und dienen zur reinen Strukturerhaltung und Suche nach den anderen Knoten. Die Zeiger auf die Subbäume werden durch die zugehörigen Schlüssel unterschieden. Der Schlüssel, der einem Zeiger auf den Unterbaum vorangeht, ist ein Duplikat des ersten Schlüssels in dem ersten formatierten Knoten dieses Unterbaums. Interne Knoten existieren nur um zu bestimmen, welcher formatierte Knoten das Objekt, das zu dem gesuchten Schlüssel korrespondiert enthält. ReiserFS beginnt an der Wurzel des Baums und verzweigt in die entsprechenden internen Knoten, bis es den gewünschten formatierten Knoten findet, der das gesuchte Objekt enthält.

Formatierte Knoten bestehen aus sogenannten „Elementen“ (engl. „item“). Es gibt vier verschiedene Elementtypen: Direkte Elemente, indirekte Elemente, Verzeichniselemente und Statistikelemente. Jedes Element besitzt einen innerhalb des Dateisystems eindeutigen Schlüssel. Dieser Schlüsel wird zur Sortierung und zum Auffinden der Elemente benötigt. Direkte Elemente enthalten kleine Dateien oder Dateienden. Indirekte Elemente bestehen aus Zeigern zu unformatierten Knoten. Alle Teile einer Datei –außer ihrem Ende– werden in unformatierten Knoten gespeichert. Verzeichniselemente enthalten den Schlüssel des ersten Verzeichniseintrags, gefolgt von einer Reihe von Verzeichniseinträgen. Statistikelemente enthalten die Metadateninformationen, die in klassischen Dateisystemen in der Inode gespeichert werden. Je nach Konfiguration von ReiserFS werden Statistikelemente als eigenständiges Elemente behandelt oder in Verzeichniseinträge eingebettet.[61]. Es ist niemals mehr als ein Element desselben Typs und des gleichen Objekts in einem Knoten

[61] Es wird derzeit noch untersucht, welche der beiden Strategien die Leseperformanz von ReiserFS erhöht. Derzeit sind es eigenständige Objekte.

gespeichert[62].

Unformatierte Datensätze stellen einen Datenblock zu den Daten einer Datei dar und entsprechend damit in gewisser Weise den Datenblöcken bei klassischen Dateisystemen. Sie sind daher auch ohne weitere Struktur. Der Reiserbaum verhält sich ihnen gegenüber wie ein Positionsbaum zur Indizierung eines BLOBs.

Eine Datei besteht aus einem Statistikelement, einer Menge von indirekten Elementen und maximal zwei direkten Elementen. Die direkten Elemente speichern die Enden der Datei. Wenn zwei direkte Elemente benutzt werden, dann bedeutet das, daß die Enden einer Datei über zwei Knoten verteilt sind. Wenn ein Ende wächst und größer wird, als die maximale Größe einer Datei, die in einen formatierten Knoten paßt, aber kleiner ist als die Größe eines unformatierten Knotens, dann wird es in einen unformatierten Knoten kopiert. Ein Zeiger darauf zusammen mit einem Zähler des verwendeten Platzes werden zur Referenzierung innerhalb eines indirekten Elements eingetragen.

Verzeichnisse bestehen aus einem Statistikelement und aus einer Menge von Verzeichniselementen. Die Verzeichniselemente bestehen aus einer Menge von Verzeichniseinträgen. Ein Verzeichniseintrag speichert den Namen und den Schlüssel der durch den Namen benannten Datei.

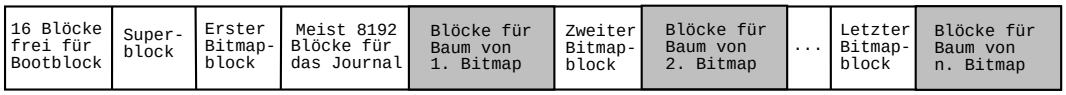

Abbildung 3.40: Das Blocklayout von ReiserFS

Datenstrukturen

Anhand von Abb. 3.40 werden Gemeinsamkeiten und Unterschiede zu den bisher besprochenen klassischen Dateisystemen noch einmal ganz deutlich. Wie jedes Dateisystem unter UNIX läßt ReiserFS am Anfang der Partition Platz für den Bootblock. Es reserviert sogar 16 Blöcke. Danach speichert es die globalen Informationen des Dateisystems in seinem Superblock ab. Zum Erfassen von durch den Baum belegten Blöcken auf der Platte benutzt ReiserFS mehrere Bitmaps, die es in regelmäßigen Abständen auf der Platte verteilt. Sie sind jeweils einen Block groß und können demnach 32768 Blöcke erfassen. Auf den ersten Bitmapblock folgen als wesentliche Neuerung die Blöcke, aus denen das Journal besteht. Der andere wesentliche Unterschied zu der klassischen Aufteilung der Partition besteht aus den grau markierten Bereichen. Diese stellen die Blöcke dar, die ReiserFS für die Bildung seines Baums benutzen kann. Demnach ist die Platte gerade nicht, wie das Bild fälschlicherweise suggerieren könnte, in statische Zonen aufgeteilt.

Der Superblock „reiserfs_super_block“ ist in Abb. 3.41 dargestellt[63]. Einige Felder

[62] Es gibt keinen Grund, warum man diese ansonsten nicht zusammenführen sollte.

[63] Es gibt mittlerweise mehrere Formate. Wir beschreiben die neuste Version 3.6.x, bei der im Programmcode „REISERFS_VERSION_2“ gesetzt ist. Unter anderem gibt es bei dieser Version

```
struct reiserfs_super_block
{
  __u32 s_block_count;
  __u32 s_free_blocks;         /*free blocks count*/
  __u32 s_root_block;          /*root block number*/
  __u32 s_journal_block;       /*journal block number*/
  __u32 s_journal_dev;         /*journal device number*/
  __u32 s_orig_journal_size;
  __u32 s_journal_trans_max;      /*max number of blocks in a transaction.*/
  __u32 s_journal_block_count;    /*total size of the journal. changes over time*/
  __u32 s_journal_max_batch;      /*max number of blocks to batch into a trans*/
  __u32 s_journal_max_commit_age; /*in seconds, how old can an async commit be*/
  __u32 s_journal_max_trans_age;  /*in seconds, how old can a transaction be*/
  __u16 s_blocksize;              /*block size*/
  __u16 s_oid_maxsize; /*max size of object id array, see get_objectid() comment*/
  __u16 s_oid_cursize; /*current size of object id array*/
  __u16 s_state;       /*valid or error*/
  char s_magic[12];    /*reiserfs magic string indicating file system is rfs*/
  __u32 s_hash_function_code; /*hash function used to sort names in a directory*/
  __u16 s_tree_height;         /*height of disk tree*/
  __u16 s_bmap_nr;             /*amount of bitmap blocks to address each block*/
  __u16 s_version;
  __u16 s_reserved;
  __u32 s_inode_generation;
  char s_unused[124] ;  /*zero filled by mkreiserfs*/
} __attribute__ ((__packed__));
```

Abbildung 3.41: Der Superblock von ReiserFS als Struktur „reiserfs_super_block“ aus „include/linux/reiser_fs_sb.h“

kennen wir schon aus den klassischen Dateisystemen. Angaben über die insgesamte Menge an Blöcken „s_block_count“ und an freien Blöcken „s_free_blocks“, sowie ein Vermerk „s_state“ über den Dateisystemzustand werden üblicherweise im Superblock gespeichert. Natürlich fehlen aber Zonenbeschreibungen oder ähnliches, da ReiserFS keine Zonen verwendet. Das für uns hier zunächst wichtige Feld ist „s_root_block“. Es gibt die Blocknummer des Blocks an, der momentan den Wurzelknoten des Baums beinhaltet. Muß eine neue Wurzel in den Baum eingefügt werden, dann wird der Eintrag im Superblock aktualisiert. Die momentane Höhe des Baums wird in „s_tree_height“ vermerkt. Sie kann die im Programmcode festgelegte Grenze von Fünf nicht überschreiten. Hinzugekommen sind auch Felder, die das Journal beschreiben. Sie definieren die Größe des Journals „s_journal_trans_max“, die maximale Blockgröße einer Transaktion „s_journal_trans_max“, sowie die maximale Anzahl von Blöcken „s_journal_max_batch“, die auf einmal einer Transaktion hinzugefügt werden können. Das Journal muß sich nicht notwendigerweise auf demselben Gerät befinden wie die Partition. Daher wird das Gerät, auf dem sich das Journal

Erweiterungen bei Größenangaben in der Inode. Im Statistikelement ist das Größenfeld einer Datei statt 32 Bit nun 64 Bit lang, und die Anzahl harter Verweise wird statt mit 16 Bit durch 32 Bit beschrieben.

befindet, in „s_journal_dev“ und die Blocknummer des Journals in „s_journal_block“ gespeichert. Im Superblock wird zudem die Hashfunktion gespeichert, mit der Namenseinträge zuletzt in Verzeichnissen sortiert worden sind.

Ein Diskblock, der bereits vom Baum beansprucht wird, gehört entweder zu einem unformatierten Knoten oder er benutzt einen Header zur Beschreibung seines Inhalts. Diese Struktur „block_head“ aus „include/linux/reiserfs_fs.h“ ist in Abb. 3.42 dargestellt. Die Angaben des Blockkopfes, insbesondere die Tiefe des Blocks inner-

```
struct block_head {
  __u16 blk_level;        /*Level of a block in the tree.*/
  __u16 blk_nr_item;      /*Number of keys/items in a block.*/
  __u16 blk_free_space;   /*Block free space in bytes.*/
  __u16 blk_reserved;
  struct key  blk_right_delim_key; /*kept only for compatibility*/
};
```

Abbildung 3.42: Die Struktur „block_head“ eines internen oder formatierten Knotens aus „include/linux/reiserfs_fs.h“

halb des Baums „blk_level“ und die Anzahl der freien Byte „blk_free_space“, sind für die Balancierungsalgorithmen wichtig.

Genauso wird jedes Element innerhalb des Baums mit einer Elementbeschreibung versehen. Ihre Struktur „item_head“ aus „include/linux/reiserfs_fs.h“ ist in Abb. 3.43 wiedergegeben. Das wichtigste Feld ist der Schlüssel „ih_key“, auf den wir weiter

```
struct item_head
{
  struct key ih_key; /*Everything in tree is found by searching on its key.*/
  union {
    __u16 ih_free_space_reserved; /*The free space in the last unformatted node of
                                    an indirect item if this is an indirect item.*/
    __u16 ih_entry_count; /*Iff this is a directory item, this field equals the
                            number of directory entries in the directory item.*/
  } __attribute__ ((__packed__)) u;
  __u16 ih_item_len;           /*total size of the item body*/
  __u16 ih_item_location;      /*an offset to the item body within the block*/
  __u16 ih_version;            /*0 for all old items, 2 for new one*/
} __attribute__ ((__packed__));
```

Abbildung 3.43: Die Struktur „item_head“ aus „include/linux/reiserfs_fs.h“

unten genau eingehen werden. In „ih_item_len“ wird vermerkt, wie lang der zu dem Element gehörende Eintrag im Knoten ist und in dem Offset „ih_item_location“, wo er im Knoten zu finden ist.

Ein Zeiger auf einen Knoten, der in einem Block auf dem Medium gespeichert, ist hat die Struktur

```
struct disk_child {
  __u32 dc_block_number; /*Disk child's block number.*/
  __u16 dc_size;         /*Disk child's used space.  */
  __u16 dc_reserved;
};
```

aus „include/linux/reiserfs_fs.h“. Er speichert die Blocknummer „dc_block_number“, in der der Knoten auf dem Medium gespeichert ist. In „dc_size“ wird zudem vermerkt, wieviel Byte der Knoten im Block bereits belegt.

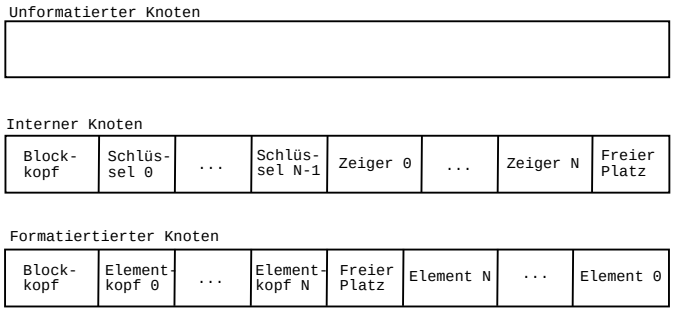

Abbildung 3.44: Die drei Formate der Knotentypen auf der Platte

Das Zusammenspiel der eben beschriebenen Strukturen wird am besten an Abb. 3.44 deutlich. Ein unformatierter Datensatz hat keine weitere Struktur und besteht einfach aus einem Speicherplatz von 4 KB. Ein interner Datensatz besteht aus einer Blockbeschreibung, gefolgt von N Schlüsseln, $N + 1$ eben beschriebenen Zeigern und freiem Speicherplatz am Ende des Blocks. Ein formatierter Datensatz besteht aus der Blockbeschreibung zu Beginn, gefolgt von den Elementbeschreibungen samt den Elementen am Ende des Blocks, die sie beschreiben. Die Anordnung mit dem Speicherplatz in der Mitte und der invertieren Reihenfolge der Elemente ist derart gewählt, das ein Einfügen ohne Umkopieren oder Änderungen an den bestehenden Strukturen möglich ist.

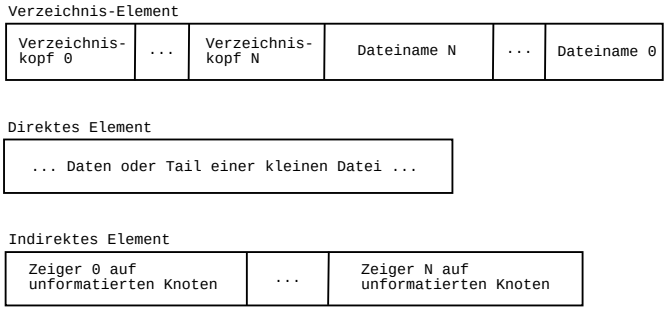

Abbildung 3.45: Die Elementstrukturen auf der Platte. Das Statistik-Element ist nicht gezeigt. Es sieht aus wie ein direktes Element, dessen Daten aus der Struktur „stat_data“ bestehen.

Die Daten, die bei klassischen Dateisystemen in der Inode notiert werden, speichert ReiserFS in dem Statistikelement ab. Seine Struktur „stat_data“ aus „include/linux/reiserfs_fs.h“ ist in Abb. 3.46 wiedergegeben. Wie wir leicht erkennen, sind

```
struct stat_data {
  __u16 sd_mode;     /*file type, permissions*/
  __u16 sd_reserved;
  __u32 sd_nlink;    /*number of hard links*/
  __u64 sd_size;     /*file size*/
  __u32 sd_uid;      /*owner*/
  __u32 sd_gid;      /*group*/
  __u32 sd_atime;    /*time of last access*/
  __u32 sd_mtime;    /*time file was last modified */
  __u32 sd_ctime;    /*time inode (stat data) was last changed
                       (except changes to sd_atime and sd_mtime)*/
  __u32 sd_blocks;
  union {
    __u32 sd_rdev;
    __u32 sd_generation;
  } __attribute__ ((__packed__)) u;
} __attribute__ ((__packed__));
```

Abbildung 3.46: Die Struktur „stat_data“ aus „include/linux/reiserfs_fs.h“

alle Felder der S5FS-Inode enthalten.[64] Die anderen Elemente sind in Abb. 3.45 dargestellt. Ein direktes Element ist nicht weiter strukturiert, sondern enthält lediglich die Daten einer kleinen Datei oder eines Tails. Ein indirektes Element speichert eine Menge von Zeigern auf unformatierte Blöcke. Diese Zeiger sind momentan 32 Bit Werte und speichern die Blocknummern von unformatierten Datensätzen. Ein Verzeichniselement besteht aus einer Menge von Verzeichniseinträgen. Diese ordnen einen Verzeichniskopf –und damit einen Schlüssel– einem Namen zu. Ein Name kann dabei maximal 4032 Byte bei einer Blockgröße von 4 KB belegen. Auf die Verzeichnisköpfe kommen wir unten noch zu sprechen.

Balancierte Bäume und große Dateien

Es ist wohl offensichtlich, daß es mithilfe eines balancierten Baums gelingt, viele kleine Dateien und Dateienden zusammen in einem Knoten zu speichern, um sie damit auf einmal effektiv schreiben oder lesen zu können.

Für größere Dateien hat das Reiser-Dateisytem allerdings die folgenden architekturbedingten Schwächen:

- Wenn das Ende einer Datei oder eine vormals kleine Datei wächst und dabei groß genug wird, einen ganzen Knoten zu belegen, dann werden sie aus den formatierten Knoten genommen und zu unformatierten konvertiert. Das kostet natürlich Verarbeitungszeit. FFS hat ähnliche Performanzeinbußen, wenn Fragmente wachsen und kopiert werden müssen.

[64] Die Union ist für uns uninteressant. Das „sd_generation“ Feld wird nur für NFS benötigt.

- Ein Dateiende, das kleiner ist als ein Knoten, kann auf zwei formatierte Knoten verteilt werden. Dies erhöht die Lesezeit, wenn die Lokalitätsreferenz schlecht ist und diese beiden Knoten nicht benachbart liegen.

- Das Zusammenführung von Tails in einen Knoten trennt die Datei von ihrem Ende. Für Dateien in der Größe der Knotengröße, kann sich das signifikant in der sequentiellen Lesezeit bemerkbar machen.

- Wenn ein Byte zu einer Datei hinzugefügt wird, welche nicht die letzte Datei in einem formatierten Knoten ist, dann wird im Schnitt die Hälfte des Knotens im Hauptspeicher kopiert werden müssen. Wenn nun eine Datei viele derartige Schreibanforderungen stellt, dann wird die Performanz durch ständige Konstruktion des formatierten Datensatzes sinken, weil ReiserFS die Schreibanforderungen nicht zwischenspeichern kann. Allerdings sind die I/O-Funktionen der Standardbibliothek „libc“ in der Regel gepuffert.

Die einzige Optimierung, die ReiserFS für große Dateien vornimmt, ist die Blockgröße auf 4 KB zu setzen. Beispielsweise erfährt Ext2 einen Geschwindigkeitsgewinn von 20%, wenn die Blockgröße von 1 KB auf 4 KB erhöht wird. Daher besteht zwischen beiden Dateisystemen für große Dateien kein Unterschied in den Kosten, eine Datei um einen Block zu erweitern. ReiserFS muß dazu einen Zeiger im Baum auf einen unformatierten Knoten einfügen und diesen Datensatz schreiben. Ext2 muß im Adressfeld einer Inode eine Blocknummer hinzufügen und den Datenblock schreiben.

Für große Dateien besteht ein Vorteil darin, daß der Baum nicht mehr balanciert werden muß als der Baum, der durch einen dreifach indirekten Verweis bei auf Inodes basierenden Dateisystemen entsteht. Für kleine Dateien und Tails allerdings gibt es die erwähnten Performanzeinbußen durch das Balancieren des Baumes im Speicher. Bei jeder Balancierung überprüft ReiserFS die Belegungsdichte des Knotens und seiner beiden Nachbarn und stellt sicher, daß die drei Knoten nicht zu zwei Knoten zusammengeführt werden.[65]

Das Layout der Dateien

Es gibt vier Ebenen, auf denen das Layout der Dateien auf dem Medium beeinflußt und optimiert wird. Diese sind:

1. Die Abbildung zwischen logischen Blocknummern und physikalischem Ort auf dem Medium.

2. Die Zuweisung von Knoten zu logischen Blöcken.

3. Die Ordnung der Objekte innerhalb des Baums.

[65] Eine höhere Komprimierung soll in der Zukunft eventuell ein Defragmentierer, ein sogenannter „Cleaner“, übernehmen. Dieser könnte in regelmäßigen Abständen arbeiten, oder wenn das System nicht belastet ist, und dabei ähnlich dem Cleaner von LFS die Datenstruktur auf der Platte analysieren und optimieren.

4. Die Verteilung und Balancierung der Objekte innerhalb der Knoten, in die sie gepackt werden.

Die Abbildung auf physische Koordinaten geschieht auf dem Laufwerk oder im Blocktreiber. Darüber hinaus kann ein LVM oder ein RAID als Softwareschicht dazwischenliegen. ReiserFS kümmert sich nicht um Zylindergrenzen oder ähnliches, sondern minimiert den Abstand gemessen in logischen Blöcken von semantisch benachbarten Knoten. Bei der Zuordnung von Knoten des Baums zu Blöcken auf dem Medium, sucht ReiserFS in dem Bitmap, das die belegten Blöcke markiert, nach einem freien Block. Diesen findet es, indem es bei der Blocknummer des linken semantischen Nachbarn startet und dann in Richtung aufsteigender Blocknummern linear nach dem nächsten freien Block sucht.

Ordnung innerhalb des Baums

Im Prinzip hängt die sinnvolle Definition des Schlüssels und seiner Berechnung von dem erwarteten Benutzungsverhalten des Dateisystems ab.[66] Betrachten wir beispielsweise die Entscheidung, ob man Verzeichniseinträge am Anfang der Partition zusammenpacken soll oder besser in der Nähe der Dateien, die sie benennen, plazieren soll. Für große und damit wenige Dateien ist ersteres sinnvoll, denn Systeme mit derartigen Benutzungsmustern können die Verzeichniseinträge meist gänzlich zwischenspeichern. Für viele kleine Dateien sollte der Namenseintrag besser in der Nähe der Datei stehen. Ähnlich verhält es sich mit den Statistikdaten. Für große Dateien sollten sie getrennt von den Daten der Datei entweder zusammen mit den anderen Statistikdaten der Dateien aus dem gleichen Verzeichnis oder beim Verzeichniseintrag selbst stehen. Für kleine Dateien sollten die Statistikdaten nahe bei den Daten der Datei gespeichert werden.

Jedes Element besitzt einen eindeutigen Schlüssel. Die Struktur „key“ ist in Abb. 3.47 gezeigt. Ein Schlüssel besteht aus der Lokalitätsnummer „k_dir_id, der Objektnummer

```
struct key {
  __u32 k_dir_id;    /*packing locality: by default parent directory object id*/
  __u32 k_objectid;  /*object identifier*/
  __u32 k_offset;
  __u32 k_uniqueness;
} __attribute__ ((__packed__));
```

Abbildung 3.47: Die Struktur „key“ aus „include/linux/reiserfs_fs.h“

„k_objectid“, einem Offset „k_offset“ und einem Eindeutigkeitsfeld „k_uniqueness“. Standardmäßig ist die Lokalitätsnummer durch die Objektnummer des Vaterverzeichnisses gegeben. Die Objektnummer identifiziert die Datei eindeutig und wird bei der Erzeugung auf die erste nicht benutzte Objektnummer des Dateisystems gesetzt[67]. Diese Wahl unterstützt die Möglichkeit, daß aufeinanderfolgende Objekterzeugungen in einem Verzeichnis benachbart plaziert werden. Das ist für viele

[66] In zukünftigen Versionen von ReiserFS wird man daher den Schlüsselalgorithmus selbst wählen können.

[67] Die Folge der benutzten und unbenutzten Objektnummern wird dabei kodiert durch „s_oid_maxsize“ und „s_oid_cursize“ im Superblock vermerkt.

Benutzungsmuster wünschenswert. Bei direkten und indirekten Elementen gibt der Offset den Ort innerhalb des logischen Objekts an, der das erste Byte des dazugehörigen Elements darstellt. Bei Verzeichniselementen besteht der Offset aus dem Hashwert des Namens des ersten Eintrags. Das Eindeutigkeitsfeld unterscheidet Verzeichniseinträge mit gleichem Offset.[68] Darüber hinaus kennzeichnet es den Typ des Elements. Für den Linksaußen eines Knotens gibt es zudem an, ob er Vorgänger von demselben Typ ist. Diese Information wird zur Ausbalancierung des Baums herangezogen. Für Verzeichnis-Elemente wird der Schlüssel des ersten Eintrags benutzt. ReiserFS benutzt die beschriebene Struktur des Schlüssels dazu, um Dateien, die aus demselben Verzeichnis stammen, zusammen abzuspeichern. Zusätzlich legt es Verzeichniseinträge, die zu demselben Verzeichnis gehören, zusammen mit den Statistikdaten des Verzeichnisses ab. Allerdings werden dabei nicht kleine Dateien und deren Statistikdaten zusammem mit dem Vaterverzeichnis abgespeichert.

Abb. 3.48 zeigt die Struktur „reiserfs_de_head“ aus „include/linux/reiserfs_fs.h“ für die Verzeichnisköpfe (siehe auch Abb. 3.45). In dem Feld „deh_offset“ speichert

```
struct reiserfs_de_head
{
  __u32 deh_offset;   /*third component of the directory entry key*/
  __u32 deh_dir_id;   /*objectid of the parent directory of the object, that is
                        referenced by directory entry*/
  __u32 deh_objectid; /*objectid of the object referenced by directory entry*/
  __u16 deh_location; /*offset of name in the whole item*/
  __u16 deh_state;    /*whether 1) entry contains stat data (for future),
                              and 2) whether entry is hidden (unlinked)*/
} __attribute__ ((__packed__));
```

Abbildung 3.48: Die Struktur „reiserfs_de_head“ aus „include/linux/reiserfs_fs.h“

ReiserFS den berechneten Hashwert über den zugehörigen Dateinamen. Dabei werden 23 Bit für den Hashwert benutzt und 7 Bit für den Überlaufspeicher. Das bedeutet, daß pro Verzeichnis nicht mehr als 127 Namen mit gleichem Hashwert gespeichert werden können. Es gibt momentan drei Hashfunktionen, die man zur Formatierungszeit oder sogar auch zur Mountzeit wählen kann. Die „Rupasov“ Hashfunktion versucht die lexikographische Ordnung von aufeinanderfolgenden Namen –soweit wie möglich– beizubehalten. Dadurch steigt bei ihrer Wahl die Kollisionsgefahr. Es ist nicht zu empfehlen, diese zu benutzen. Die „Tea“ Hashunktion erhöht die Anzahl zufälliger Bits und bewirkt dadurch deutlich weniger Kollisionen. Allerdings ist sie relativ rechenintensiv. Sie ist für große Verzeichnisse mit einer großen Anzahl von Dateien gedacht. Standardmäßig nutzt ReiserFS die „R5“ Hashfunktion, die eine Erweiterung der „Rupasov“ Funktion darstellt, die weniger Kollisionen produziert. Das Feld „deh_dir_id“ speichert die Objektnummer des Vaterverzeichnisses und „deh_objectid“ die Objektnummer, des zu dem Namen gehörenden Objekts. Damit ist ReiserFS ist der Lage, aus den Angaben im Verzeichniskopf den Schlüssel des referenzierten Objekts zu berechnen. Innerhalb eines Verzeichnisses werden die

[68] Es wird in diesem Zusammenhang auch als „Generationenzähler“ bezeichnet.

Einträge mit aufsteigendem Offset sortiert. Bei der Suche nach dem Schlüssel zu einem Namenseintrag verwendet ReiserFS binäres Suchen. Im Unterschied zu dem linearem Suchen der klassischen Systeme skaliert dieses logarithmisch mit der Eingabemenge. Damit ist ReiserFS im Unterschied zu den klassischen Systemen in der Lage, hunderttausende von Einträgen pro Verzeichnis zu verwalten, solange die Anzahl der Kollisionen und damit die Anzahl gleicher Hashwerte in einem Verzeichnis unter der erwähnten Grenze von 127 liegt.

Knotenbalancierung

Die Balancierung der internen und formatierten Knoten geschieht nach den folgenden Regeln, die in der Reihenfolge ihrer Priorität angegeben sind:

1. Minimierung der benutzten Knoten des Baumes.

2. Minimierung der Knoten, die durch die Optimierung betroffen werden.

3. Minimierung der Anzahl nicht zwischengespeicherter Knoten, die durch die Optimierung betroffen werden.

4. Wenn Daten zwischen formatierten Knoten verschoben werden müssen, dann wird die Anzahl der verschobenen Bytes maximiert.

Die vierte Regel geht davon aus, daß ein Einfügen von Daten ein Anzeichen dafür ist, daß weitere Änderungen folgen werden. Daher wird soviel wie möglich und wie mit den anderen Regeln verträglich an Platz in den Knoten freigemacht. Dieses extreme Prinzip wird nicht in den zeitkritischen internen Knoten angewandt.

Protokollierung von Metadaten

ReiserFS benutzt die in Abschnitt 3.6.2 erwähnte WAL-Strategie, um Metadatenveränderungen an dem Dateisystem im Journal zu protollieren. Auch Datenblöcke kleiner Dateien bzw. Tails werden dabei erfaßt, da Transaktionen an internen und formatierten Knoten im Logbuch vermerkt werden. Einzig eine Änderung innerhalb der unformatierten Blöcke wird vom Dateisystem nicht überwacht.

Weitere Details zu den Hintergründen und der Implementierung der Reiser-Version 3 beschreibt Hans Reiser in [Rei01b]. In [Rei01c] macht er einen Ausblick auf die Version 4 von Reiser. Diese wird Programmierern unter anderem erlauben, Plugins für Dateien, Verzeichnisse, Elemente, Hashfunktionen und sogar Schlüssel zu schreiben, die die Standardmethoden des jeweiligen Objekts überschreiben. Damit kann ein Programmierer das Dateisystem auf seine Bedürfnisse anpassen. Zudem wird ein weiterer Schritt in Richtung „Vereinheitlichung und Defragmentierung“ des Dateisystemnamensraums unternommen. Dazu wird ein neuer Systemaufruf „reiser4“ implementiert, der einen vereinheitlichten Zugriff auf Dateisystemobjekte ermöglichen soll. Zusätzlich ist der erwähnte Cleaner vorgesehen, der das Blocklayout im Hintergrund optimieren soll.

3.6.5 Konkrete Benutzung des Reiser-Dateisystems

Wir erzeugen in diesem Abschnitt eine Reiser-Partition und erstellen darin zu Demonstrationszwecken einige Dateien. Insbesondere interessiert uns dabei, wie die Reiser-Strukturen danach auf der Partition aussehen.

Wie jedes Dateisystem unter LINUX liefert auch ReiserFS eine Reihe von Programmen[69], mit denen man mit Administratorrechten das Dateisystem erzeugen und verwalten kann. Wir benutzen

```
lisa:~ # mkreiserfs -v 2 /dev/hda8
```

und formatieren dadurch die Partition „/dev/hda8“ mit der neuen Version von Reiser.[70] Um Dateien darin zu erzeugen, fordern wir den Kernel mit dem Systembefehl „mount“ durch

```
lisa:~ # mount /dev/hda8 /tmp/test/ -t reiserfs
```

auf, in das bestehende Dateisystem an die Stelle „/tmp/test“ die formatierte Partition einzuhängen. Wie das VFS dazu vorgeht, haben wir in Abschnitt 3.3.2 besprochen. Eine wichtige Option beim Einhängen ist die Option „notail“.[71] Mit dieser teilt man ReiserFS mit, daß es darauf verzichten soll, Tails oder kleine Dateien innerhalb von direkten Elementen zu speichern. Der Vorteil ist ein signifikanter Geschwindigkeitsgewinn beim Speichern von größeren Dateien, weil das ständige Balancieren und Umkopieren der formatierten Knoten durch Änderungen an direkten Elementen entfällt. Der Nachteil ist natürlich die Erhöhung der internen Fragmentierung. Nach dem Einhängen erzeugen wir innerhalb des Verzeichnisses „/tmp/test“ mit dem Befehl „dd“ folgende Dateien:

```
-rw-r--r--    1 root     root          20k Dec  2 10:45 grossedatei1
-rw-r--r--    1 root     root         3.0k Dec  2 10:45 kleinedatei1
-rw-r--r--    1 root     root         1.0k Dec  2 10:45 kleinedatei2
-rw-r--r--    1 root     root         5.0k Dec  2 10:45 kleinedatei3
```

Wir benutzen das Programm „debugreiserfs“, das Informationen über eine Reiser-Partition liefert. Unter anderem kann man den Superblock und die Bitmaps mit der Option „-m“ und bestimmte Blockinhalte mit der Option „-b“ anzeigen lassen.[72] Wir lesen also zunächst die Bitmaps aus und erhalten:

```
lisa:~ # debugreiserfs /dev/hda8 -m
Super block of format 3.6 found on the 0x3 in block 16
Block count 128512
Blocksize 4096
Free blocks 120289
```

[69] Diese sind unter ftp://ftp.namesys.com/pub/reiserfsprogs/reiserfsprogs-3.x.0j.tar.gz erhältlich.

[70] Mit „man mkreiserfs“ enthält man eine Beschreibung weiterer Optionen.

[71] Beschreibungen für andere Optionen findet unter http://www.namesys.com/mount-options.html.

[72] Siehe „man debugreiserfs“ für eine rudimentäre Beschreibung weiterer Optionen.

```
Busy blocks (skipped 16, bitmaps - 4, journal blocks - 8193
1 super blocks, 9 data blocks
Root block 8213
Journal block (first) 18
Journal dev 0
Journal orig size 8192
Filesystem state VALID
Tree height 3
Hash function used to sort names: "r5"
Objectid map size 2, max 972
Version 2
Bitmap blocks are:
#0: block 17: used 8220, free 24548
#1: block 32768: used 1, free 32767
#2: block 65536: used 1, free 32767
#3: block 98304: used 1, free 30207
```

Damit stellen wir unter anderem fest, daß die Wurzel des Baumes momentan im Block 8213 zu finden ist. Diesen lassen wir uns mit der Option „-b“ anzeigen:

```
lisa:~ # debugreiserfs /dev/hda8 -b 8213
INTERNAL NODE (8213) has level=2, nr_items=1, free_space=4040 rdkey
PTR 0: [dc_number=8211, dc_size=3500]
KEY 0:          2 4 0x1 DRCT
PTR 1: [dc_number=8212, dc_size=2304]
```

Die Wurzel ist demnach ein interner Knoten und verweist auf zwei formatierte Knoten, die im Block 8211 und 8212 zu finden sind. Beide Blöcke lesen wir nacheinander aus. Das Ergebnis der beiden Befehle ist in Abb. 3.50 abgebildet. Diesen Angaben entnehmen wir das in Abb. 3.49 gezeigte Bild des Reiser-Baums innerhalb der Partition.

Wir betrachten beispielhaft die Datei „kleinedatei3“. Bei der Namensauflösung stellt das Dateisystem über den Hashwert „h4“ fest, daß die gesuchte Datei die Objektnummer Fünf und die Vaternummer Zwei besitzt. Es sucht nun im Baum nach diesem Schlüssel und verzeigt von der Wurzel in den rechten Knoten. Dort findet es drei Elementköpfe mit dem richtigen Schlüssel. Der erste verweist auf die Statistikdaten der Datei, der zweite auf ein indirektes und der dritte auf ein direktes Element. Das indirekte Element speichert einen Zeiger auf den Block 12853, der den Hauptteil der Datei enthält, während das direkte Element das Ende der Datei enthält.

Zur Illustration der Option „notail“ stellen wir uns nun vor, daß wir eine kleine Datei von 100 Byte hinzufügen. Normalerweise würden die Daten der Datei zusammen mit ihren Statistikdaten noch im rechten formatierten Knoten in einem Statistikelement und einem direkten Element gespeichert werden. Mit der Option „notail“ wird ReiserFS statt dessen ein Statistikelement und ein indirektes Element einfügen und die Daten der Datei in einem unformatierten Knoten speichern. In beiden Fällen wird natürlich der Verzeichniseintrag im linken Knoten aktualisiert.

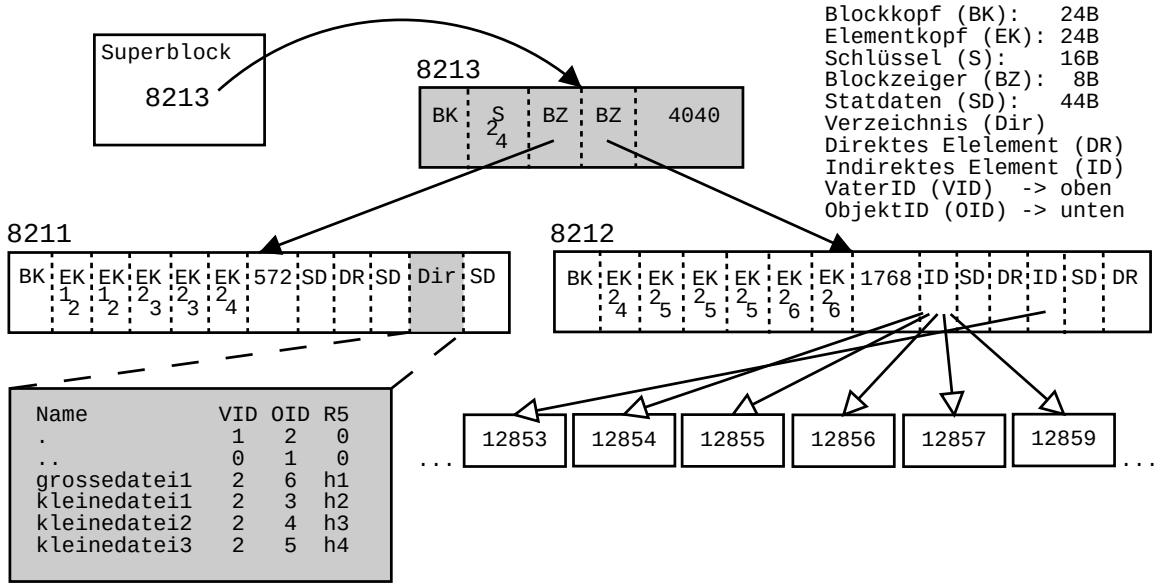

Abbildung 3.49: Der Baum von ReiserFS anhand eines konstruierten Beispiels

```
lisa:~ # debugreiserfs /dev/hda8 -b 8211
===================================================================
LEAF NODE (8211) contains level=1, nr_items=5, free_space=572 rdkey
-------------------------------------------------------------------------------
|###|type|ilen|f/sp| loc|fmt|fsck|                  key                       |
|   |    |    |e/cn|    |   |need|                                            |
-------------------------------------------------------------------------------
|  0|1 2 0x0 SD, len 44, entry count 0, fsck need 0, format new|
(NEW SD), mode drwxr-xr-x, size 176, nlink 2, mtime 12/02/2001 10:45:50 blocks 1
-------------------------------------------------------------------------------
|  1|1 2 0x1 DIR, len 176, entry count 6, fsck need 0, format old|
###: Name           length    Object key           Hash     Gen number
  0: ".             "(  1)                1 2           0    1, loc 168, state 4
  1: "..            "(  2)                0 1           0    2, loc 160, state 4
  2: "grossedatei1  "( 12)                2 6   234020480    0, loc 144, state 4
  3: "kleinedatei1  "( 12)                2 3  1989071104    0, loc 128, state 4
  4: "kleinedatei2  "( 12)                2 4  1989071232    0, loc 112, state 4
  5: "kleinedatei3  "( 12)                2 5  1989071360    0, loc 96, state 4
-------------------------------------------------------------------------------
|  2|2 3 0x0 SD, len 44, entry count 65535, fsck need 0, format new|
(NEW SD), mode -rw-r--r--, size 3072, nlink 1, mtime 12/02/2001 10:45:25 blocks 8
-------------------------------------------------------------------------------
|  3|2 3 0x1 DRCT, len 3072, entry count 65535, fsck need 0, format new|
-------------------------------------------------------------------------------
|  4|2 4 0x0 SD, len 44, entry count 65535, fsck need 0, format new|
(NEW SD), mode -rw-r--r--, size 1024, nlink 1, mtime 12/02/2001 10:45:29 blocks 8
===================================================================

lisa:~ # debugreiserfs /dev/hda8 -b 8212
===================================================================
LEAF NODE (8212) contains level=1, nr_items=6, free_space=1768 rdkey
-------------------------------------------------------------------------------
|###|type|ilen|f/sp| loc|fmt|fsck|                  key                       |
|   |    |    |e/cn|    |   |need|                                            |
-------------------------------------------------------------------------------
|  0|2 4 0x1 DRCT, len 1024, entry count 65535, fsck need 0, format new|
-------------------------------------------------------------------------------
|  1|2 5 0x0 SD, len 44, entry count 65535, fsck need 0, format new|
(NEW SD), mode -rw-r--r--, size 5120, nlink 1, mtime 12/02/2001 10:45:35 blocks 16
-------------------------------------------------------------------------------
|  2|2 5 0x1 IND, len 4, entry count 0, fsck need 0, format new|
1 pointers
[  12853]
-------------------------------------------------------------------------------
|  3|2 5 0x1001 DRCT, len 1024, entry count 65535, fsck need 0, format new|
-------------------------------------------------------------------------------
|  4|2 6 0x0 SD, len 44, entry count 65535, fsck need 0, format new|
(NEW SD), mode -rw-r--r--, size 20480, nlink 1, mtime 12/02/2001 10:45:50 blocks 40
-------------------------------------------------------------------------------
|  5|2 6 0x1 IND, len 20, entry count 0, fsck need 0, format new|
5 pointers
[  12854(5)]
===================================================================
```

Abbildung 3.50: Das Ergebnis der beiden ausgelesenen formatierten Knoten

4 Fragmentierung

4.1 Definitionen und Maße

Im letzten Kapitel haben wir bereits bei der Beschreibung verschiedener Dateisysteme von interner und externer Fragmentierung gesprochen. Beide Effekte treten im allgemeinen auf, wenn man Datenströme zur Verarbeitung in Blöcke bestimmter Größe einteilen muß, sogenanntes „Blocken“, und wenn diese Blöcke durch Direktzugriffe adressiert werden.

4.1.1 Interne Fragmentierung

Unter diesem Begriff versteht man den Verlust an Speicherplatz, der durch das Blocken der Datenströme zu Blöcken bestimmter Größe auftritt. Bei Dateisystemen ist in der Regel immer der letzte Datenblock einer Datei Quelle für interne Fragmentierung, weil dieser im Durchschnitt nur zur Hälfte gefüllt ist. Fragmente bei FFS oder Tails bei Reiser sind Versuche, die interne Fragmentierung eines Dateisystems zu verringern und damit den Anwendungen mehr Speicherplatz auf der Partition zur Verfügung zu stellen.

Bei der Berechnung des Verlusts an Speicherplatz stellt sich die Frage, welche Datenstrukturen des Dateisystems zu berücksichtigen sind. Beispielsweise stellen die Inode-Tabellen bei klassischen Dateisystemen durch ihre statische Allokation eine mögliche Speicherverschwendung dar. Eine andere Quelle sind die Blöcke, die bei Journaling-Dateisystemen für das Logbuch benötigt werden. Wir entscheiden uns in dieser Arbeit generell dafür, Metadaten oder Blöcke, die ein Dateisystem für besondere Zwecke –wie die Unterhaltung eines Logbuchs– benötigen, bei der Berechnung der internen Fragmentierung nicht zu berücksichtigen. Die Berechnung der internen Fragmentierung einer Datei ist somit gegeben durch

$$f_{\text{int}}^{\text{Dat}} = \text{Dateigröße in Bytes} / \left(\text{benötigte Blöcke} \times \text{Blockgröße in Bytes}\right) .$$

Ein Wert von Eins bedeutet demnach, daß die Datei optimal in die vorgesehenen Blöcke paßt. Allerdings sind wir an dem Verschnitt einer gesamten Partition interessiert. Dazu gibt es zwei Möglichkeiten der Berechnung. Entweder man nimmt an, daß die Dateien und ihre Blockbelegung auf der Platte statistisch unabhängig sind und berechnet den statistischen Mittelwert und bei Interesse die Standardabweichung des Mittelwerts oder man betrachtet die Gesamtheit alle Dateien:

$$f_{\text{int}} = \frac{\sum_i \text{Dateigröße der i. Datei}}{\text{Blockgröße} \times \sum_i \text{benötigte Blöcke der i. Datei}} , \tag{4.1}$$

wobei die beiden Größen wieder in Bytes ausgedrückt werden. Nach der Ungleichung $\frac{\sum a_i}{\sum b_i} \leq \frac{1}{n} \sum \frac{a_i}{b_i}$ für $a_i \leq b_i$ ist dieser Wert immer kleiner als der Mittelwert.

Man könnte sich auch vorstellen, bei der Berechnung immer nur den letzten Block einer Datei zu bewerten. In der oben angegeben Formel 4.1 wäre die benötigte Blockzahl immer Eins und die Größe der Datei die Anzahl Bytes im letzten Block. Im Mittel erwartet man damit einen Wert von 50% für die interne Fragmentierung. Bei dieser Vorgehensweise berücksichtigt man nicht die Größenverteilung der Dateien im System, da jede im System vorkommende Dateigröße modulo Blockgröße genommen wird.

4.1.2 Externe Fragmentierung

Während die interne Fragmentierung den Effekt des Blockens bewertet, gibt die externe Fragmentierung an, wie die Datenblöcke über den Direktzugriffsspeicher verteilt sind. Je nach Geschwindigkeit des Zugriffs im Unterschied zur Geschwindigkeit der Datenübertragung macht es sich deutlich in der Performanz bemerkbar, ob die Anordnung der Dateien mit ihren Datenblöcken für das jeweilige Gerät optimiert ist. Wenn zusammengehörende Datenblöcke einer Datei über das gesamte Medium verstreut liegen, muß eine Festplatte beim vollständigen sequentiellen Lesen der Datei sehr oft eine zeitaufwendige Positionierung des Kopfes vornehmen, so daß die Leseperformanz erheblich sinkt. In einem solchen Fall spricht man von einer fragmentierten Datei oder –im Fall von vielen fragmentierten Dateien– einer fragmentierten Partition.

Die Quantisierung der externen Fragmentierung ist deutlich schwieriger als die der internen. Man muß unterscheiden zwischen der abstrakten Bewertung des Blocklayouts auf der Platte und einem Zusammenhang einer Bewertung mit dem Zugriff und der Performanz des Dateisystems.

Die reine Bewertung hängt vor allem von der Beziehung der Daten –Metadaten und eigentliche Daten– und des konkreten Festplattenmodells ab. Bei heutigen Platten reicht es dabei, die logischen Blocknummern der Platte bzw. des Dateisystems zur Berechnung zu benutzen, wie wir aus dem Kapitel 2 –siehe beispielsweise Gleichung 2.1– wissen. Ein in der Literatur oft benutztes Vorgehen zur Fragmentierungsbewertung einer Datei zählt aufeinanderfolgende Blöcke einer Datei und teilt das Ergebnis durch die Anzahl der durch die Datei belegten Blöcke.[1] Ein Wert von Eins bedeutet demnach, daß die Datei überhaupt nicht fragmentiert ist. Ein Wert von Null, daß nicht ein Block der Datei direkt auf einen anderen folgt.

Es stellt sich die Frage, ob und wieweit dieses abstrakte Maß in Beziehung zu einer meßbaren Größe –wie beispielsweise der Differenz der Lesezeit einer Datei im fragmentierten und unfragmentierten Zustand– zu setzen ist. Wie wir aus den letzten beiden Kapiteln wissen, wird eine solche meßbare und damit für den Benutzer relevante Größe durch eine Menge von Faktoren beeinflußt. Die wichtigsten darunter sind das Prefetching der Platte, der Zwischenspeicher der Platte, der Puffer-, sowie der Verzeichnis- und Inode-Cache des Betriebssystems und das Vorauslesen des VFS. Andere sich im System abspielende Prozesse sind dabei nicht erwähnt, tragen aber

[1] Genauer gesagt, teilt man durch die Anzahl der Blöcke abzüglich Eins. Besteht die Datei nur aus einem Block hat sie eine externe Fragmentierung von Eins.

natürlich zu statistischen Schwankungen bei. Das oben angegebene Standardmaß ist unter diesen Gesichtspunkten in Frage zu stellen. Betrachten wir eine Datei, bei der jeder Block nur um einen Block vor dem vorangehenden getrennt liegt. Die externe Fragmentierung beträgt Null, ist damit maximal. Der Benutzer jedoch würde durch Vorauslesen und Zwischenspeichern nichts davon merken[2]. Daher definieren wir ein etwas erweitertes Maß, das wir mit „FragmentierungX“ bezeichnen. Bei diesem Maß zählt erst ein Sprung von mehr als X Blöcken zur externen Fragmentierung. Für $X = 1$ erhalten wir also das Standardmaß. Für $X \geq 2$ wäre das gerade genannte Beispiel optimal, also nicht fragmentiert. Die Intention dieses Maßes ist es, X in der Größenordnung 16 bis 32, der Anzahl der vorausgelesenen Blöcke, oder zwischen 128 und 512, der Größe des Zwischenspeichers auf der Platte, zu wählen. Allerdings ist das Layout der betrachteten Datei auch nicht optimal, obwohl die Maße $X \geq 2$ dies behaupten. Daher definieren wir noch ein weiteres Maß, das wir als „fragmentierte Länge“ oder „fragmentierten Pfad“ bezeichnen.[3] Für eine Datei aus n Blöcken mit den Blocknummern a_i ist es gegeben als

$$f^{\mathrm{Dat}} = \frac{1}{n-1} \sum_{i=2}^{n} |a_i - a_{i-1}| \ .$$

Für unser Beispiel beträgt die fragmentierte Länge Zwei, ein Wert von Eins wäre optimal. Höhere Werte bedeuten eine höhere Fragmentierung. Leider ist dieses Maß praktisch nach oben unbeschränkt, was einen gewissen Nachteil bei der Interpretation gegenüber den anderen Maßen darstellt.[4]

Bei allen Maßen wird nur der Datenanteil der Datei berücksichtigt, nicht zusätzlich die logische Blocknummer des Blocks, der die Inode der Datei enthält. Es wäre relativ aufwendig, vom Benutzermodus aus festzustellen, in welchem Block die Metadaten einer Datei stehen. Wie wir wissen, ist dafür kein VFS-Systemaufruf vorgesehen. Zudem werden die Metadaten sowohl im fragmentierten als auch im unfragmentierten Fall gelesen, so daß eventuelle Performanzeinbußen auf die fragmentierten Datenblöcke zurückzuführen sind. Den Wert für die Fragmentierung einer ganzen Partition berechnen wir in der Regel als statistischen Mittelwert über die Fragmentierung jeder Datei aus dem Dateisystem. Es ist wiederum fragwürdig, ob der Mittelwert dabei Sinn macht. Die Vorstellung dabei ist, daß viele willkürlich gewählte Dateien aus der Partition diesen Wert im Mittel aufweisen werden. Die andere Möglichkeit, die wir teilweise auch verwenden werden, summiert zunächst die Zählerwerte für die einzelnen Dateien und teilt danach durch die Anzahl der insgesamt belegten Blöcke der Dateien. Dieses Vorgehen haben wir bei der internen Fragmentierung erläutert.

Im letzten Kapitel haben wir gesehen, daß jedes Dateisystem eine ihm eigene Strategie zur Verwaltung und Allokation freier Blöcke im Dateisystem benutzt. Der optimale Fall für ein Dateisystem liegt genau dann vor, wenn es über alle Ressourcen frei

[2] Bei der Charakterisierung einer gesamten Partition kann diese Größe aber auch eine gute Wahl sein, denn wenn jede Datei maximal fragmentiert ist, werden die Gegenmaßnahmen unter starker Belastung den Performanzverlust des Dateisystems nicht verhindern können.

[3] Manchmal benutzen wir auch das englische Wort „fragmented path“ oder kurz „FragPath“.

[4] Theoretisch ist es wegen der endlichen Anzahl von Blocknummern auch nach oben beschränkt.

verfügen kann. Dieser Zustand, liegt in der Praxis nur nach der Formatierung vor. Sobald Dateisystemaktivitäten eintreten, hängt es von der Güte der Dateisystemorganisation ab, Referenzlokalität nicht nur zu erzeugen, sondern auch zu bewahren. Das ist insbesondere schwierig, wenn die Dateisystemressourcen zu mehr als 75% beansprucht sind. Entfernen und Hinzufügen von Dateien oder andere Veränderungen an bestehenden Dateien verändern die mittlere externe Fragmentierung der Partition. Der Mittelwert ist daher zeitabhängig.[5] Er hängt im Prinzip von allen vorherigen Zuständen der Partition ab. Auch die Reihenfolge ausgeführter Operationen kann entscheidend sein. Ohne besondere Maßnahmen seitens des Administrators nimmt die externe Fragmentierung bei jedem Dateisystem durch Benutzung zu. In diesem Zusammenhang spricht man vom „Altern eines Dateisystems“ oder vom „Altern einer Partition“. Es stellt sich allerdings immer die Frage, ob sich der Performanzverlust einer gealterten Partition unter normalen Systembedingungen bemerkbar macht.

Eine Möglichkeit, die externe Fragmentierung zu reduzieren und den Festplattendurchsatz zu erhöhen, kennen wir bereits aus dem letzten Kapitel. Die meisten Dateisysteme benutzen heutzutage eine logische Blockgröße von 4096 Byte. Damit werden bei einer Sektorgröße von 512 Byte immer acht aufeinanderfolgende Sektoren allokiert und der Datentransfer automatisch erhöht. Der Nachteil ist jedoch, daß dadurch die interne Fragmentierung steigt. Bei Verminderung von interner Fragmentierung durch Fragmente bei FFS und Tails bei Reiser sinkt die Performanz durch zusätzliche Suchzeiten beim Lesen der Dateienden. In diesem Sinn sind interne und externe Fragmentierung gewissermaßen Gegenspieler.

4.2 Fragmentierungseffekte beim FFS

Anhand des FFS sind in der Vergangenheit einige publizierte Untersuchungen vorgenommen worden, inwieweit Dateisystemalterungsprozesse Auswirkungen auf die Performanz des FFS-Dateisystems haben.

In [SS94] und [SS95] wird gezeigt, daß die Performanzunterschiede zwischen neu erzeugten und gealterten FFS-Dateisystemen maximal zwischen 10% und 15% betragen. Insgesamt haben die Autoren 48 verschiedene Partitionen mit teilweise sehr unterschiedlichen Beanspruchungen (engl. ”workload”) –beispielsweise als Newsserverdateisystem oder als Heimatverzeichnisdateisystem– untersucht. Die ausgewerteten Daten der gealterten Dateisysteme sind dabei über zehn Monate täglich in Form von Momentaufnahmen (engl. „snapshot“) der Metadaten aller Dateien einer Partition aufgenommen und kontrolliert reproduziert worden. Die Autoren stellen einen Zusammenhang zwischen der Blockallokationsstrategie des FFS, der konkreten Benutzung eines Dateisystems und seiner Fragmentierungsbewertung durch das Standardmaß fest. Die Performanz der neuen und alten Dateisysteme, sowie die Fragmentierung ist dabei für verschiedene Blockgrößen einzeln gemessen worden. Dateien bestehend aus zwei Blöcken sind mit einem Wert von 35% dabei deutlich höher fragmentiert als Dateien größer 256 KB, die einen Wert von über 80% erreichen.

[5] Gleiches gilt natürlich für die interne Fragmentierung.

Ein Ergebnis, das die Autoren klar auf die Benutzung von Fragmenten zurückführen können. Ihr Fazit lautet, daß Cluster-Allokationsverfahren –also Verfahren, die bevorzugt zusammenhängende Blöcke zu allokieren versuchen (siehe 3.5.2)– nur bei neu erzeugten Dateisystemen optimale Performanz aufweisen. Nach Alterung des Systems findet der Allokationsalgorithmus des öfteren nicht mehr hinreichend viele zusammenhängende Blöcke, so daß die Fragmentierung einer neu erzeugten Datei von der bestehenden externen Fragmentierung der Partition, sowie der konkreten Beanspruchung des Dateisystems abhängt. In [SS96] wird daraufhin eine Variante des FFS untersucht, die zusammen mit der Version 4.4 von BSD geliefert wird. Diese nimmt zum Zeitpunkt des Schreibens auf der Platte bei Bedarf eine Reallokation vor, wenn sie ein besseres Cluster als das ursprünglich gewählte findet. Die neue Allokationsstrategie vergleichen die Autoren mit der ursprünglichen Methode. Dabei altern sie die untersuchten Systeme künstlich, indem sie die oben erwähnten über zehn Monate aufgezeichneten Snapshots mit einem heuristischen Verfahren nachspielen. Gemäß dieser Simulation verbessert der Reallokationsalgorithmus im Vergleich zur ursprünglichen Methode die Langzeiteigenschaften des Dateisystems um 30% bis zu maximal 50%.

In [SS97] wird die im Prinzip allgemeine Methode ausführlich dargestellt, mit der die Autoren die Alterungsvorgänge bei den untersuchten Dateisystemen basierend auf echten Auslastungsdaten und Benutzungsmustern simulieren. Sie definieren eine Auslastung des Dateisystems zur simulierten Alterung als eine Folge von Dateioperationen, die vorwiegend aus dem Erzeugen und Löschen von Dateien besteht. Diese Folge von Operationen kann auf verschiedenen Dateisystemen nachgespielt werden, um Langzeiteffekte einer Anwendung zu simulieren, die über eine längere Periode das Dateisystem intensiv benutzt. Um dabei eine realistische Auslastung zu erzeugen, werten die Autoren täglich aufgenommene Metadatensnapshots echter Dateisysteme aus. Die Änderungen am Dateisystem, die an den Unterschieden zweier aufeinander folgender Snapshots zu erkennen sind, bestimmen die Folge der zu simulierenden Operationen. Da durch die täglichen Snapshots kurzzeitige Änderungen am Dateisystem nicht erfaßt werden, erzeugen die Autoren mit einem heuristischen Verfahren kurzfristige Dateisystemaktivität. Dieses Verfahren basiert auf der Protokollierung von über den Zeitraum von einer Woche aufgezeichneten NFS-Zugriffen, aus denen sie 449 verschiedene Zugriffsprofile erstellen, von denen jeweils 25 zufällig für die Simulation eines Tages gewählt werden. Zum Abspielen der Auslastung benutzen die Autoren einige Merkmale von FFS, weil sie bei den Snapshots die Pfadnamen der Dateien nicht aufgezeichnet haben. So schließen sie beispielsweise aus der aufgezeichneten Inode-Nummer auf die zu benutzende Zylindergruppe für eine zu erzeugende Datei. Damit ist die Umsetzung ihrer Methode auf andere Dateisysteme nicht trivial. Bei der Verifikation eines ihrer simuliert gealterten Dateisysteme mit dem echt gealterten Dateisystem stellen die Autoren fest, daß das simulierte Dateisystem um 10% weniger fragmentiert ist als das echte. Sie halten es für möglich, daß dies an den fehlenden Informationen über die tatsächlich kurzzeitig stattgefundenen Änderungen am Originalsystem liegen kann.

In einer Serie von Artikeln [TS97], [SGS+97] und [SKSZ99] behaupten die Autoren, daß bestehende Dateisystembenchmarks im besten Fall nutzlos und im schlechtesten

Fall irreführend sind. Diese sogenannten „Mikrobenchmarks“, die viele Aspekte eines Dateisystems einzeln und isoliert zu analysieren versuchen, machen keine Aussage über die Performanz des Gesamtsystems, weil diese Performanz von dem Kontext der Operationen und der Auslastung des Systems bestimmt wird. Sie belegen ihre Aussagen an konkreten Beispielen. Da aber auch nicht für jede Applikation ein eigener Benchmark entwickelt werden kann, schlagen sie drei applikationsspezifische Benchmarktypen vor: vektorbasierte, auswertungsbasierte und hybride Benchmarks.

- Bei vektorbasierten Benchmarks werden Systemvektoren erstellt, die in jeder Komponente des Vektors systemspezifische Kosten für eine bestimmte Primitive angeben. Diese kann beispielsweise durch einen Mikrobenchmark ermittelt werden. Für die zu testende Applikation wird ein entsprechender Applikationsvektor erstellt, der in jeder Komponente die Quantität angibt, mit der die Applikation die jeweilige Primitive benutzt. Um ein Ergebnis für die erwartete Laufzeit der Applikation zu erhalten, wird das Skalarprodukt zwischen beiden Vektoren gebildet.

- Für bestimmte Anwendungen hängt die Performanz aber nicht allein von der Benutzung einer bestimmten Ressource ab, sondern auch von der Reihenfolge ihrer Benutzung. Bei auswertungsbasierten oder spurbasierten Benchmarks wird eine für die zu testende Applikation spezifische Last generiert. Man kann beispielsweise Logdateien oder ähnlich aufschlußreiche Dateien auswerten und daraus mit stochastischen Mitteln spezifische Anfragen an das System erzeugen. Anstelle der Laufzeit der Applikation mißt man die Laufzeit der Folge dieser Anfragen. Eine Aufzeichnung einer derartigen Folge von Dateioperationen wird auch „Spur“ (engl. „trace“) genannt.

- Die hybride Methode kombiniert beide Vorgehensweisen. Mithilfe von Mikrobenchmarks erzeugt man einen Systemvektor. Die Auswertung der Applikationsspuren und die daraus generierten Anfragen speist man in einen Systemsimulator ein. Dieser erzeugt daraus den Applikationsvektor. Das Skalarprodukt aus beiden Vektoren liefert wiederum die erwartete Laufzeit der Applikation auf dem System. Der Vorteil dieser Methode liegt auf der Hand. Ein einmal programmierter Simulator kann für die Simulation unterschiedlicher Applikationen auf verschiedenen Systemen benutzt werden. Dazu muß einmal für jedes System der Systemvektor gemessen werden und einmal für jede Applikation der Applikationsvektor aus der Analyse ihrer Spuren erstellt werden.

In seiner Dissertation benutzt A. Smith die hybride Methode, um die Performanz eines Dateisystems unter bestimmter Auslastung vorherzusagen und gleichzeitig anhand des echten Dateisystems seine Vorgehensweise zu verifizieren [Smi01]. Dazu entwickelt er einen Dateisystemsimulator „HBench-FS“ für POSIX konforme Dateisysteme. Dieser erstellt aus in einem bestimmten Format übergebenen Spuren von Dateisystembefehlen einen Applikationsvektor. Dieser Vektor bestimmt nach der hybriden Methode zusammen mit auf verschiedenen Systemen gemessenen Systemvektoren die Latenzzeit, die das jeweilige Dateisystem zur Bearbeitungszeit der Applikation beiträgt. Zum Testen der Vorhersagekraft seines Benchmarks benutzt

Smith verschiedene Applikationen. Eine davon ist beispielsweise das Kompilieren des Betriebssystemkernels. Auf vier verschiedenen UNIX-Systemen mit Derivaten von FFS mißt er den Systemvektor, sowie für jede Applikation die tatsächliche Latenzzeit, die das Dateisystem bei der Bearbeitung der Applikation zur Gesamtlaufzeit beiträgt. Gleichzeitig protokolliert er durch eine Manipulation des Betriebssystemkernels die ausgeführten Systemaufrufe. Pro Applikation erstellt er mit diesem Protokoll ein Profil der Applikation. Mit diesen Profilen erzeugt der Simulator die entsprechenden Applikationsvektoren. Danach verifiziert Smith für jede Applikation pro System die gemessene Latenzzeit mit der durch das Skalarprodukt berechneten Latenzzeit. Laut seiner Angaben haben seine Vorhersagen der Latenzeit im Durchschnitt eine Abweichung von ca. 30%. Sie bestimmen aber in ca. 95% aller Fälle für die jeweilige Applikation das bessere von zwei zur Auswahl stehenden Dateisystemen. In den zwei von 48 Fällen, in denen seine Methode versagt, sind die beiden Dateisystempaare auch in der gemessenen Performanz kaum zu unterscheiden. Daher ist Smith der Auffassung, daß sein Benchmark zur Wahl des besten Dateisystems für eine bestimmte Applikation herangezogen werden kann.

4.3 Simulationen und Hilfsprogramme

In dieser Arbeit untersuchen wir, ob Fragmentierungseffekte beim ReiserFS meßbar und bewertbar sind. Die prinzipielle Meßbarkeit von derartigen Effekten auf die Datenübertragung wird durch die Anzahl der daran beteiligten Komponenten erschwert. Diese Komponenten haben wir ausführlich in den letzten beiden Kapiteln beschrieben und zusammenfassend in Abschnitt 4.1.2 erläutert. Wie wir spätestens aus dem letzten Abschnitt wissen, sind solche Effekte normalerweise langfristiger Natur. In der Regel hängen sie zudem noch von der tatsächlichen Last ab, die ein Dateisystem durch verschiedene Applikationen erfährt.

Wir haben uns in den letzten sechs Monaten mit zwei Vorgehensweisen befaßt, von denen die erste konkrete Ergebnisse geliefert hat, während die zweite, jüngere Methode, sich noch im Teststadium befindet:

1. Alterungssimulation
   Bei der „Alterungssimulation“ versuchen wir, ein Dateisystem in kurzer Zeit gezielt zu altern.[6] Dazu abstrahieren wir von einer konkreten Benutzung des Dateisystems durch eine Anwendung auf bestimmte Teilaspekte der Benutzung, von denen manche Anwendungen aber massiv Gebrauch machen. Zur Bewertung des Zustands des Dateisystems bewerten wir zwischendurch in definierten Abständen das Blocklayout und führen diverse Leseperformanzmessungen durch. Nach dem vorangegangenen Abschnitt ist diese Methode daher eher als klassischer Benchmark zu charakterisieren, der ein Dateisystem in manchen Aspekten langfristig belastet und bewertet.

[6] „Kurze Zeit“ bedeutet dabei eine Simulationszeit in der Größenordnung von maximal einigen Tagen. Diese Nebenbedingung haben wir getroffen, damit eine Simulation bei Bedarf – beispielsweise bei der Freigabe einer neuen Version eines Dateisystems oder eines Kernels– ohne großen Aufwand anwendbar ist.

2. Applikationssimulation
   Bei der „Applikationssimulation“ zeichnen wir die von einer konkreten Anwendung im laufenden Betrieb ausgeführten Dateisystemoperationen auf. Diese Aufzeichnungen können in einem zweiten Programm ausgewertet und nachgespielt werden. Damit ist es möglich, ein Dateisystem oder mehrere Dateisystem anwendungsbasiert und über einen definierten Zeitraum zu vergleichen. Nach der oben gegebenen Klassifizierung ist diese Simulation als auswertungbasierter Benchmark anzusehen.

4.3.1 Alterungssimulation

Diese Simulation dient der künstlichen Alterung einer gegebenen Partition. Ihr Ziel ist es, ein Dateisystem mit einer Folge von definierten Operationen zu belasten, eine eventuell auftretende Alterung anhand von Fragmentierungsmaßen zu bewerten und mit Performanzmessungen zu vergleichen.

Die Vorgehensweise bei einer Alterungssimulation besteht im wesentlichen aus der wiederholten Durchführung dreier Phasen:

- Änderungsphase
  In der Änderungsphase werden an dem Dateisystem in definierter Form Änderungen durchgeführt. In den im folgenden beschriebenen Tests benutzen wir das dafür entwickelte Hilfsprogramm „agesystem3“. Dies ist aber nicht zwingend notwendig. Beliebige Programme oder Skripte könnten in dieser Phase dazu ausgewählt werden, das Dateisystem nachhaltig zu belasten.

- Bewertungsphase
  Die Bewertungsphase folgt in der Regel auf die Änderungsphase. Hier wird eine Bewertung des Blocklayouts auf der Platte vorgenommen. Das dafür geschriebene Hilfsprogramm ist „fibmap“, das den in Abschnitt 3.3.2 beschriebenen gleichnamigen Gerätekontrollbefehl verwendet.

- Meßphase
  In der Meßphase werden verschiedene Leseperformanztests durchgeführt. Das dafür entwickelte Hilfsprogramm ist „read“. Bei den Messungen muß sichergestellt werden, daß der Zwischenspeicher die Meßergebnisse nicht verfälscht.

Der Applikationscharakter einer konkreten Simulation wird durch ein Shell-Skript vermittelt. Zur Durchführung unterschiedlicher Simulationstypen benutzen wir daher in der Regel verschiedene Skripte. Den Skripten ist gemeinsam, daß sie die Reihenfolge der Phasen einhalten, sowie die Aufzeichnung der auszuwertenden Daten übernehmen. Gleichzeitig stellen sie auch soweit wie möglich sicher, daß der Puffercache geleert ist, bevor die verschiedenen Messungen durchgeführt werden. Die Skripte unterscheiden sich je nach Charakter, den eine Simulation haben soll. Im einfachsten Fall wird in der Änderungsphase einfach ein anderes Programm oder wie in unserem Fall „agesystem3“ mit verschiedenen Optionen ausgeführt. In anderen Fällen müssen die verschiedenen Phasen mehrmals ausgeführt oder die Partition zu Vergleichszwecken kopiert werden.

Eine Alterungssimulation unterliegt folgenden Randbedingungen, die bei der Wahl der Szenarien und Hilfsmittel eingehalten werden sollten:

- Änderungen am Dateisystem dürfen nur durch einen Prozeß[7] und direkt auf dem Dateisystem erfolgen. Diese Restriktion soll gewährleisten, daß soweit wie möglich das zu untersuchende Dateisystem benutzt wird und nicht etwa andere Softwareschichten wie beispielsweise der Disk-Scheduler, NFS oder LVM.

- Leseperformanzmessungen müssen ebenfalls von einem Prozeß vorgenommen werden. Dabei wird eine Datei immer ganz und sequentiell gelesen. Diese Restriktion soll sicherstellen, daß unnötige und verfälschende Suchzeiten durch parallele Leseaufträge vermieden werden. Desweiteren treten Fragmentierungseffekte am ehesten beim sequentiellen Lesen auf. Nach [BHK+91] werden ohnehin die meisten Dateien sequentiell gelesen, so daß diese Einschränkung nicht realitätsfern ist.

- Die durchzuführenden Operationen auf dem Dateisystem sollen in beiden Fällen direkt die VFS-Systemaufrufe bzw. die entsprechenden einhüllenden Funktionen der Systembibliothek benutzen. Damit wird vermieden, daß eine Systembibliothek verfälschende Pufferungen durchführt. Darüber hinaus wird die Einflußnahme des Puffercaches durch Verminderung des Hauptspeichers eingeschränkt.[8] Dies verringert zwar ein wenig den Realitätsbezug, erhöht aber gleichzeitig die Meßbarkeit von Fragmentierungseffekten beim Lesen.[9]

Nach der Bekanntmachung einiger Ergebnisse der ersten Alterungssimulationen in einschlägigen Mailinglisten entstand die Notwendigkeit einer gezielteren und dauerhafteren Veröffentlichung auf einer Webseite [Loi01e]. Diese Seite ist mittlerweile relativ umfangreich. Sie dient der Vereinfachung der Kommunikation mit den Entwicklern der untersuchten Dateisysteme und den Teilnehmern der Mailinglisten. Der Programmcode aller relevanten Hilfsprogramme, Informationen zur Bedienung und eine Menge von Ergebnissen verschiedener Simulationen sind dort veröffentlicht. Wir können im Rahmen dieser Arbeit nicht auf alles eingehen, was dort zu finden ist. Statt dessen stellen wir im folgenden die oben genannten Hilfsprogramme vor und verweisen bei Bedarf auf die entsprechenden veröffentlichten Stellen.

Die Struktur unserer Hilfsprogramme ist im wesentlichen immer gleich. Es gibt eine Hauptdatei mit dem Namen des Hilfsprogramms –beispielsweise „read.cxx“, die die Hauptfunktion „main“ beinhaltet. Diese Hauptdatei fügt in der Regel die zentrale Datei „common.h“ ein, in der alle benutzten VFS-Systemaufrufe gekapselt, sowie oft benutzte andere Funktionen definiert werden. Ein Beispiel einer solchen Funktion ist

[7] Im folgenden schließen wir damit auch Leichtgewichtsprozesse ein.

[8] Da der Puffercache mit dem Seitenspeicher vereinheitlicht ist, ist diese Maßnahme die einzige Möglichkeit, dabei auch Systeminstabilitäten zu vermeiden.

[9] Natürlich wird dadurch auch die Schreibperformanz beeinflußt. Allerdings ist diese Einflußnahme relativ gering, da durch die Einteilung in Phasen die Änderungen ohnehin durch den in der Änderungsphase erzeugten Dauerstreß in einem Stück geschrieben werden müssen.

```
int walk_through_dirs(char *path, MY_FUNC *function),
```

mit der rekursiv alle Verzeichnisse beginnend mit dem Pfadnamen „path“ durchlaufen werden. Für jede gefundene Datei wird die Funktion „function“ des Typs

```
typedef int MY_FUNC(const char *,const struct stat *);
```

mit dem Pfadnamen und den Inode-Daten der Datei aufgerufen. Je nach der Implementierung dieser Funktion kann damit beispielsweise jede in einer Partition vorhandene Datei erfaßt werden. Manche Hilfsprogramme machen Gebrauch von Klassen zur Zeitmessung aus „cl_time.cxx“, zur Verwaltung von Histogrammen aus „cl_hist.cxx“ oder zur Verwaltung der Dateien einer Partition im Speicher aus „cl_partinfo.cxx“. Desweiteren gibt es noch einfache Array- und Stringklassen. Die Struktur der Quelldateien ist in Abb. 4.1 dargestellt.

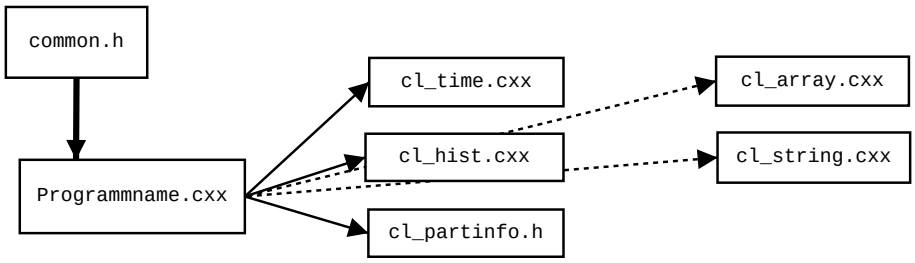

Abbildung 4.1: Die Struktur der Programmcodedateien für die Hilfsprogramme

Beschreibung von „agesystem3“

Dieses Hilfsprogramm dient dazu, ein Dateisystem auf verschiedene Weise zu belasten [Loi01b]. Die möglichen Einstellungen beschreiben wir am einfachsten anhand der Optionen, die man übergeben kann (siehe Abb. 4.2). Das Programm benutzt zur Simulation künstliche oder gemessene Verzeichnishierarchien. Künstliche Verzeichnisse werden durch die Angabe von „-c a:b“ erzeugt. Für $a = 10$ und $b = 100 \geq a$ werden beispielsweise zehn Verzeichnisse mit je zehn Verzeichnissen darin erzeugt. Bei künstlichen Verzeichnissen werden Dateien nur innerhalb der inneren Verzeichnisse geschrieben.[10] Bei der Erzeugung von Dateien legt „agesystem3“ gemessene oder berechnete Verteilungen für die Dateigrößen zugrunde. Die Option „-l“ bestimmt dieses Verhalten.

Übergibt man die Option „-l 6:a:b:c:d:e:f“, dann wird eine künstliche Verteilungsfunktion mit den Parametern a-f verwendet. Die Verteilungsfunktion ist dabei eine unsymmetrische Gaussverteilung mit der Standardabweichung „x“ und liefert einen minimalen Wert von „y“.[11] In 89% aller Erzeugungsfälle wird „y“ durch „a“ und „x“ durch „d“, in 10% alle Fälle „y“ durch „b“ und „x“ durch „e“ und in 1% aller

[10] Künstliche Verzeichnisse können nur auf diese Weise –mit einer Tiefe von Zwei– festgelegt werden. Will man beispielsweise nur ein Verzeichnis benutzen, muß man „-c 1:1“ angeben.

[11] Zum Erzeugen von statistischen Größen oder Pseudozufallszahlen verwenden wir die „GNU Scientific Library“ (GSL) in der Version 0.8. Siehe daher http://sources.redhat.com/gsl/ref/gsl-ref_18.html#SEC269 für eine Erläuterung zu der genannten Verteilungsfunktion.

```
Usage of agesystem3 v0.3: agesystem3 [options]
 -b blksize              blocksize to use for buffered i/o
 -c dir_base:dir_counter number of subdirectories to create
 -d dirname              change to directory "dirname"
 -f                      use fixed length of bytes, given in blocksize
 -h,-?                   this text
 -l string               set file size creation options coded in "string"
                         string is a) 6:a:b:c:d:e:f ->
                           a,b,c min_sizes; d,e,f max_sizes for distribution
                                   b) 1:histfilename -> use distribution of
                                      measured histogram and use -c for dirs
                                   c) 2:histdirname:histfilename -> use measured
                                      distribution for files and dirs
 -n number_of_files      maximum number of files to create
 -m mode                 set working mode to "mode"
                             0 -> get info struct
                             1 -> gauge only; use with -c and -l option
                             2 -> age fs system; use with -c and -l option
                             3 -> calc histograms; use with -l option
                             4 -> create only directories given with -l oder -c opt
 -o access_mode          set access mode to  "access_mode" (eg. cw=65, cws=4161,
 -p dirno:fileno         store max "dirno" dirs and "fileno" files in info struct
 -r file                 load partition info from "file"
 -s file                 save partition info to "file"
 -t                      simply test input and exit without any action
 -u min:max              min and max filesystem usage to work at
 -v                      be verbose
 -y seed                 set random generator seed
 -z                      sync with unmount (don't use or check/edit
                                            /root/bin/agesync.sh)
```

Abbildung 4.2: Die Optionen von „agesystem3“

Fälle „y“ durch „c“ und „x“ durch „f“ ersetzt. Damit kann man beispielsweise dafür sorgen, daß kleinere Dateien häufiger erzeugt werden als große.[12]

Bei der Simulation hält das Programm sämtliche Informationen über den Status einer Partition im Speicher. Die dazu verwendete Struktur kann durch die Option „-r“ und „-s“ aus einer Datei eingelesen oder in eine Datei geschrieben werden. Das Einlesen zu Beginn hat den Vorteil, daß die Partition nicht bei jeder Änderungsphase rekursiv durchsucht werden muß. Man spart somit Simulationszeit. Die Option „-u“ veranlaßt „agesystem3“ den benutzten Speicherbereich auf der Partition nicht unter das angegebene Minimum bzw. Maximum sinken bzw. steigen zu lassen. Durch die Option „-d“ wechselt „agesystem3“ vor der Ausführung irgendeiner Aktion in das angegebene Verzeichnis. Es gibt fünf verschiedene Modi, die mit dem Schalter „-m“ ausgewählt werden:

- Modus „-m 0“: Erfassung von Informationen über ein Verzeichnis
  In diesem Modus durchsucht „agesystem3“ das aktuelle Verzeichnis rekursiv

[12] Siehe die Beschreibung unter http://www.informatik.uni-frankfurt.de/~loizides/reiserfs/agesystem.html#1fsdist für ein oft verwendetes Beispiel.

und speichert die Informationen über gefundene reguläre Dateien und Verzeichnisse im Hauptspeicher. Dieser Modus macht eigentlich nur Sinn, wenn man gleichzeitig die Option „-s“ zum Speichern der Partitionsinformationen in einer Datei benutzt. Diese Datei kann beim Modus „-m 3“ eingelesen werden, um Informationen über Verzeichnisse und Größenverteilungen zu erzeugen, die dann zur Simulation benutzt werden.

- Modus „-m 1“: Einmaliges Erzeugen von Dateien ohne Löschen
  In diesem Modus erstellt das Programm die durch „-c“ oder „-l“ gewählten Verzeichnisse. Anschließend erzeugt es solange Dateien nach der gewählten Verteilungsfunktion, bis der benutzte Speicherplatz auf das durch die Option „-u“ spezifizierte Maximum angewachsen ist. Wir nennen diesen Modus und dieses Verfahren auch „Eichung“ (engl. „gauge“), da jede Alterung relativ zu dieser ersten Änderungsphase festgestellt wird.

- Modus „-m 2“: Erzeugen und Löschen von Dateien
  In diesem Alterungsmodus erzeugt und löscht das Programm Dateien in den vorhandenen Verzeichnissen. Dazu verfährt es wie folgt: Es berechnet den freien Speicherplatz f im durch die Option „-u“ angegebenen Interval. Mit der Wahrscheinlichkeit X erzeugt es nach der in der Abb. 4.3 gezeigten Wahrscheinlichkeitstabelle eine Datei. Mit $1 - X$ löscht es eine zufällig gewählte Datei. Ist die obere Grenze erreicht, f also zwischen 95% und 100%, wird die

f [%]	X
0-5	1.00
5-75	0.80
75-95	0.70
95-100	0.00

Abbildung 4.3: Die Wahrscheinlichkeitstabelle zum Erzeugen einer Datei

  Tabelle invertiert. Auf diese Weise ist das Löschen wahrscheinlicher als das Erzeugen. Bei Erreichen der unteren Grenze wird die Tabelle wieder wie gezeigt verwendet. Auf diese Weise erzeugt und löscht „agesystem3“ im durch die Option „-u“ definierten Intervall andauernd Dateien. Mit der Option „-n“ gibt man an, wieviele Dateien erzeugt werden sollen, bis das Programm endet. Die Größenwahl der erzeugten Dateien geschieht durch die gewählte Verteilungsfunktion.

- Modus „-m 3“: Erzeugung von Meßinformationen
  In diesem Modus werden aus einer eingelesenen Informationsdatei zwei Dateien erstellt, die mit „-l 2:dname:fname“ für die Verzeichnisse in „dname“ und für Dateien in „fname“ gespeichert werden. Bei der Simulation können dann anstelle von künstlichen Verteilungen diese gewählt werden. Die Verzeichnisse werden dann bei der Simulation genauso angelegt, wie sie in der Informationsstruktur gefunden worden sind. Für Dateien wird ein Histogramm der

Größenverteilung berechnet, daß dann zur Bestimmung der Dateigrößen anstelle der generischen Verteilungsfunktion benutzt wird.

- Modus „-m 4“: Erzeugen der Verzeichnisstruktur
  In diesem Modus erzeugt das Programm lediglich die gewünschte Verzeichnisstruktur und führt ansonsten keine Operationen aus.

Die Ausgabe von „agesystem3“ im Eich- und Alterungsmodus ist eine Tabelle, in der für alle 5% Partitionsbelegung die Anzahl der geschriebenen und gelöschten Bytes und die dafür benötigte Zeit stehen. Während das Programm läuft, wird die Belegung der Partition ständig überprüft. Bei einem Überschreiten einer Grenze –beispielsweise von 10% bis 15% auf 15% bis 20% Belegung wird der in Abschnitt 3.3.1 Befehl „sync“ zur Synchronisation benutzt. Damit wird sichergestellt, daß die Zeitmessung tatsächlich auch den Umsatz in diesem Intervall der Belegung erfasst.

Wir benutzen „agesystem3“ für drei verschiedene Tests. Wir skizzieren im folgenden die entscheidenden Stellen aus den drei verwendeten Simulationsskripten:

- Eichtest
  Beim Eichen erzeugen wir im Verzeichnis „wdir“ in den Verzeichnissen aus „dirfile“ solange Dateien mit der Größenverteilung „fhist“, bis die Partition zu 70% voll ist. Das Ergebnis speichern wir in der Datei „sfile“.

  ```
  agesystem3 -m1 -dwdir -u0:70 -l2:dirfile:fhist -ssfile
  ```

  Einzelne Eichtests haben den Charakter eines gewöhnlichen Benchmarks. Es ist nicht zu erwarten, daß dabei Fragmentierungseffekte auftreten. Bei Alterungstest wird eine Partition immer zuerst geeicht, bevor die Alterungsoperationen beginnen.

- Alterungstest
  Bei dieser Alterung erzeugen und löschen wir im Verzeichnis „wdir“ in den Verzeichnissen aus „fhist“ solange Dateien im Bereich zwischen 65% und 75%, bis insgesamt 1000 Dateien gemäß der Größenverteilung „fhist“ erzeugt sind. Wir speichern die Partitionsinformationen am Ende in „sfile“, um sie bei der nächsten Änderungsphase wieder einzulesen.

  ```
  agesystem3 -m2 -n1000 -dwdir -u65:75 -l 2:dirfile:fhist -ssfile
  ```

  Bei derartigen Tests wird untersucht, inwieweit ständiges Erzeugen und Löschen von Dateien, wie es beispielsweise in temporären Verzeichnissen oder Mail- und Newsverzeichnissen vorkommt, die Leistung eines Dateisystems senkt. Es ist ein moderater Leistungsabfall zu erwarten.

- Appendtest
  Bei diesem Test erzeugen wir in einer anfangs leeren Partition genau 1000 Dateien unterhalb des Verzeichnisses „wdir“ in einem Verzeichnis. Diese Dateien erweitern wir reihum um je einen Block von 4 KB, was durch die Option

„-o 1089“ für das Schreiben und die Optionen „-f“ für „feste Größe“ und „-b“ für die Angabe der Größe ausgedrückt wird. Das Programm endet, wenn die Partition auf diese Weise zu 99% benutzt ist.

```
agesystem3 -m1 -n1000 -dwdir -u0:99 -c1:1 -o1089 -f -b4096
```

Bei diesem Test sind große Fragmentierungseffekte zu erwarten, da ein derartiges Schreibszenario für den Blockallokationsalgorithmus eines Dateisystems sehr schwer optimierbar ist. Allerdings ist es auch nicht ungewöhnlich, Dateien durch Anhängen (engl. „append“) zu erweitern. Logbucheinträge oder allgemeine Datenbankeinträge könnten auf diese Weise erfolgen.

Bei Simulationen stellt das Verzeichnis „wdir“ sinnvollerweise den Einhängepunkt der zu testenden Partition dar. Das hat zudem den Vorteil, daß zusammen mit der Option „-z“ ein Skript mit dem Namen „agesync.sh“ dazu benutzt wird, die Partition aus- und wieder einzuhängen. Damit wird sichergestellt, daß der Puffercache zu Beginn verschiedener Phasen oder auch zwischen den verschiedenen Lesetests immer vollständig geleert ist.[13]

Beschreibung von „fibmap“

Dieses Programm benutzt den in 3.3.2 beschriebenen gleichnamigen Gerätekontrollbefehl, um das Layout einer Partition nach den in Abschnitt 4.1 definierten Maßen zu bewerten [Loi01c]. Die möglichen Optionen sind in Abb. 4.4 dargestellt. Normalerweise übergibt man „fibmap“ lediglich eine Liste von Pfadnamen.[14] Damit berech-

```
Usage of fibmap v0.2: fibmap [options] pathnames
 -b blks        work only on blocks greater than blks
 -h,-?          this text
 -s             don't show stats after fibmaping
 -v             increase verbosity by one
                output is as follows:
                name size_in_bytes blocks_total - blocknumbers - \
                f1 f10 f100 f1000 - fragmax - fragpath
```

Abbildung 4.4: Die Optionen von „fibmap“

net es die verschiedenen Maße für jede Datei, indem es rekursiv die angegebenen Verzeichnisse durchläuft. Am Ende wird eine Zusammenfassung der Ergebnisse ausgegeben, was durch die Option „-s“ unterdrückt werden kann. Mit der Option „-v“ kann man die Ausgabe der Maße für jede Datei einzeln erzwingen, mit „-v -v“ werden darüber hinaus die pro Datei belegten Blöcke angezeigt. Ein Loch wird dabei mit einem „h“ markiert und ein Tail mit einem „t“. Von den berechneten Maßen haben wir eines noch nicht angegeben. Es ist in Abb. 4.4 mit „fragmax“ gekennzeichnet.

[13] Gleichzeitig stellt dies natürlich auch eine Synchronisation dar, weil vor dem Aushängen sämtliche Puffer geschrieben werden.

[14] In unserem Fall geben wir in den Skripten immer nur den Aufhängepunkt der untersuchten Partition an.

Wir definieren es für eine Datei als die Differenz aus der größten Blocknummer minus der kleinsten normiert durch die Anzahl der Blöcke, die die Datei insgesamt belegt. Es liefert den optimalen Wert Eins, wenn die Datei nur über die belegten Blöcke verteilt ist und ist nach oben praktisch unbeschränkt. Es hat somit eine große Ähnlichkeit mit dem fragmentierten Pfadmaß.

Beschreibung „read“
Dieses Programm hat die Aufgabe, eine Partition drei verschiedenen Leseperformanzmessungen zu unterziehen und das Ergebnis in Form von Histogrammen auszugeben [Loi01d]. Die möglichen Optionen sind in Abb. 4.5 aufgeführt. Normalerweise startet man das Programm mit einer Menge von Pfaden.[14] Diese Pfade werden

```
Usage of read v0.3 (READTEST defined): read [options] pathnames
 -b blksize             blocksize to use for buffered i/o
 -d dirname             change to directory "dirname"
 -h,-?                  this text
 -v                     be verbose
 -p maxd:maxf           set # of maxdirs and maxfiles
 -n files               set # files to random read
 -l kbytes              set # kbytes to random read
 -y seed                set random generator seed
 -z                     sync with unmount (don't use or
                                check/edit agesync.sh)
```

Abbildung 4.5: Die Optionen von „read“

rekursiv durchlaufen. Jede Datei, die dabei gefunden wird, wird in einer Liste im Hauptspeicher gespeichert und gleichzeitig von „read“ sequentiell gelesen. Diesen ersten Lesetest bezeichnen wir meist mit „Dir Read“ oder „Read“. Gerade bei wenig Speicher wird auf diese Weise die Referenzlokalität zwischen Verzeichnisblöcken, Metadatenblöcken und Datenblöcken von Dateien im Verzeichnis überprüft. Die dabei erstellte Liste wird dazu benutzt, in einem zweiten Test zufällig Dateien auszuwählen und sequentiell zu lesen. Diesen Zufallslesetest nennen wir „Random Read“ oder „RRead“. Er ist weniger metadatenintensiv und begrenzt durch die großen Suchzeiten auf der Platte. Vor dem letzten Lesetest sortiert „read“ die Dateien in der Liste nach der ersten Blocknummer aufsteigend. Danach liest es sie in der aufsteigenden Reihenfolge sequentiell. Diesen Test nennen wir „Fibmap Read“ oder „FRead“. Er ist natürlich im Gegensatz zu den anderen beiden künstlicher Natur. Die Hoffnung bei der Konzeption bestand darin, daß auf diese Weise durch fragmentierte Dateien ein deutlich spürbarer Performanzverlust sichtbar wird.[15]

4.3.2 Applikationssimulation

Bei dieser Simulation zeichnen wir die von einer Anwendung im laufenden Betrieb ausgeführten Operationen auf. Diese Spuren können dann auf einem konfigurierba-

[15] Leider haben wir zur Konzeptionszeit nicht bedacht, daß die zu lesenden Metadaten die Sichtbarmachung des Effekts verhindern. Man müßte also im Voraus eine größere Menge an Dateien öffnen und dann die Leseaufträge in dieser Reihenfolge übergeben.

ren Testsystem beliebig oft mit einem zweiten Programm ausgewertet und nachgespielt werden. Je nach Menge der aufgezeichneten Operationen können damit verschiedene Dateisysteme –oder Dateisysteme mit unterschiedlichen Einstellungen– auf ihre Langzeitperformanz bezüglich bestimmter Applikationen untersucht und verglichen werden.

Anders als in Abschnitt 4.2 bei veröffentlichten Vorgehensweisen erläutert, benötigen wir einmalig einen Snapshot zu Beginn der Spurenaufzeichnung. Von da an protokollieren wir alle Änderungen am untersuchten Dateisystem kontinuierlich und inkrementell. Der Snapshot beinhaltet die Pfadnamen aller Dateien und einige interessante Metadaten wie die Größen jeder Datei. Er wird dazu benutzt, um das künstliche Dateisystem zu Beginn der Simulation in denselben Zustand zu versetzen, in dem sich das Originalsystem zu Beginn der Spurenaufzeichnung befand.[16] Die Erzeugung eines Snapshots ist im Prinzip unproblematisch.[17] Der Befehl „find wdir -ls“ kann dafür benutzt werden.[18] Bei der Aufnahme der Spuren haben wir mehrere Möglichkeiten:

- Benutzung des Programms „strace“
  Dieses Programm ermöglicht die Verfolgung von Systemaufrufen, die ein anderes Programm bei der Abarbeitung seiner Tätigkeiten benutzt. Es verwendet dazu den Systemaufruf „ptrace“, der einem Prozeß die Kontrolle über einen anderen vermittelt. Je nach Anwendung kann eine durch „strace“ verfolgte Anwendung signifikant langsamer laufen.[19] Der Ergebnisbericht von „strace“ läßt sich zudem kaum durch ein Programm verarbeiten. Diese zwei Gründe sprechen gegen die Verwendung von „strace“.

- Benutzung eines Kernelmoduls
  Die Aufgabe der Verfolgung von Systemaufrufen eines Programms kann auch auf der anderen Seite der Schnittstelle –im Kernel selbst– vorgenommen werden. Dazu müßte man die entsprechenden Stellen im Programmcode ändern, an der die Systemaufrufe einsetzen. Alternativ könnte man auch ein Modul entwickeln, daß man bei Bedarf an den Kernel binden kann.[20] Die Ausgabe erfolgt in jedem Fall über das Proc-Dateisystem. Der Nachteil dieser Vorgehensweise ist allerdings, daß auf der Kernelseite Dateien nur über Dateideskriptoren und Prozeßnummern identifiziert werden, so daß zunächst alle Aufrufe im System abgefangen werden müssen, um jeweils die Zuordnung zu der untersuchten Applikation zu ermöglichen. Entweder der Prozeß, der das

[16] Die Fragmentierungswerte beider Systeme werden sich aber unterscheiden, da der Snapshot auf ein frisch formatiertes System eingespielt wird.

[17] Da die Generierung eines Snapshots auf einem sehr belasteten Produktionssystem erhebliche zusätzliche Last erzeugt, könnte man bei tatsächlicher Verwendung unserer Methode anders vorgehen und die benötigten Daten während der Spurverfolgung aufzeichnen.

[18] Wir ziehen dieser Lösung aber aus technischen Gründen ein eigenes Hilfsprogramm mit dem Namen „rekdir“ vor, das das Ergebnis in maschinell leicht einlesbarer Weise ausgibt.

[19] Man kann „strace“ mit einem Debugger vergleichen, der an den Sprungstellen zu den interessierenden Systemaufrufen Haltepunkte setzt.

[20] Ein Beispielmodul zum Abfangen eines Systemaufrufs findet man unter http://www.kernelnewbies.org/code/intercept/.

Proc-Dateisystem ausliest, oder der Kernel selbst filtern die unnötig aufgezeichneten Informationen heraus. In jedem Fall wird dabei kostbare Rechenzeit vergeudet.

- Benutzung der Systembibliothek
  Die meisten Systeme unter LINUX verwenden die GNU-Systembibliothek „glibc“ [FSF01]. Sie kapselt über die Bereitstellung von vielen Funktionen hinaus auch alle Systemaufrufe. Daher kann man statt den Programmcode der Systemaufrufe auf der Kernelseite zu ändern, den Programmcode der Bibliothek verändern und die interessierenden Systemaufrufe durch eigene Funktionen ersetzen. Der Vorteil gegenüber der Kernelmethode liegt auf der Hand. Man programmiert im gewohnten und nichtkritischen Benutzermodus. Der Nachteil ist allerdings, daß statisch gebundene Programme neu kompiliert werden müßten, falls sie durch die geänderte Bibliothek verfolgt werden sollen.

Wir haben uns bei der Programmierung der Applikationssimulation für eine Variante der letztgenannten Methode entschieden.[21] Dabei schränken wir uns auf Programme ein, die Funktionen der Systembibliothek nicht statisch, sondern gemeinsam (engl. „shared“) einbinden. Nichtaufgelöste Symbole derartiger Programme werden zur Laufzeit durch den GNU-Linker „ld“ an die im System vorhandenen Bibliotheksfunktionen gebunden. Diesen Linker kann man durch Umgebungsvariablen steuern, so daß man in den Linkprozeß eingreifen kann, ohne etwas an dem auszuführenden Programm zu ändern. Die für unsere Zwecke interessante Umgebungsvariable ist „LD_PRELOAD“.[22] Durch das Exportieren von

```
LD_PRELOAD=/lib/libcl.so
```

veranlassen wir den Linker zuerst in der angegebenen Bibliothek „libcl“ nach aufzulösenden Symbolen zu suchen, bevor er die Bibliotheken benutzt, die standardmäßig durch „LD_LIBRARY_PATH“ bzw. „/etc/ld.so.conf“ spezifiziert werden. Damit sind wir in der Lage, beliebige Symbole aus der „glibc“ zu überschreiben, in dem wir eine Bibliothek erstellen und diese beim Programmaufruf durch „LD_PRELOAD“ dem Linker übergeben.[23] Überschriebene Symbole aus der Systembibliothek, die wir in unserer Bibliothek benutzen wollen, laden wir dynamisch mit den Befehlen „dl_open“ und „dl_sym“ nach. Für Details bezüglich des Linkens sei auf [Lev99] und [Whe00] verwiesen.

Die Struktur der Bibliothek „cllib“

Beim Aufruf einer Anwendung mit „PRE_LOAD“ wird die Bibliothek „cllib“ wie oben beschrieben vor dieser initialisiert. Sie hängt sich dadurch zwischen die Anwendung und die Systembibliotheken, wie in Abb. 4.6 dargestellt ist. Die abgefangenen Systemaufrufe sind in den bis auf das Präfix „cl_“ gleichnamigen Mo-

[21] Zu dieser Simulation gibt es noch keine veröffentlichten Ergebnisse. Der Programmcode befindet sich unter http://www.informatik.uni-frankfurt.de/~loizides/reiserfs/cllib.tar.gz. Es empfiehlt sich nach dem Entpacken die Datei „README“ zu lesen.

[22] Siehe auch „LD_DEBUG=help“ für weitere Informationen.

[23] Wiederum liegt der Vorteil auf Hand, es ist weitaus einfacher, eine eigene Bibliothek zu erstellen als in die überaus komplexe Systembibliothek einzugreifen.

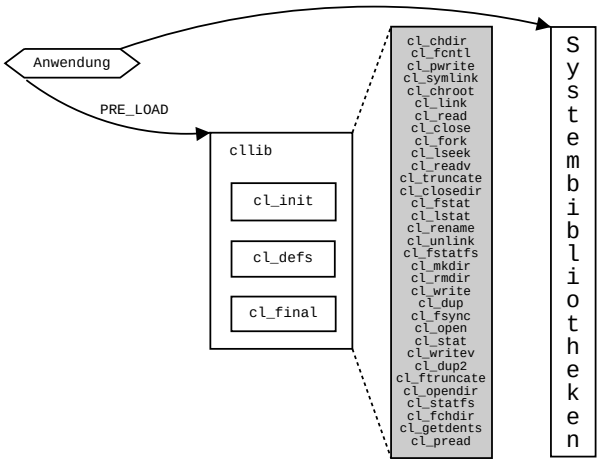

Abbildung 4.6: Die Bibliothek „cllib“

dulen definiert. Beim Start werden in der Funktion „cl_init“ die in Abb. 4.7 gezeigten Umgebungsvariablen ausgelesen, die das Verhalten der Bibliothek steuern. „CL_OUTPUT_NAME“ gibt den Pfadnamen und das Präfix für die Dateien an, in die die verfolgten Aufrufe gespeichert werden. Das Speicherformat einer Operationszeile besteht aus der Angabe eines Operationenzählers[24], eines Operationskürzels, einer Beschreibung der Operation und den Argumenten des jeweiligen Befehls, gefolgt von dem Ergebnis der Operation. Für einen Leseoperation von 4096 Byte aus dem Dateideskriptors mit der Nummer Sechs und dem Rückgabewert 321 sieht die eingetragene Zeile beispielsweise wie folgt aus:

```
1490287 6 read 6 4096 = 321
```

Mit „CL_MAX_ACTION_COUNTER“ wird angegeben, wie weit der Aktionszähler hochgezählt wird, bevor er wieder bei Eins startet. Sinnvollerweise gibt man hier einen sehr hohen Wert an. Mit „CL_OUTPUT_BUFFER_SIZE“ kann die Speicher-

```
export CL_OUTPUT_PNAME="/tmp/runtest-"  # Traces in Datei speichern
export CL_MAX_ACTION_COUNTER="1000000"  # Aktionen bis 1000000 zählen
export CL_OUTPUT_BUFFER_SIZE="64000"    # Zwischenpuffern von 64000 Byte
export CL_STATUS_PNAME="stderr"         # Statusberichte in Datei speichern
```

Abbildung 4.7: Die Umgebungsvariablen zum Steuern der Bibliothek

größe zum Zwischenpuffern eingestellt werden. Da die Bibliothek zur Vermeidung von Endlosschleifen keine höhere Ausgaberoutinen der Systembibliothek benutzen darf, muß sie die gepufferte Ausgabe selbst implementieren. Mit „CL_STATUS_PNAME“ gibt man an, auf welche Weise in der Bibliothek erkannte Fehler ausgegeben werden sollen.

[24]Wir nennen diesen auch „Aktionenzähler“.

Wiedergabe der Aufzeichnungen

Zur Wiedergabe des Snapshots wird das Programm „replay -d wdir -f sfile“ benutzt. Es liest aus der Datei „sfile“ den mit dem Programm „rekdir“ erzeugten Snapshot ein[25] und erzeugt die Kopie unter dem Verzeichnis „wdir“. Zur Auswertung und Simulation der Spuren dient das Programm „fsim“. Diesem übergibt man die aufgezeichneten Dateien in der Form „fsim -d wdir dat1 ... datN“. Um die Wiedergabe auf bestimmte Verzeichnisse mit ihren Unterverzeichnissen einzuschränken, muß man die gewünschten Verzeichnisse in „CL_MON_DIRS“ durch Doppelpunkte getrennt exportieren.

```
                Real Time    CPU Time      #
-------------------------------------------------
      open  1: 5.859928e+01 9.780000e+00 610848
     close  2: 4.026687e+01 7.350000e+00 610847
     write  3: 1.932119e+01 2.910000e+00 157904208
    pwrite  4: 0.000000e+00 0.000000e+00 0
    writev  5: 0.000000e+00 0.000000e+00 0
      read  6: 5.951661e+00 9.800000e-01 1527120
     pread  7: 0.000000e+00 0.000000e+00 0
     readv  8: 0.000000e+00 0.000000e+00 0
      stat  9: 5.990847e+00 9.100000e-01 76356
     fstat 10: 3.012778e+01 4.520000e+00 305424
     lstat 11: 6.499901e+00 1.130000e+00 76356
  truncate 12: 7.381768e+00 1.220000e+00 76356
 ftruncate 13: 5.586310e+00 9.800000e-01 76356
    rename 14: 8.467337e+00 1.200000e+00 76356
     chdir 15: 6.292020e+01 9.970000e+00 687199
     mkdir 16: 8.312465e+00 1.200000e+00 76356
     rmdir 17: 8.382329e+00 1.140000e+00 76356
    fchdir 18: 0.000000e+00 0.000000e+00 0
   opendir 19: 0.000000e+00 0.000000e+00 0
  closedir 20: 0.000000e+00 0.000000e+00 0
   readdir 21: 0.000000e+00 0.000000e+00 0
    chroot 22: 0.000000e+00 0.000000e+00 0
       dup 23: 5.099594e+00 8.700000e-01 76356
      dup2 24: 0.000000e+00 0.000000e+00 0
    statfs 25: 0.000000e+00 0.000000e+00 0
   fstatfs 26: 0.000000e+00 0.000000e+00 0
     fsync 27: 0.000000e+00 0.000000e+00 0
     fcntl 28: 1.306591e+01 2.470000e+00 152712
  getdents 29: 0.000000e+00 0.000000e+00 0
      link 30: 0.000000e+00 0.000000e+00 0
   symlink 31: 7.972190e+00 1.130000e+00 76356
    unlink 32: 3.351588e+01 5.000000e+00 381780
     lseek 33: 0.000000e+00 0.000000e+00 0
```

Abbildung 4.8: Eine Beispielsausgabe von „fsim“

Wie bereits erwähnt, ist die Bibliothek und vor allem das Auswertungsprogramm „fsim“ noch in der Entwicklungsphase. Momentan kann „fsim“ noch nicht sinnvoll

[25] Zum Erzeugen des Snapshots empfiehlt es sich, das Skript „getdirs.sh“ zu benutzen.

die Auswertungsdateien verarbeiten, wenn die untersuchte Applikation aus mehreren Prozessen besteht.[26] Bei der Implementierung der Bibliothek haben wir ein Beispielprogramm „testprg“ geschrieben, um dieses mit der Bibliothek zu verfolgen. Das Testprogramm erzeugt in einer Schleife ein Verzeichnis, erzeugt und löscht darin eine Menge von Dateien, führt einige weitere Dateisystemoperationen durch und löscht am Ende das erzeugte Verzeichnis wieder. Das Ergebnis der Simulation durch „fsim“ ist in Abb. 4.8 gezeigt. Jede Zeile gibt die Anzahl der akkumulierten Sekunden in Echt- und in CPU-Zeit, sowie die Anzahl der Aufrufe oder je nach Befehl die Menge der Bytes aus.

4.4 Messungen, Ergebnisse und Diskussion

Wir haben im Verlauf der letzten sechs Monate eine Reihe von Alterungssimulationen durchgeführt und dabei die Meßmethoden und Hilfsprogramme ständig erweitert und verbessert. Gleichzeitig ist die Entwicklung des LINUX-Kernels von Version 2.4.5 auf 2.4.16 vorangeschritten. Die Änderungen an der virtuellen Speicherverwaltung sind teilweise dramatisch gewesen, wie wir in Abschnitt 3.1.2 angedeutet haben (siehe auch [Bar01b]). Auf die dadurch auftretenden Probleme bei den Messungen sind wir bereits in [Loi01a] eingegangen. Die Entwicklung der Methoden und Programme sowie ältere Ergebnisse sind unter [Loi01e] zu finden. Wir beschreiben hier nur die neusten Messungen auf dem LINUX-Kernel in der Version 2.4.10.

4.4.1 Testsystem

Alle Simulationen sind auf dem folgenden System durchgeführt worden:

- Die Hardware besteht aus einem IBM-PC/AT kompatiblen Personalcomputer, der unter anderem die folgenden Komponenten beinhaltet:
  - AMD Duron 650 MHz
  - 128 MB RAM
  - 40 GB IDE-Festplatte
  - U2W-SCSI-Adapter (Adaptec 2940)
  - 9.1GB UW-SCSI-Festplatte (IBM Ultrastar DNES-309170W).
- Die Software besteht aus den Programmen, die bei der LINUX-Distribution von SuSE 7.2 standardmäßig installiert werden. Den Betriebssystemkern haben wir nachträglich durch den Standardkernel in der Version 2.4.10 ersetzt. Wir haben diesen durch einige Patches erweitert, um die Journaling-Dateisysteme

[26] Für jeden Prozeß wird eine eigene Ausgabedatei erzeugt. Diese Dateien müssen derzeit per Hand zusammengeführt und mit „sort“ nach den Aktionszählern aufsteigend sortiert werden. Dabei verlieren wir leider die Informationen über das aktuelle Arbeitsverzeichnis der verfolgten Prozesse. Daher muß „fsim“ so abgeändert werden, daß es die Eingabedateien parallel verarbeitet. Alternativ kann auch bei der Ausgabe die Prozeßnummer in jeder Zeile zusätzlich zum Aktionszähler notiert werden. Damit läßt sich dann in „fsim“ eine Liste der Zuordnungen zwischen aktuellem Verzeichnis des verfolgten Prozesses und seiner Prozeßnummer führen.

JFS und XFS als Module einbinden zu können.[27] Zum Kompilieren verwenden wir den Compiler „gcc“ in der stabilen Version 2.95-3.

Das System läuft im Mehrbenutzerbetrieb ohne X-Windows im Init-3-Runlevel, wobei nur die nötigsten Dienste –wie beispielsweise ein Sshd-Daemon zur Kommunikationsermöglichung über das Netzwerk– aktiv sind. Andere messungsverfälschende Dienste, vor allem der Crond-Deamon und somit die über ihn periodisch ausführbaren Wartungsarbeiten, sind deaktiviert. Zusätzlich schränken wir das System dahingehend ein, daß ihm kein Auslagerungsbereich auf Partitionen der Festplatten zur Verfügung steht. Damit kann „kswapd“ im Fall von Speichermangel keine Daten auslagern. Den Arbeitsspeicher reduzieren wir künstlich auf 32 MB, um dadurch die Wirkung des Seiten- bzw. Pufferspeichers auf Performanzmessungen zu vermindern.[28]

Die installierte Software befindet sich vollständig auf der IDE-Platte in verschiedenen mit Ext2 formatierten Partitionen. Die SCSI-Festplatte kann somit völlig unabhängig zum Testen verschiedener Partitionsgrößen und Dateisysteme benutzt werden, ohne die Stabilität des gesamten Systems zu gefährden. Bei den folgenden Simulationen benutzen wir eine oder zwei Partitionen der Größe von 1 GB am Anfang der Platte, also „/dev/sda1“ und „/dev/sda2“. Diese werden im System unter „/mnt/scsi1“ und „/mnt/scsi2“ eingehängt.

Beim Formatierten einer Partition für Reiser und XFS verwenden wir die Standardeinstellung.[29] Bei Ext2 benutzen wir zur besseren Vergleichbarkeit mit den anderen Systemen 4 KB Blöcke, reservieren 350000 Inodes[30] und erlauben gleichzeitig die maximale Benutzung des Dateisystems für ordinäre Benutzer. Daher lautet der Formatierbefehl:

```
mke2fs -b 4096 -N 300000 -m 0 /dev/sda1
```

Wenn nicht anders angegeben, benutzen wir beim Einhängen keine speziellen Optionen des VFS oder der Dateisysteme. Sprechen wir von Reiser ohne Tails, dann meinen wir das Reiser-Dateisystem mit der Einhängeoption „notail“. Wir behandeln dieses, als sei es eigenständiges Dateisystem. In der Regel beschränken wir uns bei der Darstellung von Ergebnissen auf die Reiser-Systeme mit und ohne Tails, sowie auf Ext2.

4.4.2 Eichtests

Wir besprechen im folgenden die Simulationen zweier Eichmessungen, bei denen in der Änderungsphase im wesentlichen die Zeile

[27] Für JFS benutzen wir den Patch „jfs-2.4.7-1.0.7-patch“ bzw. „jfs-2.4.7-1.0.8-patch“, für XFS den Patch „linux-2.4.10-xfs-2001-10-03.patch“, die man unter den Webadressen der beiden Dateisysteme findet.

[28] Die Kerneloption dazu lautet „mem=32“. Sie bewirkt, daß der Kernel unabhängig von der Größe des physikalischen Hauptspeichers nicht mehr als 32 MB adressiert.

[29] Die erwähnten Versionen von JFS und auch die nachfolgende Version 1.0.8 lief auf dem Testsystem nicht stabil und führte in jedem Simulationsversuch zu einem Systemabsturz.

[30] Wie in Abschnitt 3.5.3 erläutert, belegt eine Ext2-Inode 128 Byte. Somit verbraucht die statische Inode-Tabelle ca. 9000 Blöcke. Damit ist der freie Speicherplatz in etwa so groß wie bei den Journaling-Systemen.

```
agesystem3 -m1 -d/mnt/scsi1 -u0:75 -c$cparam -l6:$lparam
```

für eine Verzeichnisstruktur von “cparam“ und eine Verteilung der Dateigrößen von „lparam“ benutzt wird. Die beiden Messungen unterscheiden sich in der Wahl der Verzeichnishierarchie. Für „cparam=1:1“ werden alle Dateien in einem Verzeichnis erzeugt. Für „cparam=100:10000“ werden die Dateien in 10000 Verzeichnissen gleichmäßig erzeugt, von denen jeweils 100 ein gemeinsames Vaterverzeichnis haben.

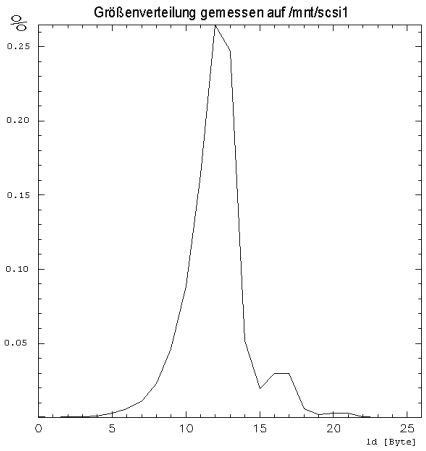

Abbildung 4.9: Die Größenverteilung der Dateien beim Eichtest

Es werden solange Dateien erzeugt, bis die in „/mnt/scsi1“ eingehängte Partition zu 75% gefüllt ist. Die gewählte Verteilungsfunktion ist dabei durch

```
lparam=0:4096:65536:4096:65536:1048576
```

parametrisiert und in Abb. 4.9 gezeigt.[31] Dateien mit einer Größe von zwischen 1 KB und 16 KB Byte sind demnach am wahrscheinlichsten, gefolgt von Dateien zwischen 32 KB und 64 KB. Zu etwas weniger als 1% sind auch Dateien in der Größenordnung von Megabyte vorhanden.

Das Ergebnis der Eichmessungen ist in Abb. 4.10 gemittelt über drei Läufe für Ext2, XFS, Reiser (RFS) und Reiser mit der Option „notail“ (RFS-notail) dargestellt. Die 1σ-Unsicherheit des Mittelwerts ist dabei gestrichelt gezeichnet. Die dunkleren Balken beziehen sich auf die Messungen mit einem Verzeichnis, die helleren auf die mit 10000 Verzeichnissen. Die Leseperformanzmessungen bestehen jeweils aus zwei Balken, die durch einen schmalen, weißen Strich voneinander getrennt sind. Der Unterschied zwischen den beiden Balken ist subtil. Er resultiert aus unterschiedlichen Zeitmessungen: Beim linken Balken ist die akkumulierte Lesezeit, während beim rechten die gesamte Laufzeit des Test bei der Performanzberechnung berücksichtigt. Damit gibt der linke Balken jeweils die tatsächliche Leseperformanz an. Der

[31] Die Wahl der Parameter ist dabei so getroffen, daß die künstliche Verteilung der Verteilung von Dateien im Verzeichnis „/usr“ ähnelt. Dies geschah zu einem Zeitpunkt, zu dem „agesystem“, der Vorgänger von „agesystem3“, nicht mit gemessenen Verteilungen umgehen konnte.

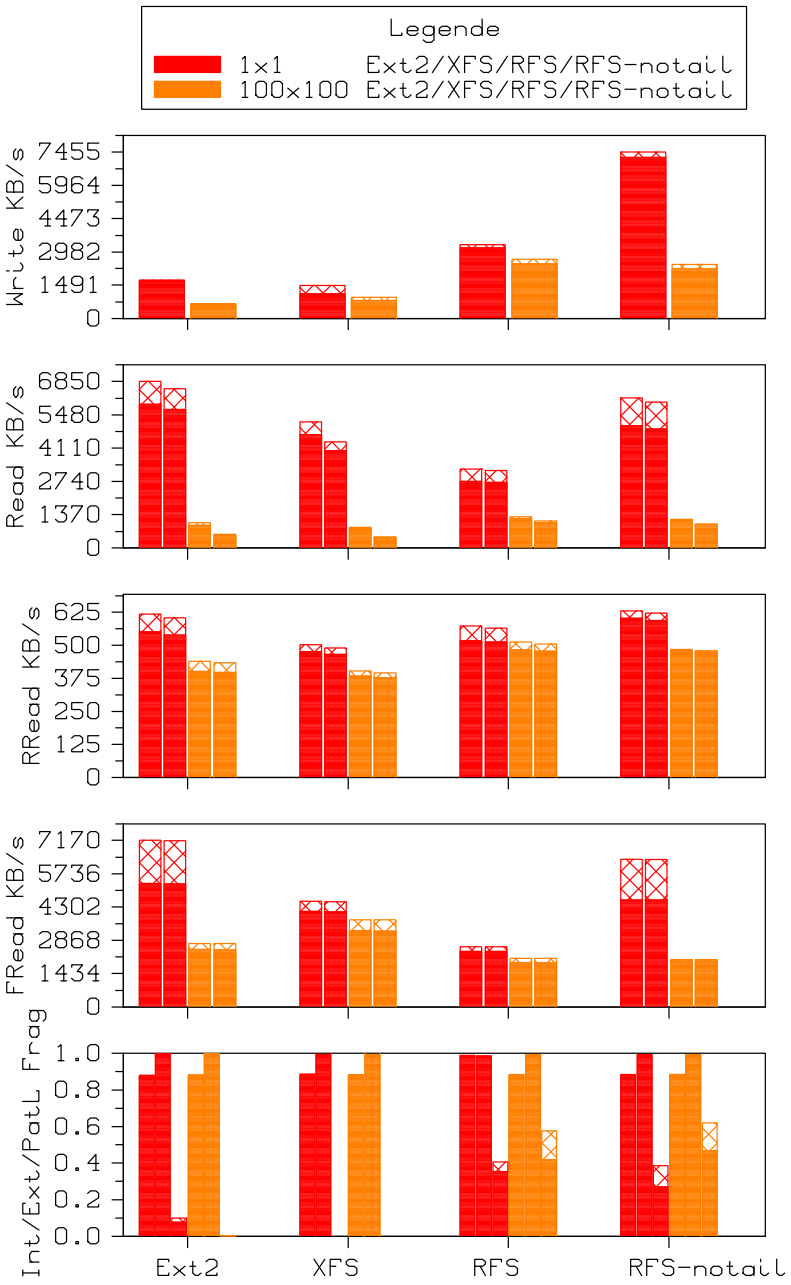

Abbildung 4.10: Ergebnis der Eichmessungen gemittelt über drei Läufe

rechte dagen zeigt das Sinken der Performanz durch die Zeit, die das Programm mit etwas anderem als Lesen verbracht hat.[32] Die unterste Zeile zeigt für jede Messung drei Balken: die interne Fragmentierung, die externe Standardfragmentierung und den Logarithmus der fragmentierten Länge[33].

Diskussion der Ergebnisse

Es ist deutlich zu sehen, daß bei der gewählten Größenverteilung Reiser ohne Tails gefolgt von Ext2 und XFS mit Einschränkung beim Schreiben im Mittel am performantesten abschneiden. Das normale Reiser ist nur beim Schreiben schneller als die beiden letztgenannten.

Betrachten wir nun die einzelnen Tests aus Abb. 4.10 getrennt. Bei der Schreibperformanz fällt auf, daß die Reiser-Dateisysteme deutlich schneller arbeiten als die beiden anderen. Sie profitieren davon, daß bei einer derartigen Schreiblast die Daten und Metadaten der Verzeichnisse im Baum sowohl im Hauptspeicher als auch auf der Platte sehr effizient behandelt werden. Deutlich ist auch der Performanzunterschied zwischen Reiser ohne Tails und mit Tails zu erkennen, da der Baum weniger oft balanciert werden muß. Beim Schreiben in 10000 Verzeichnissen sinkt die Performanz bei allen Dateisystemen signifikant, da die Inodes der Verzeichnisse nicht im Inode-Cache zwischengespeichert werden können. Dadurch müssen im Unterschied zu einem Verzeichnis die Datenblöcke der Verzeichnisse und ihre Inodes mehrmals gelesen und geschrieben werden. Der Einbruch beim Reiser ohne Tails ist derart stark, daß sogar das normale Reiser beim Schreiben schneller ist. Dies liegt daran, daß im Mittel viele kleine Dateien vorkommen, und die Kosten für das Packen der Knoten im Hauptspeicher geringer sind als die Suchzeiten auf und die Übertragungszeiten zur Platte.

Um die Leseperformanz zu diskutieren, stellen wir diese in Abb. 4.11 in Form von Histogrammen dar.[34] Dadurch wird für die einzelnen Lesetests deutlich, welche Dateigrößen von den Dateisystemen bevorzugt werden. Beim rekursiven Lesen jeder Datei innerhalb der Partition ist Ext2 gefolgt von Reiser ohne Tails für den Fall eines Verzeichnisses am schnellsten. Für den Fall von 10000 Verzeichnissen sehen wir, daß die Performanz generell um die Hälfte sinkt. Die Kosten der Suchzeiten für das Lesen der Metadaten und Daten der Verzeichnisse schlagen hier durch. Reiser ohne Tails ist hier deutlich besser als die anderen Systeme. Aber auch das normale Reiser offenbart seine Fähigkeiten. Für beide Verzeichniskonstellationen zeigt es ein konstantes Verhalten für kleine Dateien. Dies ist der Vorteil der formatierten Knoten, in denen die Dateien dicht gepackt auf einmal gelesen werden.

Beim zufälligen Lesen der Dateien ist offensichtlich, daß die Performanz nicht besonders von der Verzeichnisstruktur abhängt. Durch den zufälligen Zugriff wird die Lesezeit dominiert durch die langsamen Suchzeiten der Platte. Beim Lesen jeder

[32] Genauer gesagt, mit etwas anderem als Dateien zu öffnen, zu lesen und zu schließen.

[33] Damit ist in dieser Darstellung $\log 1 = 0$ der optimale Wert für die fragmentierte Länge.

[34] Wir haben aus Übersichtsgründen dabei Dateigrößen über 512 KB nicht berücksichtigt. Die Tendenzen für große Dateien sind bereits bei 512 KB zu erkennen. Für größere Dateien steigt der Durchsatz unabhängig von der Art des Lesetests, aber bei der gewählten Verteilung auch die Fehlerbalken. XFS gefolgt von Reiser ohne Tails und Ext2 sind für Dateien größer als 512 KB in der genannten Reihenfolge am performantesten.

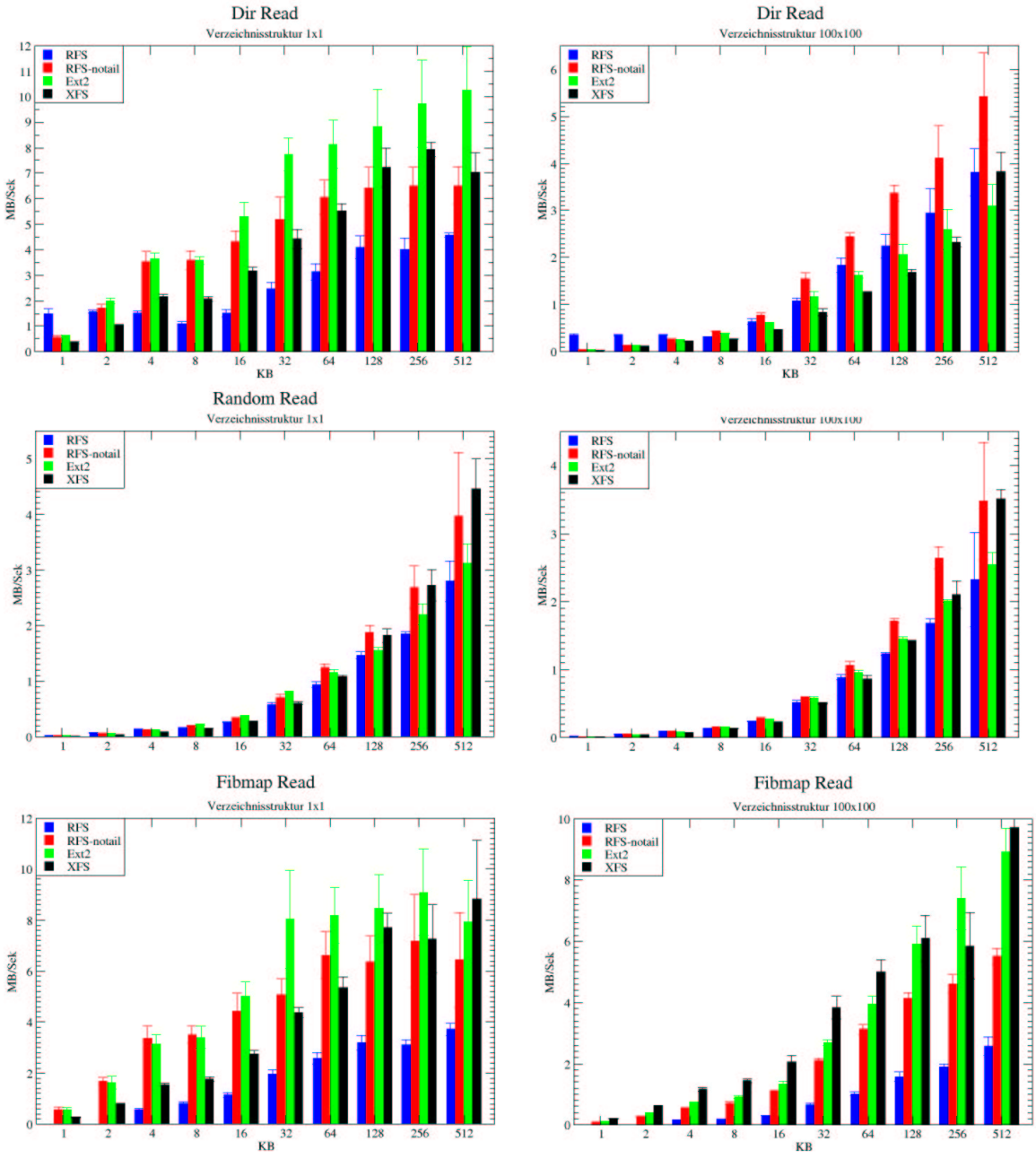

Abbildung 4.11: Histogramme der Eichmessungen gemittelt über drei Läufe

Datei muß der Kopf im Mittel über die halbe Partition bewegt werden. Trotzdem macht sich das Lesen der Metadaten für die Verzeichnisse um bis zu 25% bemerkbar. In beiden Fällen ist deutlich zu sehen, wie die Suchzeiten gerade für kleine Dateien teuer sind. Die Reiser-Dateisysteme sind bei der gewählten Verteilungsgröße im Durchschnitt etwas schneller. Es wird aber auch deutlich, daß Ext2 gerade in dem klassischen UNIX-Bereich zwischen 16 KB und 64 KB sehr performant ist, während XFS für große Dateien am besten abschneidet.

Den künstlichen Lesetest „FRead“ haben wir uns ausgedacht, weil er sensibel reagiert sollte, wenn die Platte beim Lesen von Daten zusätzliche Kopfbewegungen ausführen muß. Da die Dateien mit aufsteigenden logischen Blocknummern übergeben werden, zeigen eventuelle Performanzunterschiede deutlich, daß Metadaten oder im Fall von Fragmentierung zusätzlich Datenblöcke von Dateien gesucht werden müssen.[35] Für ein Verzeichnis schneidet Ext2 und für die 10000 Verzeichnisse XFS am besten ab. Das normale Reiser ist erst für Dateien gezeigt, die mindestens einen indirekten Block belegen. Da „fibmap“ für reine Tails nur eine Null liefert, können diese nicht sinnvoll in die Liste einsortiert werden. Wir lassen sie daher bei der Messung einfach weg. Es ist aber offensichtlich, daß das normale Reiser durch die direkten Elemente im Baum bei diesem Lesetest in der Performanz einbricht. Reiser ohne Tails dagegen ist in seiner Struktur dem Ext2 ähnlich und zeigt auch eine vergleichbare Performanz.

Zum Schluß vergleichen wir noch die Maße für die Fragmentierung. Bei der internen Fragmentierung ist Reiser natürlich mit Abstand führend, weil es keinen Platz durch im Mittel halbfreie letzte Blöcke verschwendet. Die anderen Dateisysteme haben praktisch alle einen vergleichbaren Wert von 85%. Demnach wird 15% an freiem Speicherplatz verschwendet. Externe Fragmentierung ist im wesentlichen bei allen Systemen nicht vorhanden. Dies ist bei einem ungealterten Dateisystem auch zu erwarten. Besonders auffallend ist XFS, das in beiden Fällen für beide Maße den absolut optimalen Wert erreicht[36].

Für weitere Ergebnisse, sowie den Programmcode verweisen wir auf [Loi01h]. Andere Benchmarks wie beispielsweise ”Bonnie++”[37] und ”Mongo”[38] und andere sind unter [Loi01e] referenziert.

[35] Beim Lesen von beispielsweise wenigen, sehr großen und unfragmentierten Dateien erwarten wir einen Durchsatz in der Größenordnung dessen, was maximal mit dem Laufwerk zu erzielen ist.

[36] Dies ist auf die extentbasierte Blockallokation mit später Bindung (engl. „late block allocation“) zurückzuführen. Auf diese Eigenschaft von XFS sind wir im letzten Kapitel nicht eingegangen. Im Unterschied zu den anderen Dateisystemen unter LINUX kann XFS eine Speicherseite anfordern, ohne bereits die Zuordnung zu einer logischen Blocknummer auf dem Gerät festgelegt zu haben. Beim tatsächlichen, verzögerten Schreiben ist XFS dann in der Lage, große Extents zu bilden, da es die genaue Größe der Datei zu diesem Zeitpunkt bereits kennt. Diese Eigenschaft von XFS ist nur möglich, weil die Entwickler quasi eine eigene Speicherverwaltung für XFS unter LINUX implementieren.

[37] Siehe `http://www.coker.com.au/bonnie++/` für den Programmcode.

[38] Siehe `http://www.namesys.com/benchmarks/mongo/mongo_readme.html` für den Programmcode, Erläuterungen und Ergebnisse.

4.4.3 Appendtests

Wir besprechen im folgenden die Simulationen, bei denen ein Dateisystem durch ständiges Anhängen von Blöcken an jede Datei künstlich gealtert wird. Da die Blöcke reihum angefügt werden, wachsen die Dateien gleichmäßig um jeweils einen Block. Die entscheidende Zeile in der Änderungsphase ist

```
agesystem3 -m1 -d/mnt/scsi1 -u0:$u -c$cparam -n $n -o1089 -f -b4096
```

für eine Verzeichnisstruktur von "cparam" und insgesamt „n" im Dateisystem wachsenden Dateien. Dabei führen wir in dem Simulationsskript in einer Schleife über die Werte „u=25", „u=50", „u=75" und „u=99" (oder „u=100") vier Durchläufe der verschiedenen Phasen durch, so daß wir die Entwicklung für verschiedene Füllstände und Dateilängen verfolgen können. Für „n" wählen wir die Werte „n=1000", „n=5000" und „n=10000". Bei einer Partitionsgröße von 1 GB stehen ungefähr 250000 logische Blöcke mit einer Größe von 4 KB für Datenblöcke zur Verfügung. In Abb. 4.12 wird angegeben, wieviele Blöcke eine Datei am Ende der einzelnen Änderungsphasen belegt. Diese Zahl dient uns als Richtwert bei der Interpretation der Ergebnisse. Als Verzeichnisstruktur wählen wir „cparam=1:1", so daß alle Dateien in einem Verzeichnis liegen.

Dateien	25%	50%	75%	99%
1000	63	125	188	250
5000	13	25	38	50
10000	7	13	19	25

Abbildung 4.12: Die Anzahl der Blöcke einer Datei am Ende der Änderungsphase

Um die Alterung deutlich zu machen, kopieren wir nach jeder Änderungsphase die komplette Partition auf eine benachbarte und frisch formatierte Partition des gleichen Typs. Dazu benutzen wir den Befehl

```
cd /mnt/scsi1; cp -R . /mnt/scsi2
```

und führen anschließend dieselben Messungen durch, die auch an der gealterten Partition durchgeführt werden. Das kopierte Dateisystem stellt im oben genannten Sinn das Eichsystem für die Alterungquantifizierung dar.

Das Ergebnis für 1000 Dateien ist in Abb. 4.13 dargestellt.[39] Für jede Auslastung und jedes Dateisystem sind zwei Gruppen von gleichfarbigen Balken gezeigt. Bei den Lesetests besteht eine Gruppe aus zwei Balken, bei der Fragmentierung aus drei Balken. Der linke bezieht sich auf das gealterte, der rechte auf das kopierte Dateisystem. Bei den Lesetests wird jeweils die akkumulierte und die totale Lesezeit in der Performanz berücksichtigt. Die Angaben der Fragmentierungsmaße sind

[39] Die Abbildungen der Ergebnisse für 5000 und 10000 Dateien sind unter http://www.informatik.uni-frankfurt.de/~loizides/reiserfs/apptest.html#fres4 zu finden.

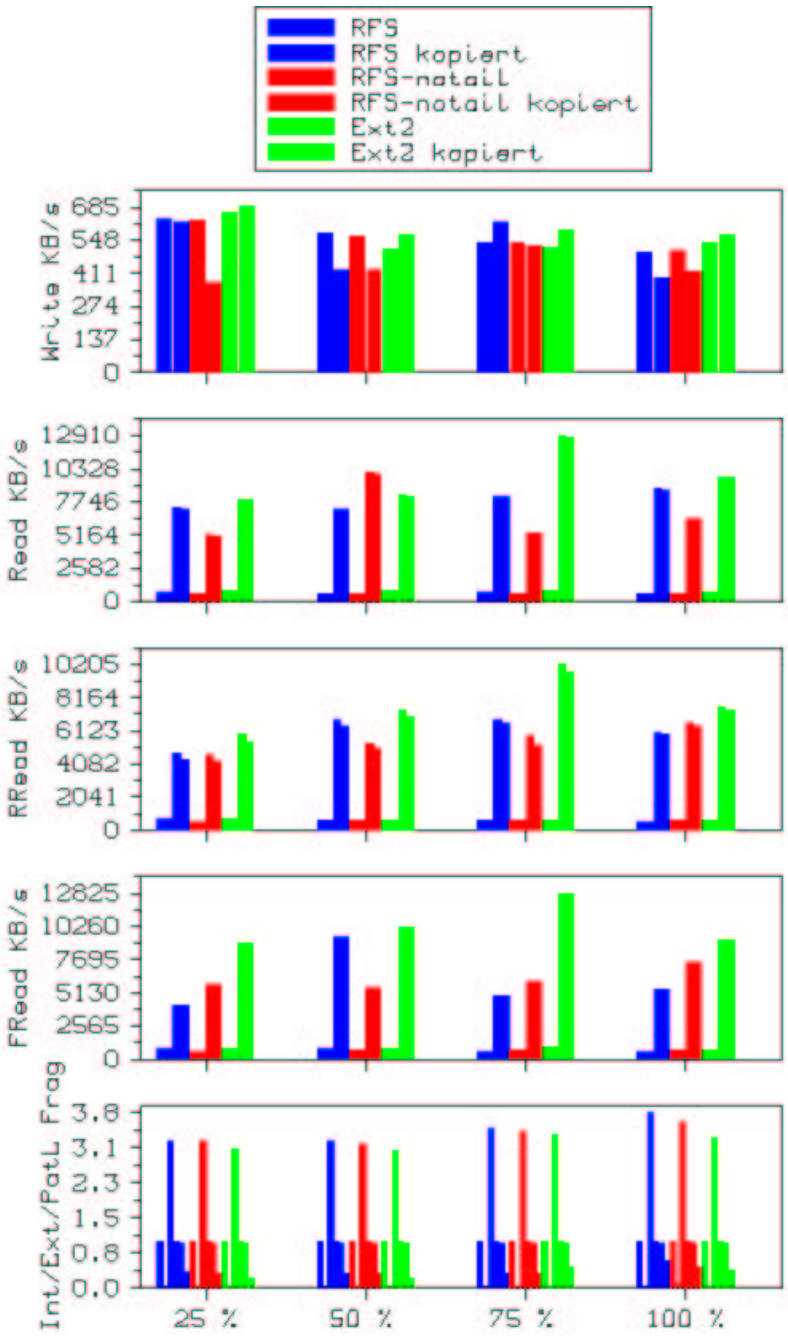

Abbildung 4.13: Das Ergebnis des Appendtests bei 1000 Dateien für die Dateisysteme im Vergleich

von links nach rechts das Maß für die interne Fragmentierung, der externen Standardfragmentierung und der fragmentierten Weglänge. Die Schreibperformanz des kopierten Systems haben wir als die Anzahl der kopierten Bytes geteilt durch die dafür benötigte Zeit angegeben. Sie läßt sich daher nicht mit der Schreibperformanz des fragmentierten Dateisystems vergleichen. Da beim Kopieren die fragmentierten Daten gelesen werden müssen, ist die Kopierzeit immer größer als die benötigte Lesezeit. Damit ist die Schreibperformanz der kopierten Partition immer niedriger als die Verzeichnisleseperformanz der fragmentierten Daten.

Diskussion der Ergebnisse

Es ist offensichtlich, daß die Dateisysteme durch die Art der Benutzung extrem fragmentiert sind. Die Performanz sinkt bei den Reiser-Dateisystemen um den Faktor Sechs bis Acht und bei Ext2 um bis zu dem Faktor Zehn.[40] Die externen Fragmentierungsmaße stehen mit diesem Performanzverlust in völliger Übereinstimmung. In den fragmentierten Systemen gibt es keine Datei, bei der benachbarte Blöcke benutzt werden. Die fragmentierte Weglänge beträgt ziemlich genau 1000. Wenn der freie Speicherplatz in der Partition knapp wird, steigt sie sogar noch etwas an. Sie steht damit in deutlichem Einklang mit der Art der Blockallokation. Wir wissen aus dem letzten Kapitel, daß sowohl Ext2 als auch Reiser in ihren Bitmaps zunächst nach dem nächsten freien Block suchen. Da die Dateien quasi gleichzeitig wachsen, finden sie freie Blöcke frühestens im Abstand von 1000 Blöcken.[41]

Beide Dateisysteme verwenden eine Preallokation von acht Blöcken. Wie wir im letzten Kapitel erläutert haben, ist diese aber nur gültig, solange die zur Datei gehörende Inode im Dcache vorhanden ist. Die Vorbestellung wird demnach bei unserem Szenario nach jedem Anhängen aufgehoben, da die Dateien jedesmal geöffnet und geschlossen werden. Es wäre interessant, die Preallokation der Dateisysteme über das VFS zu testen. Da allerdings pro Prozeß ohne Änderungen am Kernel nur 1024 Dateien gleichzeitig göffnet sein können, gehen wir statt dessen anders vor.[42] Wir führen vor dem ersten Anhängevorgang eine Art von Preallokation aus, indem wir reihum jede Datei mit ihrer maximal zu erwartenden Größe erzeugen. Vor der ersten Änderungsphase verkleinern wir die Dateien auf einen Block und beginnen dann mit dem Test in der oben beschriebenen Form. Damit sollten die zielbasierten Blockallokationsstrategien in der Lage sein, ein effizienteres Layout bei gleicher Belastung des Dateisystems zu erzeugen. Die Ergebnisse der Appendtests mit Preallokation sind in Abb. 4.14 für 1000, in Abb. A.2 für 5000 und in Abb. A.3 für 10000 Dateien dargestellt.

Anhand von Abb. 4.14 sieht man deutlich, daß im Fall von 1000 Dateien alle Datei-

[40] Man beachte, daß wir wegen der Dauer einer Simulation nur eine Messung pro Dateisystem durchgeführt haben und somit keine Aussagen über die Abweichungen machen können. Bei den fragmentierten Systemen ist diese mit Sicherheit gering. Bei den kopierten Systemen wissen wir nach Abb. 4.11, daß relative Fehler von bis zu 20% bei Dateien der Größe zwischen 32 KB und 512 KB vorkommen.

[41] Auch die Organisation des Mediums in Gruppen hilft Ext2 bei diesem Szenario nicht, da auch bereits die entsprechenden Blöcke innerhalb der nächsten Gruppe belegt sein werden.

[42] Man kann das Limit mittlerweile durch „/proc/sys/fs/file-max“ als Administrator einstellen, aber die Implikationen für den Inode-Cache sind nicht offensichtlich.

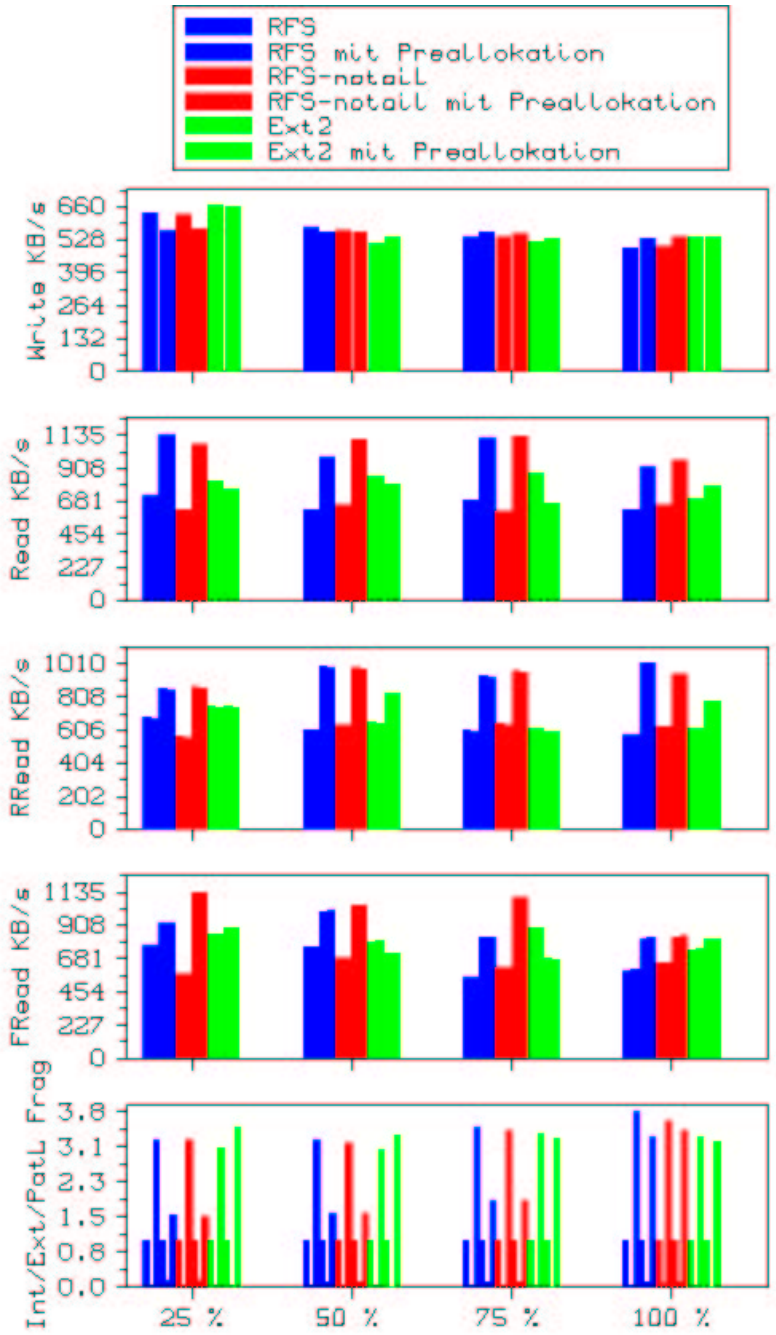

Abbildung 4.14: Die Ergebnisse des Appendtests für 1000 Dateien in einem Verzeichnis mit und ohne Preallokation

systeme von der einmaligen Preallokation profitieren. Durch die Preallokation beim Lesen sind Reiser ohne Tails zwischen 25% und 100% und das normale Reiser zwischen 30% und 80% schneller als zuvor. Ext2 dagegen profitiert nur bis zu 25% und ist in einem Fall sogar langsamer. Dieser Ausreiser ist für uns momentan nicht zu erklären. Wir kommen weiter unten darauf zurück. Beim Schreiben überwiegen anfangs bei den kleinen Dateien die Kosten der durch die Preallokation längeren Wege auf der Platte. Am Schluß setzen sich bei den Reiser-Systemen sogar die Vorteile der Preallokation durch.

Beim Betrachten von Abb. A.2 für das Ergebnis von 5000 Dateien fällt auf, daß nur noch die Reiser-Dateisysteme von der Preallokation profitieren. Ihr Leseperformanzgewinn beträgt zwischen 35% und über 200%. In extremen Fällen sind sie sogar bis zu Faktor Fünf schneller. Die hohe Schreibperformanz von Ext2 im ersten Durchgang erklärt sich dadurch, daß die Dateien in diesem Fall genau durch die direkten Blöcke einer Inode addressiert werden können. Beim Schreiben macht sich in diesem Fall wieder die Distanz der Datenblöcke zu den Inodes bemerkbar.

Abb. A.3 zeigt das Ergebnis für 10000 Dateien. Hierbei wird noch deutlicher, daß die Reiser-Systeme aus der Preallokation einen Nutzen ziehen. Selbst bei voller Auslastung sind sie um mehr als Faktor Zwei schneller als ohne Preallokation. Ext2 dagegen zeigt wie schon bei 5000 Dateien keinen Performanzgewinn durch die Preallokation. Bei sehr kleinen Dateien ist die Schreibperformanz bei allen Systemen unverhältnismäßig groß. Ext2 profitiert wieder durch die direkten Blöcke und durch die geringe Distanz zwischen Inodes und Datenblöcken.

Zusammenfassung

Aber auch die preallokierten Systeme sind teilweise stark von der Fragmentierung betroffen. Ein Blick auf Abb. A.4 für Ext2, Abb. A.5 für Reiser und Abb. A.6 für Reiser ohne Tails macht dies insbesondere für Ext2 deutlich. Wir beschränken uns im folgenden bei der zusammenfassenden Betrachtung der Performanz auf die zufälligen Lesetests. Je nach Größe der Dateien bewirkt die Fragmentierung bei Ext2 einen Performanzverlust von einem Faktor 3 bis 14, wenn man dabei auch die kopierten preallokierten Partitionen berücksichtigt. Für Ext2 ist die Preallokation ganz offensichtlich eher hinderlich bei der Suche nach freien Speicherplatz. Die Ursache liegt in der Verteilung der Datenblöcke und Inodes über die verschiedenen Blockgruppen, die bei Ext2 die Preallokation sinnlos machen. Auffällig ist aber, daß statistisch in mehr als 65% der Fälle die Kopie der preallokierten Partition performanter ist als die Kopie der einfach fragmentierten Partition. Wir haben dafür keine Erklärung. Es kann auch nur ein statistisches Artefakt sein oder mit der Implementierung des Kopierbefehls und damit der Reihenfolge des Lesens der Dateien beim Kopieren zusammenhängen. Darüber hinaus korrelieren die Fragmentierungswerte nicht in jedem Fall mit der gemessenen Performanz.

Für die Reiser-Systeme ergibt sich ingesamt ein sehr geschlossenes Bild. Beide profitieren für alle Dateilängen aus Abb. 4.12 von der Preallokation. Die Fragmentierung bewirkt je nach Größe der Dateien einen Performanzverlust von einem Faktor Fünf bis Sieben. Die Preallokation hat je nach Größe der Dateien einen Performanzgewinn von 50% bis 200% zur Folge. Die Fragmentierungsmaße korrelieren sowohl für

die fragmentierte als auch für die preallokierte Partition mit der Performanzmessung. Darüber hinaus korrespondiert der Meßwert des fragmentierten Pfads in allen Fällen außer bei den komplett gefüllten Partitionen mit der Anzahl der Dateien im System. Dieses Ergebnis deckt sich mit der Allokationsstrategie von Reiser. Die Bitmaps werden aufsteigend nach freien Blöcken durchsucht. Durch die Preallokation können die Dateien dadurch während des Tests vorwiegend in aufsteigender Richtung wachsen. Für weitere Ergebnisse, sowie den Programmcode verweisen wir auf [Loi01g].[43]

4.4.4 Alterungstests

Nachdem wir die Eichtests als „best case“ und die Appendtests als „worst case“ für Dateisysteme untersucht haben, kommen wir nun zum dritten vorgestellten Typ von Simulationen. Bei diesen Alterungstests versuchen wir durch eine Folge von Erzeugungs- und Löschvorgängen in der Änderungsphase ein Dateisystem relativ moderat zu altern. Die entscheidende Stelle in dem Alterungsskript besteht aus dem folgenden Ausschnitt:

```
if test "$run" = "0"; then
 agesystem3 -m 1 -c 1:1 -d /mnt/scsi1 -u 0:$gaugeu -l $lparam
else
 agesystem3 -m 2 -c 1:1 -d /mnt/scsi1 -u $ageu -l $lparam -n $nrun
fi
```

Die Simulationsschritte werden in „run“ gezählt. Für „run=0“ zu Beginn des Test wird die Partition bis zu „gaugeu“ Prozent geeicht. Für späteren Zeitpunkte wird die Partition im durch „ageu“ gegebenen Intervall durch Einfüge- und Löschoperationen nach dem in 4.3.1 erläuterten Verfahren gealtert. Dabei bestimmt „nrun“ die Anzahl der pro Simulationszeit zu erzeugenden Dateien. Als Verzeichnisstruktur wählen wir für den Moment ein Verzeichnis, in dem die Dateien erzeugt und gelöscht werden. Die generische Verteilungsfunktion parametrisieren wir durch „lparam=6:2048:65536:524288:65536:524288:4194304“. Die durch sie erzeugten Dateigrößen sind immer größer als 2 KB und im Mittel um den Faktor 4 bis 16 größer als bei der in Abb. 4.9 dargestellten Funktion. Wir wählen „gaugeu=65“ und „ageu=65:85“. Der Gedanke dabei ist, daß ein Dateisystem beim Anlegen in der Regel derart dimensioniert wird, daß es aus Platzverschwendungsgründen nicht dauerhaft relativ leer bleibt und aus Effizienzgründen aber auch nicht zu voll werden soll. Daher eichen wir die Partitionen bis zu einer Blockbelegung von 65% und lassen danach die Belegung zwischen 65% und 85% schwanken. Insgesamt führen wir 45 Änderungsphasen durch. Die Anzahl der pro Durchgang erzeugten Dateien ist in Abb. 4.15 dargestellt. Über die gesamte Simulation betrachtet, werden demnach nach der Eichung 60000 Dateien erzeugt und ungefähr auch wieder gelöscht.

Die Ergebnisse der Simulation sind für Ext2 in Abb. 4.16, für Reiser in Abb. 4.17 und für Reiser ohne Tails in Abb. A.7 abgebildet. In den Abbildungen sind auf

[43] Insbesondere unter http://www.informatik.uni-frankfurt.de/~loizides/reiserfs/apptest.html#meas4 sind weitere zusammenfassende Tabellen und Ergebnisse mit anderen Verzeichnisstrukturen zu finden.

„run“	0	1-20	21-30	31-43	43-44
„nrun“	Eichung	500	1000	2500	5000

Abbildung 4.15: Die Anzahl der Erzeugungen pro Durchlauf

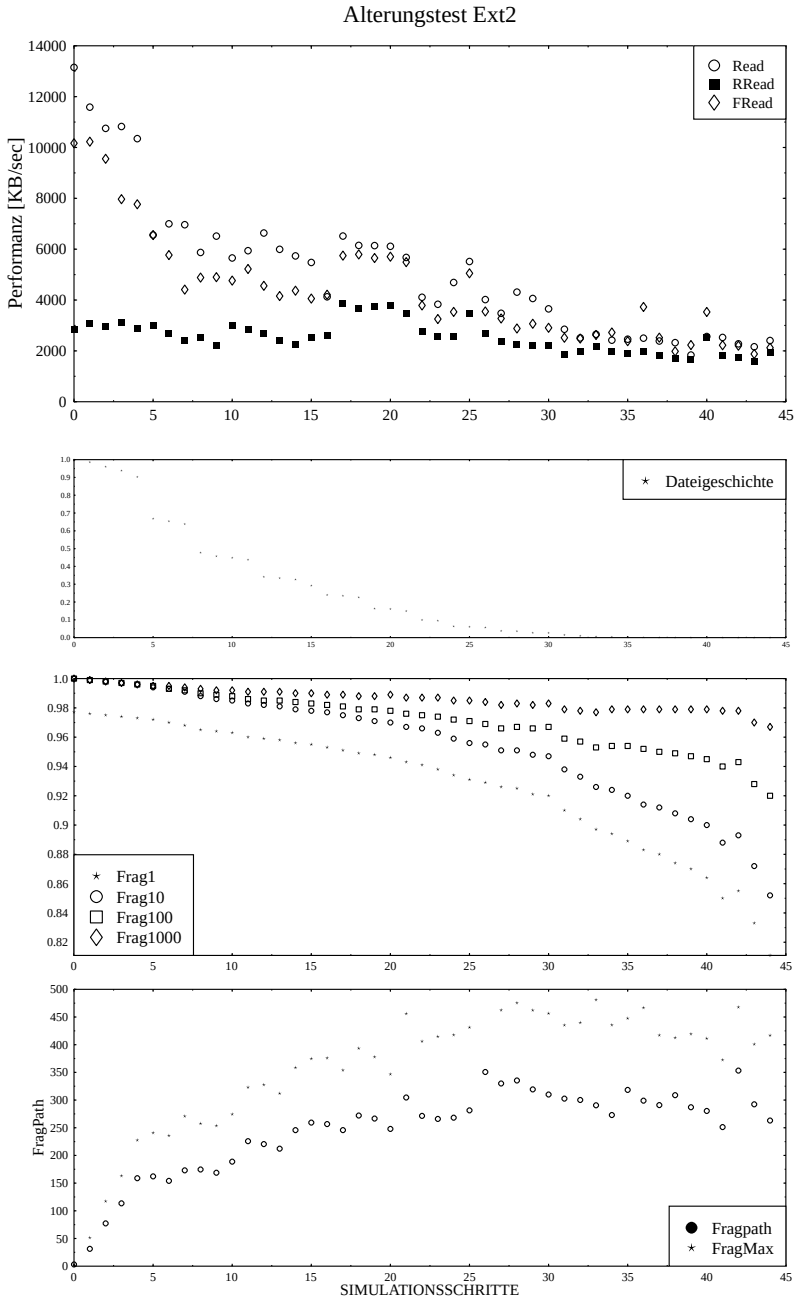

Abbildung 4.16: Das Ergebnis des Alterungstests für Ext2

der X-Achse immer die Simulationsschritte aufgetragen. Damit zeigen die einzelnen Graphen jeweils die zeitliche Entwicklung der Partition für die abgebildete Größe in Simulationsschritten. Im obersten Graphen sind die gemessenen Performanzkurven für die drei verschiedenen Lesetests dargestellt. Die Schreibperformanz kann wegen der Löschvorgänge schlecht als solche interpretiert werden, daher geben wir sie nicht an. Direkt unterhalb des Graphen für den Lesetests ist die Dateihistorie der Partition gezeigt. Bei dieser berechnen wir pro Simulationsschritt den Anteil der im Eichvorgang erzeugten Dateien bezogen auf alle Dateien der Partition. Nach der Eichung ist dieser Anteil natürlich Eins, gegen Ende der Simulation Null. Die unteren beiden Graphen zeigen den Verlauf der verschiedenen Fragmentierungsmaße.

Diskussion

Beim Betrachten der Performanzkurven sieht man, daß die Dateisysteme im Verlauf der Zeit an Performanz verlieren. Nimmt man den zufälligen Lesetest als Grundlage, so erreicht Ext2 am Ende der Simulation nach 60000 erzeugten Dateien eine Performanz von 80% bezogen auf die Werte, die es nach der Eichung erreicht hat. Bei den Reiser-Systemen ist der Verlust dramatischer. Sie erreichen nur etwa 60% ihres Eichwerts. Die anderen Lesetests reagieren naturgemäß sensitiver auf die Veränderungen am Dateisystem und zeigen daher dramatischere Verluste. Diese sind größtenteils durch die Durchmischung der Metadaten verursacht, was Messungen mit anderen Schrittweiten –beispielsweise 100 Erzeugungsvorgänge pro Durchgang– belegen. Die Maße korrelieren mit der Gesamttendenz und zeigen steigende Fragmentierung mit zunehmender Alterung. Am Ende der Simulation gibt das Standardmaß der externen Fragmentierung bei allen Dateisystemen einen Wert von 80% bis 85% an.

Wir betrachten nun die Systeme in einzelnen. Bei Ext2 in Abb. 4.16 fällt auf, daß es einen Zeitraum zwischen den Simulationsschritten 17 bis 21 gibt, bei der die Performanz plötzlich signifikant steigt. Für das zufällige Lesen liegen die Werte sogar um knapp 40% oberhalb des Eichwerts. Die Fragmentierungsmaße zeigen keine Korrelation dazu. Es ist nicht ganz klar, wodurch dieser Performanzsprung verursacht wird. Wir betrachten daher in Abb. A.1 die Standardfragmentierung und das Leseperformanz für den Zufallstest für die Dateigrößen einzeln. Es ist deutlich zu sehen, daß die Performanz über alle Dateigrößen ein sehr ähnliches Muster besitzt. Bei den Fragmentierungswerten ist dies nicht ganz so offensichtlich. Zwischen den erwähnten Werten von 17 und 21 Schritten ist das Blocklayout bei den Dateien kleiner 32 KB und bei den Dateien größer als 256 KB besser als in der Umgebung. Trotzdem ist der Performanzsprung nicht befriedigend erklärbar. Da wir keine Informationen über die Metadaten haben, ist zu vermuten, daß ein großer Teil auch auf eine bessere Namensauflösung zurückzuführen ist.

Bei Reiser in Abb. 4.17 ist der kontinuierliche Performanzabfall ganz deutlich bei allen Lesemessungen festzustellen. Durch das lexikographische Einsortieren der Dateinamen in den Baum mit einem Verzeichnis werden die Wege von den Verzeichniseinträgen zu den Statdaten immer länger, genauso wie die Wege von den Statdaten zu den indirekten Datenblöcken. Die darauf sensibel reagierenden Messungen für „Read“ und „Fread“ sind daher stark mit der Dateihistorie korreliert. Die steigende fragmentierte Weglänge ist ein Charakteristikum dafür. Die Auflösung pro

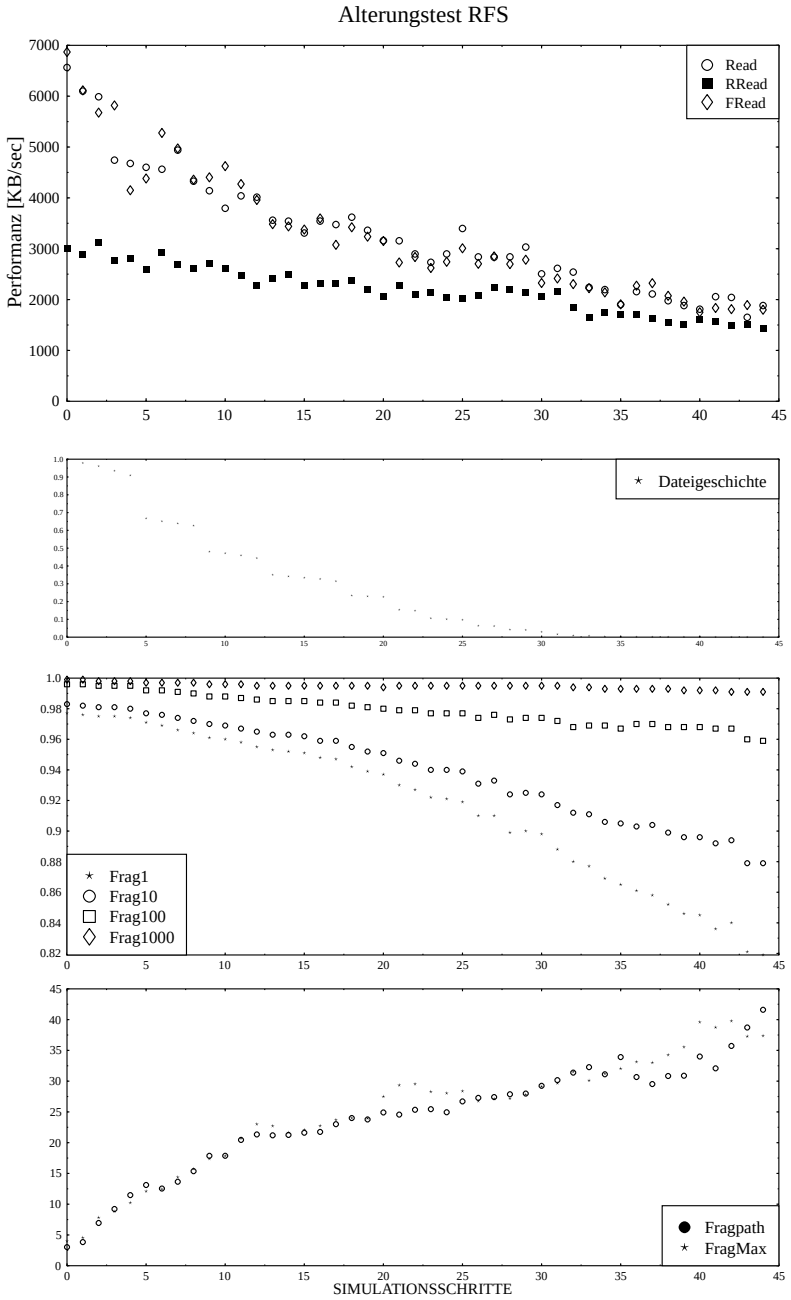

Abbildung 4.17: Das Ergebnis des Alterungstests für Reiser

Dateigrößen in Abb. 4.17 zeigt deutlich, daß der Performanzabfall bei Reiser auf die Fragmentierung der Dateien größer als 16KB –insbesondere größer als 256 KB– zurückzuführen ist.

Für Reiser in Abb. A.7 ohne Tails gelten im wesentlichen dieselben Ausagen. Der Unterschied zum normalen Reiser besteht lediglich darin, daß die Performanzschwankungen ähnlich wie bei Ext2 größer sind. Der Grund ist, daß jede Datei um einen indirekten Datenblock länger ist. Damit gibt es im Baum mehr Platz für die Verzeichnis- und Statdaten, so daß die Performanzunterschiede je nach Lage deutlicher ausfallen.

Von uns vorgenommene Meßergebnisse mit echten Verteilungen und Verzeichnisstrukturen lassen darauf schließen, daß die Alterungs- und Fragmentierungseffekte sich weniger stark bemerkbar machen als für die gezeigten Simulationen. Allerdings haben wir bei diesen Durchführungen die Anzahl der pro Schritt zu erzeugenden Dateien auf nur 100 eingestellt. Beim Untersuchen der Ergebnisse hat sich dieser Wert als zu klein erwiesen, um Fragmentierungseffekte meßbar zu machen. Das Standardmaß zur Fragmentierungsbewertung zeigte bei diesen –hier nicht vorgestellten– Messungen allerdings einen Wert von bereits 90%. Der Programmcode sowie weitere Ergebnisse für andere Dateisysteme und Parameter finden sich unter [Loi01f].

4.4.5 Berechnung des Performanzverlusts

Wir haben in den letzten Abschnitten gesehen, daß gealterte Systeme weniger performant sind als geeichte und daß die Fragmentierungsmaße besonders bei Reiser mit den Performanztendenzen korrelieren. Bisher haben jedoch nicht den Versuch unternommen, die Performanzverluste genauer zu quantifizieren. Dazu entwickeln wir in diesem Abschnitt ein einfaches Festplattenmodell mit dem Ziel, aus der Kenntnis der Zuordnung zwischen Dateien und logischen Blocknummern auf die von unseren Lesetests gemessene Performanzen zu schließen.

In unserem Modell bestehe eine Datei aus den Datenblöcken mit den logischen Nummern b_1 bis b_n. Zum Lesen dieser Datei benötigen wir die Zeit

$$T_{\mathrm{Dat}} = \sum_i S(b_{i+1}, b_i) + n \cdot T_L \,.$$

Dabei sei T_L die reine Übertragungszeit eines Blocks, und $S(x, y)$ die Suchzeit, die das Dateisystem und die Festplatte benötigen, den Block mit der Nummer x zu suchen, wenn sich der Kopf momentan über S(y) befindet.Die Zeitspanne, in der alle Dateien der Partition gelesen sind, besteht aus der Summe der Lesezeit der einzelnen Dateien, wobei wir die Suchzeiten zwischen dem letzten Datenblock einer Datei und dem ersten Datenblock der nächsten gelesenen Datei innerhalb der Summe berücksichtigen.Die Performanz der gelesenen Datenblöcke berechnen wir als den Quotienten aus allen Datenblöcken B mit der gesamten Laufzeit des Tests zu

$$T_{\mathrm{Sys}} = 1/(\underbrace{1/B \cdot \sum_D \sum_i S(b_{i+1}, b_i)}_{K} + T_L) \,. \tag{4.2}$$

Unser Ziel ist es nun, die Kosten K für die drei verschiedenen Testtypen zu bestimmen. Während der Bewertungsphase speichern wir für jeden Simulationsschritt des Alterungstests die vollständige Zuordnung zwischen Dateien und ihren Blöcken in einer Datei. Mit dieser Liste ist es keine Schwierigkeit, die verschiedenen Lesetests nacheinander zu simulieren. Sie ist anfangs in der Reihenfolge sortiert, in der die Dateien beim Auslesen des Verzeichnisses gefunden wurden. Damit ist die Simulationsreihenfolge für die Berechnung der Verzeichnisperformanz bestimmt. Für „FRead“ sortieren wir diese Liste vor der Berechnung nach aufsteigenden ersten Blocknummern. Für den zufälligen Lesetest führen wir die Auswahl der zu lesenden Dateien auf die gleiche Weise wie bei „RRead“ durch. Die Berechnung der Summen in Gleichung 4.2 ist demnach in allen Fällen einfach durchführbar.

Als nächstes wollen wir nun die Konstanten in Gleichung 4.2 bestimmen. Die Anzahl der gelesenen Blöcke stellt dabei das geringste Problem dar. Sie wird ohnehin bei jedem Meßdurchlauf angegeben. Schwieriger dagegen sind die Blocklesezeit T_L und die Suchfunktion $S(x, y)$ zu bestimmen. Beides sind Größen, die von der von uns verwendeten Festplatte abhängen. Sie müssen somit gemessen werden. Wir verwenden dazu die Informationen aus Abschnitt 3.4 über die Programmierung des SCSI-Treibers unter LINUX, um damit die in Abschnitt 2.2.8 beschriebenen Befehle direkt an die Festplatte abzusetzen.

Bestimmung der Übertragungszeit eines Blocks

Wir benutzen „READ_CAPACITY“ einerseits im normalen Modus, um die maximale Anzahl an Sektoren auf der Platte herauszufinden und andererseits im PMI-Modus, um die Verzögerungen auf der Platte zu messen. Der Befehlsblock ist in Abb. 2.18 dargestellt. Der in Abb. A.8 gezeigte Programmcodeausschnitt unseres Hilfsprogramms zur Ansteuerung der Festplatte verdeutlicht die Implementierung des Zugriffs über die in Abschnitt 2.2.8 vorgestellte Schnittstelle des SCSI-Subsystems für den „READ_CAPACITY“ Befehl im PMI-Modus.

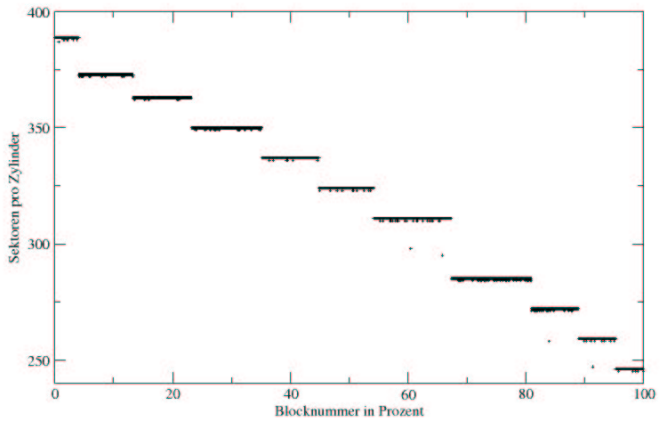

Abbildung 4.18: Die Rückgabewerte von „READ_CAPACITY“ im PMI-Modus

Das Ergebnis ist in graphischer Form in Abb. 4.18 dargestellt. Auf der X-Achse sind die Blocknummern bezüglich der maximalen Blockzahl in Prozent aufgetragen. Die Platte verfügt 17916239 Blöcke einer Größe von 512 Byte. Damit ist ein Block auf der Platte identisch mit einem Sektor. Der Speicherbereich ist offensichtlich in elf Notches eingeteilt. Wir wollen ihre Auswirkungen auf die Übertragungsrate überprüfen (siehe dazu auch [Met97]).

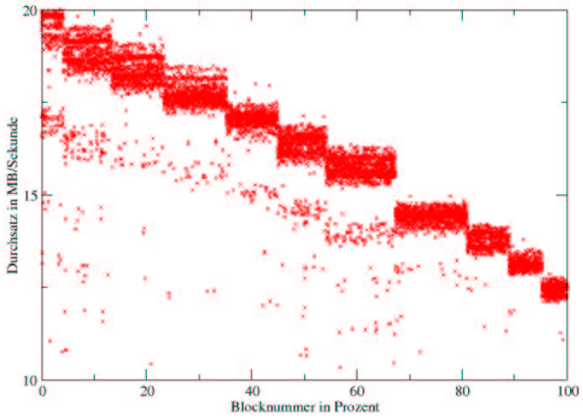

Abbildung 4.19: Der Durchsatz der SCSI-Platte gemessen in Blöcken von je 2 MB

Dazu öffnen wir die Gerätedatei der Platte und lesen in einer Schleife sequentiell Blöcke von 2 MB, um den Datendurchsatz zu messen.[44] Das Ergebnis dieser Durchsatzmessung haben wir in Abb. 4.19 dargestellt. Trotz der zahlreichen Aussetzer durch Cache-Effekte und zu weitem Vorlaufen sind die einzelnen Geschwindigkeitszonen deutlich zu erkennen. Die erste Zone ist die einzige, bei der die Platte den vollen Durchsatz von 20 MB pro Sekunde leistet. Sie reicht ungefähr 5% weit. In der letzten Zone liefert die Platte nur noch 65% Prozent ihres maximalen Durchsatzes. Der Ort der Partitionen auf der Platte beinflußt somit gravierend die Übertragungsrate. Die beiden von uns verwendeten Partitionen sind jeweils ein 1 GB groß und liegen am Anfang der Platte.[45] Damit beträgt der uns interessierende Durchsatz ungefähr 19 MB pro Sekunde. Die gesuchte Übertragungszeit für einen Block ist somit

$$T_L = 0.00021\,[\mathrm{s}]\,.$$

Bestimmung der Suchfunktion

Bei Messung der Suchfunktion gehen wir ähnlich vor. Wir benutzen das Interface aus Abschnitt 2.2.8, um den Befehlsblock für ein einfaches Suchkommando aus Abb. 2.18

[44] Wir wählen eine Blockgröße von 2 MB, weil der Festplattencache gerade 2 MB faßt.

[45] Bei manchen Sprüngen in der Performanz, die wir oben nicht befriedigend mit dem Blocklayout erklären konnten, ist die Ursache möglicherweise in dem Zonenwechsel der ersten Zone auf die zweite zu suchen. Dieser Wechsel liegt ungefähr in der Mitte von „/dev/sda1“.

zu implementieren. Damit sind wir in der Lage, die Platte bestimmte Sektoren aufsuchen zu lassen. Wir erstellen eine sogenannte „Suchkurve“ (engl. „seek curve“), indem wir der Platte zufällig ausgewählte logische Blocknummern zum Ansteuern übergeben (siehe auch [WGPW96]). Dabei speichern wir die benötigte Suchzeit für die prozentuale Differenz der momentanen und zu zur suchenden Blocknummer bezogen auf die maximale Differenz. Das Ergebnis ist in Abb. 4.20 dargestellt. Man

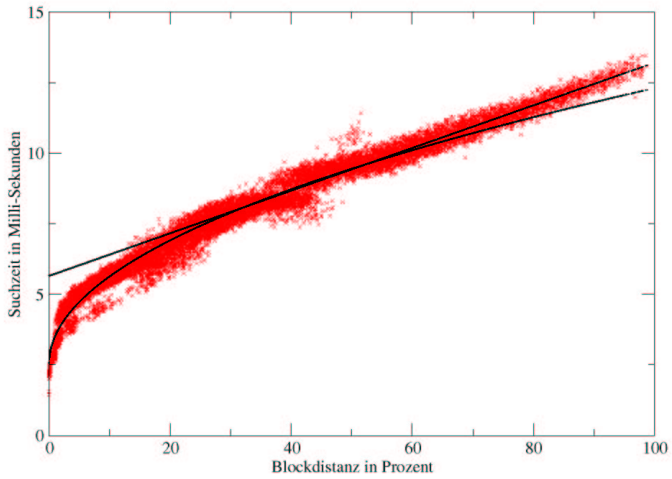

Abbildung 4.20: Die gemessene Suchkurve zusammen mit den Fits an die parametrisierte Suchfunktion

sieht deutlich den nichtlinearen Zugriffscharakter: Auch sehr kurze Suchwege kosten bereits über zwei Millisekunden. Das ist ungefähr ein Sechstel der Zeit, die der Kopf für den maximalen Suchweg benötigt. Die Bewegung des Kopfarms wird durch vier Phasen beschrieben:

- In der Beschleunigungsphase wird der Arm solange konstant beschleunigt, bis er entweder die halbe Strecke zurückgelegt oder die maximale Geschwindigkeit erreicht hat. Daher verhält sich die Suchzeit in diesem Bereich proportional zu der Wurzel aus der Distanz.
- In der Bewegungsphase wird der Arm für lange Suchstrecken mit maximaler Geschwindigkeit bewegt. Hier steigt die Suchzeit linear mit der Distanz.
- In der Abbremsphase wird der Arm in der Nähe der gesuchten Spur gebremst. Daher gilt für die Suchzeit wiederum, daß sie proportional zur Wurzel aus der Distanz steigt.
- In der Berühigungsphase wird der Kopf vom Controller auf die gewünschte Adresse gelenkt. Die dafür benötigte Zeit ist konstant. Typischerweise liegt sie bei ca. einer Millisekunde.

Sehr kurze Suchwege sind durch die Berühigungszeit, kurze Suchwege durch die Beschleunigungsphase und lange Suchwege durch die Bewegungsphase dominiert. Daher parametrisieren wir die Suchfunktion ähnlich wie [SSS99] durch

$$S'(d) = \begin{cases} 0 & \text{für} \quad d = 0 \\ p + q\sqrt{d} & \text{für} \quad 0 < d \leq d_c \\ p' + q' \cdot d & \text{für} \quad d > d_c \end{cases} ,$$

wobei wir $S(x, y) = S'(|x - y|)$ setzen. Wir bestimmen die Parameter der Suchfunktion durch einen nichtlinearen Fit. Das Ergebnis ist bereits in Abb. 4.20 dargestellt. Die numerischen Werte nach der Minimierung des Fehlerquadrats lauten

$$\begin{aligned} d_c &= 0.25 \\ p &= 0.00256 \\ q &= 0.00098 \\ p' &= 0.00566 \\ q' &= 0.75525 \,. \end{aligned}$$

Anwendung des Modells

Damit sind nun alle Größen der Gleichung 4.2 bekannt oder berechenbar. Ein Programm, das die in der Bewertungsphase gespeicherten Blockdaten einliest, kann je nach simulierten Lesetest die Summen in Gleichung 4.2 ausführen, und damit das Performanzergebnis vorhersagen. Leider können wir das Modell nicht an den im letzten Abschnitt besprochenen Systemen testen, da wir zu dem Zeitpunkt der obigen Messung die Blocknummern noch nicht mitaufgezeichnet haben. Statt dessen verwenden wir die aufgezeichneten Blockdaten einer ähnlichen Simulation.
Das untersuchte Dateisystem ist ein Reiser-Dateisystem mit einem speziellen Patch. Dieser Patch stellt einen Versuch dar, die bitmapbasierte Blockallokation von Reiser zu verbessern. Dazu pflegt er im laufenden Betrieb des Systems Hinweise, wo noch freie Blöcke im Bitmap sein könnten und wo unter keinen Umständen. Auf diese Weise soll die Suchzeit nach einem freien Block verringert werden.[46] Die Unterschiede zum normalen Reiser sind bei der Alterungssimulation marginal. Die Anzahl der insgesamt durchgeführten Simulationsschritte beträgt 80. In jedem Durchlauf werden 500 Dateien erzeugt. Insgesamt werden in der Simulation also 40000 Dateien erzeugt und ungefähr genausoviele gelöscht. Das Ergebnis dieser Simulation und der Berechnung des Modells sind in Abb. 4.21 dargestellt.

Diskussion

Dieses einfache Modell funktioniert offensichtlich erstaunlich gut. Trotz aller Vernachlässigungen –wie beispielsweise der Nichtberücksichtigung von Metadaten oder Zwischenspeicherstrategien– kann es das Ergebnis von „RRead“ grob im Verlauf widergeben. Mit Sicherheit liegt das an dem starken Einfluß des Charakters des Zufallslesetests: Da die Platte bei diesem ohnehin viel suchen muß, funktioniert das

[46] Siehe http://marc.theaimsgroup.com/?l=reiserfs&m=100093425211776&w=4 für eine detaillierte Erläuterung.

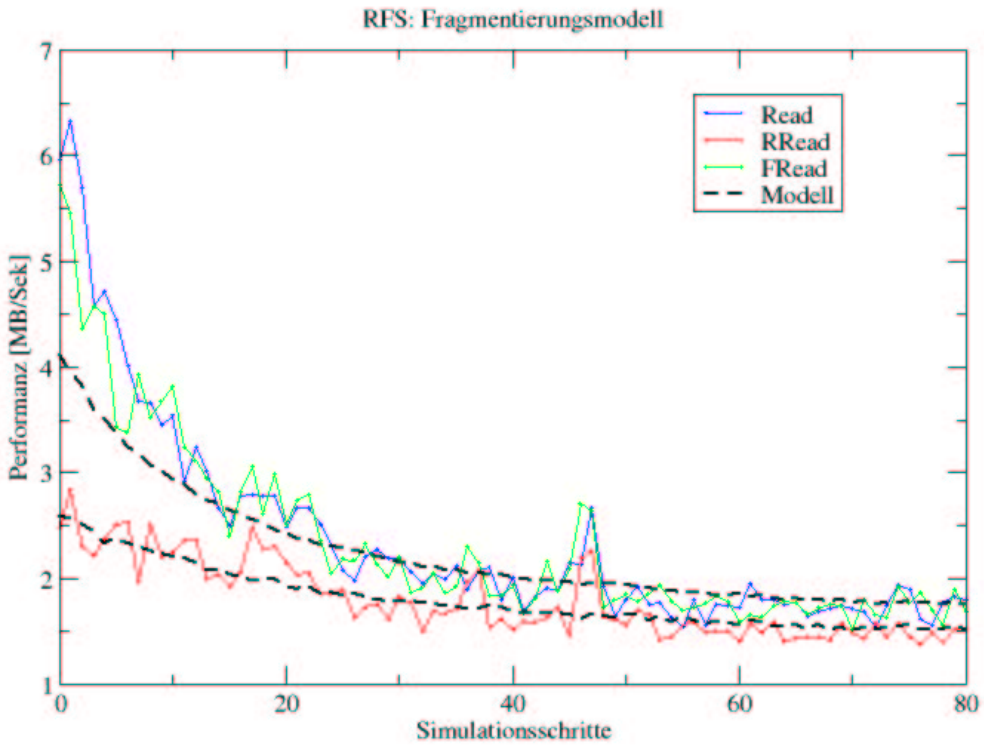

Abbildung 4.21: Simulierte und berechnete Performanzen im Vergleich

darauf getrimmte Modell gut. Bei den anderen Lesetests zeigt sich, die Schwachstelle des Modells deutlich. Die berechneten Kurven für diese Test sind komplett identisch, die Meßwerte aber nicht. Darüberhinaus weichen die Berechnungen gerade im interessanten Anfangsteil von der gemessenen Kurve ab. Dies ist eine Bestätigung dafür, daß diese beiden Lesetests keine gute Vergleichbarkeit zwischen Fragmentierung und Performanzmessung zulassen, da zu viele Metadateneffekte eine Rolle spielen, die das Modell komplett vernachlässigt. Es wäre daher höchst interessant, die Lesetests insgesamt in zwei Versionen durchzuführen: Einmal in der momentanen Form und einmal auf eine leicht modidifizierte Weise, bei der eine große Anzahl von Dateien immer im Voraus geöffnet und dann auf einmal gelesen werden. Damit müßten die Metadateneffekte eine geringere Rolle spielen. Vergleiche zwischen beiden Arten von Lesetests wären mit Sicherheit aufschlußreich.

5 Zusammenfassung

Das Ziel dieser Arbeit ist die Analyse und Simulation von Fragmentierungseffekten am Reiser-Dateisystem sowie ihre objektive Bemaßung.

Zu diesem Zweck untersuchen wir die beteiligten Komponenten aus Hardware und Software. Insbesondere vertiefen wir Festplatten- und Dateisystemtechniken sowie ihre Zusammenarbeit im virtuellen Dateisystem unter LINUX. Dabei arbeiten wir die Unterschiede zwischen dem fortschrittlichen ReiserFS und dem klassischen Ext2FS Dateisystem heraus.

Wir definieren verschiedene Maße zur Zustandscharakterisierung eines Dateisystems und geben eine Einführung in die Problematik der kontrollierten und anwendungsbasierten Langzeitsimulation von Alterungseffekten. Daraufhin stellen wir zwei von uns entwickelte Simulationsmethoden vor: Die Alterungssimulation und die Applikationssimulation.

Die Ergebnisse der durchgeführten Alterungssimulationen zeigen, daß externe Fragmentierung bei ReiserFS zu Performanzverlusten führt. Bei den Tests durch Anhängen sinkt die über alle Dateigrößen gemittelte Leseperformanz des gealterten Dateisystems im schlechtesten Fall um 600% bezogen auf die Referenzkopie. Bei der im Vergleich moderaten Belastung der Alterungstests durch Erzeugen und Löschen sinkt dieselbe Meßgröße um 40% im Vergleich zur Referenz. Ext2FS schneidet bei den Appendtests wesentlich schlechter ab. Im schlechtesten Fall sinkt die Leseperformanz um den Faktor 14. Bei der moderaten Alterungsmessung dagegen sinkt die Performanz nur auf 80% des Referenzwerts. Allerdings sind die absoluten Meßwerte beider Systeme bei dem gewählten Szenario in derselben Größenordnung. Die Fragmentierungsmaße korrelieren bis auf wenige Ausnahmen bei Ext2FS in allen Fällen mit den Meßwerten.

Für die Appendtests zeigen wir, daß sich durch eine einmalig zu Beginn der Simulation für jede Datei vorgenommene Preallokation von Blöcken mit anschließender Freigabe des Speicherplatzes die Leseperformanz bei ReiserFS um bis zu 200% im Vergleich zu dem ohne Preallokation gealterten gemessenen Wert erhöht. Bei Ext2FS hat dieser einmalige Vorgang keine meßbaren Vorteile.

Wir führen darüber hinaus eine Methode vor, mit der wir die Performanz der Lesetests durch das gemessene Blocklayout berechnen können. An Hand eines Simulationsergebnisses für ReiserFS zeigen wir, daß wir damit den Verlauf der Meßwerte grob bestimmen können.

A Abbildungen

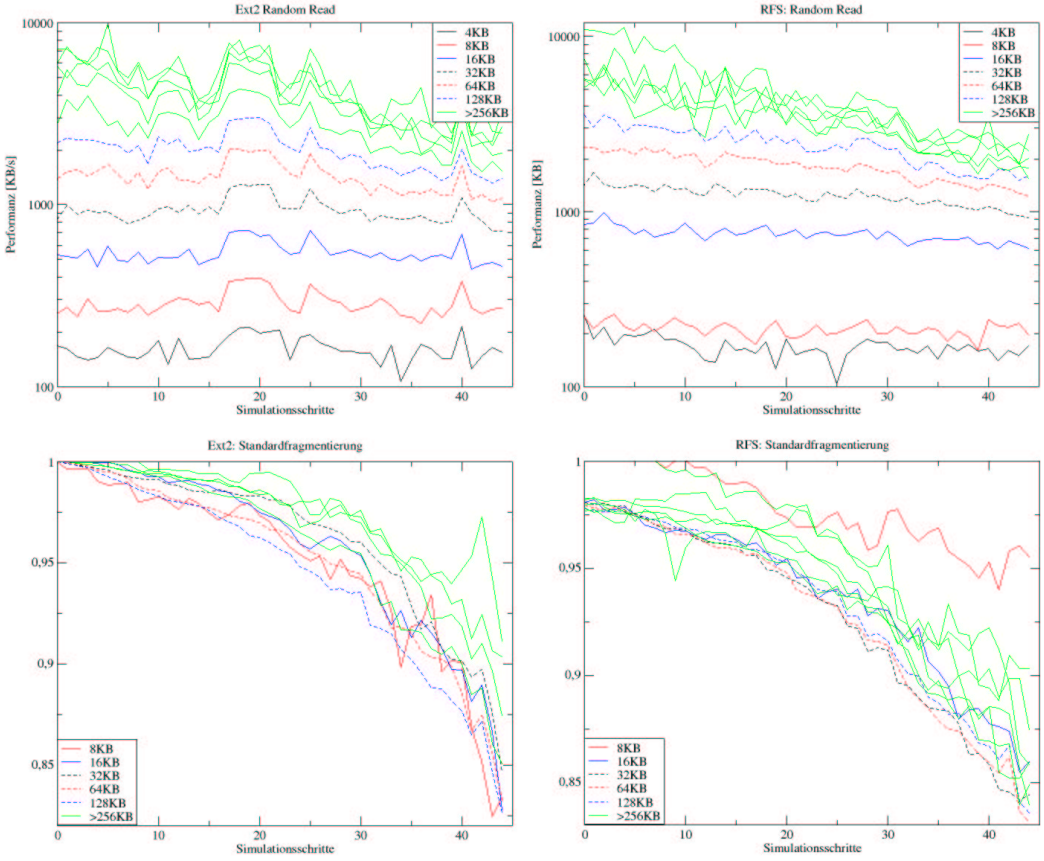

Abbildung A.1: Die Zufallsleseperformanz und die Standardfragmentierung des Alterungstests für Ext2 und Reiser in Histogrammform

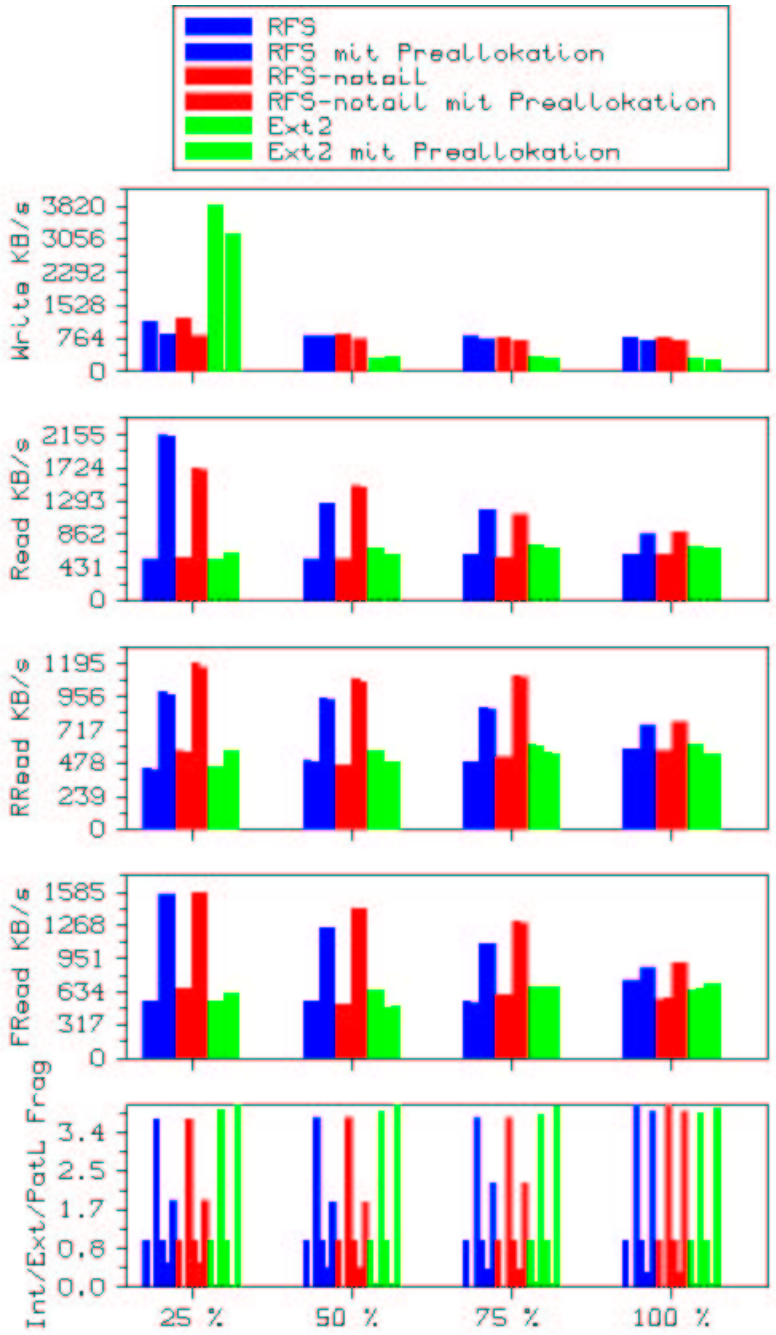

Abbildung A.2: Das Ergebnis des Appendtests für 5000 Dateien in einem Verzeichnis mit und ohne Preallokation

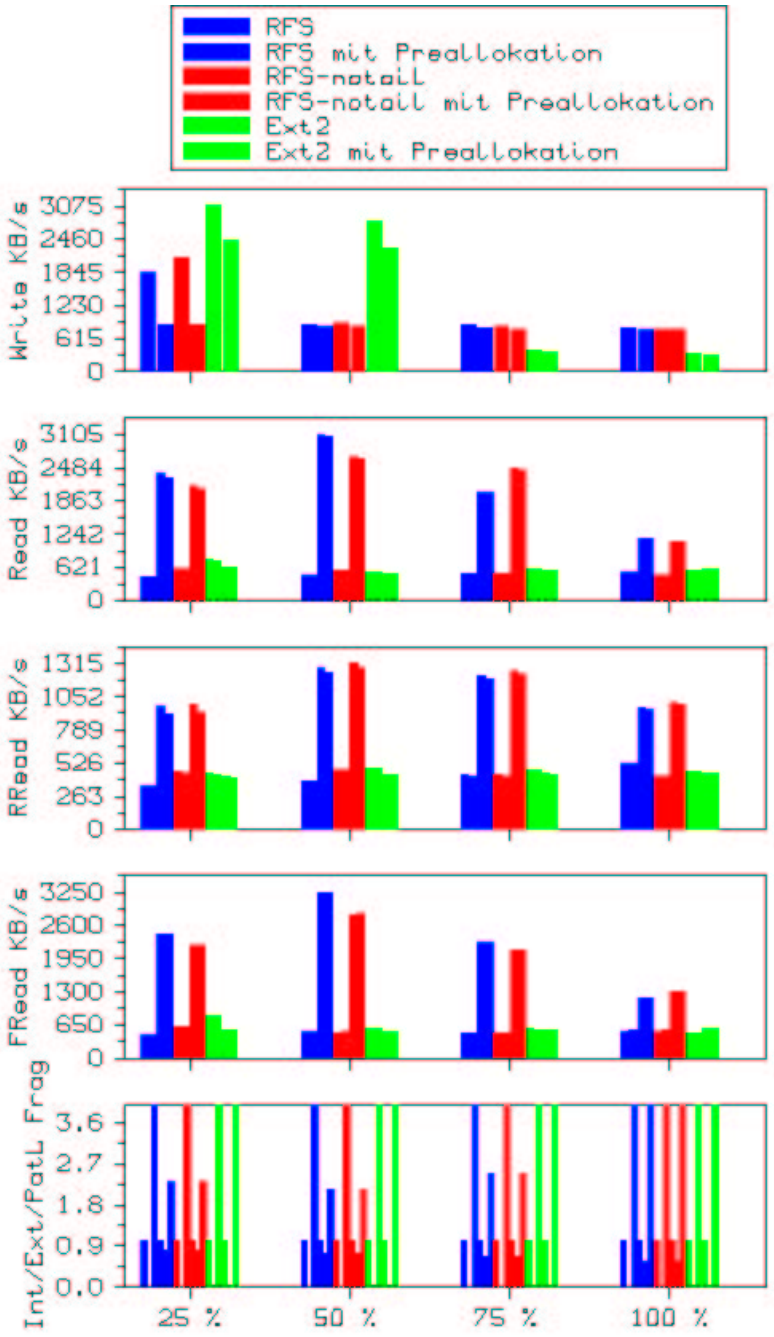

Abbildung A.3: Das Ergebnis des Appendtests für 10000 Dateien in einem Verzeichnis mit und ohne Preallokation

Dateisystemname		Ext2FS															
Auslastung von 1GB		25%				50%				75%				99%			
Allokationsart		Nur Append		Mit Preallok		Nur Append		Mit Preallok		Nur Append		Mit Preallok		Nur Append		Mit Preallok	
Dateien	Messung	app	cpy	app	cpy	app	cpy	app	cpy	app	cpy	app	cpy	app	cpy	app	cpy
1000	Dir-Read	823	7969	762	15267	856	8265	797	15231	875	12912	662	14159	701	9721	781	11767
	Random-Read	750	5882	748	8116	651	7375	832	11012	620	10207	596	10554	612	7620	778	10822
	Fibmap-Read	844	8937	886	12086	799	10149	716	16748	890	12826	680	14480	744	9247	816	14807
	Frag1	0.019	0.984	0.016	0.984	0.038	0.992	0.014	0.992	0.040	0.995	0.011	0.995	0.031	0.996	0.010	0.996
	Frag10	0.019	1.000	0.022	1.000	0.040	1.000	0.019	1.000	0.044	1.000	0.016	1.000	0.036	1.000	0.015	1.000
	Frag100	0.019	1.000	0.053	1.000	0.040	1.000	0.050	1.000	0.046	1.000	0.047	1.000	0.038	1.000	0.045	1.000
	Frag1000	0.951	1.000	0.629	1.000	0.956	1.000	0.638	1.000	0.935	1.000	0.601	1.000	0.936	1.000	0.577	1.000
	FragPath	1059	1	3131	1	1045	1	2069	1	2289	2	1721	2	1991	2	1559	2
	Append/Copy	663	686	658	681	511	569	537	675	519	587	533	629	539	571	534	572
5000	Dir-Read	535	8555	624	12899	687	10444	609	11357	726	12320	680	11277	699	11334	672	11545
	Random-Read	454	1656	567	2594	566	3090	486	2533	605	3520	551	2630	605	4541	541	2776
	Fibmap-Read	540	9213	620	9823	643	6637	489	7474	685	14615	680	8856	656	8231	702	8038
	Frag1	0.040	0.971	0.036	0.971	0.080	0.957	0.062	0.957	0.090	0.972	0.083	0.972	0.078	0.979	0.079	0.979
	Frag10	0.048	1.000	0.061	1.000	0.087	1.000	0.088	1.000	0.096	1.000	0.109	1.000	0.082	1.000	0.104	1.000
	Frag100	0.054	1.000	0.187	1.000	0.107	1.000	0.214	1.000	0.121	1.000	0.235	1.000	0.113	1.000	0.236	1.000
	Frag1000	0.134	1.000	0.193	1.000	0.146	1.000	0.221	1.000	0.149	1.000	0.244	1.000	0.135	1.000	0.245	1.000
	FragPath	7463	1	16688	2	6882	1	11120	3	5952	2	8992	2	6526	2	8322	2
	Append/Copy	3823	441	3174	382	294	497	359	388	332	556	318	434	326	504	278	395
10000	Dir-Read	767	12410	631	7847	536	9284	511	8873	594	7014	575	8253	574	8434	587	10810
	Random-Read	445	1479	423	1283	483	1988	438	1753	471	1897	442	1461	463	1751	445	1958
	Fibmap-Read	826	7437	558	11860	585	8195	511	10128	569	10447	541	6395	497	7548	570	6314
	Frag1	0.038	1.000	0.028	1.000	0.048	0.971	0.037	0.971	0.050	0.941	0.041	0.941	0.044	0.957	0.037	0.957
	Frag10	0.040	1.000	0.072	1.000	0.055	1.000	0.085	1.000	0.056	1.000	0.098	1.000	0.049	1.000	0.091	1.000
	Frag100	0.050	1.000	0.271	1.000	0.076	1.000	0.267	1.000	0.080	1.000	0.284	1.000	0.075	1.000	0.284	1.000
	Frag1000	0.051	1.000	0.292	1.000	0.089	1.000	0.300	1.000	0.096	1.000	0.319	1.000	0.088	1.000	0.326	1.000
	FragPath	27485	1	31268	2	17001	1	20795	2	16232	2	17829	2	14821	2	16155	2
	Append/Copy	3079	464	2446	350	2799	460	2282	341	410	480	364	369	326	372	307	334

Abbildung A.4: Das Ergebnis des Appendtests für Ext2 im Überblick

Dateisystemname		ReiserFS															
Auslastung von 1GB		25%				50%				75%				99%			
Allokationsart		Nur Append		Mit Preallok		Nur Append		Mit Preallok		Nur Append		Mit Preallok		Nur Append		Mit Preallok	
Dateien	Messung	app	cpy	app	cpy	app	cpy	app	cpy	app	cpy	app	cpy	app	cpy	app	cpy
1000	Dir-Read	725	7330	1139	6542	624	7211	987	10430	686	8154	1114	6733	623	8740	917	7222
	Random-Read	680	4702	862	5404	606	6825	992	6099	604	6844	937	5337	572	6036	1013	7325
	Fibmap-Read	768	4155	925	8524	761	9448	1006	7820	552	4877	826	7793	601	5382	820	10654
	Frag1	0.038	0.978	0.135	0.978	0.034	0.983	0.101	0.988	0.032	0.987	0.087	0.990	0.045	0.987	0.094	0.991
	Frag10	0.573	0.982	0.936	0.982	0.612	0.993	0.911	0.994	0.558	0.993	0.865	0.994	0.478	0.994	0.747	0.995
	Frag100	0.610	0.999	0.973	1.000	0.643	0.996	0.964	0.996	0.591	0.995	0.927	0.995	0.508	0.997	0.813	0.997
	Frag1000	0.730	1.000	0.999	1.000	0.746	1.000	0.998	1.000	0.709	1.000	0.982	1.000	0.654	1.000	0.877	1.000
	FragPath	1702	2	37	2	1614	1	39	2	3115	2	75	2	6846	3	1977	4
	Append/Copy	634	623	564	611	574	426	558	612	537	622	556	823	494	393	533	621
5000	Dir-Read	540	6799	2157	4741	534	4278	1262	5708	597	4749	1176	6048	605	9069	868	7591
	Random-Read	433	1751	989	2124	494	2232	942	2358	491	3087	874	2518	580	4989	752	3589
	Fibmap-Read	538	4974	1560	7058	543	4280	1247	4785	536	4380	1092	4511	735	8955	862	7053
	Frag1	0.000	0.948	0.529	0.953	0.000	0.959	0.428	0.962	0.002	0.972	0.387	0.973	0.003	0.976	0.322	0.977
	Frag10	0.006	0.956	0.954	0.956	0.030	0.964	0.921	0.965	0.048	0.976	0.876	0.976	0.052	0.980	0.728	0.980
	Frag100	0.123	0.999	0.977	1.000	0.136	0.998	0.949	0.998	0.142	0.997	0.911	0.997	0.141	1.000	0.782	0.999
	Frag1000	0.189	1.000	0.990	1.000	0.202	1.000	0.982	1.000	0.214	1.000	0.940	1.000	0.218	1.000	0.817	1.000
	FragPath	4887	1	80	1	4986	2	72	2	4986	2	192	2	9806	3	6725	4
	Append/Copy	1173	313	839	855	814	363	808	793	816	471	745	685	780	459	715	514
10000	Dir-Read	450	4232	2375	4164	467	5544	3105	4129	492	5938	2024	6668	526	4588	1157	5208
	Random-Read	349	1171	975	1226	386	1587	1280	1733	429	1784	1220	2053	518	2569	961	2685
	Fibmap-Read	464	9928	2422	11102	509	5344	3251	6117	493	6496	2259	7086	532	6075	1192	5461
	Frag1	0.000	0.973	0.800	0.969	0.000	0.942	0.729	0.947	0.000	0.971	0.670	0.975	0.000	0.959	0.548	0.963
	Frag10	0.000	0.978	0.938	0.975	0.000	0.947	0.917	0.948	0.000	0.976	0.834	0.977	0.003	0.964	0.679	0.965
	Frag100	0.000	1.000	0.938	1.000	0.000	1.000	0.939	1.000	0.002	0.998	0.895	0.998	0.007	0.998	0.744	0.998
	Frag1000	0.000	1.000	0.972	1.000	0.003	1.000	0.973	1.000	0.009	1.000	0.935	1.000	0.019	1.000	0.778	1.000
	FragPath	9784	1	201	2	9993	2	132	2	10008	2	292	2	19611	4	13198	4
	Append/Copy	1859	211	867	460	878	292	848	1159	853	400	793	1060	823	407	792	803

Abbildung A.5: Das Ergebnis des Appendtests für Reiser im Überblick

Dateisystemname		ReiserFS Notail															
Auslastung von 1GB		25%				50%				75%				99%			
Allokationsart		Nur Append		Mit Preallok		Nur Append		Mit Preallok		Nur Append		Mit Preallok		Nur Append		Mit Preallok	
Dateien	Messung	app	cpy	app	cpy	app	cpy	app	cpy	app	cpy	app	cpy	app	cpy	app	cpy
1000	Dir-Read	623	5208	1068	7180	653	10025	1098	8935	612	5280	1121	7547	655	6437	965	9799
	Random-Read	567	4622	866	4688	633	5305	984	6217	642	5791	968	6981	627	6613	948	7635
	Fibmap-Read	570	5681	1138	5262	683	5427	1041	7257	621	6049	1102	8974	650	7481	829	7777
	Frag1	0.039	0.974	0.134	0.979	0.046	0.984	0.101	0.989	0.038	0.986	0.086	0.988	0.064	0.986	0.088	0.991
	Frag10	0.579	0.982	0.937	0.982	0.635	0.992	0.916	0.994	0.611	0.993	0.865	0.994	0.547	0.994	0.734	0.996
	Frag100	0.586	1.000	0.977	1.000	0.651	0.995	0.968	0.995	0.631	0.995	0.929	0.995	0.576	0.997	0.811	0.997
	Frag1000	0.736	1.000	0.999	1.000	0.753	1.000	0.995	1.000	0.741	1.000	0.980	1.000	0.717	1.000	0.870	1.000
	FragPath	1629	2	35	2	1431	2	41	2	2660	2	76	2	4296	2	2689	4
	Append/Copy	628	372	567	764	563	422	558	792	539	522	549	664	501	420	537	775
5000	Dir-Read	551	5875	1717	5253	542	4320	1478	6095	557	3771	1118	4965	588	6418	893	7036
	Random-Read	565	2146	1196	1721	469	1948	1078	2286	525	2535	1112	2372	560	3440	771	3366
	Fibmap-Read	660	4677	1586	6795	510	5240	1429	4167	602	4897	1295	5613	563	5788	909	5715
	Frag1	0.000	0.950	0.528	0.951	0.000	0.959	0.427	0.963	0.003	0.971	0.386	0.971	0.004	0.975	0.321	0.976
	Frag10	0.038	0.956	0.954	0.956	0.038	0.964	0.921	0.965	0.052	0.975	0.876	0.976	0.065	0.980	0.727	0.980
	Frag100	0.166	0.999	0.977	0.999	0.129	0.998	0.949	0.998	0.141	0.997	0.910	0.997	0.147	0.999	0.780	0.999
	Frag1000	0.223	1.000	0.990	1.000	0.204	1.000	0.986	1.000	0.211	1.000	0.943	1.000	0.222	1.000	0.817	1.000
	FragPath	4834	1	74	1	4975	2	72	2	4978	2	190	2	9815	3	6760	4
	Append/Copy	1233	408	811	600	846	377	770	769	804	436	729	461	794	461	708	467
10000	Dir-Read	589	5384	2147	5577	552	3439	2698	4396	499	6244	2465	8281	477	5249	1090	6067
	Random-Read	463	1341	987	1210	472	1577	1316	1660	426	1993	1255	2202	424	2081	996	2533
	Fibmap-Read	612	8314	2209	7737	505	4999	2818	6315	482	6660	2124	8240	533	6182	1291	6316
	Frag1	0.000	0.970	0.805	0.973	0.000	0.943	0.729	0.946	0.000	0.971	0.669	0.975	0.000	0.957	0.548	0.962
	Frag10	0.006	0.976	0.937	0.977	0.010	0.947	0.920	0.948	0.009	0.976	0.834	0.977	0.012	0.964	0.679	0.964
	Frag100	0.006	0.999	0.938	1.000	0.014	1.000	0.940	1.000	0.013	0.997	0.894	0.998	0.017	0.998	0.742	0.998
	Frag1000	0.016	1.000	0.972	1.000	0.019	1.000	0.969	1.000	0.018	1.000	0.928	1.000	0.021	1.000	0.772	1.000
	FragPath	9709	2	197	1	10008	2	139	2	10008	2	293	2	19667	4	13219	4
	Append/Copy	2114	297	870	419	902	395	837	957	831	416	790	976	784	307	778	710

Abbildung A.6: Das Ergebnis des Appendtests für Reiser ohne Tails im Überblick

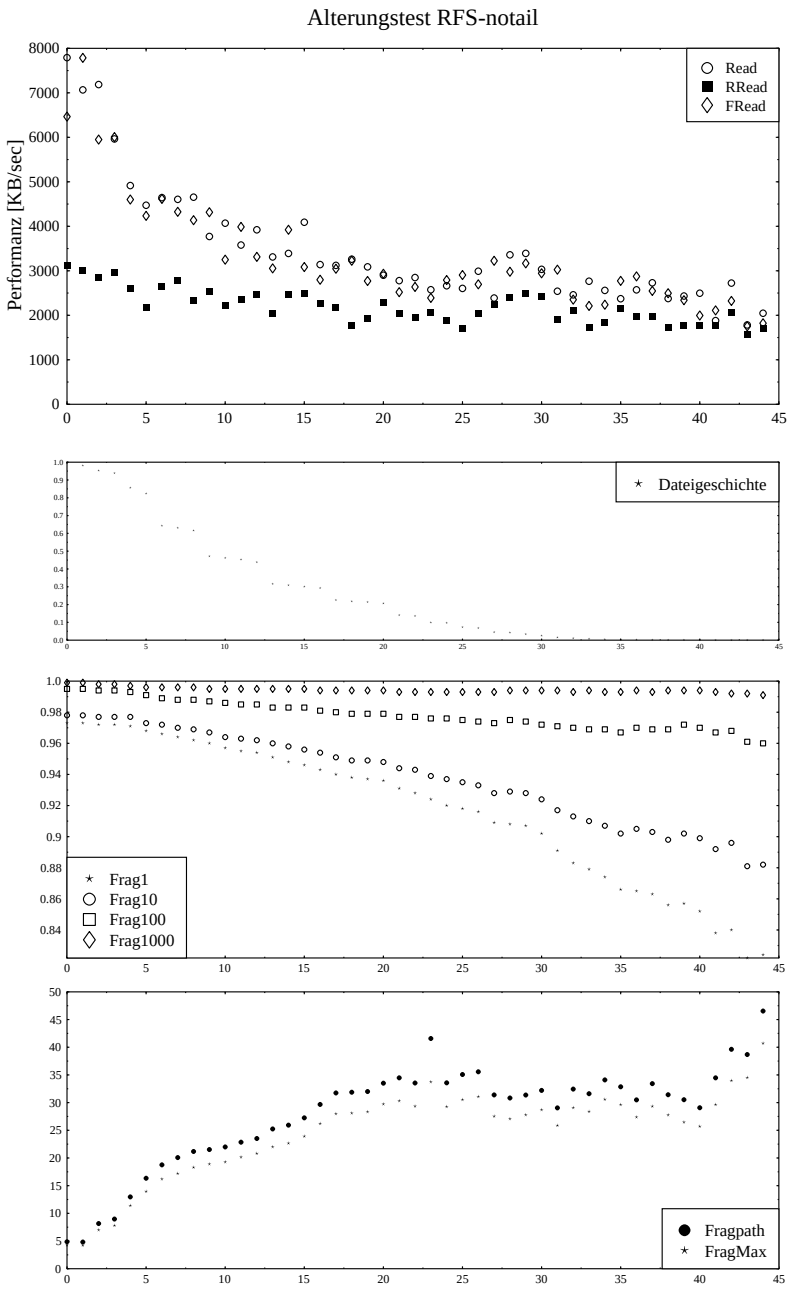

Abbildung A.7: Das Ergebnis des Alterungstests für Reiser ohne Tails

```
int read_pmi_capacity(int lbn,int *lbn_next,int *lbn_next_len)
{
 unsigned char buffer[100];
 int status;
 unsigned char *cmd;

 memset(buffer, -1, sizeof(buffer));

 *((int *) buffer) = 10;      /*length of input data*/
 *(((int *) buffer) + 1) = 8; /*length of output buffer*/

 cmd = (char *) (((int *) buffer) + 2);

 cmd[0] = READ_CAPACITY;
 cmd[1] = 0x00;         /*LUN / (reserved)*/
 putnbyte(cmd+2,lbn,4); /*LBN*/
 cmd[6] = 0x00;         /*(reserved)*/
 cmd[7] = 0x00;         /*(reserved)*/
 cmd[8] = 0xff;         /*(reserved)*/
 cmd[8] = 0x01; /*(reseved) / PMI Bit*/
 cmd[9] = 0x00;         /*(reserved)*/

 status = ioctl(fd, 1 /*SCSI_IOCTL_SEND_COMMAND*/ , buffer);
 if (!status){
  *lbn_next = getnbyte(buffer + 8, 4);
  *lbn_next_len = getnbyte(buffer + 12, 4);
  return -1;
  } /*check status and return error*/
}
```

Abbildung A.8: Ausschnitt der Funkion „read_pmi_capacity“ zum Absetzen des PMI-Befehls an die SCSI-Platte

B Akronyme und Abkürzungen

ATA AT Attachment: Standard, der die früher als IDE oder EIDE bezeichnete Schnittstelle zum Anschluß von Festplatten und CD-ROM-Laufwerken an IBM-PC/ATs beschreibt.

ATAPI ATA Packet Interface: Standardisierte Schnittstelle, die den Anschluß von CD-ROM-Laufwerken an den IDE-Controller ermöglicht.

ANSI American National Standards Institute: Amerikanisches Normungsgremium vergleichbar mit dem deutschen DIN-Institut (siehe http://www.ansi.org/).

BIOS Basic Input Output System: Fest im ROM oder EEPROM gespeichertes Programm, das beim Hochfahren des Computers die Hardware überprüft, die Systemdateien lädt und den Rechner betriebsbereit macht. Es ermöglicht die Zugriffe auf alle Systemkomponenten (siehe http://www.bios-info.de/).

BSD Berkely Software Distribution: Weitverbreitetes UNIX-System, das die Entwicklung von UNIX maßgeblich geprägt hat.

Cache Zwischenspeicher: Cache bzw. Caching nennt man ein allgemeines Vorgehen zur Beschleunigung der Informationsverarbeitung. In einem Cache werden jeweils die von der darunterliegenden im Zugriff oder in der Verarbeitung meist langsamen Software- oder Hardwarekomponente geholten Informationen vorsorglich gespeichert, um sie im Falle eines erneuten Zugriffs auf dieselben Daten dann von dort und nicht wieder von der langsameren Komponente zu holen.

CD-ROM CD-Massenspeicher: Nur lesbarer Massenspeicher mit 640 MB Kapazität, der mit einem (oder mehreren) Laserstrahl(en) abgetastet wird (engl. „compact disk read only memory“).

CHS Geometrieangabe einer Festplatte: Die sogenannten „CHS-Koordinaten“ definieren einen Sektor auf einer Festplatte durch die Angabe seiner Zylinder, Kopf und Sektornummer (engl. „cylinder“, „head“ und „sector“). Adressierung durch Angabe der CHS-Koordinaten wird auch physikalische Adressierung genannt.

CPU Hauptprozessor: Zentrale Programmiereinheit eines Personalcomputers (engl. „central processing unit“).

CRC Fehlererkennungsverfahren: Es benutzt einen vordefinierten mathematischen Divisor (Generatorpolynom), um die Integrität eines übertragenen Blocks zu prüfen (engl. „cylic redundancy code“, siehe http://www.informatik.uni-frankfurt.de/~haase/crc.html).

DMA Hauptspeicherdirektzugriff: Der direkte Speicherzugriff bietet eine Alternative zum Pollingverfahren oder Handshakeverfahren und ermöglicht den direkten Datentransfer zwischen einem Interface und dem Arbeitsspeicher des PC ohne Umweg über die CPU. Dadurch wird diese zwar entlastet, sie hat aber keinen Zugriff auf den Speicher, während der DMA-Controller arbeitet (engl. „direct memory access“).

DVD-ROM DVD-Massenspeicher: Multimediascheibe mit verschiedensten Formaten, seit 1997 abwärts kompatibel zur CD-ROM (engl. „digital video disk read only memory“).

ECC Fehlerkorrekturcode: Ähnlich dem CRC werden einer Informationseinheit weitere Kontrollbits angefügt, die eine Prüfsumme darstellen. Beim Erkennen eines Fehlers kann aber bis zu einer verfahrensabhängigen Bitzahl der Fehler automatisch korrigiert werden (engl. „error correction code“), siehe `http://www.csl.sony.co.jp/person/morelos/ecc/codes.html`).

EEPROM Electrically Erasable Read Only Memory: Speicher, der gewöhnlich nur gelesen wird, aber elektrisch löschbar und beschreibbar ist.

EIDE Enhanced IDE: EIDE ist eine Eigenentwicklung von Western Digital zur Verbesserung von IDE und ist kompatibel zu ATA-2 und ATAPI.

ESDI Enhanced Small Drive Interface: Von Maxtor 1983 präsentierte gegenüber ST506/412 verbesserte Festplattenschnittstelle. ESDI hat sich im Gegensatz zu SCSI-Schnittstellen nicht auf dem Markt durchgesetzt.

Ext2 Ext2FS: In dieser Arbeit verwendete Abkürzung für Ext2FS.

Ext3 Ext3FS: In dieser Arbeit verwendete Abkürzung für Ext3FS.

Ext2FS Second Extended File System: Standarddateisystem unter LINUX.

Ext3FS Third Extended File System: Erweitert das Standarddateisystem Ext2 um ein Journal zur Protokollierung von Daten- und Metadatenänderungen.

FAT File Allocation Table: Dateizuordnungstabelle bei Microsoft-Dateisystemen. Sie gibt für jede Datei in einer Liste an, welche Datenblöcke der Datei zugeordnet sind.

FFS Fast File System: Das Standarddateisystem bei BSD. Es wertet Geometriedaten von Festplatten aus, um die Performanz zu erhöhen.

GPL General Public License: Die GPL wurde von der 1985 von Richard Stallman gegründeten Stiftung „Free Software Foundation“ formuliert und führt juristisch die Veröffentlichung freier Software an. Das „Copyleft“ erlaubt, geistiges Eigentum anderer beliebig zu modifizieren und auch profitabel zu vertreiben, solange das neue Produkt freien Zugang zum Quellcode gewährt.

I/O Input/Output: Abkürzung für „Eingabe/Ausgabe“, auch „E/A“ genannt. Bezeichnung für Vorgänge, Programme und Komponenten, die Daten in den Computer übertragen (Eingabe) bzw. für Benutzer darstellen (Ausgabe). E/A-Aufgaben werden in der Regel von Peripheriegeräten und Geräte-Treibern übernommen.

IBM-PC IBM-Personalcomputer: Am 12.August 1981 wurde in New York der erste PC, IBM 5150 PC, als Konkurrenz zu Apple vorgestellt. Mit dem IBM-PC wurde das Betriebssystem MS-DOS von Microsoft ausgeliefert. Merkmale des IBM-PC waren der 8088 Prozessor, ein 64 KB großer Arbeitsspeicher und ein eigens für diesen PC entwickeltes BIOS. Dieser erste PC war so erfolgreich, daß er schnell von anderen Herstellern nachgebaut wurde und diese „Kopien“ zum Teil günstiger als das Original verkauft wurden (IBM-kompatibel).

IBM-PC/AT IBM Personalcomputer/Advanced Technology: Der IBM-PC/AT stellt die Weiterentwicklung des IBM-PC durch IBM dar. Die Hauptmerkmale des am 14. August 1984 vorgestellten Gerätes IBM AT-5170 waren der 80286 Prozessor von Intel, eine maximale Größe des Arbeitsspeichers von 16 MB, eine Festplatte, ein 5 1/4 Zoll Diskettenlaufwerk für 1.2 MB Disketten eine weiterentwickelte Datenbusbreite von 16 Bit.

IDE Intelligent Drive Electronics: Mittlerweile Synonym für ATA. Ursprünglich von Compaq 1984 entwickelte Festplatte mit integriertem Controller.

IEEE Institute of Electrical and Electronics Engineers: Eine Standardisierungsorganisation für Elektronik ähnlich der ISO. Insbesondere für Netzwerke sind viele Normen bekannt (IEEE 802 Normen).

IPC Interprozeßkommunikation: Werkzeuge und Methoden zur Kommunikation von Prozessen (engl. „inter process communication“).

ISA Industry Standard Architecture: Standardisierter PC-Bus, den IBM 1981 mit anfangs 8 Bit für den XT und ab 1984 mit 16 Bit für den AT-Bus eingeführte.

ISO International Standards Organisation: Freiwillige, nicht staatlich geregelte Organisation, deren 89 Mitglieder die nationalen Normungsinstitute der beteiligten Länder sind.

JFS Journaled File System: Dateisystem von OS/2, das seit kurzem von IBM auf LINUX portiert wird.

LBA Logische Blockadresse: Bei ATA-Geräten verwendete logische lineare Adressierung der Sektoren, die vom Laufwerkscontroller in die Geometriekoordinaten umgerechnet wird (engl. „logical block address“).

LBN Logische Blocknummer: Bei SCSI verwendete lineare Adressierung der logischen Blöcke (engl. „logical block number“). Diese Blöcke stellen bei Direktzugriffsgeräten die kleinste, beliebig adressierbare Einheit dar.

LUN Logische Gerätenummer: SCSI-Targets können bis zu acht physikalische Geräte unterstützen, die durch ihre LUN unterschieden werden (engl. „logical unit number“).

LFS Logbuchbasiertes Dateisystem: Dateisystem, das im wesentlichen aus einem Logbuch besteht (engl. „log structured file system“, LFS).

LVM Logical Volume Manager: Softwareschicht unter UNIX, durch die Dateisysteme transparent über Gerätegrenzen hinweg installiert werden können.

NFS Networking File System: Softwareschicht unter UNIX, die Dateisysteme transparent über Netzwerkgrenzen exportiert oder auch importiert.

MFM Modifizierte Frequenzmodulation: Kodierungsverfahren bei dem im Unterschied zum älteren Frequenzmodulationsverfahren das jeweilige zu kodierende Datenbit und das vorhergehende Datenbit eingeht, so daß die Taktrate effektiv halbiert werden kann (engl. „modified frequency modulation“).

MMU Memory Management Unit: Sie ermöglicht die Übersetzung zwischen physikalischen und virtuellen Speicherseiten.

MS-DOS Microsoft-Disk Operating System: Altes Betriebssystem für IBM-PC kompatible Systeme.

NCITS National Committee for Information Technology Standards: Katalog von Standardisierungen, die von 1961-1996 „X3-Standards„ hießen. Momentan gibt es 56 NCITS Standards, 202 X3-Standards und 354 ANSI/ISO-Standards (siehe `http://www.ncits.org/`).

NRZ Pulscodekodierverfahren: Variante des digitalen Pulscodekodierverfahrens, bei dem der Signalpegel direkt das zu übertragende Bit kodiert („non return to zero“).

OSI Open Systems Interconnection: Ein Begriff, der eine Reihe von Protokollen und Referenzmodellen bezeichnet, die von der ISO entwickelt wurden. Im Rahmen seiner Tätigkeit hat dieses Komitee unter anderem 1984 das OSI-Schichtenmodell entwickelt. Es beschreibt die Zusammenarbeit zwischen Hardware und Software in einem mehrschichtigen Aufbau, um die Kommunikation über Netzwerke zu realisieren.

PCI Peripheral Component Interconnect: Standardisierter 32 Bit breiter PC-BUS, der von Intel 1991 eingeführt wurde.

POSIX Portable Operating System Interface: Ein Standard für verschiedene Versionen von UNIX-Betriebssystemen. Er ermöglicht Softwareentwicklern die portable Erstellung von Anwendungen.

PIO Programmierbare Eingabe/Ausgabe: Bei dieser Art der Datenübertragung steuert die CPU den Zugriff auf die ATA-Festplatte im Unterschied zum DMA-Modus (engl. „programmable input/output“).

RAID Redundant Array of Inexpensive Devices: Fehlertolerante Systeme schützen Daten, indem sie diese duplizieren oder auf unterschiedlichen physischen Medien speichern. Dadurch bleiben Daten auch dann verfügbar, wenn das Datensystem teilweise ausfällt. Es gibt verschiedene Stufen der Redundanz: Datenverteilung, Plattenspiegelung, Ersatzsektoren und Clustering.

RAM Arbeitspeicher: Der flüchtige Arbeitsspeicher bzw. Hauptspeicher ist beliebig adressierbar und enthält nur solange Daten, wie elektrische Spannung anliegt. Die Spannung muß die Speicherzustände im RAM periodisch auffrischen. Die Zugriffszeit ist im Vergleich zum Massenspeicher sehr hoch (engl. „random access memory“).

RFS ReiserFS: In dieser Arbeit verwendete Abkürzung für ReiserFS.

Reiser ReiserFS: Modernes Dateisystem unter LINUX, das seine gesamte Organisation durch einen balancierten Baum vornimmt, der zu diesem Zweck auf die Datenblöcke der Platte abgebildet wird.

ROM Festwertspeicher: Festspeicher, der nur gelesen werden kann (engl. „read only memory“)

RLL Begrenzte Laufzeitmethode: Nachfolgeverfahren von MFM, bei dem aufeinanderfolgende Nullen nicht einzeln kodiert werden, sondern aus der Taktlänge auf die Anzahl von übertragenden Nullen geschlossen werden kann. Das geht aus Synchronisationsgründen nicht beliebig lange. Beim weitverbreiteten RLL 2,7 befinden sich zwischen zwei gesetzen Bits mindestens zwei und maximal sieben Nullbits, daher der Name des Verfahrens. Dadurch kann im Mittel trotz längerer Codewörter statistisch gesehen bei gleicher Taktrate 50% mehr Daten übertragen werden (engl. „run length limited“).

SAN Storage Area Network

SCSI Small Controller Systems Interface: Eine 1983 eingeführte Schnittstelle zum Anschluß von Peripheriegeräten an den Computer, insbesondere von Festplatten (auch Scanner, optische Laufwerke und Drucker).

S5FS System-V File System: Altes Dateisystem unter UNIX.

SMD Storage Module Drive

ST412 ST412-Festplatte: Modell 412 der Festplatte mit Schnittstelle zu entsprechenden Controller als Nachfolger zu ST506 („seagate technology“).

ST506 ST506-Festplatte: Modell 506 der Festplatte mit Schnittstelle zu entsprechendem Controller („seagate technology“).

ST506/412 ST506/412-Schnittstelle: Durch die Festplatten ST506 und ST412 von Seagate um 1980 herum definierte Schnittstelle („seagate technology“). : Dieser Übertragungsmodus wird in ATA-ATAPI-4 definiert und läßt das übertragende Gerät die Taktrate des IDE-Busses für die Dauer der Übertragung

übernehmen. Dadurch können beide Flanken des Taktsignals zur Übertragung benutzt werden, so daß die Datenrate verdoppelt wird. Dies macht ein 80-poliges Kabel mit 40 Masseleitungen und den üblichen 40 Leitungen des ATA-Standards notwendig.

VFS Virtuelles Dateisystem: Dateisystem, das bei UNIX-Systemen als Zwischenschicht fungiert und den Betrieb mehrerer Dateisysteme ermöglicht (engl. „virtuell filesystem switch“).

XFS Irix File System: Modernes Multimediadateisystem mit Echtzeitfähigkeiten von SGI, das seit jüngster Zeit nach LINUX portiert wird.

ZBR Zong Bit Recording: Verfahren neuerer Festplatten, bei dem die Sektoranzahl auf den äußeren Zylindern erhöht wird. Damit wird in diesen Bereichen die Datendichte effektiv erhöht. Bei gleicher Rotationsgeschwindigkeit steigt dadurch die Datenrate.

Literaturverzeichnis

[Bac86] M. J. Bach. *The Design of the Unix Operating System*. Prentice-Hall International Editions, 1. Auflage, 1986. ISBN 0-13-201757-1. 39, 47, 74, 79

[Bar00a] M. Bar. Journaling File Systems For Linux. *BYTE Magazine*, Mai 2000. Online unter http://www.byte.com/documents/s=365/byt20000524s0001/. 105

[Bar00b] M. Bar. Journaling Filesystems: The Future of Storage Under Linux. *Linux Magazine*, August 2000. Online unter http://www.linux-mag.com/2000-08/journaling_01.html. 105

[Bar01a] M. Bar. *Linux Files Systems*. Osborne/McGraw-Hill, 1. Auflage, 2001. ISBN 0-07-212955-7. 70, 97

[Bar01b] M. Bar. Linux Kernel Pillow Talk. *BYTE Magazine*, Oktober 2001. Online unter http://www.byte.com/documents/s=1436/byt20011024s0002/. 42, 143

[BBD+01] M. Beck, H. Böhme, M. Dziadzka, U. Kunitz, R. Magnus, C. Schröter, und D. Verworner. *Linux Kernelprogrammierung: Algorithmen und Strukturen der Version 2.4*. Addison-Wesley, 6. Auflage, 2001. ISBN 3-8273-1659-6. 43, 70

[BC01] D. P. Bovet und M. Cesati. *Understanding the Linux Kernel*. O'Reilly and Associates, 1. Auflage, 2001. ISBN 0-596-00002-2. 43, 70, 97

[Bes00a] S. Best. JFS Log: How the Journaled File System performs logging. Oktober 2000. Online unter http://oss.software.ibm.com/developer/opensource/jfs/project/pub/jfslog/jfslog.pdf. 106

[Bes00b] S. Best. JFS Overview: How the Journaled File System cuts system restart times to the quick. *IBM developerWorks*, Januar 2000. Online unter http://www-106.ibm.com/developerworks/library/jfs.html. 106

[BHB99] I. T. Bowman, R. C. Holt, und N.V. Brewster. Linux as a Case Study: Its Extracted Software Architecture. *ICSE'99, Los Angeles, May 16-22, 1999*, 1999. Online unter http://plg.uwaterloo.ca/~itbowman/papers/linuxcase.html. 43

[BHK+91] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, und J. K. Ousterhout. Measurements of a Distributed File System. In *Proceedings*

of 13th ACM Symposium on Operating Systems Principles, Seiten 198–212. Association for Computing Machinery SIGOPS, 1991. 132

[BHS95] T. Blackwell, J. Harris, und M. Seltzer. Heuristic Cleaning Algorithms in Log-Structured File Systems. *Proceedings of the USENIX 1995 Technical Conference, New Orleans, LA, USA*, Seiten 277–288, 1995. Online unter http://citeseer.nj.nec.com/blackwell95heuristic.html. 106

[BK00] S. Best und D. Kleikamp. JFS Layout: How the Journaled File System handles the on-disk layout. *IBM developerWorks*, Mai 2000. Online unter http://www-106.ibm.com/developerworks/library/jfslayout/. 106

[Bra98] P. J. Braam. Linux Virtual File System. http://www.coda.cs.cmu.edu/doc/talks/linuxvfs/, 1998. 70

[Bra01] R. Brause. *Betriebssysteme - Grundlagen und Konzepte*. Springer Verlag, 2. Auflage, 2001. ISBN 3-540-67598-1. 36, 37

[Bro99] N. Brown. The Linux Virtual File-system Layer. http://www.cse.unsw.edu.au/~neilb/oss/linux-commentary/vfs.html, 1999. 70

[Bög96] H. Bögeholz. Platten-Karussel; Aktuelle Festplatten im Vergleich. *c't, Magazin für Computertechnik, Heise Verlag*, April, 1996. 20

[Com94] T10 Technical Committee. ATA-1 attachment interface for disk drives. http://www.t13.org/project/d0791r4c.pdf, 1994. This standard defines the AT Attachment Interface. It defines an integrated bus interface between disk drives and host processors. It provides a common point of attachment for systems manufacturers, systems integrators, and suppliers of intelligent peripherals. AT Attachment Interface for Disk Drives (ATA-1) was withdrawn as a standard on 6 August 1999. 18, 20, 21, 23

[Com96] T10 Technical Committee. ATA-2 attachment interface for disk drives. http://www.t13.org/project/d0948r4c.pdf, 1996. This standard defines an integrated interface between devices and host processors. It provides a common point of attachment for systems manufacturers, systems integrators, and suppliers of intelligent devices. It was withdrawn as a standard in 2001. 21

[Com01] T10 Technical Committee. SCSI-3 Standards Architecture. http://www.t10.org/scsi-3.htm, 2001. 6, 9, 24

[Cox99a] A. Cox. Advanced SCSI Drivers And Other Tales. *Linux Magazine*, September 1999. Online unter http://www.linux-mag.com/1999-09/gear_01.html. 72

[Cox99b] A. Cox. An Introduction to SCSI Drivers. *Linux Magazine*, August 1999. Online unter http://www.linux-mag.com/1999-08/gear_01.html. 72

[CTT94] R. Card, T. Ts'o, und S. Tweedie. Design and Implementation of the Second Extended Filesystem. 1994. Online unter http://web.mit.edu/tytso/www/linux/ext2intro.html. 89, 97

[Cus94] H. Custer. *Inside the Windows NT File System*. Microsoft Press, 1994. 108

[Del00] M. Dell'Omodarme. Journalling Filesystems for Linux. *LinuxGazette, Published by Linux Journal*, 68, Juli 2000. Online unter http://www.linuxgazette.com/issue68/dellomodarme.html. 105

[Dem01] K. Dembowski. *Computerschnittstellen und Bussysteme*. Hüthig Verlag, Heidelberg, 2. Auflage, 2001. ISBN 3-778-52782-7. 8

[Die00] O. Diedrich. Von Blöcken und Knoten: Das Linux-Dateisystem. *CT, Heise Verlag*, 6:138, 2000. 97

[Flo01] J. I. S. Florido. Journal File Systems. *LinuxGazette, Published by Linux Journal*, 55, Juli 2001. Online unter http://www.linuxgazette.com/issue55/florido.html. 105

[FSF01] Free Software Foundation. The GNU C Library Manual. http://www.gnu.org/manual/glibc-2.2.3/libc.html, 2001. 140

[Gil01] D. Gilbert. The Linux 2.4 SCSI subsystem HOWTO. http://www.linuxdoc.org/HOWTO/SCSI-2.4-HOWTO/index.html, 2001. 72

[GK97] Gregory R. Ganger und M. Frans Kaashoek. Embedded Inodes and Explicit Grouping: Exploiting Disk Bandwidth for Small Files. *USENIX Annual Technical Conference*, Seiten 1–17, 1997. Online unter http://citeseer.nj.nec.com/ganger97embedded.html. 108

[GMSP00] Gregory R. Ganger, Marshall Kirk McKusick, Craig A. N. Soules, und Yale N. Patt. Soft Updates: A Solution to the Metadata Update Problem in File Systems. *ACM Transactions on Computer Systems*, 18(2):127–153, 2000. 88

[GP94] G. R. Ganger und Y. N. Patt. Metadata Update Performance in File Systems. *Proceedings of the USENIX 1994 Symposium on Operating Systems Design and Implementation, Monterey, CA, USA*, Seiten 49–60, 1994. Online unter http://citeseer.nj.nec.com/ganger94metadata.html. 88

[Hal01] J. Hall. Gedanken zum Linux-Jubiläum: Happy Birthday. *IX, Heise Verlag*, 10:88, 2001. Online unter http://www.heise.de/ix/artikel/2001/10/088/. 40

[Haw01] C. Hawklord. Filestore: System V. http://www.hawklord.uklinux.net/internals/filestore/fs3.htm, 2001. 79

[HK01] T. Harris und K. Koehntopp. Linux Partition HOWTO. http://surfer.nmr.mgh.harvard.edu/partition/Partition.html, 2001. 73

[HR99] T. Härder und E. Rahm. *Datenbanksysteme - Konzepte und Techniken der Implementierung*. Springer Verlag, 1. Auflage, 1999. ISBN 3-540-65040-7. 3, 99, 101

[Ins85] American National Standards Institute. Small Computer System Interface (SCSI-1) - Draft Proposed December 16, 1985. ftp://ftp.cs.uni-sb.de/misc/doc/SCSI/SCSI-1-Rev17B/SCSI17B.Z, 1985. Abstract: This standard defines mechanical, electrical, and functional requirements for attaching small computers with each other and with low- to medium-performance intelligent peripherals such as rigid disks, flexible disks, magnetic tape devices, printers, and optical disks. The resulting interface facilitates the interconnection of small computers and intelligent peripherals and thus provides a common interface specification for both systems integrators and suppliers of intelligent peripherals. Revision 17B consists of changes made by the X3T9.2 task group at their December 10, 1985 meeting. These changes were made in order to make the X3T9.2 draft proposed standard consistent with the ISO/TC97/SC13 draft proposal. See page 1.1 for a changed page list. 24

[Ins90] American National Standards Institute. Small Computer System Interface (SCSI-2) - Draft Proposed March 9, 1990. ftp://ftp.cs.uni-sb.de/misc/doc/SCSI/SCSI-2/10ctxt.tar.Z, 1990. Abstract: This standard defines mechanical, electrical, and functional requirements for attaching physically small computers with each other and with intelligent peripherals such as rigid disks, flexible disks, magnetic tape devices, printers, optical disks, and scanners. The resulting interface facilitates the interconnection of physically small computers and intelligent peripherals and thus provides a common interface specification for both systems integrators and suppliers of intelligent peripherals. This is a draft proposed American National Standard of Accredited Standards Committee X3. As such, this is not a completed standard. The X3T9 Technical Committee may modify this document as a result of comments received during X3 approval as a standard. v, 24, 29, 31, 33, 34

[Joh01] M. K. Johnson. Whitepaper: Red Hat's New Journaling File System: ext3. *Red Hat Whitepapers*, 2001. Online unter http://www.redhat.com/support/wpapers/redhat/ext3/index.html. 106

[Kle86] S. Kleiman. Vnodes: An Architecture for Multiple File System in SUN UNIX. In *Proceedings of the 1986 USENIX Summer Technical Conference*. 1986. 48, 70

[KR78] B. W. Kernighan und D. M. Ritchie. *The C Programming Language*. Prentice Hall, Englewood Cliffs N.Y., 1. Auflage, 1978. 38

[Kro00] A. Kroschel. *Festplatten*. Professional Series. Franzis' Verlag, 1. Auflage, 2000. ISBN 3-7723-6215-X. 24

[Köh94] K. Köhntopp. UNIX Dateisysteme. *CT, Heise Verlag*, 2, 1994. 47

[Köh96] K. Köhntopp. Dynamisierung im UNIX-Dateisystem, Dezember 1996. Online unter http://www.koehntopp.de/kris/artikel/diplom/. 105

[Lan01] H. Landis. ATA-ATAPI.COM. http://ata-atapi.com/, 2001. Information, Software and Consulting Service for ATA/ATAPI (IDE/EIDE), SCSI, 1394 and Other Interfaces. 9

[Lev99] J. R. Levine. *Linkers and Loaders*. Morgan-Kauffman, 1. Auflage, 1999. ISBN 1-55860-496-0. 140

[LMKQ89] S. J. Leffler, M. K. McKusick, M. J. Karels, und J. S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 2. Auflage, 1989. 80, 88

[Loi01a] C. Loizides. Bericht für August: Betriebssystem-Probleme bei den simulationen. Private Kommunikation, August 2001. 143

[Loi01b] C. Loizides. Das Hilfsprogramm: agesystem3, Beschreibung und Programmcode. http://www.informatik.uni-frankfurt.de/~loizides/reiserfs/agesystem3.html, 2001. Dieses Tool ist der Nachfolger von „agesystem“ und führt auf verschiedene Weisen Änderungen an einem Dateisystem durch. 133

[Loi01c] C. Loizides. Das Hilfsprogramm: fibmap, Beschreibung und Programmcode. http://www.informatik.uni-frankfurt.de/~loizides/reiserfs/fibmap.html, 2001. Dieses Tool bewertet auf verschiedene Arten das Blocklayout einer Datei oder Partition. 137

[Loi01d] C. Loizides. Das Hilfsprogramm: read, Beschreibung und Programmcode. http://www.informatik.uni-frankfurt.de/~loizides/reiserfs/read.html, 2001. Dieses Tool führt drei verschiedene Lesetests an einer gegebenen Partition aus. 138

[Loi01e] C. Loizides. Journaling-Filesystem Fragmentation Project. http://www.informatik.uni-frankfurt.de/~loizides/reiserfs/, 2001. 132, 143, 149

[Loi01f] C. Loizides. Journaling-Filesystem Fragmentation Project: Aging Tests. http://www.informatik.uni-frankfurt.de/~loizides/reiserfs/agetest.html, 2001. 159

[Loi01g] C. Loizides. Journaling-Filesystem Fragmentation Project: Append Tests. http://www.informatik.uni-frankfurt.de/~loizides/reiserfs/apptest.html, 2001. 155

[Loi01h] C. Loizides. Journaling-Filesystem Fragmentation Project: Gauge Performance. http://www.informatik.uni-frankfurt.de/~loizides/reiserfs/gaugeperf.html, 2001. 149

[MEL+00] J. Mostek, B. Earl, S. Levine, S. Lord, R. Cattelan, K. McDonell, T. Kline, B. Gaffey, und R. Ananthanarayanan. Porint The SGI XFS File System To Linux. In *Proceedings of FREENIX Track: 2000 USENIX Annual Technical Conference*. San Diego, CA, USA, Juni 18–23 2000. Online unter http://www.usenix.org/events/usenix2000/freenix/full_papers/mostek/mostek.pdf. 106

[Mes93] H. P. Messmer. *PC-Hardwarebuch - Aufbau, Funktionsweise, Programmierung: Ein Handbuch nicht nur für Profis*. Addison-Wesley, 2. Auflage, 1993. ISBN 3-89319-528-9. 16, 18

[Met97] Rodney Van Meter. Observing the Effects of Multi-Zone Disks. In *Proceedings of the 1997 USENIX Conference*, Seiten 19–30. 1997. 161

[MJLF84] M. K. McKusick, W. N. Joy, S. J. Leffler, und R. S. Fabry. A Fast File System for Unix. *ACM - Transaction on Computer Systems*, 2(3):181–197, 1984. 79, 80, 81, 85, 88

[NS01] J. Nehmer und P. Sturm. *Systemsoftware - Grundlagen moderner Betriebssysteme*. dpunkt.verlag, 2. Auflage, 2001. ISBN 3-89864-115-5. 36, 37

[OW93] T. Ottmann und P. Widmayer. *Algorithmen und Datenstrukturen*. Wissenschaftsverlag, 2. Auflage, 1993. ISBN 3-411-16602-9. 100

[PPT+00] R. Pike, D. Presotto, K. Thompson, H. Trickey, und P. Winterbottom. The Use of Name Spaces in Plan 9, 2000. Informationen online unter http://www.cs.bell-labs.com/plan9dist/. 106

[PW85] R. Pike und P. Weinberger. The Hideous Name. In *Summer 1995 USENIX Conference Proceedings*, Seiten 563–568. Salt Lake City, Utah, 1985. 106

[RC01] A. Rubini und J. Corbet. *Linux Device Drivers*. Addison-Wesley, 2. Auflage, 2001. ISBN 0-59600-008-1. Online einsehbar unter http://www.xml.com/ldd/chapter/book/. 43, 70

[Rei01a] H. Reiser. Future Vsision. http://www.namesys.com/whitepaper.html, Januar 2001. 106

[Rei01b] H. Reiser. ReiserFS. http://www.namesys.com/res_whol.shtml, Januar 2001. 119

[Rei01c] H. Reiser. ReiserFS Vs.4. http://www.namesys.com/v4/v4.html, September 2001. 119

[RO92] Mendel Rosenblum und John K. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992. Online unter http://citeseer.nj.nec.com/rosenblum91design.html. 105

[Rob01] D. Robbins. Advanced Filesystem Implementor's Guide, Part 7: Introducing ext3. November 2001. Online unter http://www-106.ibm.com/developerworks/linux/library/l-fs7/. 106

[Ron01] F. Ronneburg. Debian GNU/Linux Anwenderhandbuch. http://www.openoffice.de/linux/buch/, 2001. 40

[RT74] D. M. Ritchie und K. Thompson. The UNIX Time-Sharing System. *Communications of the ACM*, 17(7):365–375, 1974. 38, 74

[Sch95] F. Schmidt. *The SCSI Bus and IDE Interface*. Addison-Wesley, 1. Auflage, 1995. ISBN 0-201-42284-0. 8, 16, 18, 20, 25, 26, 28, 29, 33, 35

[SDH+96] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, und G. Peck. Scalability in the XFS File System. In *Proceedings of the USENIX 1996 Technical Conference*, Seiten 1–14. San Diego, CA, USA, Januar 22–26 1996. 106, 108

[SGS+97] C. Small, N. Ghosh, H. Saleeb, M. Seltzer, und A. K. Smith. Does System Research Measure Up? *Computer Science Department Technical Report, Harvard University*, TR-16-97, 1997. 128

[SH99] G. Saacke und A. Heuer. *Datenbanken: Implementierungstechniken*. MITP Verlag, 1. Auflage, 1999. ISBN 3-8266-0513-6. 100, 107

[SKSZ99] M. Seltzer, D. Krinsky, K. Smith, und X. Zhang. The Case for Application-Specific Benchmarking. *Proceedings of the 1999 Workshop on Hot Topics in Operating Systems, Rico, AZ, 1999*, 1999. Online unter http://citeseer.nj.nec.com/seltzer99case.html. 128

[Smi01] K. A. Smith. *Workload Specific File System Benchmarks*. Doktorarbeit, Computer Science Department, Harvard University, 2001. 129

[SS94] A. K. Smith und M. Seltzer. File Layout and File System Performance. *Computer Science Department Technical Report, Harvard University*, TR-35-94, 1994. 88, 127

[SS95] M. Seltzer und K. A. Smith. File System Logging Versus Clustering: A Performance Comparism. *Proceedings of the 1995 USENIX Technical Conference, New Orleans, Jan. 1995*, 1995. 106, 127

[SS96] A. K. Smith und M. Seltzer. A Comparison of FFS Disk Allocation Policies. *Proceedings of the 1996 USENIX Technical Conference, San Diego*, Januar 1996. 88, 128

[SS97] A. K. Smith und M. Seltzer. File System Aging - Increasing the Relevance of File System Benchmarks. *Proceedings of the 1997 ACM SIGMETRICS Conference*, Juni 1997. 128

[SSS99] E. Shriver, C. Small, und K. Smith. Why Does File System Prefetching Work. In *USENIX 1999 Technical Conference, Monterey, California*, Seiten 71–83. 1999. 163

[Tan87] A. S. Tanenbaum. *Operating Systems - Design and Implementation*. Prentice-Hall International Editions. Prentice-Hall, 3. Auflage, 1987. ISBN 0-13-637331-3. 39, 89

[Tan92] A. S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall International Editions. Prentice-Hall, 4. Auflage, 1992. ISBN 0-13-595752-4. 39

[Tec00] Lucent Technologies. The Creation of the UNIX* Operating System: The famous PDP-7 comes to the rescue. http://www.bell-labs.com/history/unix/pdp7.html, 2000. 38

[TS97] D. Tang und M. Seltzer. Lies, Damned Lies and File System Benchmarks. *Computer Science Department Technical Report, Harvard University*, TR-16-97, 1997. 128

[Vah95] U. Vahalia. *UNIX Internals: The New Frontiers*. Prentice Hill, 1. Auflage, 1995. ISBN 0-13-101908-2. 38, 39, 70

[vR01] R. van Riel. Page replacement in Linux 2.4 memory management. 2001. 42, 54

[WGPW96] B. L. Worthington, G. R. Ganger, Y. N. Patt, und J. Wilkes. On-Line Extraction of SCSI Disk Drive Parameters. Technischer Bericht CSE-TR-323-96, 19 1996. 162

[Whe00] D. A. Wheeler. Program Library HOWTO. http://www.dwheeler.com/program-library/, 2000. 140

Danksagung

Ich bedanke mich bei Herrn Stephan Wolf, Vorstandsvorsitzender der Innovate Software AG, für die mir für dieses Projekt zur Verfügung gestellten finanziellen Mittel.

Mein besonderer Dank gilt Herrn Prof. Dr. Kurt Geihs, Technische Universität Berlin (vorher Universität Frankfurt), und Herrn Lutz Vieweg, Leiter der Forschungsabteilung der Innovativen Software AG, für die Betreuung während der letzten Monate.