# Dynamisierung im UNIX-Dateisystem

Diplomarbeit vorgelegt von

Kristian Köhntopp

Institut für Informatik und Praktische Mathematik
der Christian-Albrechts-Universität zu Kiel

Dezember 1996

# Inhaltsverzeichnis


**1 Einführung** **7**

**2 Festplattenhardware** **9**

2.1 Geometrie und Sektoradressierung . . . . . 10

2.2 Datenrate und Aufzeichnungsdichte . . . . . 12

2.3 RAID . . . . . 16

2.4 Partitionen . . . . . 18

2.5 Fragmentierung . . . . . 20

**3 Konventionelle UNIX–Dateisysteme** **22**

3.1 Struktur des System V–Dateisystems . . . . . 22

3.1.1 Inodes und Datenblöcke . . . . . 22

3.1.2 Verzeichnisse und Pfadnamen . . . . . 24

3.2 BSD Fast Filing System . . . . . 27

3.2.1 Inodes und Datenblöcke . . . . . 29

3.2.2 Verzeichnisse und Pfadnamen . . . . . 34

3.3 Second Extended Filesystem . . . . . 35

3.3.1 Inodes und Datenblöcke . . . . . 36

3.3.2 Verzeichnisse und Pfadnamen . . . . . 40

**4 Dynamisierung der Anzahl der Inodes** **41**

4.1 Vorüberlegungen . . . . . 42

4.1.1 Methoden der Strukturbeschreibung . . . . . 42

4.1.2 Ansatz für eine dynamisch wachsende Inodetabelle . . . . . 43

4.1.3 Positionierung von Inodes und Datenblöcken . . . 44
4.1.4 Auffinden freier Inodes . . . 48
4.1.5 Freigabe von Inodes . . . 51
4.1.6 Selbstbezüglichkeit . . . 52
4.2 Modell des herkömmlichen Dateisystems . . . 53
4.2.1 Konstanten und elementare Funktionen . . . 54
4.2.2 Blockgruppen . . . 56
4.2.3 Superblöcke . . . 57
4.2.4 Deskriptoren und Bitmaps . . . 58
4.2.5 Inodetabelle . . . 60
4.2.6 Dateien . . . 62
4.2.7 Allozierung und Freigabe von Inodes . . . 65
4.2.8 Allozierung und Freigabe von Blöcken . . . 68
4.2.9 Vergrößerung von Dateien . . . 71
4.2.10 Integritätsbedingungen . . . 75
4.3 Modellierung der Änderungen . . . 78
4.3.1 Konstanten und elementare Funktionen . . . 79
4.3.2 Blockgruppen . . . 79
4.3.3 Superblöcke . . . 80
4.3.4 Deskriptoren und Bitmaps . . . 80
4.3.5 Inodetabelle . . . 81
4.3.6 Allozierung und Freigabe von Inodes und Blöcken . . . 84
4.3.7 Dynamische Inodetabelle . . . 88
4.3.8 Integritätsbedingungen . . . 89

**5 Dateisystem mit variabler Anzahl Inodes 90**
5.1 Entwicklungsumgebung . . . 90
5.2 Vorgehensweise . . . 91
5.2.1 Dateisystemsimulator als NFS–Server . . . 91
5.2.2 Dateisystem als Kernelmodul . . . 93

5.3 Die Kernelumgebung . . . 93
5.3.1 Das Interface zum VFS . . . 94
5.3.2 Von Dateisystemen verwendbare Kernelfunktionen . . . 98
5.4 Aufbau und Funktionsweise des ext2–Dateisystems . . . 99
5.4.1 Aktivierung und Deaktivierung des Dateisystems . . . 99
5.4.2 Aktivierung und Deaktivierung von Inodes . . . 100
5.4.3 Belegung und Freigabe von Inodes . . . 102
5.4.4 Belegung und Freigabe von Blöcken . . . 103
5.4.5 Verwaltung der Bitmaps . . . 104
5.5 Von `ext2` zu `ext3` . . . 104
5.5.1 Erzeugung eines Entwicklungsumfeldes . . . 104
5.5.2 Aggressive Vorbestellung von Blöcken . . . 107
5.5.3 Änderungen an den Datenstrukturen . . . 110
5.5.4 Aktivierung und Deaktivierung des Dateisystems . . . 112
5.5.5 Aktivierung und Deaktivierung von Inodes . . . 113
5.5.6 Belegung und Freigabe von Inodes . . . 114
5.5.7 Erweitern der Inodetabelle . . . 118
5.5.8 Verwaltung der Bitmaps . . . 120
5.6 Aufbau und Funktionsweise der Hilfsprogramme . . . 121
5.6.1 Anlegen eines Dateisystems . . . 123
5.6.2 Dateisystemprüfung . . . 124

**6 Dateisysteme im Vergleich 125**
6.1 Meßverfahren . . . 125
6.2 Meßergebnisse . . . 127
6.2.1 Gemessene Konfigurationen . . . 127
6.2.2 Ergebnisse von Test 1 . . . 129
6.2.3 Ergebnisse von Test 2 . . . 129
6.2.4 Ergebnisse von Test 3 . . . 130
6.2.5 Ergebnisse von Test 4 . . . 132
6.2.6 Strukturelle Unterschiede . . . 134
6.2.7 Bewertung . . . 136

**7 Zusammenfassung und Ausblick** **137**

7.1 Zusammenfassung . . . 137

7.2 Erweiterungsmöglichkeiten . . . 138

7.3 Verwandte Arbeiten . . . 138

7.3.1 LFS — Log–Structured File System in Sprite und BSD . . . 139

7.3.2 Episode – lokale Dateisystemkomponente von OSF DCE . . . 141

7.3.3 XFS — auf B+–Bäumen basierendes Dateisystem in Irix 5.3 . . . 142

7.4 Ausblick . . . 144

**Literaturverzeichnis** **145**

# Abbildungsverzeichnis


2.1 Aufbau einer Festplatte . . . . . 9
2.2 Gebräuchliche Abbildungen von LBA– auf CHS–Koordinaten (*mappings*) . . . 10
2.3 Herkömmliche Festplatte und Festplatte mit ZBR . . . . . 13
2.4 Transferleistung einer ZBR–Festplatte . . . . . 14
2.5 Geometrieauskunft einer Maxtor 7850 AR Festplatte . . . . . 15
2.6 RAID–Controller als SCSI–auf–SCSI–Adapter . . . . . 17
2.7 Verteilung der Streifen bei RAID 0 . . . . . 19
2.8 Verteilung der Streifen bei RAID 1 . . . . . 19
2.9 Verteilung der Streifen bei RAID 3 . . . . . 19
2.10 Verteilung der Streifen bei RAID 5 . . . . . 19
2.11 Fragmentierung von Dateien . . . . . 21

3.1 Inode–Datenstruktur im System V–Dateisystem . . . . . 23
3.2 Inode und Datenblöcke . . . . . 24
3.3 Verzeichnis in S5FS . . . . . 25
3.4 Auflösung eines absoluten Pfadnamens . . . . . 26
3.5 Inode–Datenstruktur Berkeley FFS . . . . . 28
3.6 Vergleich der Struktur von S5FS und FFS . . . . . 30
3.7 Rotational Delay . . . . . 31
3.8 Wachsende fragmentierte Dateien . . . . . 32
3.9 Verwaltung freier Blöcke durch Free Lists oder durch Bitmaps . . . . . 33
3.10 Verzeichniseinträge in FFS . . . . . 34
3.11 Aufbau einer Blockgruppe im Linux Second Extended Filesystem . . . . . 36
3.12 Deskriptor für eine Blockgruppe in ext2 . . . . . 37

3.13 Vorbestellung von Blöcken . . . . . . . . . . 38
3.14 Inode–Datenstruktur in ext2 . . . . . . . . . . 40

4.1 Mappingfunktion für die Inodetabelle im S5FS . . . . . . . . . . 42
4.2 Dateisystem mit Inodetabellendatei . . . . . . . . . . 44
4.3 Positionierung neuer Inodes und Datenblöcke, Verfahren 1 . . . . . . . . . . 45
4.4 Positionierung neuer Inodes und Datenblöcke, Verfahren 2 . . . . . . . . . . 47
4.5 Hierarchie von Verwaltungsstrukturen . . . . . . . . . . 48

5.1 Verwendete Konfiguration . . . . . . . . . . 90
5.2 Aufteilung der verwendeten Dateisysteme . . . . . . . . . . 91
5.3 Gliederungsschichten für ein Dateisystem . . . . . . . . . . 92
5.4 Unterstützte Dateisystemtypen . . . . . . . . . . 94
5.5 VFS Inode und File (gekürzt) . . . . . . . . . . 95
5.6 VFS Superblock und Filesystemtyp (gekürzt) . . . . . . . . . . 96
5.7 Verkettung der VFS–Strukturen . . . . . . . . . . 97
5.8 Unabhängige Übersetzung eines Dateisystems als Kernelmodul . . . . . . . . . . 105
5.9 Ausgabe von `/proc/filesystems` . . . . . . . . . . 106
5.10 `ioctl()`–Codes zur Steuerung des Debugsystems . . . . . . . . . . 107
5.11 Debuggingsausgabe des ext3–Dateisystems . . . . . . . . . . 108
5.12 Wirkungsweise der Preallocation . . . . . . . . . . 109
5.13 Reservierte Inodes im ext3–Dateisystem . . . . . . . . . . 111

6.1 Geschwindigkeitsvergleich ext2/ext3, Gesamtzeit und Systemzeit . . . . . . . . . . 128
6.2 Geschwindigkeitsvergleich ext2/ext3, Tabelle der Meßwerte . . . . . . . . . . 129
6.3 Größe der Inodetabelle nach Test 2 . . . . . . . . . . 130
6.4 ext3-m: Auslastung der Blockgruppen nach Test 3 . . . . . . . . . . 131
6.5 ext3-o/ext3-m: Zusammenhang der Inodetabellendatei nach Test 3 . . . . . . . . . . 132
6.6 Auswirkungen der aggressiven Vorbestellung von Blöcken . . . . . . . . . . 133
6.7 Zerstörung und Rekonstruktion eines ext2–Dateisystems . . . . . . . . . . 135

# Kapitel 1

# Einführung

Dateisysteme sind ein wichtiger Bestandteil von Betriebssystemen. Sie stellen insbesondere im Kontext von UNIX [Ta87] [Th78] den zentralen Mechanismus zur Benennung von Betriebssystemobjekten zur Verfügung[1] und realisieren zugleich einen persistenten Datenspeicher. Bei der Entwicklung eines Dateisystems kommt es daher nicht nur auf Performance und Flexibilität an, sondern auch auf Zuverlässigkeit.

Die vorliegende Diplomarbeit ist Bestandteil eines größeren Projektes, das sich mit der formalen Modellierung von Betriebssystemen beschäftigt. Durch die vollständige formale Spezifikation aller Teile eines Betriebssystems mit funktionalen Methoden werden unterschiedliche Implementierungen von Betriebssystemmechanismen vergleichbar und einer systematischen Einordnung zugänglich. Zugleich bildet eine solche Modellierung die Grundlage eines Verifikationsprozesses, der integraler Bestandteil einer revisionsfähigen Softwareentwicklung sein sollte. Im Rahmen dieses Projektes entstehen zur Zeit eine Studienarbeit zum Thema Speicherverwaltung [Wu i.Vorb.], zwei Diplomarbeiten zu verschiedenen Aspekten von Kernel– und Userlevel–Threads [Wa i.Vorb.] [Hp i.Vorb.] sowie eine Dissertation, die sich mit Sicherheitsaspekten von UNIX [Hd i.Vorb.] befaßt.

Diese Arbeit behandelt die Dynamisierung von Datenstrukturen im UNIX–Dateisystem. In konventionellen Implementierungen des UNIX–Dateisystems muß die maximal mögliche Anzahl von Dateien bereits beim Anlegen des Systems festgesetzt werden. Stellt sich später ein größerer Bedarf an Dateien heraus, kann diese Grenze nicht ohne eine Betriebsunterbrechung geändert werden. Das Ziel der Arbeit ist es, ein existierendes UNIX–Dateisystem so zu erweitern, daß die maximale Anzahl von Dateien den Bedürfnissen zur Laufzeit im Rahmen der durch die Hardware vorgegebenen Begrenzungen angepaßt wird.

Um dieses Ziel zu erreichen, wurde ein existierendes Dateisystem mit formalen Methoden modelliert und das entstandene Modell so modifiziert, daß es den neuen Anforderungen zur dynamischen Veränderung der maximalen Anzahl von Dateien genügt. Anschließend wurden diese Veränderungen am Modell auf die vorliegenden Originalquelltexte übertragen. Durch eine vergleichende Analyse von altem und neuem Dateisystem wurde geprüft, inwieweit der höhere

[1] „In UNIX everything is a file.“, Ken Thompson, [Pi90]

Grad der Flexibilisierung im neuen System Leistungseinbußen nach sich zieht. Die Motivation für diese Vorgehensweise resultiert aus den Bestrebungen, formale Modellierungsmethoden und praktische Implementierungen zu integrieren und auf diese Weise zu einem leistungsfähigen und real einsetzbaren System zu kommen.

Diese Vorgehensweise spiegelt sich in der Gliederung der Arbeit direkt wieder:

In Kapitel 2 wird zunächst ein kurzer Überblick über die derzeit verfügbare Festplattentechnik gegeben, denn diese Technik bildet die Rahmenbedingungen für die Entwicklung von Dateisystemen.

Im folgenden Kapitel 3 werden konventionelle UNIX–Dateisysteme mit einer statischen Maximalzahl von Dateien pro Dateisystem eingeführt. Insbesondere wird das *Second Extended Filesystem* „ext2" [Ca95] des Linux–Kernels [Li96] vorgestellt, das die Ausgangsbasis für die Entwicklung des in dieser Arbeit entwickelten Dateisystems ist. Außerdem werden die grundlegenden Datenstrukturen von UNIX–Dateisystemen und ihr Zusammenspiel zur Organisation der Daten beschrieben.

Die Motivation für ein dynamisches Dateisystem wird in Kapitel 4 dargestellt. Weiterhin geht dieses Kapitel auf die verschiedenen Möglichkeiten ein, eine solche Dynamisierung zu realisieren. Die Bildung eines formalen Modells des Linux *Second Extended Filesystem* schließt sich an. Dazu wurde der vorhandene Quelltext des Linux–Dateisystems analysiert und daraus ein funktionales Modell des Dateisystems entwickelt. Das darauf aufbauende funktionale Modell eines Dateisystems mit einer variablen Maximalzahl von Dateien wird im Anschluß erläutert.

Die Abbildung dieses Modells auf in der Linux–Kernelumgebung lauffähigen Code wird in Kapitel 5 vorgenommen. Nach einer Einführung in die Struktur der Kernelumgebung und das für Dateisysteme verwendbare Programmierinterface werden der Kernelcode des weiterentwickelten Dateisystems „ext3" sowie die im Usermode ablaufenden Werkzeuge zur Pflege von Dateisystemen diskutiert.

In Kapitel 6 werden die Geschwindigkeit und verschiedene strukturelle Aspekte beider Dateisysteme miteinander verglichen und bewertet.

Abschließend faßt das Kapitel 7 die Ergebnisse dieser Arbeit zusammen. Darüber hinaus werden verwandte Arbeiten zu diesem Thema vorgestellt und Erweiterungsmöglichkeiten aufgezeigt.

# Kapitel 2

# Festplattenhardware

Dieses Kapitel gibt einen kurzen Überblick über die aktuelle Festplattentechnik.

Festplatten für PCs und Workstations bestehen aus einem Stapel von starr verbundenen Metallscheiben, die beidseitig mit extrem fein polierten Metalloxiden beschichtet sind. Ein Kamm von beweglichen Schreib–/Leseköpfen greift seitlich in den rotierenden Plattenstapel hinein und kann so konzentrische Kreise von Magnetisierungsmustern abtasten oder erzeugen.

Der von einem Kopf (*head*) in einer Umdrehung überstrichene Bereich der Festplatte heißt Spur (*track*). Die im Plattenstapel übereinander angeordneten Spuren bilden einen Zylinder (*cylinder*). Abbildung 2.1 illustriert dies: Der Kamm der Schreib–/Leseköpfe erzeugt konzentrische Kreise auf der Oberfläche des Mediums. Denkt man sich den Rest der Platte weg, erscheinen diese Kreise als ein Zylinder im Raum.

Festplatten können Daten nicht beliebig fein aufgelöst lesen oder schreiben. Statt dessen ist eine Spur in Sektoren (*sectors*) fester Größe unterteilt, die die kleinste adressierbare Einheit darstellen. Gebräuchlich ist dabei eine Größe von 512 Byte. Gelegentlich findet man auch Festplatten mit Sektorgrößen von 1024 oder gar 2048 Byte.

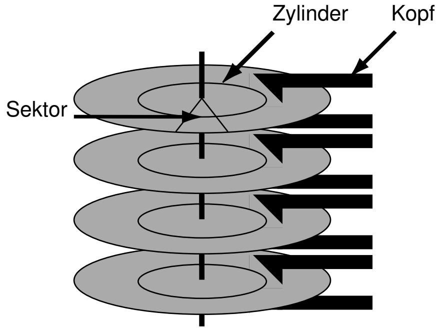



Abbildung 2.1: Aufbau einer Festplatte: Ein Kamm von Schreib–/Leseköpfen greift seitlich in einen Plattenstapel hinein.

## 2.1 Geometrie und Sektoradressierung

Ein Sektor auf einer Festplatte ist durch die drei Koordinaten Zylindernummer, Kopfnummer und Sektornummer (nach den englischen Bezeichnungen *CHS–Modell* genannt) bestimmt. Die maximalen CHS–Koordinaten sind dabei Kenngrößen der Festplatte und werden in der Literatur gewöhnlich als Geometriedaten (*disk geometry*) der Platte bezeichnet. Festplatten mit SCSI–Schnittstelle und neuere Festplatten mit der im PC–Bereich üblichen IDE/EIDE–Schnittstelle können dem Gerätetreiber auf Anfrage ihre Geometriedaten übermitteln. Manche Gerätetreiber benutzen diese Information, um Plattenzugriffe zu optimieren. Außerdem stehen sie Wartungsprogrammen zum Anlegen von Partitionen oder zur Generierung von Dateisystemen über den Gerätetreiber zur Verfügung. Auf diese Weise läßt sich die Konfiguration ganz oder teilweise automatisieren.

Die Adressierung von Sektoren durch den Gerätetreiber erfolgt jedoch bei allen SCSI–Festplatten und zunehmend auch bei IDE/EIDE–Festplatten über lineare Blockadressen (*linear block addresses, LBA*). Dabei handelt es sich um Blocknummern zwischen Null und einer Maximalzahl, die von der Kapazität der Platte abhängig ist. Die Übersetzung von linearen Blockadressen in CHS–Koordinaten erfolgte früher im Gerätetreiber. Im Jargon der Hersteller wird die Abbildung von linearen Blockadressen auf CHS–Koordinaten als *mapping* der Platte bezeichnet. Dabei ist es aufgrund der zunehmenden Komponentenintegration inzwischen üblich, daß diese Umrechnung von der Festplatte selbst vorgenommen wird und sie vorzugsweise mit LBA angesteuert werden sollte.

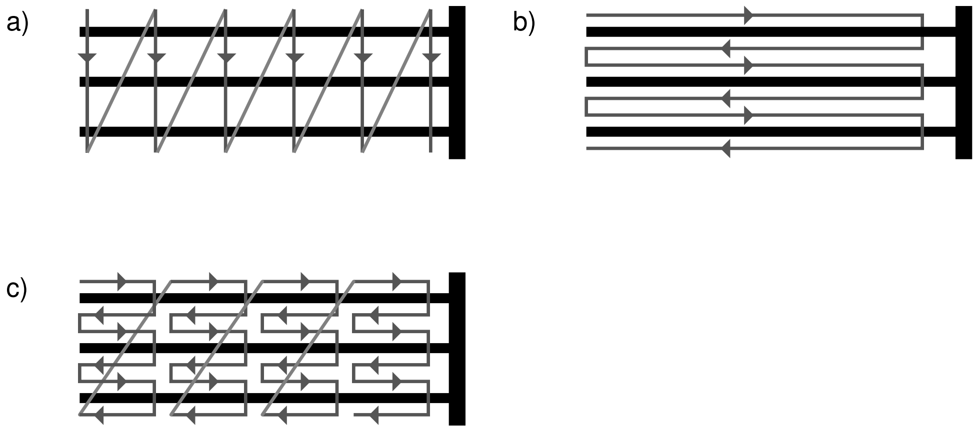



Abbildung 2.2: Drei gebräuchliche Abbildungen von LBA– auf CHS–Koordinaten:
(a) Traditionell ordnen Festplatten die Kapazität „vertikal“ an.
(b) Beim „horizontalen Mapping“ arbeitet die Platte erst die ganze Oberfläche ab, bevor sie auf den nächsten Kopf wechselt.
(c) Einige Plattenhersteller setzen ein zonenweises horizontales Mapping ein.

Bögeholz nennt in [Bo96a] drei verschiedene gängige Abbildungen von linearen Blockadressen: Beim sogenannten *vertikalen Mapping* wird zunächst eine Spur eines Zylinders vollgeschrieben, danach wechselt die Festplatte auf den nächstniedrigeren Kopf desselben Zylinders usw. Ist die Abbildung auf dem untersten Kopf des Zylinders angekommen, wird der oberste Kopf des nächstinneren Zylinders angesprochen. Dieses Mapping wird hauptsächlich von älteren Festplatten verwendet. Bei diesen Modellen werden die Schreib–/Leseköpfe von Schrittmotoren von einem Zylinder zum nächsten bewegt. Die Lage der Spuren auf der Festplatte ist dabei durch die Schrittweite und die Justage der Schrittmotoren bestimmt. 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 rein logische Umschaltungen von einem Kopf zum nächsten schneller als Bewegungen des Kopfes, die den Einsatz von träger Mechanik notwendig machen.

Neuere Platten haben keine Schrittmotoren mehr, sondern verwenden schnellere Linearmotoren. Diese Art von Motoren läßt außerdem eine höhere Spurdichte zu. 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. Einige Hersteller haben ihre Festplatten deswegen eine Zeit lang mit einem *horizontalen Mapping* ausgeliefert. Dabei wird zunächst eine Spur belegt und danach auf die nächstinnere Spur desselben Kopfes gewechselt. Ist die Festplatte bei der innersten Spur des ersten Kopfes angekommen, wechselt sie auf den nächstunteren Kopf und belegt diese Ebene der Festplatte von innen nach außen.

Horizontales Mapping verträgt sich jedoch schlecht mit dem im nächsten Abschnitt erläuterten *zone bit recording* (ZBR) und macht Geschwindigkeitsprognosen bzw. verbindliche Bandbreitenzusagen sehr schwer. Daher sind die einige Hersteller, die ein horizontales Mapping einsetzten, dazu übergegangen, statt dessen *zonenweises horizontales Mapping* zu verwenden[1]. Bei diesem Verfahren wird die Festplatte in Zonen unterteilt (die den ZBR–Zonen entsprechen). Innerhalb dieser Zonen wird ein horizontales Mapping verwendet, der Wechsel von einer Zone zur nächsten ist jedoch ein „vertikaler“ Übergang. Abbildung 2.2[2] illustriert die einzelnen Abbildungsverfahren.

[1] In [Bo96a] werden die Firmen Maxtor und Seagate als Anwender solcher Mischformen genannt, während Quantum und Micropolis zu diesem Zeitpunkt nach eigenen Angaben nur einfache vertikale oder horizontale Mappings verwendet haben. Bögeholz präsentiert in diesem Artikel ebenfalls Meßdaten einer teilweise beschädigten Maxtor 72004A, an denen das gemischte Mapping sichtbar gemacht werden kann.

[2] Grafik nach [Bo96a].

## 2.2 Datenrate und Aufzeichnungsdichte

Festplatten haben zwar meistens mehrere Schreib–/Leseköpfe, aber die Verstärkerelektronik hinter diesen Köpfen ist in der Regel nur in einfacher Ausführung vorhanden[3]. Dies bedeutet: Es kann nur ein Kopf zur Zeit aktiv sein. Die Geschwindigkeit, mit der Daten von einer Festplatte gelesen oder geschrieben werden können, hängt in diesem Fall nur von der Anzahl der Sektoren auf einer Spur und der Umdrehungsgeschwindigkeit des Plattenstapels ab. Festplatteninterface und Bussystem können zu begrenzenden Faktoren werden, wenn sie nicht mehr schnell genug sind, um mit der Geschwindigkeit der Festplatte als Datenquelle oder -ziel mitzuhalten.

Um die Datenrate von Festplatten zu steigern, haben sich die Hersteller in den vergangenen Jahren folgerichtig auf die Steigerung der Umdrehungsgeschwindigkeit und auf die Erhöhung der Anzahl der Sektoren pro Spur konzentriert. Dabei ist der Umdrehungsgeschwindigkeit von Festplatten, die am Arbeitsplatz betrieben werden sollen, aus ergonomischen Gründen eine Grenze gesetzt. Während ältere Festplatten noch mit 3600 Umdrehungen/Minute betrieben wurden, arbeiten die meisten moderneren Platten mit einer Rotationsgeschwindigkeit von 7200 U/min. Diese schnelleren Geräte schneiden bei Geräuschmessungen jedoch deutlich schlechter ab als langsamer drehende Modelle. Insbesondere bei Lautheitsmessungen[4], die im Gegensatz zu einfachen Schalldruckmessungen nicht nur die abgestrahlte Schallenergie messen, sondern die anatomische Hörcharakteristik des Ohres berücksichtigen, schneiden die schneller drehenden Platten sehr viel schlechter ab. Außerdem benötigen alle diese schnell drehenden Festplatten eine separate aktive Kühlung, um innerhalb des vom Hersteller vorgegebenen Temperaturbereiches zu bleiben.

Die Hersteller von Festplatten haben die Konsequenzen gezogen und sich den Kundenwünschen angepaßt: Festplatten für den Consumer– und Desktop–Markt werden in Regel mit geringeren Drehzahlen von 4500 oder 5400 U/min produziert. Neben der geringeren Geräuschbelastung haben diese Geräte wegen der geringeren Drehzahl auch eine geringere Datenrate und können so günstiger produziert werden und mit dem einfacheren, aber im Vergleich zu SCSI langsameren IDE/EIDE–Interface ausgestattet werden. Langsam drehende Platten benötigen außerdem keine aktive Kühlung, die wiederum weitere Lärmbelästigung erzeugen und die Kosten in die Höhe treiben würde. Festplatten größerer Kapazität, die für den Einsatz in Serverrechnern gedacht sind und damit nach Erwartung der Hersteller in geschlossenen Rechnerräumen oder verdämmten Gehäusen eingesetzt werden, arbeiten weiterhin mit hohen Drehzahlen (und sind mit SCSI–Interfaces ausgestattet). In diesem Bereich kommt es vorrangig auf Transferleistung und Kapazität an, während Kosten und Kühlung untergeordnete Bedeutung haben.

Auch die Anzahl der Sektoren pro Spur hat sich in vergangenen Jahren erhöht: Während bei typischen Festplatten für den PC–Markt Ende der 80er Jahre 17 oder 26 Sektoren zu 512 Byte pro Spur die Regel waren, hatten Festplatten für dasselbe Marktsegement um 1993 37 Sektoren pro Spur. Aktuelle Festplatten bedienen sich einer *zone bit recording* (ZBR) genannten Technik, um noch höhere Sektorzahlen zu erreichen: In [Bo95b] werden Zahlen von 69 bis 124 Sektoren pro Spur genannt.

[3] In [Bo95a] wird die ST12450W „Barracuda“ der Firma Seagate Benchmarksieger. Diese Festplatte hatte als einzige Platte im Testfeld zwei parallel arbeitende Köpfe.

[4] Siehe [Wi94].

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 2.3).

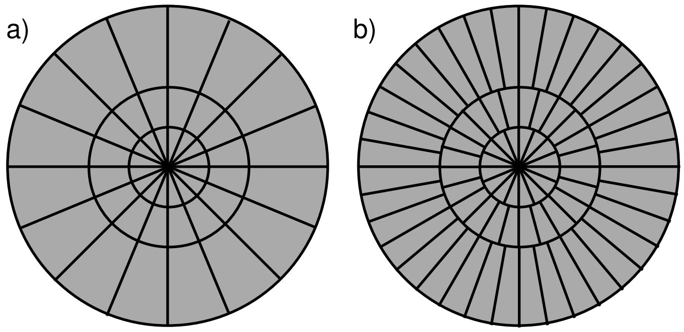



Abbildung 2.3: Aufteilung einer herkömmlichen Festplatte und einer Festplatte mit ZBR:
a) Traditionell arbeiten Festplatten mit einer festen Anzahl von Sektoren pro Spur.
b) Festplatten mit ZBR (*zone bit recording*) arbeiten auf den äußeren Spuren mit einer höheren Bitrate und mehr Sektoren pro Spur.

Diese Aufteilung in Zonen und die damit verbundene Änderung der Datentransfergeschwindigkeit kann man durch einen einfachen Benchmark detailliert sichtbar machen. Liest man nacheinander alle linearen Blockadressen der Festplatte durch und mißt man dabei jeweils die zur Datenübertragung benötigte Zeit, erhält man ein Diagramm, das die erzielte Datenrate in Relation zur LBA darstellt. Abbildung 2.4 zeigt ein solches Diagramm aus [Bo96b] für eine IBM–Festplatte. Deutlich ist die stufenweise Abnahme der Datenrate beim Voranschreiten des Benchmarks auf die inneren Zylinder der Disk zu erkennen. Bögeholz merkt an, daß die beiden schnellsten Zonen dieser speziellen Festplatte anteilig sehr groß sind: Sie nehmen 60% der gesamten Fläche ein. Außerdem hat diese Platte nur sehr wenige Zonen (acht Stück). Er vermutet, daß der Hersteller die zur vollen Ausnutzung der äußeren Plattenbereiche notwendigen Datenraten noch nicht im Griff hat und daß mit veränderten Schreib–/Leseköpfen oder schnellerer Plattenelektronik noch weitere Steigerungen in Datenrate und Kapazität möglich sind.

Die gezeigte Leistungskurve ist typisch für ein vertikales oder für gemischtes Mapping (siehe Abbildung 2.2). Sie macht die Planung von Partitionen einfach und die innerhalb der Partition erreichbaren Datenraten abschätzbar. Auf einer IBM DCHS–04W würde man beispielsweise Dateisysteme, die schnellen Datentransfer erfordern (etwa den Paging–Bereiche des Kernels oder häufig benutzte Dateisysteme), in die ersten 60% des Adreßraumes legen, während man weniger häufig genutzte Dateisysteme (etwa Boot– oder Dump–Dateisysteme) auf den langsamen inneren Zylindern installieren kann. Hätte die Festplatte dagegen ein horizonales Mapping, wäre die Kennlinie nicht monoton fallend, sondern würde für jede Scheibe des Plattenstapels

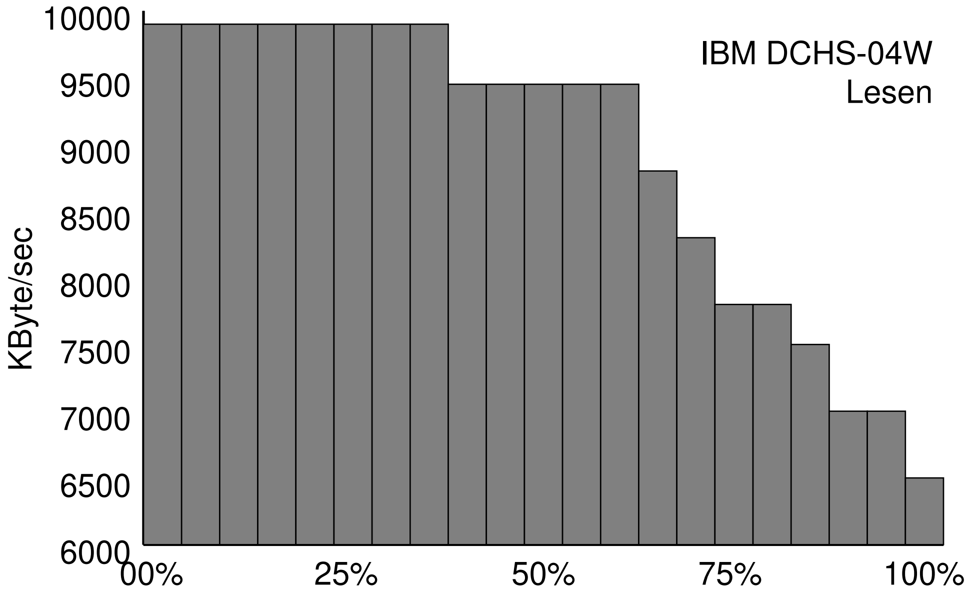



Abbildung 2.4: Messung der Lesegeschwindigkeit einer IBM–Festplatte (64 KB Transfereinheiten) mit ZBR. Die ZBR–Zonen der Festplatte sind an der von außen nach innen abnehmenden Lesegeschwindigkeit deutlich zu erkennen. Diese Festplatte verwendet nach Herstellerangaben ein zonenweise horizontales Mapping.

einmal vom Maximum zum Minimum fallen und wieder auf den Maximalwert ansteigen. Die schnellen Bereiche der Festplatte wären aus der Sicht der linearen Blockadressierung nicht zusammenhängend und daher nicht für ein zusammenhängendes Dateisystem nutzbar.

Das gemischte Mapping verwendet innerhalb einer ZBR–Zone horizontales Mapping. Innerhalb der Zone 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. In der Kennlinie einer fehlerfreien Festplatte ist diese Art des Mapping nicht von einem traditionellen vertikalen Mapping zu unterscheiden.

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 [Li96]. Dies trifft insbesondere für das verbreitete MS–DOS zu [MSDOS], das mit nicht quaderförmigen Geometrien nicht arbeiten kann. Aber auch der in Kapitel 3 beschriebene BSD–FFS Allokator [BSD96] hat Probleme mit andersförmigen Geometrien: Er geht von einer festen Anzahl der Sektoren pro Spur und einer konstanten Rotationsverzögerung von Sektoren bei Spurwechseln aus, um Zugriffszeiten zu optimieren. Diese Annahmen gelten bei ZBR–Geometrien in dieser einfachen Form nicht mehr, was zur Folge hat, daß die Optimierungen sinnlos sind[5].

Um trotzdem noch mit CHS–Ansteuerung nutzbar zu sein, „lügen" viele Festplatten, wenn sie nach ihrer Geometrie befragt werden. Sie geben bei Abfrage ihrer Geometriedaten eine quaderförmige Phantasiegeometrie an, bei der letztlich nur die Größenangabe (Produkt $C*H*S$) verläßlich 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, wahren CHS–Koordinaten umwandelt. Abbildung 2.5 zeigt die Auskunft, die eine Maxtor 7850 AR Festplatte erteilt, wenn sie nach ihrer Geometrie befragt wird. Die Festplatte (Bauformat 3.5 $\times$ 1 Zoll) hat bei dieser Bauhöhe schon aus physikalischen Gründen keine acht Scheiben und 16 Köpfe.

```
root@white:~# hdparm -g /dev/hda
 geometry     = 1654/16/63, sectors = 1667232, start = 0
```

Abbildung 2.5: Quaderförmige Geometrieauskunft einer Maxtor 7850 AR Festplatte (814 MB, EIDE–Interface, ZBR mit zonenweise horizontalem Mapping)

[5] Sie sind meistens nicht schädlich, weil praktisch alle modernen Festplatten interne Zwischenspeicher haben, die ganze Spuren aufnehmen können und auch spurweise voraus lesen. Dadurch spielt die Anordnung der Sektoren innerhalb einer Spur letztlich keine Rolle.

## 2.3 RAID

Aktuelle Festplattendesigns erreichen Kapazitäten von etwas mehr als 8 GB für ein einzelnes Gerät. Die erzielten Datentransferraten liegen bei Spitzenwerten von knapp 10 MB/sec. Diese Beschränkungen für Einzelgeräte lassen sich durch Kombination von Festplatten zu Arrays umgehen. Diese *RAID* (*Redundant Array of Inexpensive Disks*) genannte Technik wurde in [Pa88] vorgestellt. Patterson, Gibson und Katz präsentieren dort verschiedene Verfahren, mit denen mehrere einzelne Festplatten gleicher Bauart zu einer virtuellen, größeren Platte zusammengefaßt werden können. Die einzelnen Konfigurationen, RAID–Level genannt, unterscheiden sich dabei in der Parallelisierbarkeit von Lese– und Schreibzugriffen sowie in der Toleranz gegenüber Ausfällen einzelner Platten im Array. Von den bei Patterson vorgestellten Verfahren haben die von ihm RAID 1, 3 und 5 genannten Verfahren größere Beachtung in der Industrie gefunden und sind inzwischen Standardlösungen[6], wenn große Festplattenkapazitäten realisiert werden müssen. Pattersons RAID 2 und 4 sind nur von akademischem Interesse gewesen, da sie in der Realisierung zu aufwendig sind, ohne zusätzliche Datensicherheit anzubieten.

Die Grundidee hinter RAID besteht darin, einen Satz von Standardfestplatten gleichen Typs zusammenzufassen und gegenüber dem Hostrechner als eine große Festplatte erscheinen zu lassen. So sind heute erhältliche RAID–Systeme auch meistens in Form von SCSI–auf–SCSI–Adaptern implementiert (Abbildung 2.6 zeigt eine konkret eingesetzte Konfiguration): Der Hostrechner steuert das RAID–Array wie eine gewöhnliche SCSI–Festplatte. Der Controller des Arrays verarbeitet die Anforderungen des Hostrechners und generiert sekundäre Anforderungen, die über einen sekundären SCSI–Bus an die Platten des RAID–Arrays verteilt werden. Der Arraycontroller ist dabei in der Regel ein spezialisierter kleiner Computer mit einer großen Menge Cache–RAM und einer variablen Anzahl von Adapterkarten zur Kommunikation mit Festplatten und Hostrechnern.

RAID–Arrays können höhere Datentransferraten erreichen als einzelne Festplatten, wenn es möglich ist, aufeinanderfolgende Zugriffe auf unterschiedliche Laufwerke zu verteilen. Im einfachsten Fall teilt man die Laufwerke dazu in Streifen (*stripes*) auf, die dann reihum auf den einzelnen Laufwerken positioniert werden[7]. Abbildung 2.7 illustriert dies: Lineare Zugriffe in der Reihenfolge $A$, $B$, $C$, ... werden nacheinander auf alle vorhandenen Festplatten verteilt. Dieses RAID 0 genannte Verfahren ist strenggenommen kein RAID–Verfahren, denn es wird keine Redundanz erzeugt: Fällt auch nur eine der am Array teilnehmenden Disks aus, ist das darauf enthaltene Dateisystem zerstört. Da sich die einzelnen Ausfallwahrscheinlichkeiten der Disks im Array addieren, wird das Risiko bei größeren Installationen schnell untragbar.

Eine einfache, aber sehr teure und aufwendige Lösung zur Erhöhung der Zuverlässigkeit ist die Spiegelung der einzelnen Platten. Dieses bei Patterson unter dem Namen RAID 1 aufgeführte Verfahren ist jedoch für große Arrays wegen des hohen Overheads nicht praktikabel: Die Kosten für die Medien verdoppeln sich gegenüber RAID 0, hinzu kommen Kosten für

[6]In der Industrie werden außerdem andere Verfahren zur redundanten Datenhaltung unter anderen, höheren „RAID"–Nummern verwendet. Diese Verfahren werden in [Pa88] nicht diskutiert und haben teilweise mit den dort diskutierten Dingen nur den Namen gemeinsam.

[7]Ähnliche Verfahren werden auch in der RAM–Ansteuerung von PCs und Workstations eingesetzt, wo man verschiedene RAM–Bänke ineinander verschränkt, um bei linearen Zugriffen Wartezyklen zu vermeiden.

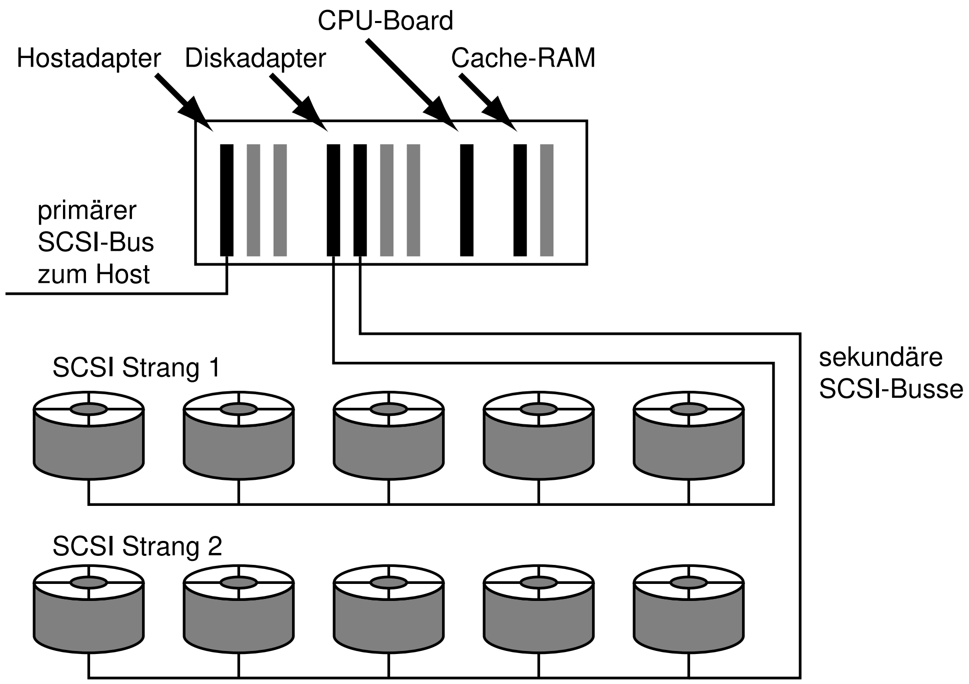



Abbildung 2.6: RAID–Systeme werden zur Zeit hauptsächlich in der Form von SCSI–auf–SCSI–Adaptern implementiert. Die hier gezeigte Beispielkonfiguration ist mit 10 Quantum „Fireball"–Festplatten zu je 4 GB Kapazität seit Ende letzten Jahres im Projektzentrum Ökosystemforschung an der Universität Kiel im Einsatz. Gegenüber dem Hostrechner erscheint das Array als zwei große Festplatten mit jeweils 20 GB (RAID 0) bzw. 16 GB (RAID 3 oder 5) Kapazität.

zusätzliche Infrastruktur wie Controller, Netzteile, Stellfläche usw. RAID–1–Systeme können geringfügig schneller arbeiten als RAID–0–Systeme, weil Leseanforderungen auf beide Kopien der Daten verteilt werden können. Ebenso ist es bei richtiger Implementierung und Inkaufnahme eines geringen zusätzlichen Risikos möglich, Schreibzugriffe zu verteilen und die zweite Kopie asynchron zu schreiben. Im Fehlerfall kann RAID 1 den Ausfall einer beliebigen Disk pro Paar verkraften. Die Daten werden dann von der Kopie dieser Disk gelesen, und es ist kein zusätzlicher Aufwand zur Rekonstruktion der Daten notwendig.

RAID 3 realisiert ebenfalls ein gewisses Maß an Ausfallsicherheit, aber zu geringeren Kosten als RAID 1. In RAID 3 wird einem analog zu RAID 0 organisierten Festplattenarray eine zusätzliche Paritätsdisk zur Seite gestellt. Auf dieser Festplatte wird in jedem Streifen die XOR–Summe der entsprechenden Streifen aller anderen Disks gespeichert (d.h. in Abbildung 2.9 wird in Streifen 1 die XOR–Summe von $A$, $B$ und $C$ gespeichert, in Streifen 2 die XOR–Summe von $D$, $E$ und $F$ usw.). In normalem Betrieb und bei Ausfall der Paritätsdisk können Lesezugriffe wie bei RAID 0 abgehandelt werden. Fällt genau eine beliebige andere Disk aus, kann die Information der fehlenden Streifen aus der Paritätsinformation und den noch vorhandenen Informationen vollständig rekonstruiert werden. Diese Lesezugriffe sind jedoch langsamer, da die Daten der fehlenden Disk aus den Daten aller entsprechenden Streifen der anderen Disks rekonstruiert werden müssen, d.h. Lesezugriffe auf die fehlende Disk müssen durch Lesezugriffe auf alle anderen Disks simuliert werden.

Schreibzugriffe sind bei RAID 3 relativ langsam, da zum einen beim Schreiben eines Streifens alle entsprechenden Streifen aller anderen Disks gelesen werden müssen, um die geänderte Paritätsinformation zu berechnen, und da zum anderen die Schreibbandbreite durch die Geschwindigkeit der Paritätsdisk begrenzt ist.

Diese Begrenzung der Schreibbandbreite kann dadurch abgemildert werden, daß man die Streifen mit Paritätsinformation nicht auf einer dedizierten Paritätsdisk speichert, sondern die Paritätsinformation reihum auf alle am Array beteiligten Disks verteilt. Dadurch verteilen sich ebenfalls die Schreibzugriffe für diese Paritätsinformation. Noch immer müssen jedoch alle Disks eines Arrays gelesen werden und zwei Disks des Arrays beschrieben werden, um einen Streifen zu aktualisieren (Beispiel: Um Streifen $A$ zu ändern, müssen $A$, $B$, $C$ und 1 gelesen, die neue Paritätsinformation $\bar{1}$ aus dem neuen $\bar{A}$ und den alten $B$, und $C$ neu berechnet und $\bar{A}$ und $\bar{1}$ geschrieben werden). Dieses Verfahren entspricht RAID 5.

Um die Verzögerung durch eine große Anzahl von Lese– und Schreiboperationen abzumildern, sind praktisch alle RAID–Controller mit vergleichsweise großen RAM–Caches ausgestattet, in denen sie die gelesenen Streifen gepuffert speichern. Einige Hersteller experimentieren außerdem mit batteriegesichertem Cache–RAM und verzögerten Schreibzugriffen, um den Durchsatz zu erhöhen.

## 2.4 Partitionen

Für einige Zwecke kann es nützlich oder notwendig sein, statt einer großen Festplatte mehrere kleinere Platten zu haben. 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. Außerdem kann man Daten mit verschieden langer Lebensdauer durch Verwendung mehrerer Partitionen voneinander trennen. Dies hilft, Fragmentierung (siehe Abschnitt 2.5) zu vermeiden.

Die meisten Betriebssysteme stellen dazu den Mechanismus der *Partitionierung* von Festplatten zur Verfügung. Eine Partition ist dabei ein zusammenhängender Abschnitt von Blöcken einer Festplatte. So könnte man sich die Platte aus Abbildung 2.1 in zwei Partitionen unterteilt vorstellen. Die erste Partition würde auf den äußeren Zylindern beginnen und sich bis zum markierten Zylinder erstrecken, die zweite würde dort beginnen und die gesamten inneren Zylinder der Platte umfassen.

Die benutzten Partitionen einer Festplatte dürfen sich niemals überlappen. Sie stoßen im Normalfall nahtlos aneinander, aber es gibt keinen Mechanismus, der dies erzwingt. Normalerweise wird man eine Festplatte vollständig partitionieren, d.h. keinen Platz ungenutzt lassen, aber dies wird ebenfalls nicht erzwungen.

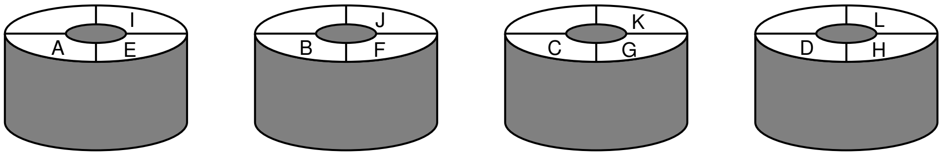



Abbildung 2.7: Bei „RAID 0“ werden die Streifen reihum auf die beteiligten Laufwerke verteilt. „RAID 0“ ist strenggenommen kein RAID–Verfahren, da keine Redundanz vorhanden ist.

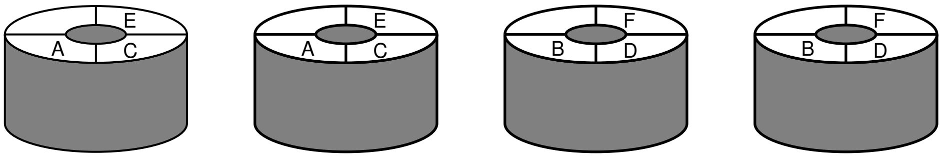



Abbildung 2.8: RAID 1 bezeichnet die Spiegelung von Informationen.

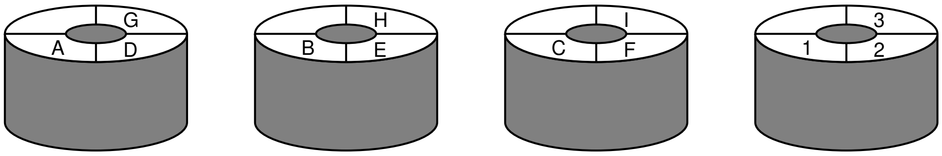



Abbildung 2.9: RAID 3 ist weniger aufwendig, aber auch etwas unsicherer als RAID 1: Für jeweils eine Gruppe von Festplatten wird eine zusätzliche Festplatte mit Paritätsinformationen installiert.

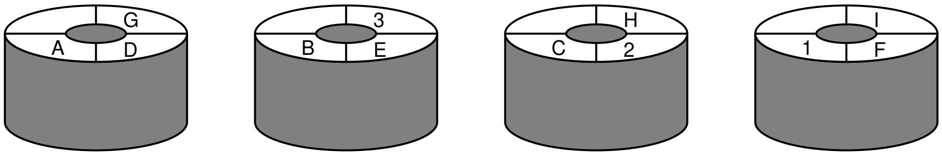



Abbildung 2.10: Bei RAID 3 können Schreibzugriffe zum begrenzenden Faktor werden, da in jedem Fall die Paritätsdisk aktualisiert werden muß. RAID 5 umgeht dieses Hindernis, indem die Paritätsinformation „untergemischt“ wird.

Eine Partition einer Festplatte wird vom Betriebssystem wie eine einzelne kleine Festplatte behandelt. Auf einer Partition kann in der Regel genau ein Dateisystem eingerichtet werden[8]. Üblicherweise gibt man beim Einschalten des Rechners an, von welcher Partition man den Rechner starten möchte.

Der Umgang mit Partitionen erfolgt in den meisten UNIX–Versionen bereits im Gerätetreiber, der für jede Partition ein (major, minor)–Paar und damit eine Gerätedatei zur Verfügung stellt. 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.

## 2.5 Fragmentierung

Weil Festplattenplatz nur in Einheiten zu jeweils einem Sektor oder ganzzahligen Vielfachen davon zur Verfügung steht, verwalten praktisch alle Dateisysteme Platz in diesen Einheiten. Dabei geht Platz verloren: Eine Datei, die nur ein einzelnes Byte umfaßt, belegt trotzdem die Mindestgröße von einem Plattensektor und verschwendet damit bei einer Sektorgröße von 512 Byte 511 Bytes. Unter der Annahme, daß Dateigrößen modulo Sektorgröße gleichmäßig verteilt sind, wird im Mittel pro Datei jeweils ein halber Sektor verschwendet. Faßt man mehrere physikalische Plattensektoren zu logischen Blöcken zusammen, steigt der Platzverlust durch ungenutzte Endblöcke stark an. [BSD89, 7.1] gibt ein Beispiel für den erhöhten Platzverbrauch durch große Blöcke: Der Platzverlust des gewählten Beispieldateisystems mit 775,2 MB Nettodaten steigt von 6,9% (512 Byte große Blöcke) auf 45,6% (4096 Byte große Blöcke) an.

Auf der anderen Seite ist es wünschenswert, für einen schnellen Zugriff auf die Daten möglichst große Blöcke zu verwenden. [BSD84] zeigt, daß eine simple Verdoppelung der Blockgröße von 512 Byte auf 1024 Byte in einem normalen UNIX-Dateisystem den Durchsatz schon mehr als verdoppelt. Platzausnutzung und Transfergeschwindigkeit stehen also grundsätzlich miteinander in Konflikt: Während eine kleine Blockgröße für eine optimale Ausnutzung des Plattenplatzes gut ist, sind nur mit großen Blöcken hohe Datentransferraten möglich.

Für eine zügige Übertragung von Daten ist es außerdem von Vorteil, wenn die Daten einer Datei auf der Platte zusammenhängend abgelegt sind. Dateioperationen, die eine Datei von vorne nach hinten durchlesen, sind relativ häufig [Ou85]. In diesem Fall ist es von Vorteil, wenn die Datei zusammenhängend auf der Platte abgelegt ist und ohne zusätzliche Kopfbewegungen eingelesen werden kann.

Wenn Dateien jedoch nahtlos hintereinander angeordnet werden, ist es sehr wahrscheinlich, daß eine Datei nicht mehr in einem zusammenhängenden Stück auf der Platte abgelegt werden kann, wenn sie wächst. Auch durch das Löschen und Neuanlegen von größeren Dateien kann es zur *Fragmentierung* von Dateien kommen (siehe Abbildung 2.11).

[8] Es existieren Versionen von SCO XENIX/386, die mit dem Programm `divvy` eine Partition in mehrere Dateisysteme unterteilen können, aber das ist die Ausnahme.

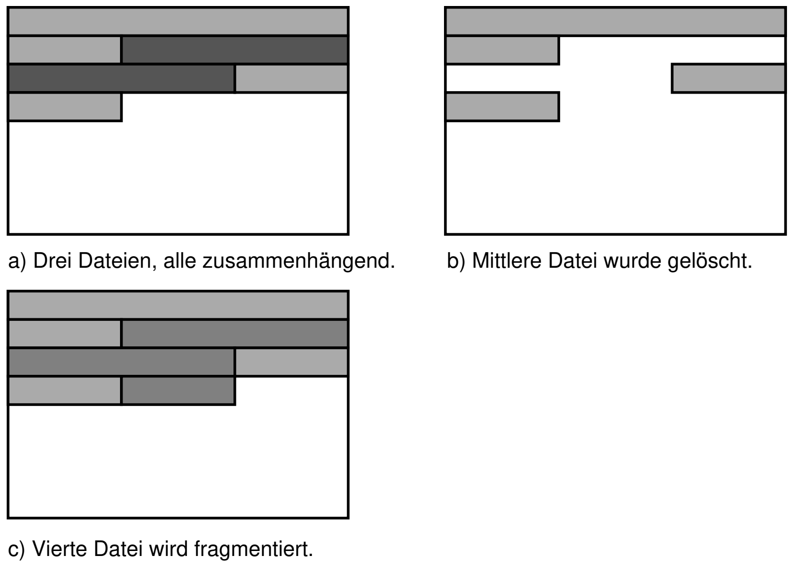



Abbildung 2.11: Fragmentierung von Dateien

Dateisysteme müssen beide widersprüchlichen Optimierungsziele bei der Positionierung von Datenblöcken berücksichtigen: Einerseits soll der vorhandene Platz möglichst effektiv genutzt werden, andererseits sollen die Daten möglichst so plaziert werden, daß sie in großen, zusammenhängenden Stücken angeordnet sind, damit sie schnell gelesen werden können.

# Kapitel 3

# Probleme und Beschränkungen konventioneller UNIX–Dateisysteme

Dieses Kapitel gibt einen Überblick über das klassische *System V–Dateisystem* (S5FS), das *BSD Fast Filing System* (FFS) und das Linux *Second Extended Filesystem* (ext2). Dabei sollen besonders die Probleme und Beschränkungen dieser Dateisysteme angesprochen werden und bereits dafür existierende Lösungen Erwähnung finden.

## 3.1 Struktur des System V–Dateisystems

### 3.1.1 Inodes und Datenblöcke

Das Dateisystem von AT&T System V UNIX (S5FS) entspricht dem ursprünglichen UNIX–Dateisystem, wie es in [Ba86] vorgestellt wird. Neben der ursprünglichen Variante mit einer Blockgröße von 512 Bytes sind auch noch Abwandlungen mit einer Blockgröße von 1 KB (S51K) und einer Blockgröße von 2 KB (S52K) gebräuchlich. Den folgenden Größenberechnungen liegt jedoch immer die Originalblockgröße von 512 Bytes zugrunde. Die wesentlichen Konzepte von S5FS haben sich ausgezeichnet bewährt und sind auch in FFS und ext2 zu finden.

UNIX definiert eine Datei als eine geordnete, endliche Folge von Bytes mit einer Reihe von weiteren Eigenschaften (Metainformationen), die zur Verwaltung dieser Bytes notwendig sind. Diese Eigenschaften sind der *Dateityp*, der *Dateieigentümer*, die *Dateigruppe*, die *Basiszugriffsrechte* für den Dateieigentümer, Angehörige der Dateigruppe und alle anderen Benutzer, die *Anzahl der Namen* der Datei, die *Größe der Datei* sowie die *Zeitstempel* des letzten Lesezugriffs auf die Daten der Datei, des letzten Schreibzugriffs auf die Daten der Datei und des letzten Schreibzugriffs auf die Metainformationen der Datei.

Festplatten und andere persistente Speichermedien bieten den Speicherplatz in adressierbaren Einheiten an, die größer als ein Byte sind. Typischerweise sind die kleinsten adressierbaren Einheiten Blöcke von 512 Bytes oder einem ganzzahligen Vielfachen davon. Die physikalische Repräsentation einer Datei besteht dann aus einer Folge von nicht notwendigerweise zusammenhängenden Datenblöcken des Speichermediums sowie einer Verwaltungsstruktur, der *Inode*[1]. Die Datenblöcke enthalten nur die in der Datei gespeicherten Daten.

Alle oben erwähnten Metainformationen und die Adressen der zur Datei gehörenden Datenblöcke sind direkt oder indirekt in der Inode abgelegt. Abbildung 3.1 zeigt die C–Datenstruktur, die von System V UNIX zur Speicherung dieser Informationen auf Festplatte verwendet wird. Sie belegt in dieser Version 64 Bytes. Bei einer Blockgröße von 512 Bytes können also acht Inodes in einem Diskblock gespeichert werden.

```
/*
 * Inode structure as it appears on
 * a disk block.
 */
struct dinode
{
  unsigned short  di_mode; /* mode and type of file */
  short   di_nlink;        /* number of links to file */
  short   di_uid;          /* owner's user id */
  short   di_gid;          /* owner's group id */
  off_t   di_size;         /* number of bytes in file */
  char    di_addr[40];     /* disk block addresses */
  time_t  di_atime;        /* time last accessed */
  time_t  di_mtime;        /* time last modified */
  time_t  di_ctime;        /* time created */
};
#define INOPB   8                /* 8 inodes per block */
/*
 * the 40 address bytes:
 *      39 used; 13 addresses
 *      of 3 bytes each.
 */
```

Abbildung 3.1: Inode–Datenstruktur im System V–Dateisystem

Die in Abbildung 3.1 gezeigte Inode stellt 40 Bytes zur Speicherung von Adressen von Datenblöcken zur Verfügung. Eine Adresse eines Blocks belegt dabei 3 Bytes, so daß ein Dateisystem bei 512 Bytes Blockgröße insgesamt 8 GB groß sein darf. In der Inode werden die Adressen der ersten zehn Datenblöcke direkt gespeichert (*direct blocks*). Die elfte Adresse ist die Adresse eines *indirect block*, der die Adressen von 128 weiteren Datenblöcken[2] enthält. Entsprechend ist die zwölfte Adresse die Adresse eines *double indirect block*, der die Adressen von 128 Indirect Blocks enthält, und die dreizehnte Adresse ist die Adresse eines *triple indirect block*, der die Adressen von 128 Double Indirect Blocks enthält. Abbildung 3.2 stellt diesen Sachverhalt graphisch dar.

[1]Der Begriff *Inode* ist eine Zusammenziehung der Worte *index node* ([Ba86], Seite 22).

[2]Inkonsequenterweise verwendet S5FS hier 4 Bytes zur Speicherung einer Adresse. Das höchstwertigste Byte dieser Adressen ist immer 0.

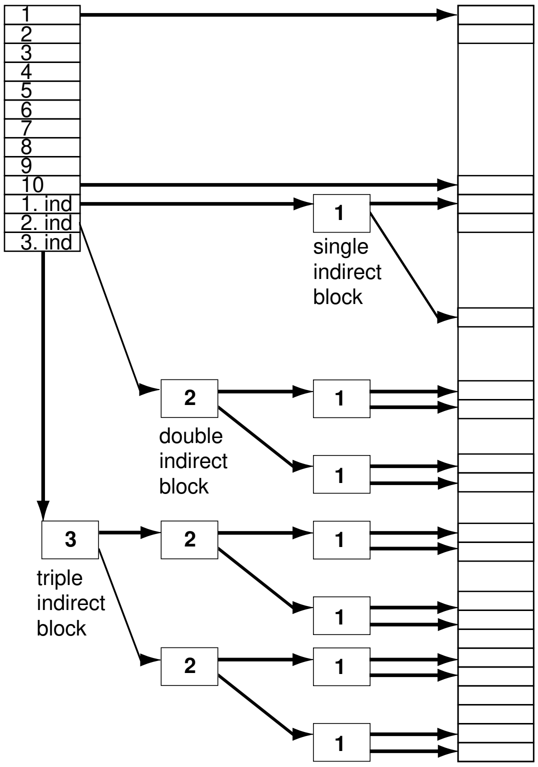



Abbildung 3.2: Inode und Datenblöcke

In S5FS werden beim Anlegen des Dateisystems die ersten beiden Blöcke der Festplatte für einen Bootloader zum Laden und Starten des Betriebssystems und für den *Superblock* des Dateisystems reserviert. Der Superblock enthält die Metainformationen über das gesamte Dateisystem: Dies sind im wesentlichen der Name des Dateisystems, seine Größe in Blöcken und die Größe der Inodetabelle (*inode table*). Hinzu kommen weitere Informationen, die jedoch keine Kenngrößen des Dateisystems sind, sondern bei denen es sich um dynamische Daten handelt, die bei Verlust regeneriert werden können. Abbildung 3.6 gibt einen graphischen Überblick über die Organisation des S5FS im Vergleich mit dem moderneren Berkeley Fast Filing System.

Die folgenden Blöcke des Dateisystems bilden die bereits erwähnte Inodetabelle. Ihre Größe wird bei der Erzeugung des Dateisystems mit dem Systemkommando *mkfs* festgelegt, und sie belegt bei Verwendung der Defaultparameter von `mkfs` etwa zwei Prozent der Gesamtgröße des Dateisystems. Es ist jedoch möglich, bei der Einrichtung eines Dateisystems eine größere Menge an Inodes anzufordern, und für einige Systemkonfigurationen können die Inodes bis zu 10% des Dateisystems belegen.

### 3.1.2 Verzeichnisse und Pfadnamen

Mit der Inodetabelle hat das Betriebssystem bereits alle benötigten Informationen zur Verwaltung von Dateien. Diese Dateien haben aber noch keine Namen und sind auch noch nicht in der unter UNIX gewohnten Weise baumförmig strukturiert angeordnet. Diese Strukturbildung erfolgt über Verzeichnisse.

Ein Verzeichnis ist in UNIX eine Datei von einem bestimmten Typ (Typ `d` für *directory*) und besitzt einen festgelegten Aufbau. Dadurch, daß ein Verzeichnis eine Datei ist, hat jedes Verzeichnis eine Inode und Datenblöcke wie jede andere Datei auch. Ein Verzeichnis ist jedoch eine besondere Datei, an deren Aufbau das Betriebssystem einige Anforderungen stellt: In S5FS besteht ein Verzeichnis aus einer Folge von Datensätzen zu jeweils 16 Bytes. Die ersten zwei Bytes enthalten die Inodenummer einer Datei, die folgenden 14 Bytes 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 **.** (Punkt), der die Inodenummer des Verzeichnisses selbst enthält. Der zweite Eintrag ist immer der Name **..** (Punkt Punkt), der die Inodenummer des übergeordneten Verzeichnisses speichert.

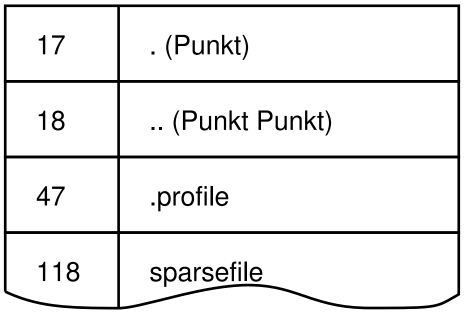



Abbildung 3.3: Verzeichnis in S5FS

Dadurch, daß alle Informationen über die Datei in der Inode gespeichert sind, ist es möglich, mehr als einen Verzeichniseintrag für eine Datei zu haben, ohne Inkonsistenzen durch doppelte Datenhaltung befürchten zu müssen. Weitere Verzeichniseinträge für eine existierende Datei erzeugt man mit dem Systemaufruf `link()`, der den alten und einen neuen Namen für die Datei mitgeteilt bekommen muß. Der Aufruf schlägt die Inodenummer der Datei über den alten Pfadnamen nach und erzeugt einen Eintrag für den neuen Namen einer Datei mit derselben Inodenummer. Beide Verzeichniseinträge verweisen auf dieselbe Inode, daher haben beide Dateien bis auf den Namen dieselben Attribute. Beide Namenseinträge sind gleichberechtigt, d.h. es ist nachträglich nicht feststellbar, welcher Name der Datei der originale Name war. Mit dem Systemaufruf `unlink()` können Namen einer Datei entfernt werden. Sobald eine Datei keine Namen mehr hat, also ihr Link–Count–Feld in der Inode auf den Wert 0 sinkt, wird der Speicherplatz für die Datei wieder freigegeben.

Alle betriebssysteminternen Funktionen von UNIX arbeiten mit Inodes bzw. Inodenummern[3]. Es existiert aber keine Kernelfunktion, die eine Inodenummer von einem Benutzerprozeß entgegennehmen würde. Statt dessen müssen Prozesse dem Kern gegenüber Dateien immer über Pfadnamen identifizieren. In UNIX übernimmt eine betriebssysteminterne, zentrale Funktion, `namei()`, die Umwandlung von Pfadnamen in Inodenummern und prüft dabei gleichzeitig Zugriffsrechte entlang des Pfades.

[3] Genaugenommen arbeiten sie mit Paaren (`dev_t`, `ino_t`), d.h. mit Paaren von Gerätenummern und Inodenummern. Inodenummern sind nur innerhalb eines Dateisystems eindeutig und werden erst in Verbindung mit einer eindeutigen Gerätebezeichnung zu systemweit eindeutigen Dateiidentifikatoren.

Wenn `namei()` mit einem Pfadnamen, der mehrere Komponenten enthält, aufgerufen wird, beginnt UNIX mit der Umsetzung des Namens in eine Inodenummer bei der ersten Namenskomponente im Startverzeichnis der Suche, also entweder im Wurzelverzeichnis oder im aktuellen Verzeichnis. Die Datensätze dieses Verzeichnisses werden linear nach einem passenden Namen durchsucht. Ist er gefunden, kennt `namei()` die Inodenummer des nächsttieferen Verzeichnisses und kann sich selbst rekursiv mit dieser Inodenummer und dem Restpfad aufrufen[4].

Auf diese Weise können beliebig lange Pfade aufgelöst werden. Abbildung 3.4 zeigt, wie die Inodes dabei auf die Datenblöcke eines Verzeichnisses zeigen, während die Datenblöcke des Verzeichnisses wiederum auf Inodes zurückverweisen. Die sich so ergebende induktiv definierte Struktur „springt“ zwischen dem Inodebereich und dem Datenbereich einer Platte hin und her. Der Induktionsanfang ist die Inodenummer des Wurzelverzeichnisses (wenn der Pfadname mit einem „/“ beginnt) oder des aktuellen Verzeichnisses (in allen anderen Fällen), die in der Prozeßstruktur enthalten sind und deswegen als bekannt vorausgesetzt werden können.

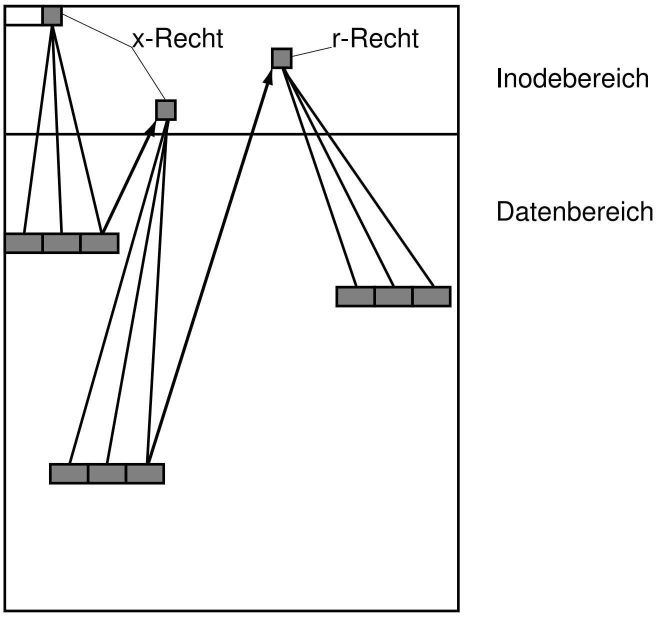



Abbildung 3.4: Auflösung eines absoluten Pfadnamens

In S5FS liegen die Inodes konzentriert am Anfang einer Festplatte und belegen in der Regel etwa zwei Prozent des gesamten Plattenplatzes. Die Datenblöcke von Verzeichnissen können an beliebiger Stelle auf dem Rest der Platte liegen. Deswegen sind beim Auflösen von Pfadnamen unter Umständen umfangreiche Kopfbewegungen notwendig. Ein Aufruf von `namei()` zur Auflösung eines Pfadnamens ist also eine vergleichsweise aufwendige Operation.

Für eine effiziente Abwicklung von `namei()` ist es daher entscheidend, daß ein Cache für häufig benutzte Verzeichnisblöcke existiert. Datenblöcke und Inodes, die kürzlich erst im Zugriff waren, bleiben im RAM des Rechners zwischengespeichert und müssen bei erneuter Anforderung nicht

[4]Da der Kernelstack begrenzt ist, löst man diese endständige Rekursion in eine `while`–Iteration auf.

noch einmal von der Festplatte gelesen werden. Dadurch wird insbesondere das Öffnen von Dateien sehr stark beschleunigt[5].

## 3.2 BSD Fast Filing System

In [BSD84] wurde erstmals das *Fast Filing System (FFS)* vorgestellt, das eine ganze Reihe von Schwächen des klassischen UNIX–Dateisystems behebt und einige neue Features zur Verfügung stellt, die im originalen System V–Dateisystem nicht vorhanden waren.

Der Hauptgrund für die Entwicklung von FFS war die Beobachtung, daß die Datentransferleistung eines typischen UNIX–Systems stark hinter dem berechneten Maximaldurchsatz seiner Festplatten zurückbleibt. Das ursprüngliche Dateisystem arbeitete mit einer Blockgröße von 512 Byte. Schon durch eine Vergrößerung der Blockgröße auf ein Kilobyte konnte die Transfergeschwindigkeit mehr als verdoppelt werden: Zum einen konnten jetzt größere Blöcke von der Festplatte in den Speicher bewegt werden, wodurch der Verwaltungsaufwand des Betriebssystems geringer wurde. Zum anderen waren durch die größeren Blöcke bei gleichbleibender mittlerer Dateigröße im Schnitt weniger Zugriffe auf indirekte Blöcke notwendig.

Ein anderer Mangel, der mit den wachsenden Festplattengrößen zutage trat, ist die Begrenzung von Inodenummern auf nur 16 Bit breite Worte. Dies schränkt die Anzahl der Dateien in einem Dateisystem auf 65536 Stück ein. Auf großen Festplatten mit einer geringen mittleren Dateigröße wird diese Grenze leicht überschritten. Eine Erweiterung des Datentyps „Inodenummer" auf 32 Bit erforderte eine Überarbeitung des Verzeichniskonzepts und der anderen Datenstrukturen eines Dateisystem. Bei dieser Gelegenheit erweiterte man zugleich auch viele Datenfelder in der Inode selbst (siehe Abbildung 3.5). Weiterhin erlaubte eine neue Verzeichnisstruktur lange Dateinamen von bis zu 255 Zeichen Länge. Durch die Änderungen am Dateisystem sind Inodes beim FFS 128 Bytes groß.

Ein weiteres Problem, das in [BSD84] und in [BSD89] 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. Das alte Dateisystem speichert freie Blöcke in einer Liste, der sogenannten *free list* ab. Diese Liste ist in Form eines LIFO–Speichers organisiert[6]: Freiwerdende Blöcke werden vorne in die Liste eingefügt, benötigte Blöcke werden ebenfalls von vorne her der Liste entnommen. Wenn in einem System häufig Dateien erzeugt und gelöscht werden, ist die Liste freier Blöcke schon bald weitgehend durchmischt, und damit werden die Datenblöcke neu angelegter Dateien über die ganze Platte verstreut. Um Dateien zusammenhängend anzulegen,

[5]Dieselbe Beobachtung macht man auch bei anderen Betriebssystemen: Unter MS–DOS hat man sich vor der Einführung des Buffer Cache `SMARTDRV` mit dem Namenscache `FASTOPEN` beholfen, der eine Tabelle von kompletten Pfadnamen und Plattenblöcken gespeichert hat. Wenn auf einen Pfadnamen wiederholt zurückgegriffen wurde, war es dadurch nicht mehr notwendig, einen kompletten Durchlauf durch die Verzeichnisse vorzunehmen. [Ta87]

[6]Der wirkliche Sachverhalt ist etwas komplizierter, aber für die Zwecke dieser Diskussion genügt diese Vereinfachung.

```
/*
 * A dinode contains all the meta-data associated with a UFS file.
 * This structure defines the on-disk format of a dinode.
 */

#define NDADDR  12            /* Direct addresses in inode. */
#define NIADDR  3             /* Indirect addresses in inode. */

struct dinode {
    u_short     di_mode;      /*   0: IFMT and permissions. */
    short       di_nlink;     /*   2: File link count. */
    union {
        u_short oldids[2];    /*   4: Ffs: old user and group ids. */
        ino_t   inumber;      /*   4: Lfs: inode number. */
    } di_u;
    u_quad_t    di_size;      /*   8: File byte count. */
    struct timespec di_atime; /*  16: Last access time. */
    struct timespec di_mtime; /*  24: Last modified time. */
    struct timespec di_ctime; /*  32: Last inode change time. */
    daddr_t     di_db[NDADDR];/*  40: Direct disk blocks. */
    daddr_t     di_ib[NIADDR];/*  88: Indirect disk blocks. */
    u_long      di_flags;     /* 100: Status flags (chflags). */
    long        di_blocks;    /* 104: Blocks actually held. */
    long        di_gen;       /* 108: Generation number. */
    u_long      di_uid;       /* 112: File owner. */
    u_long      di_gid;       /* 116: File group. */
    long        di_spare[2];  /* 120: Reserved; currently unused */
};
```

Abbildung 3.5: Inode–Datenstruktur Berkeley FFS müßte diese Liste immer sortiert gehalten werden. Dies ist mit dieser Darstellungsform der Liste nicht effizient implementierbar [BSD84].

Außerdem werden in S5FS die Inodes eines Dateisystems in einer Tabelle gesammelt am Anfang des Dateisystems abgelegt. Dies erfordert beim Zugriff auf die Datei weite Kopfbewegungen über die Oberfläche des Mediums. Es wäre effizienter, wenn die Inodes in der Nähe der Daten, die sie referenzieren, angelegt werden könnten.

Die Praxis hat schließlich gezeigt, daß beim Einsatz von UNIX auf Rechnern, auf denen mehrere Gruppen arbeiten, eine stärkere Einflußnahme der Systemverwaltung auf den Ressourcenverbrauch der Benutzer notwendig ist. Das neue Dateisystem sollte es erlauben, den Verbrauch an Datenblöcken und Inodes einzelner Benutzer zu begrenzen.

Zusammengefaßt ergeben sich die die folgenden Verbesserungen von FFS gegenüber S5FS, auf die in den folgenden Abschnitten teilweise detaillierter eingegangen wird:

- Maximal $2^{32}$ statt $2^{16}$ Inodes pro Dateisystem.
- Maximal 255 statt 14 Zeichen pro Dateiname.
- Geschwindigkeitssteigerung durch Verwendung einer größeren Blockgröße; trotzdem geringe Platzverschwendung durch ungenutzte Blockenden wegen der Verwendung von Blockfragmenten.
- Verbesserte, zusammenhängende Anordnung von Blöcken wegen der Verwendung einer Bitmap statt einer Free List.
- Speicherung von Inodes näher an den Daten.
- Auswertung von Informationen über die Plattengeometrie.
- Verweise über Dateisystemgrenzen hinweg.
- Bessere administrative Kontrolle über den Platzverbrauch einzelner Benutzer durch die Einführung eines Quotasystems.

### 3.2.1 Inodes und Datenblöcke

#### 3.2.1.1 Grobstruktur

Um den Durchsatz des neuen Dateisystems zu erhöhen, entschieden sich die Entwickler des FFS zunächst dafür, die Blockgröße drastisch zu vergrößern. Das neue Dateisystem verwendete zuerst Blöcke von 4 KB, heutige FFS–Versionen nehmen standardmäßig sogar Blöcke von 8 KB Größe. Anders als in S5FS werden die Inodes nicht in einer Tabelle am Anfang des Dateisystems gesammelt. Statt dessen teilt man die Platte in Gruppen von Zylindern auf. Jede Zylindergruppe enthält eine Kopie des Superblocks[7], eine *bitmap*, die die freien Blöcke bezeichnet, einen Anteil

[7] Die Superblock–Kopien sind unter Kenntnis der Geometrie so verschoben, daß sie von jeweils unterschiedlichen Köpfen gelesen werden. Diese Maßnahme soll zusätzliche Sicherheit gegen partiell zerstörte Festplatten geben.

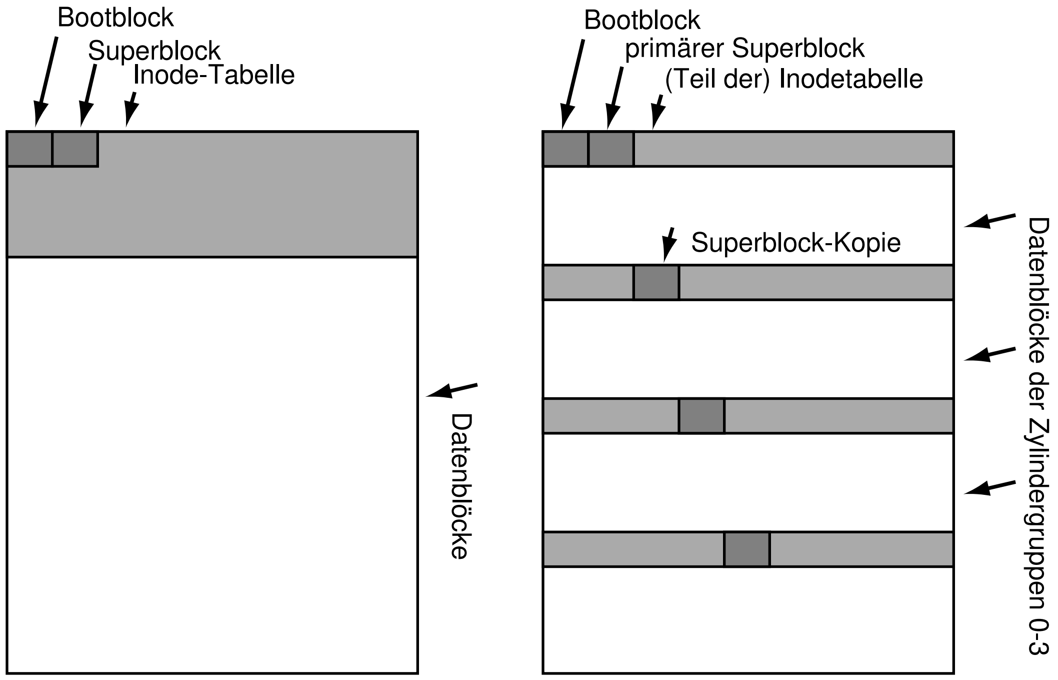



Abbildung 3.6: Im S5FS werden die Inodes gesammelt am Anfang des Dateisystems angelegt. FFS unterteilt die Festplatte in Zylindergruppen, von denen jede Inodes, Datenblöcke und eine Kopie des Superblockes enthält.

an den Inodes des Dateisystems und Datenblöcke. Dies bedeutet, daß jede Zylindergruppe praktisch ein kleines Dateisystem für sich allein ist, das jedoch mit den anderen Zylindergruppen des Dateisystems gemeinsame Block– und Inode–Adressen teilt.

Durch das Konzept der Zylindergruppen verteilen sich die Inodes über die gesamte Festplatte, anstatt sich am Anfang einer Platte zu sammeln. Beim Anlegen von Dateien versucht FFS, Datenblöcke in derselben Zylindergruppe zu finden, in der die Inode für die Datei liegt. Auf diese Weise stehen die Verwaltungsinformationen einer Datei in der Nähe der Daten der Datei, und die Schreib–/Lesekopfbewegungen, die zum Zugriff auf die Daten notwendig sind, werden minimiert.

Dadurch entsteht jedoch ein Problem mit großen Dateien: Wenn in einer Zylindergruppe eine sehr lange Datei angelegt wird, belegt sie alle dort vorhandenen Datenblöcke. In dieser Zylindergruppe sind jedoch wahrscheinlich noch viele Inodes frei. 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 *maxcontig* Bytes ein *long seek* erzwungen, bei dem die Zylindergruppe gewechselt wird. *maxcontig* ist eine dateisystemspezifisch einstellbare Variable, für die erfahrungsgemäß mit einem Wert von $1/8$ der Größe einer Zylindergruppe der gewünwschte Effekt erreicht wird. 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 Daten zu gruppieren: Dateien eines Verzeichnisses werden in derselben Zylindergruppe plaziert. Auf der anderen Seite darf die Gruppierung von Daten aber nicht übertrieben werden: Würde man dies auf die Spitze treiben, fänden sich alle Dateien in einer Gruppe wieder, und man hätte gegenüber S5FS nichts gewonnen. Um zu verhindern, daß die Gruppierungsmaßnahmen zu große Gruppen erzeugen, werden neue Verzeichnisse in Zylindergruppen angelegt, die überdurchschnittlich viele freie Inodes und möglichst wenige Verzeichnisse enthalten.

#### 3.2.1.2 Feinstruktur

Die Entwickler von FFS gingen davon aus, daß eine Festplatte eine quaderförmige Geometrie hat. FFS arbeitet mit Daten über die Umdrehungsgeschwindigkeit der Festplatte und über die Zeit, die das System braucht, um von einer Spur auf eine andere zu wechseln. Mit diesen Informationen wird versucht, die Anordnung von Blöcken innerhalb einer Zylindergruppe zu optimieren: Während die Schreib–/Leseköpfe 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 (siehe Abbildung 3.7).

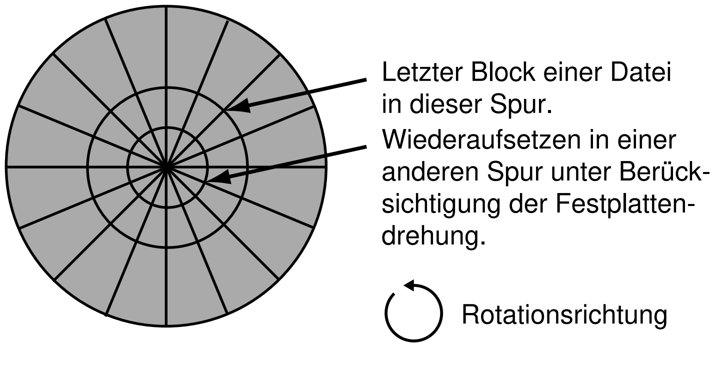



Abbildung 3.7: Während die Schreib–/Leseköpfe der Festplatte die Spur wechseln, dreht sich das Medium unter den Köpfen weiter. FFS versucht, diese Verzögerung bei der Anordnung der Blöcke einer Datei zu berücksichtigen.

Zur Zeit der Entwicklung von FFS war dies durchaus eine sinnvolle Optimierung, aber mit einer nicht quaderförmigen Geometrie oder gar 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.

#### 3.2.1.3 Fragmente

Durch den Übergang auf eine Blockgröße von 8192 Bytes ist es den BSD–Entwicklern gelungen, die Transfergeschwindigkeit sogar bei verstreut angeordneten Dateien stark zu erhöhen. Jedoch steigt der Platzverlust durch verschwendete Blockenden stark an.

Betroffen von dieser Verschwendung sind 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. Wenn bei der Speicherung einer Datei der letzte Datenblock nicht ganz gefüllt werden kann, bricht das Dateisystem einen Datenblock in die vorgegebene Anzahl Fragmente auf und speichert das Dateiende in so wenigen Fragmenten wie möglich. Der Rest des Datenblocks steht zur Speicherung weiterer 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 Bytes großen Fragmenten in derselben Größenordnung zu halten wie bei einem alten Dateisystem mit 1024 Bytes Blockgröße.

Es ist jedoch ausgesprochen ungünstig, wenn eine Datei wächst, die ihren letzten Datenblock mit einem anderen Dateiende teilt (Abbildung 3.8). Da in FFS teilweise belegte Blöcke nur als Dateienden vorkommen dürfen, kann FFS nicht einfach einen anderen Block belegen und die Datei dort fortsetzen. Statt dessen muß das Dateiende in einen leeren Block umkopiert werden, der dann erweitert werden darf. Wenn keine besonderen Vorkehrungen getroffen werden, kann man leicht einen pathologischen Fall konstruieren, in dem sich die Dateienden zweier langsam wachsender Dateien wieder und wieder gegenseitig behindern und in dem exzessives Umkopieren notwendig wird.

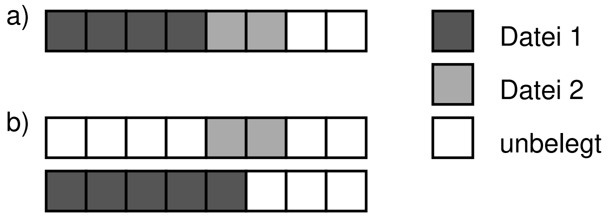



Abbildung 3.8: Wächst eine fragmentierte Datei, kann es notwendig sein, das Dateiende umzukopieren.
a) Ein Block aus acht Fragmenten wird von zwei Dateien geteilt.
b) Wächst Datei 1, muß FFS die vier Fragmente umkopieren, da im fragmentierten Block kein zusammenhängender Platz mehr frei ist und nur der letzte Block einer Datei fragmentiert sein darf.

Solange ausreichend freier Platz im Dateisystem vorhanden ist, versucht FFS, diesen pathologischen Fall zu verhindern, indem es eine Datei blockweise wachsen läßt, sobald das zweite Fragment eines Blocks belegt wird. Sinkt der freie Platz in einem Dateisystem unter eine einstellbare Grenze, unterläßt das Dateisystem diese Optimierung. Das Zeitverhalten von FFS wird außerdem durch eine Veränderung der C–Bibliothek *stdio* unterstützt: Die Bibliothek fragt die Blockgröße des unterliegenden Dateisystems ab und paßt ihre Puffergröße so an, daß möglichst immer ganze Blöcke auf einmal geschrieben werden.

#### 3.2.1.4 Verwaltung freier Blöcke und Inodes

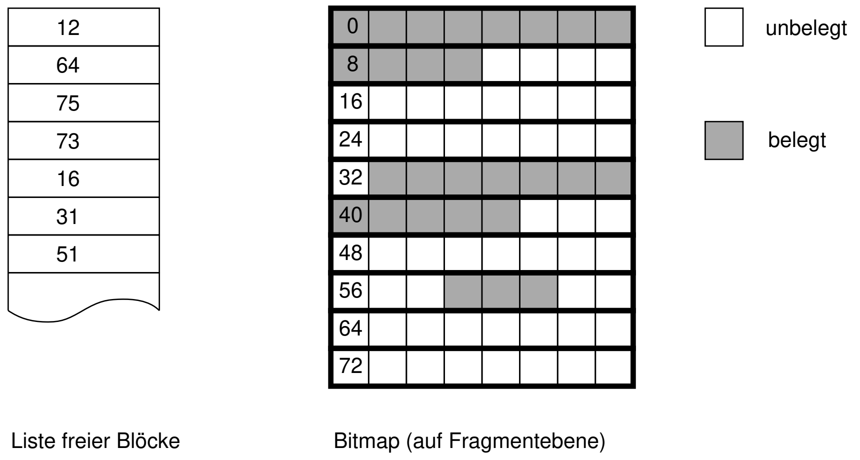



Abbildung 3.9: Verwaltung freier Blöcke durch Free Lists oder durch Bitmaps

Die Verwaltung freien Platzes erfolgt in FFS auf der Ebene von Fragmenten und mit Hilfe von Bitmaps, d.h. 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 Free Lists sind Bitmaps implizit geordnet, und zusammenhängende freie Bereiche lassen sich im Vergleich zum S5FS leichter zu erkennen. Abbildung 3.9 zeigt ein Beispiel einer Free List und der dazu gehörenden Bitmap. Die dargestellte Bitmap arbeitet auf der Ebene von Fragmenten; Blockgrenzen sind durch die etwas dickere Umrahmung angedeutet.

Auch die Adressen, die in den Inodes gespeichert werden, um die zu einer Datei gehörenden Datenblöcke zu bezeichnen, sind fragmentweise auflösend, d.h. zwei aufeinanderfolgende Datenblöcke unterscheiden sich in ihren Adressen nicht um eins, sondern um „Anzahl der Fragmente pro Block“.

#### 3.2.1.5 Quotas

Die Inode– und Blockverwaltung von FFS wurde so modifiziert, daß sie über den Verbrauch von Inodes und Blöcken durch Benutzer und Benutzergruppen Buch führt. Das sogenannte *Quotasystem* erlaubt dem Systemverwalter einen schnellen Überblick darüber, wie viele Inodes oder Blöcke einem einzelnen Benutzer oder seiner Benutzergruppe zugeordnet sind.

Außerdem 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 unter die durch das Limit gesetzte Grenze gegangen ist. Ein hartes Limit kann auf keinen Fall überschritten werden. Das Dateisystem meldet beim Anlegen von Dateien oder beim Schreiben in eine Datei die neue spezielle Fehlermeldung `EDQUOT` („quota exceeded“), die von Anwendungsprogrammen wie `ENOSPC` („no space left on device“) behandelt werden sollte.

### 3.2.2 Verzeichnisse und Pfadnamen

#### 3.2.2.1 Lange Dateinamen

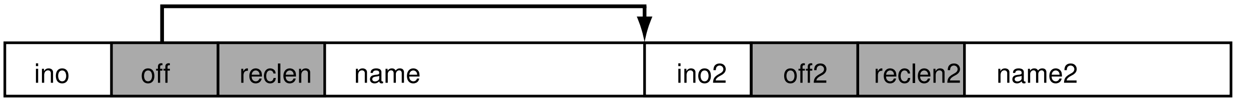



Abbildung 3.10: Ein Verzeichniseintrag besteht aus einer Inodenummer und einem Datenfeld, in dem der bis zu 255 Zeichen lange Name gespeichert werden kann. FFS vermerkt die Länge des Datenfeldes in `off` und die Länge des Namens in `reclen`.

Das BSD FFS läßt Dateinamen von bis zu 255 Zeichen Länge zu. Würde man Namen dieser Länge mit derselben Verzeichnisstruktur verwalten wie S5FS, wäre die Datenspeicherung wenig effizient.

FFS verwendet daher eine Struktur, in der Namenseinträge in Verzeichnissen eine variable Länge haben können (siehe Abbildung 3.10): Das Feld `off` gibt an, wie lang der Verzeichniseintrag ist, und das Feld `reclen` beinhaltet die Information über die Länge des im Verzeichniseintrag gespeicherten Namens. Wird ein Verzeichniseintrag gelöscht, wird der von ihm belegte Platz dem `off`–Feld des vorhergehenden Eintrages zugeschlagen. Verzeichnisblöcke, die vollständig leer sind, werden freigegeben und in der Inode des Verzeichnisses auf Null gesetzt. Verzeichniseinträge überspannen 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 mit den Funktionen der `opendir()`–Familie realisiert.

#### 3.2.2.2 Symbolische Verweise

Neu in FFS gegenüber S5FS sind außerdem symbolische Verweise (*symbolic links*). Es handelt sich um einen neuen Dateityp, der von `ls` als `l` gelistet wird. Die Verweisdatei enthält einen Pfadnamen. Wenn das Dateisystem bei der Auflösung eines Pfadnamens auf eine solche Verweisdatei trifft, wird der Inhalt der Verweisdatei dem noch zu interpretierenden Restpfadnamen vorangestellt und der so erzeugte Name als Pfadname interpretiert.

Symbolische Verweise können im Gegensatz zu gewöhnlichen Verweisen Dateisystemgrenzen überwinden. Außerdem sind bei einem symbolischen Verweis der originale Dateiname und der Verweis immer klar zu unterscheiden, und es kann nicht durch `unlink()` und `create` zu einer Desynchronisation der beiden Namen kommen.

## 3.3 Second Extended Filesystem

Rémy Card, Theodore Tsó und Stephen Tweedie entwickelten 1993 das *Second Extended Filesystem (ext2)* [Ca95][8]. Obwohl die Entwicklung ursprünglich für das frei verfügbare Linux–Betriebssystem gedacht war, ist sie in der Zwischenzeit auf eine Reihe von anderen Plattformen portiert worden. Es existieren unter anderem Kernelserver für den GNU Hurd Microkernel, den Mach Microkernel und das MASIX–Projekt [Ca93].

Linux wurde in der Frühphase unter Andrew Tanenbaums Minix–Betriebssystem [Ta87] entwickelt. Es war daher naheliegend, für Linux ebenfalls das dem S5FS sehr ähnliche Minix–Dateisystem zu verwenden. Da das Minix–Dateisystem unter denselben Nachteilen und Beschränkungen litt wie S5FS, entschloß man sich 1993 zur Entwicklung eines neuen, leistungsfähigeren Dateisystems. Das Resultat, ext2, ist strukturell dem BSD FFS sehr ähnlich, basiert aber nicht auf BSD–Quelltexten. Gegenüber dem BSD–Code ist der ext2–Quelltext kleiner und einfacher. Trotzdem verallgemeinert ext2 einige Konzepte von FFS und ist schneller als das BSD–Dateisystem[9].

Das Linux–Second Extended Filesystem unterscheidet sich vom Berkeley Fast Filing System in den folgenden Punkten:

- ext2 verfügt nicht über den FFS–Code zur kleinräumigen Optimierung der Blockanordnung.
- ext2 arbeitet mit Blöcken in der Größenordnung von FFS–Fragmenten und verwendet andere Hilfsmittel als FFS zur Erzeugung von zusammenhängend angeordneten Datenbereichen.
- ext2 kennt keine Fragmente. Der umständliche Code zur Fragmentverwaltung und das Umkopieren von Fragmenten bei langsam wachsenden Dateien entfällt damit.
- ext2 kennt besondere Dateitypen (unveränderbare Dateien, nicht kürzbare Dateien) und besondere Dateiattribute.

[8] [Ca95] erklärt auch die Herkunft des Namens: Das *Extended Filesystem* war eine Designstudie, die den dringenden Bedarf der Linux–Anwender nach einem Dateisystem mit langen Namen befriedigen sollte. Es war jedoch niemals als Dauerlösung gedacht.

[9] In [Ca95] wird die Geschwindigkeit von BSD FFS und Linux–ext2 verglichen. Ein allgemeiner Geschwindigkeitsvergleich zwischen Linux, FreeBSD und Solaris auf Intel–Pentium–Plattformen findet sich in [La96].

## 3.3.1 Inodes und Datenblöcke

### 3.3.1.1 Grobstruktur

Das Layout von ext2 auf der Festplatte ist ähnlich dem Layout von FFS: ext2 besteht aus Blockgruppen (*block groups*), die konzeptuell den Zylindergruppen von FFS entsprechen. ext2 wertet jedoch keine Geometrieinformationen des Mediums aus, so daß keine Entsprechung zwischen dem Layout des Dateisystems und der physikalischen Geometrie der Festplatte zu erwarten ist. Das ext2–Dateisystem kann mit Blöcken von 1, 2 oder 4 KB Größe arbeiten. Standardmäßig wird eine Blockgröße von 1 KB verwendet.

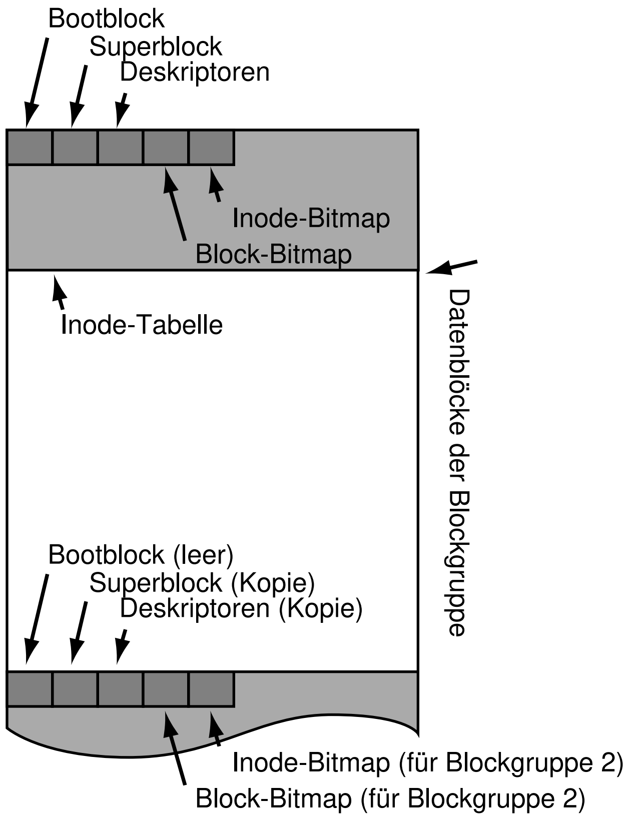



Abbildung 3.11: Aufbau einer Blockgruppe im Linux Second Extended Filesystem

Eine Blockgruppe besteht in ext2 aus einer Kopie des Superblocks, einer Kopie aller Blockgruppendeskriptoren (siehe unten), einer Blockbelegungsbitmap für diese Blockgruppe, einer Inodebelegungsbitmap für die Inodes dieser Blockgruppe und einem Teil der Inodetabelle. Abbildung 3.11 gibt einen Überblick über das Layout einer Blockgruppe.

Eine Blockgruppe wird durch ihren Deskriptor beschrieben. Der Deskriptor enthält neben den Blockadressen der Blockbelegungsbitmap und Inodebelegungsbitmap auch die Startadresse der Inodetabelle für diese Blockgruppe (siehe Abbildung 3.12). Außerdem sind einige statistische Informationen über die Auslastung der Blockgruppe gespeichert: Neben der Anzahl der freien Blöcke und der freien Inodes in der Blockgruppe wird die Anzahl der in dieser Blockgruppe angelegten Verzeichnisse mitgeführt. Das Dateisystem verwendet diese Informationen bei der

Suche nach freien Blöcken und freien Inodes. Die Anzahl der Verzeichnisse kann beim Anlegen neuer Verzeichnisse verwendet werden. Der Deskriptor belegt insgesamt 32 Bytes; bei einer Blockgröße von 1 KB passen also 32 Deskriptoren in einen Block.

```
/*
 * Structure of a blocks group descriptor
 */
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 3.12: Ein Deskriptor für eine Blockgruppe in ext2 gibt die Adressen der Bitmapblöcke und der Inodetabelle an. Er enthält außerdem summarische Informationen über die Auslastung der Blockgruppe. Diese Informationen werden beim Anlegen neuer Verzeichnisse und bei der Suche nach freien Blöcken und Inodes ausgewertet.

Die Größe der Blockgruppen wird durch die Tatsache beschränkt, daß die Inode– und Blockbitmaps der Blockgruppe in jeweils einen Datenblock passen müssen. Bei einer Blockgröße von 1024 Bytes kann eine Blockgruppe also maximal 8192 Inodes und 8192 Blöcke enthalten; bei größeren Blockgrößen liegt das Limit entsprechend bei 16384 bzw. 32768 Objekten pro Bitmapblock. Ein mit Defaultparametern angelegtes Dateisystem wird also aus Blockgruppen von jeweils 8 MB bestehen (8192 Blöcke zu jeweils 1 KB pro Blockgruppe).

Beim Anlegen von Dateien und Verzeichnissen verwendet ext2 einen ähnlichen Mechanismus wie FFS: Inodes von Dateien versucht ext2 in derselben Blockgruppe einzuordnen wie die Inode des Verzeichnisses, in dem sie enthalten sind. Datenblöcke von Dateien und Verzeichnissen versucht ext2 in derselben Blockgruppe zu belegen, in der die Inode dieser 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[10]. Dazu werden in allen Blockgruppendeskriptoren die Felder `bg_free_blocks_count` und `bg_free_inodes_count` ausgewertet[11].

[10] ext2 führt in den Deskriptoren der Blockgruppen einen Zähler für die Anzahl der in dieser Blockgruppe angelegten Verzeichnisse mit, wertet diesen aber nicht aus.

[11] ext2 hält zur Beschleunigung dieses Vorgangs nicht nur den Superblock des Dateisystems, sondern auch alle Blockgruppendeskriptoren im Arbeitsspeicher.

#### 3.3.1.2 Feinstruktur

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 (*extents*) zu belegen. Dazu verwendet das Dateisystem einen Blockbelegungsalgorithmus, der mit einem Zielblock arbeitet (*goal based allocator*). Dieser Algorithmus versucht zunächst, den Zielblock zu belegen. Ist dieser nicht verfügbar, werden in der Umgebung des Zielblocks freie Blöcke gesucht. Ist auch dort kein freier Platz vorhanden, wird auf eine lineare Suche nach freiem Platz zurückgegriffen.

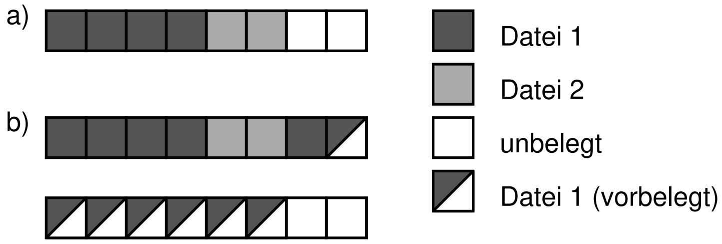



Abbildung 3.13: Bildung von zusammenhängenden Bereichen durch Vorbestellung von Blöcken im ext2–Dateisystem
a) Eine Situation ähnlich der in Abbildung 3.8.
b) ext2 kann die Datei verlängern, ohne umzukopieren. Gleichzeitig werden die folgenden Blöcke vorbestellt, um sicherzustellen, daß weitere Schreibanforderungen Extents bilden.

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 Vorbestellung von Blöcken (*preallocation*): Wann immer ein neuer Block für eine Datei angefordert wird, versucht das Dateisystem, bis zu acht zusammenhängende Blöcke in Folge zu belegen (siehe Abbildung 3.13). Nachfolgende Anforderungen von Blöcken für diese Datei werden dann aus dem Pool dieser vorbestellten Blöcke befriedigt. Die Vorbestellung wird aufgegeben (die vorbestellten Blöcke werden freigegeben), wenn der Zielblock nicht im vorbestellten Bereich liegt oder wenn die Datei geschlossen wird[12].

Dateien, die aus zusammenhängenden Stücken bestehen, werden vom Dateisystem und den Gerätetreibern der Festplatte in Linux besonders effizient behandelt: Das ext2–Dateisystem versucht, aufeinanderfolgende `read(2)`–Aufrufe innerhalb einer Datei zu erkennen, und erzeugt in diesem Fall auf Verdacht Leseanforderungen, um den Buffer Cache im voraus zu laden (*read ahead*). Gleichzeitig versucht Linux, aufeinanderfolgende Leseanforderungen zusammenzuziehen (*read clustering*), so daß der Gerätetreiber statt vieler kleiner, aufeinanderfolgender Anforderungen nur eine große Anforderung zu sehen bekommt.

[12]Nach [Ca95] liegt die Trefferrate für Preallocation auch bei stark gefüllten Dateisystem noch bei ca. 75 Prozent. Bei gefüllten Dateisystemen gelingt es allerdings immer seltener, die Vorbestellung auf die volle Länge von acht Blöcken auszudehnen.

Beide Maßnahmen zusammen (Preallocation und Read Clustering) 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–Dateisystem 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. Während das FFS in dem Beispiel in Abbildung 3.8 dargestellten Situation Fragmente umkopieren muß, handhabt das ext2–Dateisystem diese Situation, ohne Daten verschieben zu müssen.

#### 3.3.1.3 Inode

Die Inode–Datenstruktur des ext2–Dateisystems gleicht im wesentlichen der des FFS. Sie ist ebenso wie die FFS–Inode 128 Bytes groß. Unterschiede bestehen lediglich 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 ist geplant, die überzähligen Bytes der Inode für *POSIX.1e Access Control Lists (ACLs)* [POSIX.1e] zu verwenden. In dieser Richtung ist jedoch bislang kein Code veröffentlicht worden.

Einen Unterschied zum FFS bildet das zusätzliche Feld `i_dtime`. Das ext2–Dateisystem 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 (zusätzlich ist `i_links_count` auf Null gesetzt). Mit geeigneten Werkzeugen ist es dem Systemverwalter möglich, nach gelöschten Dateien zu suchen und die Datei wiederherzustellen, sofern die durch das Löschen freigegebenen Datenblöcke noch nicht überschrieben sind[13]. Zum ext2–Dateisystem gehört neben dem Kernelcode auch eine Usermode–Bibliothek, die privilegierten Programmen den strukturierten Zugriff auf die Datenstrukturen des Dateisystems erlaubt. Diese Bibliothek, die von den Usermode–Programmen des ext2–Dateisystems benutzt wird, erleichtert die Entwicklung solcher Hilfsprogramme.

Das ext2–Dateisystem sieht ein Feld `i_flags` in der Inode–Struktur vor. Diese Flags können verwendet werden, um einer Datei eine Reihe von besonderen Eigenschaften zu verleihen. Dazu gehören unter anderem Flags, die eine Datei als unveränderlich und als Logbuch kennzeichnen. Eine unveränderliche Datei (*immutable file*) kann weder gelöscht noch umbenannt noch auf eine andere Art verändert werden, sobald das System in den Mehrbenutzerbetrieb gewechselt ist. Eine Logbuchdatei (*append only file*) kann ebenfalls weder gelöscht noch umbenannt werden. Als einzige Schreiboperation ist das Anhängen an das Dateiende erlaubt. Andere Flags erzwingen für bestimmte Dateien synchrone Updates, d.h. jeder `open(2)`–Aufruf auf diese Datei verhält sich so, als sei die Datei zusätzlich mit dem Modus `O_SYNC` geöffnet worden. Ein weiteres Flag kennzeichnet die Datei als aus dem Backup auszuschließen. Das ext2–Dateisystem stellt

[13] Dazu genügt es im einfachsten Fall, die `i_dtime` der Datei auf Null zu setzen und einen Filesystemcheck einzuleiten. Der `fsck` erkennt die Datei dann als nicht von der Wurzel des Dateibaums aus erreichbare Datei und legt sie im `/lost+found`–Verzeichnis ab.

```
/*
 * Constants relative to the data blocks
 */
#define    EXT2_NDIR_BLOCKS        12
#define    EXT2_IND_BLOCK            EXT2_NDIR_BLOCKS
#define    EXT2_DIND_BLOCK            (EXT2_IND_BLOCK + 1)
#define    EXT2_TIND_BLOCK            (EXT2_DIND_BLOCK + 1)
#define    EXT2_N_BLOCKS           (EXT2_TIND_BLOCK + 1)
/*
 * Structure of an inode on the disk
 */
struct ext2_inode {
    __u16   i_mode;      /* File mode */
    __u16   i_uid;       /* 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;       /* Group Id */
    __u16   i_links_count; /* Links count */
    __u32   i_blocks;    /* Blocks count */
    __u32   i_flags;     /* File flags (chflags) */
    /* gekürzt */
    __u32   i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
    __u32   i_version;  /* File version (for NFS) */
    __u32   i_file_acl; /* File ACL, unused */
    __u32   i_dir_acl;  /* Directory ACL, unused */
    __u32   i_faddr;    /* Fragment address, unused */
    /* gekürzt */
};
```

Abbildung 3.14: Inode–Datenstruktur in ext2. Ausgelassen wurden zwei `union`–Blöcke mit betriebssystemabhängiger Information (in Linux nicht verwendet).

`ioctl(2)`–Aufrufe zur Verfügung, um die Flags einer Datei zu setzen und zu lesen. Das Kommando `chflags(1)` nutzt dieses Interface und erlaubt Benutzern den Zugriff auf die Flags.

### 3.3.2 Verzeichnisse und Pfadnamen

Ebenso wie FFS läßt das ext2–Dateisystem bis zu 255 Zeichen lange Dateinamen zu. Die Organisation der Verzeichnisse erfolgt genau wie in FFS (siehe Abbildung 3.10) als lineare Liste von Inodenummern und Namen.

# Kapitel 4

# Dynamisierung der Anzahl der Inodes

Bei allen bisher betrachteten UNIX–Dateisystemen wird die Anzahl der Inodes bei der Generierung des Dateisystems mit dem betreffenden `mkfs`–Kommando statisch festgelegt. Diese Anzahl der Inodes ist dann für den Rest der Lebensdauer des Dateisystems Maximalzahl der Dateien und Verzeichnisse, die zu einem gegebenen Zeitpunkt in diesem Dateisystem existieren können. Um die Anzahl der Inodes in einem Dateisystem zu verändern, ist es notwendig, den Inhalt des Dateisystems zu sichern, das Dateisystem neu anzulegen (der alte Inhalt wird dabei zerstört) und den Inhalt von der Sicherung zu rekonstruieren.

Dies ist insbesondere bei großen Dateisystemen ein sehr aufwendiger Prozeß, der außerdem mit einer Betriebsunterbrechung verbunden ist. Daher schätzt ein erfahrener Systemverwalter die Anzahl der benötigten Inodes in der Regel sehr großzügig und reserviert um den Faktor zwei oder drei mehr Inodes, als das Dateisystem voraussichtlich benötigen wird.

Das Ziel dieser Arbeit ist es, eine variable Anzahl der Inodes in einem Dateisystem vorzusehen, d.h. ein vorhandenes Dateisystem so zu modifizieren, daß es ohne Betriebsunterbrechung nach Bedarf freie Datenblöcke in Inodeblöcke umwandeln kann und diese in die Inodetabelle einsortiert. Diese Arbeit geht dabei von einem bereits existierenden Dateisystem aus, nämlich dem ext2–Dateisystem von Rémy Card [Ca93][Ca95], das im Kontext eines Linux–Kernels [Li96] abläuft. Durch direkte Gegenüberstellung mit dem konventionellen Ausgangsdateisystem mit einer statischen Anzahl von Inodes wird verdeutlicht, welche Annahmen und Eigenschaften von konventionellen Dateisystemen durch diese Dynamisierung verändert werden oder ihre Gültigkeit verlieren.

Viele der Überlegungen in diesem Kapitel sind unabhängig von einer konkreten Implementierung. Die Darstellung der Sachverhalte lehnt sich jedoch an Ideen und Terminologien an, wie sie im Zusammenhang mit dem Linux Second Extended Filesystem (ext2) verwendet werden. Im Folgenden wird unter dem *konventionellen* oder *alten* Dateisystem das ext2–Dateisystem verstanden, wie es in Abschnitt 3.3 vorgestellt wurde. Die meisten genannten Konzepte sind auf das BSD FFS übertragbar. Das in Kapitel 5 vorgestellte ext3–Dateisystem wird auch *Dateisystem mit dynamischer Inodeanzahl* oder *neues* Dateisystem genannt.

## 4.1 Vorüberlegungen

### 4.1.1 Methoden der Strukturbeschreibung

Dateisysteme organisieren die Datenblöcke einer Festplattenpartition zu höheren Datenstrukturen. Jedem Datenblock kommt dabei im Rahmen des Dateisystems genau eine Aufgabe zu. Man kann Funktionen angeben, die jeder Datenstruktur eines Dateisystems absolute Blocknummern[1] zuordnen, die diese Datenstruktur also auf eine Reihe von Datenblöcken der Festplatte abbilden. Eine solche Funktion wird im Folgenden nach der Kernelfunktion `bmap()` *mapping* genannt.

```
mode  inode#     = {0 .. NINODES-1};
mode  abs_block# = {0 .. NBLOCKS-1};

let int i_per_b  = BSIZE/ISIZE;
func s5imap = (inode# i) abs_block#:
     int(i/i_per_b) + ISTART;
```

Abbildung 4.1: Mappingfunktion für die Inodetabelle im S5FS: Die Funktion `s5imap` liefert für eine gegebene Inodenummer eine absolute Blocknummer. Das Dateisystem besteht dabei aus `NBLOCKS` Datenblöcken der Größe `BSIZE` Bytes und enthält `NINODES` Inodes der Größe `ISIZE` Bytes beginnend an der Position `ISTART`.

Für einfache, zusammenhängende Datenstrukturen ist das Mapping trivial. Es genügt, den Startblock einer solchen Datenstruktur und ihre Länge anzugeben. Ein Beispiel für eine solche Struktur ist die Inodetabelle im System V–Dateisystems (siehe Abbildung 4.1[2]) oder die Deskriptortabelle des ext2–Dateisystems.

Ein komplizierteres Mapping zählt die Datenblöcke einer Struktur einzeln in einer tabellenartigen Struktur auf. Ein Beispiel dafür ist die Inode, die eine Tabelle der Datenblöcke einer Datei darstellt. Im Gegensatz zu einem Paar (Startblock, Länge) brauchen diese Datenblöcke nicht zusammenhängend aufeinanderfolgend abgespeichert zu werden, und die Struktur ist problemlos erweiterbar. Allerdings benötigt die Abbildungstabelle, die einer relativen Blocknummer[3] eine absolute Blocknummer zuordnet, selbst Speicherplatz auf der Festplatte. Die Inode enthält deswegen noch ein zweites Mapping, das einer Datei Datenblöcke mit Verwaltungsinformationen zuordnet, nämlich die indirekten Blöcke auf den verschiedenen Indirektionsebenen.

Beide Methoden der Realisierung von Mappings stellen Extreme dar: Während sich bei der System V–Inodetabelle die Lage der folgenden Datenblöcke aus der Lage des ersten Blocks implizit ergibt, zählt das Mapping einer Inode jeden Datenblock einzeln auf.

[1] Unter einer absoluten Blocknummer wird in dieser Arbeit eine lineare Blockadresse relativ zum Anfang der Partition verstanden.

[2] Die verwendete Notation wird in Abschnitt 4.2 eingeführt.

[3] Relativ zum Anfang der Datei.

Zwischenstufen sind denkbar und werden auch tatsächlich eingesetzt: Die Lage der Sub–Inodetabelle einer Blockgruppe wird im ext2–Dateisystem durch ein Paar (Startblock, Länge) beschrieben. Dabei ist der Startblock der Sub–Inodetabelle für diese Blockgruppe im Deskriptor dieser Blockgruppe genannt (siehe Abbildung 3.12, Feld `bg_inode_bitmap`), die Länge der Tabelle ist für alle Blockgroups außer der letzten gleich und wird im Superblock des Dateisystems notiert. Die gesamte Inodetabelle für ein ext2–Dateisystem wird also durch die Liste der Startblöcke der Sub–Inodetabellen in den Deskriptoren beschrieben. Dies ist eine kompliziertere und flexiblere Struktur als ein einfaches (Startblock, Länge)–Paar, erreicht aber nicht die Komplexität und Flexibilität einer Inode.

### 4.1.2 Ansatz für eine dynamisch wachsende Inodetabelle

Wenn man die Inodetabelle eines Dateisystems flexibler gestalten möchte, ist es naheliegend, den Übergang zu einer vollen Beschreibung der Inodetabelle mit Hilfe einer Inode zu machen, denn dies ist die flexibelste Struktur im Dateisystem zur Beschreibung eines Mappings. Wie in Abbildung 4.2 dargestellt, müßte dazu im Dateisystem eine ausgezeichnete Inode vorhanden sein, deren Datenblöcke genau die Blöcke der Inodetabelle des Dateisystems sind. Die Inodetabelle ist dann strukturell eine gewöhnliche Datei.

Die Inode, die die Inodetabelle beschreibt, enthält dabei wie jede normale Inode die Blocknummern von Datenblöcken. Die Inode der Inodetabellendatei ist dabei jedoch eine ausgezeichnete Inode, an die besondere Anforderungen gestellt werden. Es wird verlangt, daß die Daten in diesen Datenblöcke eine bestimmte Form haben müssen, denn es muß sich bei diesen Datenblöcken um Inodeblöcke handeln, d.h. sie müssen mit den On–Disk–Repräsentationen von Inodes des Dateisystems gefüllt sein. Dadurch ist es möglich, die Inodeblöcke des Dateisystems frei anzuordnen, denn die spezielle Inode der Inodetabellendatei führt über jeden einzelnen Inodeblock Buch. Die Einschränkung, daß die Inodetabelle ein zusammenhängender Bereich von Blöcken sein muß, fällt weg. Ebenso wie bei einer gewöhnlichen Datei ist es auch bei der Inodetabellendatei möglich, die Anzahl der Blöcke in der Datei zu vergrößern oder zu verkleinern. Dadurch läßt sich die Anzahl der Inodes im Dateisystem nachträglich verändern.

Daraus ergeben sich folgende Fragen, die in den folgenden Kapiteln beantwortet werden:

- Die Blöcke der Inodetabelle sind nun frei positionierbar. Gibt es Regeln, nach denen sie sich günstig positionieren lassen? (siehe Abschnitt 4.1.3)
- Wie findet man schnell freie Inodes in der Inodetabellendatei? (siehe Abschnitt 4.1.4)
- Wie handhabt das Dateisystem die Freigabe von Inodes? (siehe Abschnitt 4.1.5)
- Die Inodetabelle wird durch eine Inode beschrieben. Um diese Inode zu finden, muß man in der Inodetabelle nachschlagen. Wie löst man die Selbstbezüglichkeit dieser Definition auf? (siehe Abschnitt 4.1.6)

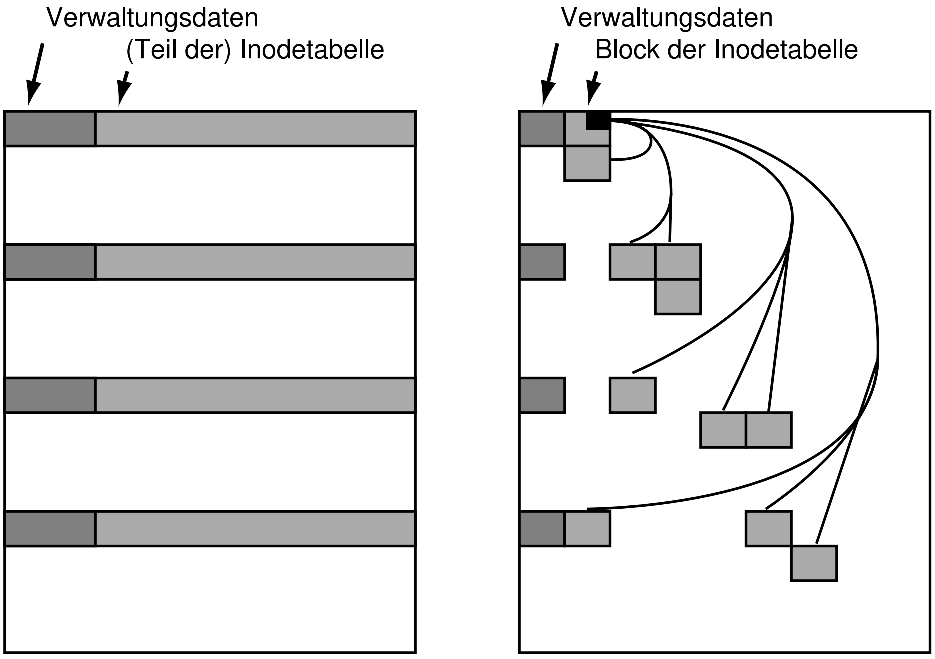



Abbildung 4.2: (links) In herkömmlichen Dateisystemen werden am Anfang der Blockgruppe Verwaltungsdaten plaziert (dunkelgrau), gefolgt von den Inodeblöcken dieser Blockgruppe (hellgrau). Der Rest der Blockgruppe besteht aus Datenblöcken und kann zur Datenspeicherung genutzt werden.
(rechts) Beschreibt man die Inodeblöcke mittels einer ausgezeichneten Inode (schwarz), können die Inodeblöcke im Dateisystem frei plaziert werden. Inodeblöcke lassen sich nicht mehr fest einer Blockgruppe zugeordnen. Die Bögen deuten an, wie die Inode der Inodetabellendatei auf ihre Datenblöcke verweist. Jeder dieser Datenblöcke kann weitere Inodes enthalten, die ihrerseits auf Datenblöcke von Dateien verweisen (nicht dargestellt).

### 4.1.3 Positionierung von Inodes und Datenblöcken

In den in Kapitel 3 beschriebenen Dateisystemen war die Position jeder Inode im Dateisystem durch ihre Inodenummer immer implizit vorgegeben. Beispielweise hat im ext2–Dateisystem die Inodenummer implizit die Blockgruppe angegeben, der die Inode angehörte. Dies ist nun nicht mehr der Fall: Wird die Inodetabelle selbst durch eine Inode beschrieben, sind die Blöcke der Inodetabelle frei positionierbar. Damit stellt sich die Frage, wie man sie positioniert bzw. ob man ihre Positionierung überhaupt sinnvoll steuern kann.

Dabei gelten einige Randbedingungen: Eine Inode ist nur einen Bruchteil eines Datenblocks groß. So ist beispielsweise im typischen ext2–Dateisystem in einem Datenblock Platz für acht Inodes[4]. Die Umwandlung von Datenblöcken in Inodeblöcke kann nur auf der Basis von ganzen Datenblöcken geschehen. Das bedeutet, Inodes entstehen immer in Gruppen von einem Datenblock — bei Parameterwahl für Block– und Inodegröße wie beim ext2–Dateisystem entstehen also immer acht zusammenhängende Inodes auf einmal.

[4] 128 Bytes pro Inode und eine angenommene Blockgröße von 1024 Bytes.

Dies erschwert die Positionierung von Inodeblöcken. Wünschenswert wäre es, die Inode einer Datei möglichst nahe dem ersten Datenblock einer Datei zu finden. Diese Inode hängt jedoch in jedem Fall mit einigen weiteren Inodes desselben Inodeblockes zusammen, die dann nicht notwendigerweise nahe dem ersten Datenblock der durch sie beschriebenen Datei liegen. Hinzu kommt, daß keine Datenstruktur existiert, mit der schnell feststellbar wäre, ob in der Nähe eines bestimmten Datenblockes freie Inodes vorhanden sind, und die Einführung solcher Datenstrukturen würde eine grundlegende Veränderung des zugrundeliegenden Paradigmas notwendig machen. Kapitel 7 geht auf mögliche Konstruktionen von Dateisystemen ein, die so etwas leisten können. Es handelt sich dabei jedoch um Projekte, die den Umfang dieser Arbeit sprengen.

#### 4.1.3.1 Triviale Numerierung

Die einfachste Lösung ist es, das Problem der Plazierung von Inodes zunächst einmal zu ignorieren und die Inodetabellendatei nach denselben Regeln zu erweitern, die auch für normale Dateien gelten. Neue Inodeblöcke werden demnach möglichst in der Nähe des jeweils vorhergehenden Blockes der Inodetabellendatei gesucht. Die Inodetabellendatei wird nach diesem Verfahren möglichst kompakt und zusammenhängend angelegt. Es ist zu untersuchen, ob dies eine ausreichend gute Lösung ist (siehe Kapitel 6).

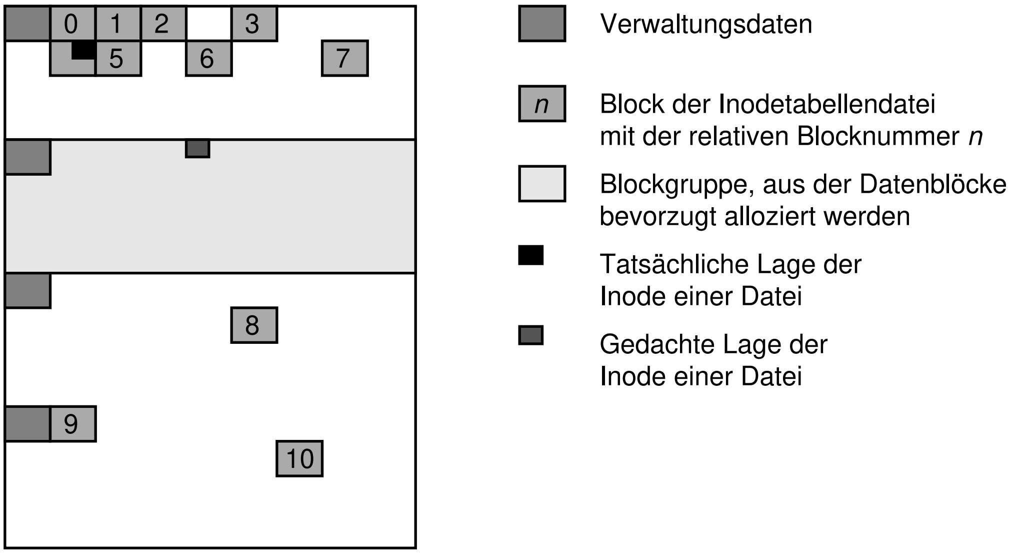



Abbildung 4.3: In konventionellen Dateisystemen werden die Datenblöcke einer Datei aus derselben Blockgruppe alloziert, in der sich die Inode der Datei befindet. Im dynamischen Dateisystem ist bei „trivialer Numerierung“ die Zuordnung von Inodes zu einer Blockgruppe nicht mehr implizit gegeben. Beim Anlegen der Inode wird daher eine Blockgruppe bestimmt, aus der die Datenblöcke der Inode bevorzugt belegt werden sollen, um die Blockgruppen der Festplatte gleichmäßig auszunutzen.

In konventionellen Dateisystemen hat die Position der Inode jedoch Rückwirkungen auf die Positionierung der Datenblöcke der Datei: Normalerweise wird der Blockallokator versuchen, Datenblöcke für eine Datei in derselben Blockgruppe zu finden, in der die Inode der Datei plaziert ist. Diese Strategie ist im neuen Dateisystem nutzlos: Verwendet man die eben angesprochene triviale Positionierungsregel für Inodeblöcke, wird das Dateisystem die Blöcke der Inodetabellendatei möglichst dicht beieinander anlegen, also in derselben Blockgruppe, sofern es die Platzverhältnisse zulassen. Indem die Datenblöcke der Dateien in derselben Blockgruppe angelegt werden wie ihre Inode, läuft diese Blockgruppe sehr schnell über, während andere Blockgruppen gar nicht genutzt werden.

Einen recht einfachen Ausweg aus dieser Situation findet man, indem man beim Anlegen einer Inode wie beim konventionellen Dateisystem berechnet, in welcher Blockgruppe die Inode der Datei zu liegen kommen würde, hätte man noch statische Inodetabellen. Diese Blockgruppe muß in einem neu zu schaffenden Hinweisfeld in der Inode abgespeichert werden, und dieser Hinweis muß als Ausgangspunkt für die Blockallokation dieser Inode dienen (siehe Abbildung 4.3).

Diese Lösung minimiert zwar den Änderungsaufwand an der Verwaltung freier Datenblöcke, ist aber insofern unbefriedigend, als der Abstand zwischen einer Inode und den durch die Inode verwalteten Datenblöcken sehr groß werden kann.

#### 4.1.3.2 Umlaufende Numerierung

Im konventionellen Dateisystem ergibt sich aus der Inodenummer implizit die Blockgruppe, in der die Inode liegt: Eine Inode mit der Inodenummer `i` entstammt der Blockgruppe `int(i/inodes_per_group)`. Die Inodenummern sind also an Blockgruppen gebunden, und diese Eigenschaft läßt sich ausnutzen, um Inodes relativ nahe an den von ihnen referenzierten Datenblöcken zu positionieren.

In einem Dateisystem mit einer dynamischen Inodeanzahl läßt sich nicht vorhersagen, wie viele Inodes pro Blockgruppe angelegt werden. Die Zuordnung von Inodenummern zu Blockgruppen muß also ohne eine Konstante `inodes_per_group` auskommen. Verwendet man statt dessen die Anzahl der Blockgruppen im Dateisystem `ngroups`, ergibt sich für die Inodenummer `i` die Blockgruppe `int(i/inodes_per_block) mod ngroups` (*umlaufende Numerierung von Inodeblöcken*). Die ersten `inodes_per_block` Inodes liegen demnach in der Blockgruppe `0`, die folgenden `inodes_per_block` Inodes in der Blockgruppe `1` usw. Nach jeweils `ngroups` Inodeblöcken wird wieder auf die Blockgruppe `0` zugegriffen (siehe Abbildung 4.4).

Es läßt sich nicht garantieren, daß der Inodeblock tatsächlich in der vorherbestimmten Blockgruppe zu liegen kommt, denn es besteht immer die Möglichkeit, daß in dieser Blockgruppe keine freien Blöcke mehr vorhanden sind. Der Blockallokator wird dann eine andere Blockgruppe auswählen. Trotzdem wird die Inode nahe ihren Datenblöcken zu liegen kommen, denn diese werden aus derjenigen Blockgruppe entnommen, in der die Inode tatsächlich angelegt worden ist. Beim Anlegen einer neuen Inode kann der Allokator also wie vom konventionellen Dateisystem gewohnt eine Zielblockgruppe `j`$\in \{$`0.. ngroups`$\}$ berechnen und würde dann die Inodeblöcke nach freien Inodes durchsuchen, für die gilt: `b mod ngroups == j`, `b` ist eine relative Blocknummer in der Inodetabellendatei. Findet er in der Zielblockgruppe keine freien Inodes, würde er einen neuen Inodeblock in dieser Blockgruppe anlegen.

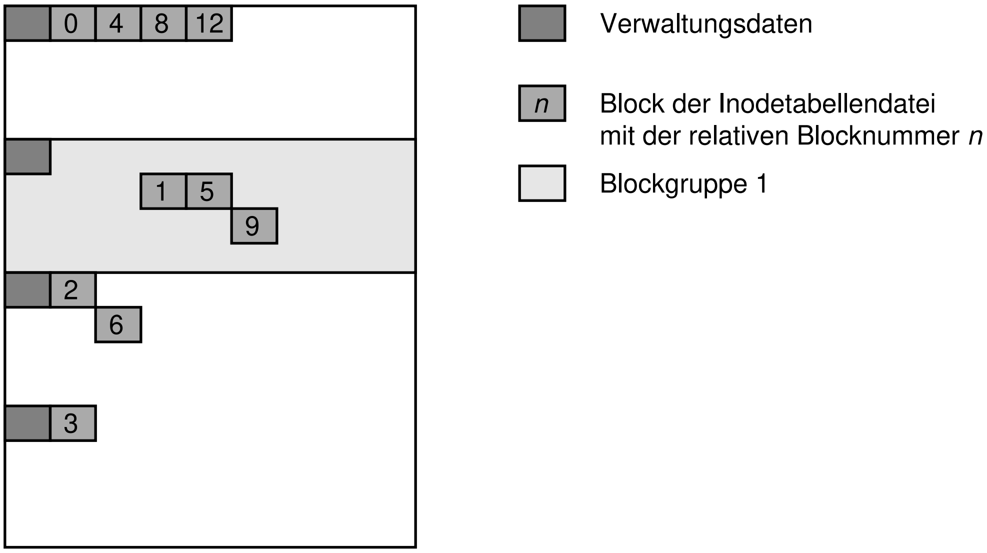



Abbildung 4.4: Bei umlaufender Numerierung von Inodeblöcken wird der `j`-te Inodeblock nach Möglichkeit in der Blockgruppe `j mod ngroups` angelegt. Das Beispiel zeigt ein Dateisystem mit vier Blockgroups, bei dem in der Blockgruppe 1 drei Inodeblöcke mit den relativen Blocknummern 1, 5, und 9 der Inodetabellendatei angelegt sind. Die gezeigte Inodetabellendatei hat ungleichmäßig viele Inodeblöcke pro Blockgruppe. Sie ist eine dünn besetzte Datei: Die Blöcke mit den relativen Nummern 7, 10, und 11 fehlen.

Es besteht kein Grund anzunehmen, daß der Verbrauch an Inodes in den Blockgruppen des Dateisystems gleichmäßig sein wird: Wie im Abschnitt 3.3.1.1 erläutert, legt das ext2–Dateisystem Inodes von Dateien nach Möglichkeit in derselben Blockgruppe an, in der auch das Verzeichnis liegt, das sie enthält. Man kann also davon ausgehen, daß bei einer umlaufenden Numerierung von Inodeblöcken die Inodetabellendatei eine dünn besetzte Datei sein wird. Das ist ungünstig, wenn man Garantien abgeben möchte, die die Form haben: „Wenn die Inodetabellendatei die Länge `size` Bytes hat, können auch mindestens `size/isize` Inodes in diesem Dateisystem verwendet werden.“ Ein weiterer Nachteil ergibt sich daraus, daß der Zahlenraum der Inodenummern nicht so dicht besetzt ist wie bei trivialer Numerierung. Dadurch werden bei umlaufender Numerierung im Schnitt höhere Inodenummern verwendet. Die Folge ist, daß Zugriffe auf die Inodetabellendatei bei umlaufender Numerierung eher Zugriffe auf ein– und mehrfach indirekte Blöcke erfordern.

Bei umlaufender Numerierung von Inodeblöcken wird weiterhin vorausgesetzt, daß die Anzahl der Blockgruppen im Dateisystem `ngroups` eine Konstante ist. Es ist also nicht möglich, das Dateisystem durch Hinzufügen von Blockgruppen zur Laufzeit zu vergrößern, ohne die Numerierung der Inodes zu reorganisieren. Eine solche Reorganisation kann, wie in Abschnitt 4.1.5 dargestellt wird, nicht zur Laufzeit erfolgen.

Umlaufende Numerierung arbeitet außerdem nicht gut mit den in Abschnitt 4.1.4 dargestellten Inodebelegungsbitmaps und Summenzählern zusammen: Die Inodebelegungsbitmap bekäme durch umlaufende Numerierung und eine dünn besetzte Inodetabellendatei schnell unnötige Einträge, und Summenzähler pro Block der Inodebitmap sind keine Hilfe beim Auffinden von freien Inodes mit einer Suchschrittweite von `ngroups` Blöcken in der Inodetabellendatei.

Wegen dieser Komplikationen wurde für die weitere Entwicklung die triviale Numerierung der Inodetabellendatei gewählt. Wie in Kapitel 6 gezeigt wird, ist dies ausreichend, um eine Systemleistung zu erzielen, die der des ext2–Dateisystems vergleichbar ist.

### 4.1.4 Auffinden freier Inodes

Um freie Inodes schneller aufzufinden, verwaltet das konventionelle Dateisystem eine Belegungsbitmap. Bei einer festen Anzahl von Inodes hat diese Bitmap eine feste Länge. Für eine variable Anzahl der Inodes ist auch die Bitmap von variabler Länge. Den größten Freiheitsgrad bei der Positionierung dieser Daten erhält man, wenn man für diese Bitmap ebenfalls eine Inode anlegt, die die Inodebelegungsbitmapdatei[5] beschreibt.

Jede Inode, die angelegt wird, benötigt ein Bit in dieser Bitmap. Je 8192 Inodes benötigen also bei einer angenommenen Blockgröße von 1 KB einen Datenblock in dieser Datei. Man kann daher erwarten, daß für diese Datei in den allermeisten Fällen keine indirekten Blockzugriffe oder doch wenigstens nur einfach indirekte Zugriffe notwendig werden.

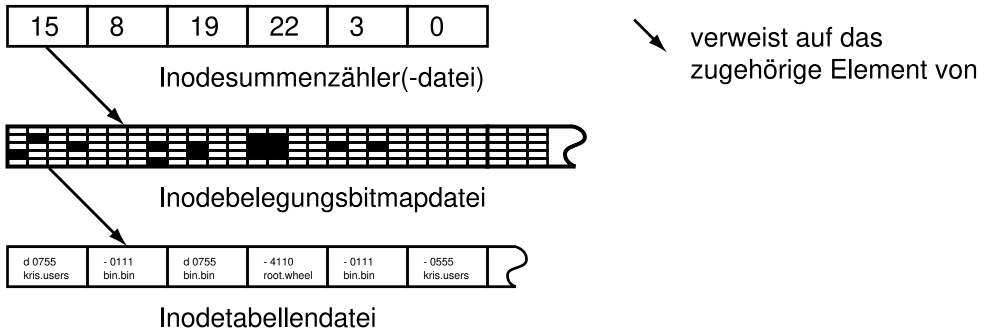



Abbildung 4.5: Zum schnellen Auffinden von freien Inodes wird eine Inodebelegungbitmap geführt. Zum schnellen Auffinden von Bitmapblöcken mit freien Inodes werden Summenzähler geführt. Diese Hierarchie von Verwaltungsinformationen existiert sowohl im konventionellen Dateisystem als auch im Dateisystem mit dynamischer Anzahl von Inodes. Im konventionellen Dateisystem werden diese Datenstrukturen jedoch nicht sauber getrennt.

Eine Inodebelegungsbitmap erleichtert das Auffinden von freien Inodes. Bei Dateisystemen mit vielen Dateien ist dies jedoch nicht ausreichend: Auf der Suche nach freien Inodes muß unter Umständen die gesamte Bitmap untersucht werden, die sehr lang sein kann.

Im herkömmlichen Dateisystem existieren zur Beschleunigung dieses Vorgangs in den Blockgruppendeskriptoren statistische Informationen, die Auskunft darüber geben, ob in einer bestimmten Blockgruppe freie Inodes vorhanden sind. Nur diejenigen Bitmapblöcke von Blockgruppen müssen durchsucht werden, in denen der Summenzähler für die freien Inodes größer

[5] Fortan auch kürzer als „Inodebitmapdatei", „Inodebelegungsbitmap", „Inodebitmap" oder „Bitmap" bezeichnet.

als Null ist. Außerdem wird die Anzahl der freien Inodes als Entscheidungshilfe für die Positionierung der Inodes von neuen Verzeichnissen ausgewertet. Die Speicherung dieser Statistikinformation in den Deskriptoren von Blockgruppen ist durchaus sinnvoll, da die Lage der Inodetabellen an die Blockgruppen gekoppelt ist.

Diese Kopplung zwischen Blockgruppen und Inodetabelle existiert jedoch nicht mehr, wenn man Inodeblöcke mit einer gesonderten Inode verwaltet. Trotzdem wird man die Summeninformationen in irgendeiner Form verwalten müssen, wenn man schnell freie Bits in der Inodebelegungsbitmap auffinden möchte. Tatsächlich wird diese Summeninformation sogar noch bedeutsamer: In einem Dateisystem mit dynamisch wachsender Inodetabelle wird diese immer nur gerade so groß sein wie notwendig. Man kann also davon ausgehen, daß die meisten Bitmapblöcke der Inodebelegungsbitmap keine freien Einträge aufweisen. Um schnell entscheiden zu können, ob überhaupt freie Inodes zu finden sind oder ob die Inodetabelle zu vergrößern ist, sind solche Summeninformationen erforderlich (siehe Abbildung 4.5).

Analog zu konventionellen Dateisystemen wird man einen Summenzähler pro Datenblock der Inodebitmapdatei mitführen. Ein solcher Zähler kann Werte zwischen 0 und 8192 annehmen, belegt also sinnvollerweise 2 Bytes. Mit wachsender Anzahl von Inodes und damit wachsender Anzahl von Bitmapblöcken wächst auch die Anzahl dieser Zähler. Um die Anzahl der Inodes unbegrenzt wachsen lassen zu können, müßte man diese Zähler streng genommen ebenfalls wieder mit einer Inode in einer Datei verwalten. Dieser Aufwand ist in den meisten Fällen jedoch nicht gerechtfertigt: Bei einer angenommenen Blockgröße von 1 KB finden 512 Zähler (entsprechend 512 Blöcken mit Inodebitmaps) in einem Block Platz. Demnach wäre die Anzahl der Inodes in einem Dateisystem auf $512 * 8192 = 2^9 * 2^{13} = 2^{22} = 4194304$ beschränkt, würde man nur einen Block für diese Summenzähler zulassen.

Es ist denkbar, die Summenzähler gar nicht abzuspeichern, sondern beim Anmelden des Dateisystems im Hauptspeicher zu erzeugen und nur für die Dauer der Nutzung des Dateisystems vorzuhalten. Dazu ist es notwendig, die Inodebelegungsbitmap beim Anmelden des Dateisystems komplett durchzulesen und die Anzahl der freien Inodes pro Block zu zählen. Dies verlangsamt den `mount(2)`–Systemaufruf.

Die in Abbildung 4.5 dargestellte Hierarchie von Verwaltungsstrukturen existiert ebenfalls in konventionellen Dateisystemen. Sie ist allerdings versteckt in den Verwaltungsinformationen der Blockgruppen: Die Summenzähler sind Bestandteil der Blockgruppendeskriptoren, und die Bitmapblöcke sind am Beginn einer jeden Blockgruppe abgelegt. In einem Dateisystem mit einer dynamischen Anzahl von Inodes existiert die Bindung an Blockgruppen nicht mehr; daher müssen diese Informationen gesondert in Dateien abgelegt werden.

Umgekehrt lassen sich auch die meisten Strukturen eines ext2–Dateisystems schon als Dateien deuten. Es ist beispielsweise möglich, aus den in den Blockgruppendeskriptoren (siehe Abbildung 3.12, Datenfelder `bg_block_bitmap` und `bg_inode_bitmap`) gespeicherten Blocknummern der Blockbelegungsbitmaps und Inodebelegungsbitmaps jeweils eine Inode für eine Blockbelegungsbitmapdatei und eine Inodebelegungsbitmapdatei zu konstruieren. Beide Dateien haben in ext2 eine feste Länge, und die Lage ihrer Datenblöcke ist vorhersagbar, solange keine Fehler auf dem Medium eine nicht standardgemäße Plazierung der Bitmaps beim Anlegen des Dateisystems erzwingen. Man kann sich diese beiden Datenfelder der Deskriptoren als zwei ineinander verschränkte Inodes zweier Bitmapdateien vorstellen. Die Datenblöcke, die die Deskriptoren ausmachen, können wiederum als eine in jeder Blockgruppe replizierte, zusammenhängende Datei fester Länge angesehen werden. Trennt man Block- und Inodeverwaltung voneinander, muß man auch die Deskriptorendaten auftrennen und auf unterschiedliche Strukturen verteilen.

Die Hierarchie der Verwaltungsstrukturen aus Abbildung 4.5 ist nicht nur auf Inodes anwendbar. Im VIVA–Dateisystem [He93] wird die äquivalente VIVA–Datenstruktur zur Konstruktion einer Datei zur Verwaltung von Fragmenten verwendet, und in der geplanten POSIX.1e–Erweiterung [POSIX.1e] des ext2–Dateisystems ist eine ähnliche Struktur zur Verwaltung der *access control entries*[6] (*ACEs*) vorgesehen. Man kann die Hierarchie von Verwaltungsstrukturen aus Abbildung 4.5 also allgemeiner wie folgt beschreiben:

- Es ist ein Array dynamischer Größe aus Datenstrukturen (Inodes, ACEs, Dateifragmente) zu verwalten, die einen ganzzahligen Bruchteil der Blockgröße des Dateisystems belegen. Das Array wird in Form einer Datei angelegt.
- Die Datei zur Speicherung dieses Arrays kann belegte und freie Speicherplätze von der Größe der zur verwaltenden Datenstruktur enthalten. Zum Auffinden freier Speicherplätze wird eine Bitmap in einer weiteren Datei angelegt.
- Wenn die Größe der Bitmap anwächst, wird zur Beschleunigung der Suchprozesse eine Reihe von Summenzählern angelegt, die Informationen über die Anzahl freier Speicherplätze für jeweils eine Gruppe von Bits liefert.

[6]Ein ACE ist ein atomarer Bestandteil einer *access control list* (*ACL*). Er legt Rechte für einen einzelnen Benutzer oder eine einzelne Benutzergruppe fest.

### 4.1.5 Freigabe von Inodes

In einem Dateisystem mit dynamisch wachsender Inodetabelle kann der Fall eintreten, daß sehr viele Dateien gelöscht werden und dementsprechend viele Inodes freigegeben werden. Es ist auf Anhieb klar, daß lediglich partiell freie Inodeblöcke nicht freigegeben werden können: Solange auch nur eine Inode in einem Inodeblock belegt ist, muß dieser Inodeblock als solcher weitergeführt werden.

Es ist theoretisch denkbar, die Inodetabelle zu komprimieren, d.h. leere Inodes in der Inodetabellendatei mit belegten Inodes vom Ende der Inodetabellendatei zu vertauschen. Auf diese Weise würden sich leere Inodes am Ende der Inodetabellendatei sammeln. Dort könnten sie freigegeben werden könnten, um die Inodetabellendatei zu verkürzen.

In der Praxis ist dies jedoch eine sehr aufwendige Operation, da die relative Position einer Inode in der Inodetabellendatei direkt mit der Inodenummer zusammenhängt. Diese Inodenummer wird aber in den Verzeichniseinträgen für eine Datei benutzt. Wird eine Inode innerhalb der Inodetabellendatei verschoben, müßten auch alle Vorkommen ihrer Inodenummer in allen Verzeichnissen gefunden und angepaßt werden. Während es vergleichsweise einfach ist, vom Pfadnamen einer Datei auf die zugehörige Inodenummer zu kommen (Algorithmus `namei()`), ist die umgekehrte Operation erheblich aufwendiger. Es existiert in einem UNIX–Dateisystem keine Datenstruktur, von der man von einer Inode ausgehend effizient auf alle Pfadnamen einer Datei schließen könnte.

Programme wie `ncheck(8)` oder `fsck(8)`, die diese Operation durchführen können, müssen dazu rekursiv die gesamte Verzeichnisstruktur eines Dateisystems lesen und invertieren. Die Programmlaufzeiten liegen dabei abhängig von der Größe des untersuchten Dateisystems typischerweise im Minutenbereich. Wenn sich während dieser Zeit die Verzeichnisstruktur des Dateisystems verändert, d.h. wenn in dieser Zeit Dateien erzeugt oder gelöscht werden, können die gelesenen Daten inkonsistent sein, und die Ergebnisse dieser Programme sind ungültig. Wenn man also eine Operation zur Bereinigung der Inodetabellendatei vorsieht, kann diese sicherlich nur auf einem abgemeldeten Dateisystem durchgeführt werden.

Es bereitet jedoch keine Probleme, vollständig ungenutzte Inodeblöcke freizugeben und den entsprechenden Datenblockzeiger in der Inode der Inodetabellendatei auf Null zu setzen. Der betreffende Inodeblock braucht dazu nicht am Ende der Inodetabellendatei stehen. Die Inodetabellendatei wäre in diesem Fall zwar nicht verkürzt, würde als dünn besetzte Datei allerdings weniger Platz belegen[7]. Diese Art der Speicherfreigabe ist aber weniger effizient als die oben vorgeschlagene Bereinigung der Inodetabellendatei: Während das Bereinigungsverfahren maximal viele vollständig ungenutzte Inodeblöcke erzeugt, werden im zweiten Verfahren keine Anstrengungen zur Sortierung von freien und belegten Inodes unternommen. Es bleibt also der Dynamik der Benutzung überlassen, ob vollständig freie Inodeblöcke entstehen, die freigegeben werden können.

Es ist leicht möglich, einen pathologischen Fall zu konstruieren, in dem die Inodetabellendatei zunächst auf eine gewünschte Zielgröße gebracht wird und dann von jeweils acht Inodes sieben gelöscht werden[8]. Das Resultat ist eine Inodetabelle, in der jede achte Inode belegt ist und daher

[7] Mit den am Ende von 4.1.3 diskutierten Nachteilen für dünn besetzte Dateien.

[8] Unter der Annahme, daß bei einer Blockgröße von 1024 Bytes acht Inodes pro Block plaziert werden können.

kein Block der Inodetabelle freigegeben werden kann, die Inodetabelle also achtmal größer als benötigt ist.

Im normalen Betrieb reorganisiert das System die Inodetabelle jedoch selbst, da neue Inodes immer den Platz mit der niedrigsten freien Inodenummer belegen. Dadurch ist die Inodetabellendatei im vorderen Teil meistens dicht besetzt, während im hinteren Teil überwiegend freigegebene Blöcke zu finden sind.

### 4.1.6 Selbstbezüglichkeit

Die Definition der Inodetabelle durch eine Inode dieser Tabelle ist selbstbezüglich in dem Sinne, daß man auch zum Auffinden dieser Inode in der Inodetabelle nachsehen müßte. Zur Auflösung dieser Selbstbezüglichkeit gibt es zwei mögliche Ansätze:

- Man speichert die Adresse des Datenblocks, in dem die spezielle Inode innerhalb der Inodetabelle enthalten ist, an einer definierten Stelle außerhalb der Inode und Inodetabelle. Es ist zum Beispiel möglich, die Adresse dieses Datenblocks im Superblock zu hinterlegen.
- Man verwendet zwar eine Inodestruktur zur Verwaltung der Inodetabelle, aber speichert diese Struktur nicht in der Inodetabelle, sondern an einem bekannten Ort. Es ist zum Beispiel möglich, die Inode der Inodetabelle im Superblock zu hinterlegen.

Verwendet man das erstgenannte Verfahren, kann die Inodetabelle nie leer sein. Sie muß mindestens die Inode zur Beschreibung der Inodetabelle enthalten. Diese Beobachtung ist jedoch ohne Bedeutung, da eine Inodetabelle ohnehin nicht leer sein kann: Außer der Inode zur Beschreibung der Inodetabelle enthält sie in jedem Fall weitere Inodes mit spezieller Bedeutung, etwa die Inode zur Beschreibung des Wurzelverzeichnisses. Das ext2–Dateisystem kommt sogar auf ein gutes Dutzend Inodes mit besonderer Bedeutung.

Verwendet man das zweite Verfahren, ist eine Erweiterung des Dateisystemkonzeptes denkbar, die in [OSF92] beschrieben wird: Das „Episode"–Dateisystem der Open Software Foundation ermöglicht es, für eine Festplattenpartition eine gemeinsame Blockbelegungsbitmap zur Freispeicherverwaltung zu haben, aber mehrere Inodes von Inodetabellendateien. So ist es möglich, daß sich mehrere Dateisysteme den freien Speicher einer Partition untereinander aufteilen. Episode läßt auch das *cloning* von Inodetabellen zu. Bei diesem Verfahren wird die Inodetabellendatei dupliziert. Fortan wird auf alle direkten Datenblöcke und alle indirekten Blöcke zweifach verwiesen, jedoch markiert Episode alle Diskadressen in der originalen Inodetabellendatei und in der Kopie als *copy on write* (*COW*). Bei verändernden Zugriffen auf einzelne, nicht duplizierte Blöcke wird der entsprechende Block kopiert, und die COW–Markierungen werden entfernt. Mit Hilfe dieser Technik ist es möglich, Schnappschüsse von Dateisystemen zu machen, die dann an anderer Stelle im Namensraum angemeldet und z.B. ohne Zeitdruck gesichert werden können. Die Sicherungskopie ist garantiert konsistent, da sie einen Schnappschuß zu einem bestimmten Zeitpunkt darstellt. Trotzdem ist das originale Dateisystem während der Zeit, die die Datensicherung benötigt, weiterhin für Benutzer verfügbar. Kapitel 7 geht näher auf Episode und andere weiterentwickelte Dateisysteme ein.

## 4.2 Modell des herkömmlichen Dateisystems

Im Folgenden soll ein Modell des ext2–Dateisystems gebildet werden, indem die Mappingfunktionen des Dateisystems angegeben werden. Kennt man diese Funktionen, kann man zu jeder Komponente[9] des Dateisystems die absolute Blocknummer bestimmen, an der diese Komponente gespeichert wird.

Die Umkehrung aller Mappings, die jeder absoluten Blocknummer eine Komponente zuordnet, wird nicht angegeben, da sie von einem Dateisystem normalerweise nicht verwendet wird. Statt dessen wird in real existierenden Dateisystemen im Normalfall mit einem externen Programm (`fsck`) sichergestellt, daß die Umkehrung aller Mappings auch wieder eine Funktion ist (d.h. jeder absoluten Blocknummer wird höchstens eine Komponente zugeordnet) und daß diese Funktion total ist (d.h. jeder absoluten Blocknummer wird eine Komponente zugeordnet).

Die Notation der verwendeten Datenstrukturen orientiert sich an [Ja90] und [Bau84]. Es werden die folgenden Konstruktionen verwendet, die hier nur zur Erinnerung noch einmal vorgestellt, aber nicht formal eingeführt werden sollen:

```
mode zusammengesetzter_typ = (
    typ1 name1,
    typ2 name2,
    ...);
mode aufzaehlungstyp = { eins, zwei, drei };
mode einschraenkungstyp = {0 .. 9};
```

`mode` definiert analog zum `struct` in C einen neuen Datentyp `zusammengesetzter_typ`, der sich aus den aufgelisteten einzelnen Feldern `name1`, `name2`, `...` zusammensetzt. Die einzelnen Felder sind jeweils vom angegebenen Typ `typ1`, `typ2`, `...`. Die Auswahl von Komponenten eines zusammengesetzten Typs zur Verwendung in Ausdrücken erfolgt wie in C oder Pascal über den Punktoperator: `t.name1`.

Analog zu Pascal werden Aufzählungstypen durch Aufzählung aller möglichen Werte definiert, und Einschränkungstypen sind Einschränkungen des Ganzzahltyps `int` auf einen zusammenhängenden Teilbereich.

Funktionsdefinitionen werden in der folgenden Form notiert:

```
func name = (typ1 name1, typ2 name2, ...) ergebnistyp:
    rumpf;
```

[9] Eine Komponente eines Dateisystems ist eine Datenstruktur wie Superblock, Deskriptortabelle, Inodetabelle, Datei usw.

Dies definiert eine Funktion `name`, die die Definitionsmenge {`typ1`×`typ2`×`...`} auf die Bildmenge `ergebnistyp` abbildet und deren Berechnungsvorschrift in `rumpf` gegeben ist. Ist der Ergebnistyp einer Funktion als `ergebnistyp`|`e:` notiert, so ist damit die Erweiterung der Ergebnismenge um ein ausgezeichnetes Element zur Signalisierung von Fehlerbedingungen gemeint.

Funktionen können mehr als ein Ergebnis liefern. Ist dies der Fall, dann ist der Ergebnistyp der Funktion von der Form `paar(typ1, typ2)`, falls die Funktion ein Paar von Funktionsergebnissen liefert oder von der Form `tupel(typ1, typ2, ...)`, falls die Funktion ein mehrstelliges Ergebnis liefert.

Mit dem Operator `let typ name = ausdruck` wird das Ergebnis von `ausdruck` an den Namen `name` vom Typ `typ` gebunden.

Als einzige Kontrollstrukturen werden in diesem Text Funktionsaufruf und `if bedingung then wahr-zweig else falsch-zweig fi` sowie zur Bequemlichkeit das verkürzte `if` (nach Bedarf ohne `then` oder ohne `else`) verwendet.

Das Modell wird an einigen Stellen von einem Mechanismus Gebrauch machen, der es erlaubt, das Ergebnis eines Funktionsaufrufes im Typ zu wandeln. Ein solcher Mechanismus kann entweder ein *casting*-Mechanismus wie in C sein, oder es müssen wie in Pascal explizite Typwandlungsfunktionen für jede vorkommende Typwandlung vorausgesetzt werden. Zur Notation der Typumwandlung wird die Syntax von C verwendet.

### 4.2.1 Konstanten und elementare Funktionen

Ein ext2–Dateisystem wird durch einige Konstanten charakterisiert, die der Systemverwalter beim Anlegen des Dateisystems bestimmen kann. Diese Konstanten sind die Größe des Dateisystems in Blöcken (`NBLOCKS`), die Größe eines Blocks in Bytes (`BSIZE`) und für jeweils wie viele Kilobytes Dateisystemgröße eine Inode angelegt werden soll (`IRATIO`). Die Blockgröße ist dabei nicht frei wählbar, sondern es muß sich um eine der von ext2 unterstützten Blockgrößen handeln: 1024, 2048 oder 4096 Bytes pro Block. Zusätzlich kann am Anfang einer jeden Blockgruppe noch Platz für einen Urlader freigehalten werden, so daß das Dateisystem erst an der Stelle `FIRSTBLOCK` beginnt.

Ein `offset` bezeichnet die Byteposition, auf der eine Datenstruktur in einem Block beginnt. Mappingfunktionen, die Objekte bezeichnen, die kleiner als ein Block sind, liefern statt einer absoluten Blocknummer ein Paar bestehend aus einer absoluten Blocknummer und einem Offset zurück.

```
mode bsize = { 1024, 2048, 4096 };

let int   NBLOCKS    = <userinput>, // Dateisystems in Blocks.
    bsize BSIZE      = <userinput>, // Blocks in Bytes.
    int   IRATIO     = <userinput>, // Je IRATIO Blöcke eine Inode.
    int   FIRSTBLOCK = <userinput>; // Beginn des Dateisystems.
```

```
mode abs_block# = {0 .. NBLOCKS-1};
mode offset     = {0 .. BSIZE-1};
```

Ein Datenblock auf dem Medium wird durch ein Bytefeld von `BSIZE` Elementen modelliert. Das ganze Medium (eine Partition einer Festplatte) wird als Feld von `NBLOCKS` Blöcken dargestellt.

```
mode block = byte[offset];
mode disk  = block[abs_block#];
```

Das Dateisystem arbeitet weiterhin mit einigen Konstanten, die sich aus den Größen der verwendeten Datenstrukturen ergeben: die Größe einer Inode in Bytes (`ISIZE`), die Größe eines Blockgruppendeskriptors in Bytes (`DSIZE`) und die Größe einer Blockadresse in Bytes (`ASIZE`).

```
let int ISIZE = 128,  // Größe einer Inode in Bytes.
    int DSIZE = 32,   // Größe eines Deskriptors in Bytes.
    int ASIZE = 4;    // Größe einer Blockadresse in Bytes.
```

Aus diesen primitiven Konstanten lassen sich weitere Konstanten ableiten, die die Notation vieler Ausdrücke verkürzen: die Anzahl der Bits in einem Block (`BITS_PER_B`), die Anzahl der Inodes in einem Block (`I_PER_B`), die Anzahl der Deskriptoren in einem Block (`D_PER_B`) und die Anzahl der Adressen in einem Block (`A_PER_B`).

```
let int BITS_PER_B = BSIZE*8,      // Bits pro Block.
    int I_PER_B    = BSIZE/ISIZE, // Inodes pro Block.
    int D_PER_B    = BSIZE/DSIZE, // Deskriptoren pro Block.
    int A_PER_B    = BSIZE/ASIZE; // Adressen in pro Block.
```

Das Modell wird den Buffercache von UNIX nicht mitmodellieren, da wir uns nur für die Anordnung von Datenstrukturen im Dateisystem interessieren. Mit Hilfe einer Funktion `bread` wird ein Block einer Partition zurückgeliefert, mit Hilfe von `bwrite` wird eine Partition durch Ersetzen eines Blocks in eine neue Partition überführt[10].

```
func bread(abs_block# b, disk d) block:
    d[b];

func bwrite(abs_block# b, block bl, disk d) disk:
    alt(d, b, bl);
```

[10] Der Operator `alt(feld, pos, el)` ist in der verwendeten Notation vordefiniert und ersetzt im Feld `feld` das an der Position `pos` vorhandene Element durch `el`.

### 4.2.2 Blockgruppen

Wegen der Beschränkung der Größe einer Blockbelegungsbitmap einer Blockgruppe auf einen Datenblock kann eine Blockgruppe maximal `BGSIZE` Blöcke umfassen. Daraus ergibt sich dann die Anzahl der Blockgruppen `NGROUPS`. Aus der Anzahl der Blöcke und der Inoderatio errechnet sich die gewünschte Anzahl von Inodes für das Dateisystem und mit der Anzahl der Blockgruppen die Anzahl an Inodes und Inodeblöcken pro Blockgruppe (`I_PER_G`, `IB_PER_G`). Da alle Blockgruppen gleich viele Inodes enthalten sollen, kann die tatsächliche Anzahl von Inodes (`NINODES`) etwas höher liegen als die gewünschte Anzahl von Inodes.

```
// Die Blockgruppe muß durch einen Bitmapblock
// beschrieben werden können.
let int BGSIZE  = BSIZE * BITS_PER_B;
mode bitnr = {0 .. BGSIZE-1};

// Bestimme die Anzahl der Inodes pro Blockgruppe:
// Bestimme erst die gewünschten Inodes per Blockgruppe
let int WINODES  = NBLOCKS/IRATIO,
    int WI_PER_G = int(WINODES/(int(NBLOCKS/BGSIZE)+1)),
// und daraus dann die tatsächlichen Inodes per Blockgruppe.
    int IB_PER_G = int(WI_PER_G/I_PER_B),
    int I_PER_G  = IB_PER_G * I_PER_B;
```

Wenn die Anzahl der Blöcke im Dateisystem kein ganzzahliges Vielfaches der maximalen Blockgruppengröße ist, hat die letzte Blockgruppe des Dateisystems nicht die volle Größe, sondern umfaßt nur `SBGSIZE` Blöcke ($50 < \mathtt{SBGSIZE} \leq \mathtt{BGSIZE}$)[11]. Wenn die letzte Blockgruppe des Dateisystems zu klein ist, um die Verwaltungsinformationen einer Blockgruppe aufzunehmen und Platz für einige Datenblöcke zu bieten, wird sie gänzlich verworfen.

[11] Die Konstante 50 ist von den Autoren des ext2–Dateisystems willkürlich gewählt.

```
// Verwaltungsoverhead pro Blockgruppe: Deskriptoren und
// Superblock + Inodebitmapblock + Blockbitmapblock.
let int GBLOCKS    = int((int(NBLOCKS/BGSIZE)+1)/D_PER_B),
    int G_OVERHEAD = GBLOCKS+FIRSTBLOCK+3;

// Jetzt kann das Dateisystem in Gruppen
// der Größe BGSIZE unterteilt werden.
let int NGROUPS = int(NBLOCKS/BGSIZE) +
    if (NBLOCKS mod BGSIZE) > G_OVERHEAD+50 then
        1; // Fall 1: Blockgruppe groß genug.
    else
        0; // Fall 2: Blockgruppe zu klein.
    fi,

// Die letzte Blockgruppe des Dateisystems ist
// möglicherweise kleiner als die anderen Blockgruppen.
let int SBGSIZE =
    if (NBLOCKS mod BGSIZE) > G_OVERHEAD+50 then
        NBLOCKS mod BGSIZE; // Fall 1.
    else
        BGSIZE;             // Fall 2.
    fi;

// Tatsächliche Anzahl der Inodes.
let int NINODES = I_PER_G * NGROUPS;
```

Damit kann man nun endlich eine Hilfsfunktion `gmap` angeben, die die Anfangsposition der einzelnen Blockgruppen festlegt. Anders als die Mappings der folgenden Funktionen definiert `gmap` keine Komponente des Dateisystems, d.h. es ist kein Fehler, wenn sich das Mapping für eine Komponente und eine Blockgruppe überlappen.

Um die Funktion `gmap` definieren zu können, müssen zunächst die Datentypen für Definitions– und Wertemenge der Abbildung definiert werden. Die Funktion bildet eine Blockgruppennummer auf eine absolute Blocknummer ab.

```
mode gruppen#   = {0 .. NGROUPS-1};

func gmap = (gruppen# g) abs_block#:
    g * BGSIZE;
```

### 4.2.3 Superblöcke

In diesem und den folgenden Abschnitten werden nacheinander die Komponenten des Dateisystems modelliert. Dazu wird jeweils eine Mappingfunktion angegeben, die die Position der Komponente im Dateisystem festlegt. Zusätzlich werden *Accessorfunktionen* zum Lesen und Schreiben dieser Komponente angegeben.

In jeder Blockgruppe wird im ersten freien Block eine Kopie des Superblocks installiert. Die Funktion `smap` ist das (triviale) Mapping für diese Superblöcke.

```
func smap = (gruppen# s) abs_block#:
    gmap(s)+FIRSTBLOCK;
```

Das Dateisystem selbst greift immer nur auf den Superblock an der Position `smap(0)` zu. Die anderen Kopien des Superblocks werden bei der Erzeugung des Dateisystems durch Usermode–Werkzeuge angelegt und danach nie mehr verändert. Wird das Dateisystem beschädigt, verwendet das Usermode–Programm `fsck` gegebenenfalls Informationen über die Größe des Dateisystems und der Inodetabelle aus Kopien des Superblocks. Zur Reparatur eines Dateisystems ist es jedoch nicht notwendig, aktuelle statistische Informationen über das Dateisystem zu haben, denn diese Informationen werden durch den `fsck`–Lauf in jedem Fall vollständigt regeneriert. Daher brauchen die Kopien des Superblocks auch nicht auf Stand gebraucht zu werden.

Die hier definierten Accessorfunktionen greifen implizit auf den Superblock 0 zu.

```
mode super_block = (
...
);

func sread(disk p) super_block:
    (super_block) bread(smap(0), p);

func swrite(super_block s, disk p) disk:
    (super_block) bwrite(smap(0), (block) s, p);
```

### 4.2.4 Deskriptoren und Bitmaps

Die Deskriptoren der Blockgruppen sind unmittelbar hinter dem Superblock gespeichert und belegen `int((NGROUPS+D_PER_B-1)/D_PER_B)` Blöcke. Das ext2–Dateisystem dupliziert diese Desriptoren ebenso wie Superblöcke, d.h. nur an der Stelle `dmap(0, g)` finden sich aktuelle Versionen der Deskriptoren. Die anderen Kopien werden bei der Erzeugung des Dateisystems geschrieben und nie verändert. Sie dienen nur Speicherung der Blockadressen der Inode– und Blockbelegungsbitmaps und werden von `fsck` zur Rekonstruktion verwendet.

```
func dmap(gruppen# g, gruppen# d) paar (abs_block#, offset):
    (smap(g)+1+int(d/D_PER_B), (d mod D_PER_B) * DSIZE);
```

Ein Deskriptor für eine Blockgruppe enthält Zeiger auf die andere Komponenten dieser Blockgruppe sowie einige statistische Informationen über die Blockgruppe. Die hier gezeigte Definition entspricht Abbildung 3.12.

```
mode ext2_group_desc = (
    abs_block# bg_block_bitmap,
    abs_block# bg_inode_bitmap,
    abs_block# bg_inode_table,
    int        bg_free_blocks_count,
    int        bg_free_inodes_count,
    int        bg_used_dirs_count);
```

Mit Hilfe der Accessorfunktionen `dread` und `dwrite` können Deskriptoren gelesen und beschrieben werden. Es wird implizit auf die Deskriptoren an der Position `dmap(0, x)` zugegriffen.

```
mode dbnr = {0 .. D_PER_B-1};
mode desc_block = ext2_group_desc[dbnr];

func dread(gruppen# g, disk p) ext2_group_desc:
    let (abs_block# b, offset o) = dmap(0, g),
         dnbr d_in_b             = o/DSIZE;

    ((desc_block) bread(b, p))[d_in_b];

func dwrite(gruppen# g, ext2_group_desc d, disk p) disk:
    let (abs_block# b, offset o) = dmap(0, g),
        dnbr d_in_b              = o/DSIZE,
        desc_block db            = (desc_block) bread(b, p),
        desc_block new_db        = alt(desc_block, d_in_b, d);

    bwrite(b, (block) new_db, p);
```

Diese Definitionen zeigen ein Muster, das sich im folgenden einige Male wiederholen wird: Ein Datenblock enthält eine ganzzahlige Anzahl von Elementen vom Typ `x`, auf die zugegriffen werden soll. Über den Bereichstyp `xnr` lassen sich diese Elemente aufzählen. Der Typ `x_block`, der einen als kleines Feld dieser Elemente interpretierten Datenblock darstellt, ist die Einheit, über die Medienzugriffe stattfinden. Die Accessorfunktionen bestimmen mit Hilfe der Mappingfunktion den Datenblock und Offset, in dem das gesuchte Element enthalten ist. Über `bread` wird der gewünschte Datenblock gelesen und als kleines Feld von Elementen des Zieltyps interpretiert. Für einen Lesezugriff wird nur das Element am Zieloffset zurückgeliefert. Bei Schreibzugriffen wird das Element am Zieloffset ausgetauscht und der gesamte Block geschrieben.

Die Angabe von Mappings für die Belegungsbitmaps der Blöcke und Inodes einer Blockgruppe ist mit Hilfe der Funktion `dread` trivial:

```
func bbmap(gruppen# g, disk p) abs_block#:
    dread(g, p).bg_block_bitmap;

func ibmap(gruppen# g, disk p) abs_block#:
    dread(g, p).bg_inode_bitmap;
```

Der Bereichstyp `bitnr` ist bereits weiter oben definiert.

```
mode bitmap_block = bit[bitnr];
```

Für Bitfelder werden die Funktionen `test_bit(pos, bitfeld) bit:`, `set_bit(pos, bitfeld) bitfeld:`, `clear_bit(pos, bitfeld) bitfeld:` sowie `ffz(pos, bitfeld) pos:` [12] ohne Erklärung vorausgesetzt.

Die Funktion `alloc_bit` findet eine freie Position in der Bitmap und belegt sie. Die Freigabe von Bits kann direkt mit `clear_bit` erfolgen.

```
func alloc_bit(abs_block# start, bitmap_block bm)
    paar(abs_block#, bitmap_block):

    let abs_block# pos      = ffz(start, bm),
        bitmap_block new_bm = set_bit(pos, bm);
    (pos, new_bm);
```

### 4.2.5 Inodetabelle

Eine Inodenummer ist vom Typ `inode#`:

```
mode  inode#     = {0 .. NINODES-1};
```

Die Funktion `inmap` ähnelt stark den Funktionen `ibmap` und `bbmap`.

```
func inmap(gruppen# g, disk p) abs_block#:
     dread(g, p).bg_inode_table;
```

Kennt man den Beginn der Inodetabelle in einer Blockgruppe, kann man die Funktion `imap` angeben, die das Mapping für die Inodes darstellt. Ähnlich wie `dmap` adressiert sie Datenstrukturen innerhalb von Blöcken und liefert deswegen als Ergebnistyp ein Paar bestehend aus einer absoluten Blocknummer und einem Offset in diesen Block.

[12] *find first zero* (`ffz`) liefert die Position des ersten Nullbits nach der Position `pos` im Bitfeld. Die vorderste Position des Bitfeldes hat den Index 0.

```
func imap(inode# i, disk p) paar(abs_block#, offset):
    let gruppen#   g       = int(i/I_PER_G),
        inode#     i_in_g  = i mod I_PER_G,
        abs_block# istart  = inmap(g, p);
    (istart+i_in_g, i_in_g*ISIZE)
```

Das folgende Modell einer `ext2_inode` entspricht nicht ganz genau dem Aufbau einer Inode im ext2–Dateisystem (siehe Abbildung 3.14): Derzeit unbenutzte Felder sind weggelassen, andere Felder sind in den erweiterten Möglichkeiten der gewählten Notation angepaßt. Für die Zwecke von `bmap` ist es ausreichend, auf das Feld `i_block` zugreifen zu können, aber die in 4.2.10 besprochenen Integritätsbedingungen benötigen zum Teil detailliertere Informationen über die Inode, so daß sie hier mitmodelliert werden.

```
mode ftype  = { fifo, chr, dir, blk, reg, lnk, sock };
mode perms  = bit[12];
mode time_t = ulong;

let EXT2_N_BLOCKS = 15;

mode ext2_inode = (
    ftype  i_type,  // Typ der Datei
    perms  i_perms, // Zugriffsrechte sstrwxrwxrwx
    ushort i_uid,   // Eigentümer
    ulong  i_size,  // Größe der Datei in Byte
    time_t i_atime, // Letzter Lesezugriff auf Datenblock
    time_t i_mtime, // Letzter Schreibzugriff -"-
    time_t i_ctime, // Letzter Schreibzugriff auf Inode
    time_t i_dtime, // Löschzeitpunkt
    ushort i_gid,   // Gruppenzugehörigkeit
    ushort i_links_count,
    ulong  i_blocks,// Anzahl der zugewiesenen Blöcke
    bit    i_flags[32],
gruppen# i_block_group,

    abs_block# i_block[0 .. EXT2_N_BLOCKS-1]
);
```

Kennt man die Position der Inode auf der Festplatte, kann diese geladen werden. Dies erfolgt mit Hilfe der Modellfunktion `iread`, die eine `inode#` auf eine `ext2_inode` abbildet. Die wirkliche Kernelfunktion `iget` ist komplizierter, denn sie arbeitet mit einer anderen Darstellungform für Inodes (*in-memory* Inodes) und verwendet einen Inodecache.

```
mode inr = {0 .. I_PER_B-1};
mode inode_block = ext2_inode[inr];

func iread(inode# i, disk p) ext2_inode:
```

```
    let (abs_block# b, offset o) = imap(i, p),
        inr i_in_b = o/ISIZE;

    ((inode_block) bread(b, p)[i_in_b];

func iwrite(inode# i, ext2_inode in, diskp) disk:
    let (abs_block# b, offset o) = imap(i, p),
        inr i_in_b         = o/ISIZE,
        inode_block ib     = (inode_block) bread(b, p),
        inode_block new_ib = alt(inode_block, i_in_b, in);

    bwrite(b, (block) new_ib, p);
```

### 4.2.6 Dateien

Der Wertebereich einer relativen Blocknummer entspricht (nahezu) dem einer absoluten Blocknummer, denn es muß möglich sein, mit einer Datei die gesamte Platte zu füllen. Liegt die relative Blocknummer für eine konkrete Inode `i` außerhalb von {`0 .. i.i_blocks-1`}, wächst die betreffende Datei.

Ein *indirect block* von beliebiger Ebene ist ein Feld von absoluten Blocknummern.

```
mode rel_block# = abs_block#;
mode ind_block  = abs_block#[A_PER_B];
```

Die Funktion `bmap` bildet mit Hilfe der Inode eine relative Blocknummer auf eine absolute Blocknummer ab. Der hier angegebene Code entspricht fast genau dem Code des ext2–Dateisystems mit Ausnahme der Speicherverwaltung des Buffer Cache für die geladenen indirekten Blöcke, die hier nicht mitmodelliert wird. `bmap` braucht lediglich festzustellen, über welche Indirektionsebene ein relativer Block angesprochen werden muß, und dann nacheinander diese Indirektionsebenen aufzulösen. Die einzige zusätzliche Komplikation, die dabei auftreten kann, besteht darin, daß an der angeforderten relativen Blocknummer noch kein Datenblock alloziert worden ist. Dies kann nicht nur auf der Datenblockebene geschehen, sondern es ist genausogut möglich, daß auch Indirekt–Blöcke noch nicht bereitgestellt wurden. In jedem dieser Fälle liefert `bmap` das Mapping `0` zurück.

```
func bmap = (ext2_inode i, rel_block# r, disk p) abs_block#:
  if (r < 12) then
    // direct block
    i.i_block[r];
  else
    // single indirect
    let abs_block# r1 = r-12;
    if (r1 < A_PER_B) then
      let abs_block# d1 = i.i_block[12];
```

```
      if (d1 == 0) then 0; else
        ((ind_block) bread(d1, p))[r1];
      fi
    else
      // double indirect
      let abs_block# r2 = r1-A_PER_B;
      if (r2 < A_PER_B^2) then
        let abs_block# d2 = i.i_block[13];
        if (d2 == 0) then 0; else
          let abs_block# d21 =
            ((ind_block) bread(d2, p)[int(r2/A_PER_B)];
          if (d21 == 0) then 0; else
            ((ind_block) bread(d21, p))[r2 mod A_PER_B];
          fi
        fi
      else
        // triple indirect
        let abs_block# r3 = r2 - A_PER_B^2,
            abs_block# d3 = i.i_block[14];
        if (d3 == 0) then 0; else
          let abs_block# d32 =
            ((ind_block) bread(d3, p)[int(r3/A_PER_B^2)];
          if (d32 == 0) then 0; else
            let abs_block# d31 =
              ((ind_block) bread(d32, p))[int(r3/A_PER_B)
                                          mod A_PER_B];
            if (d31 == 0) then 0; else
              ((ind_block) bread(d31, p))[r3 mod A_PER_B];
            fi
          fi
        fi
      fi  // of 3-ind
    fi  // of 2-ind
  fi // of 1-ind
```

Man erkennt leicht, daß `bmap` Regelmäßigkeiten enthält. Zieht man diese Regelmäßigkeiten heraus, bekommt man eine übersichtlichere Funktion `bmap`, die zusätzlich noch mit beliebig vielen Indirektionsebenen arbeiten kann. Die Teilfunktion `indirection` bestimmt per Endrekursion, welche Indirektionsebene für eine gegebene relative Blocknummer `r` notwendig ist. Die Teilfunktion `absteigen` führt dann den rekursiven Abstieg über die ermittelte Anzahl von indirekten Blöcken durch.

```
// Steige eine Kette von j Indirektionsebenen herab.
func absteigen(disk p, int j,
    abs_block# r, abs_block# d) abs_block#:

    // Block nicht vorhanden.
```

```
    if (d == 0) then 0;
    else
        // Gewünschtes A_PER_B Segment ausschneiden.
        let int index = int(r/A_PER_B^(j-1)) mod A_PER_B;
        let abs_block# next =
            ((ind_block) bread(d, p))[index];
        // Unterste Indirektionsebene erreicht.
        if (j == 1) then
            next;
        else
            // Tiefer absteigen.
            absteigen(p, j-1, r, next);
        fi
    fi

// Ermittlung der benötigten Indirektionsebene.
func indirektion(disk p, int n,
    abs_block# ro, ext2_inode i) abs_block#:

    // Normalisiere die relative Blocknummer.
    let abs_block# rn = ro - A_PER_B^(n-1);

    // Liegt der gesuchte Block im Zielbereich?
    if (rn < A_PER_B^n) then
        // Ja, dann absteigen.
        absteigen(p, n, rn, i.i_block[11+n];
    else
        // Nein, dann I-Ebene erhöhen.
        indirektion(p, n+1, rn, i);
    fi;

// bmap ist nun trivial.
func bmap(ext2_inode i, rel_block# r, disk p) abs_block#:
    if (r < 12) then
        i.i_block[r]; // Sonderfall
    else
        // r-11: Korrektur für Sonderfall
        indirektion(disk p, 1, r-11, ext2_inode i);
    fi;
```

Der Aufruf von `indirektion` in `bmap` erfolgt nicht mit `r`, sondern mit `r-11`. Bei der Ermittlung von `rn` wird zusätzlich $\texttt{A\_PER\_B}^0$ abgezogen, so daß `rn` insgesamt um `12` (die Anzahl der direkt über die Inode ansprechbaren Blöcke) gegenüber `ro` vermindert ist, d.h. die relative Koordinate des ersten über den single indirect block angesprochenen Datenblocks ist wie für die Rechnung erforderlich `0`.

Durch geeignete Modifikation der Funktionen `absteigen` und `indirektion` kann man Mappingfunktionen für die indirekten Blöcke einer Inode konstruieren, die hier aber nicht ausdrücklich angegeben werden, da sie bis auf die Rekursionstiefe mit `bmap()` identisch sind.

### 4.2.7 Allozierung und Freigabe von Inodes

Wenn eine neue Datei angelegt werden soll, muß das Dateisystem eine Inode für diese Datei aus dem Pool der freien Inodes allozieren. Dies erfolgt mit dem Algorithmus `ialloc`, der im realen ext2–Dateisystem in der Funktion `ext2_new_inode` realisiert ist. Die Funktion ermittelt zunächst die für den Dateityp der neuen Inode optimale Blockgruppe und alloziert dann eine Inode aus dieser Blockgruppe. Die Belegung der Inode erfolgt durch die Modifikation der Inodebelegungsbitmap für diese Blockgruppe und macht außerdem die Anpassung des Deskriptors für diese Blockgruppe notwendig. Auch der Zähler für die freien Inodes im Superblock muß nachgeführt werden.

Die Hilfsfunktion `ialloc_desc` wird in `ialloc` verwendet und erzeugt abhängig vom Dateityp aus einem gegebenen Deskriptor einen neuen Deskriptor, in dem die Zähler für freie Inodes und belegte Verzeichnisse angepaßt sind:

```
func ialloc_desc(ftype t, ext2_group_desc d) ext2_group_desc:
    ( d.bg_block_bitmap,
      d.bg_inode_bitmap,
      d.bg_inode_table,
      d.bg_free_blocks_count,
      d.bg_free_inodes_count-1,
      if (t == dir) then d.bg_used_dirs_count+1;
      else           d.bg_used_dirs_count; fi;
    )
```

Ebenso existiert eine Hilfsfunktion `ialloc_super`, die einen entsprechend angepaßten Superblock erzeugt; auf die Darstellung dieses Codes wird hier verzichtet.

Die Ermittlung der optimalen Blockgruppe für die neue Inode erfolgt unterschiedlich für Verzeichnisse und andere Dateitypen. Für neue Verzeichnisse wird unter den Blockgruppen mit überdurchschnittlich vielen freien Inodes diejenige Blockgruppe mit den meisten freien Blöcken ausgewählt. Der Algorithmus ist hier rekursiv notiert, in Wirklichkeit aber iterativ implementiert.

```
func dir_preferred_bg(disk p) gruppen#|e:
    let super_block sb = sread(p),
        // avefreei: Average Free Inodes
        int avefreei = sb.free_inodes_count/NGROUPS;

    func group_scan(gruppen#|-1 i, ext2_group_desc bestd,
        int best, disk p) gruppen#|e:
```

```
        // Abbruchbedingung.
        if (i == -1) then
            // Noch freie Inodes vorhanden?
            if (bestd.bg_free_inodes_count == 0) then
                // Abbruch mit Fehler.
                e;
            else
                // Abbruch mit Erfolg.
                best;
        else
            let ext2_group_desc d = dread(i, p);
            if  (d.bg_free_inodes_count >= avefreei) then
            and d.bg_free_blocks_count > bestd.free_blocks_count)
                // Schritt mit neuem Favoriten.
                group_scan(i-1, d, i, p);
            else
                // Schritt ohne neuen Favoriten.
                group_scan(i-1, bestd, best, p);
            fi;
        fi;

    // Start
    group_scan(NGROUPS, dread(NGROUPS, p), NGROUPS, p);
```

Für alle anderen Dateitypen wird die Blockgruppe durch das Verzeichnis vorgegeben, in dem sie (beim Anlegen) enthalten sind. Ist in dieser Blockgruppe keine freie Inode mehr verfügbar, wird zunächst mit quadratischem Hashing (siehe [BSD84]) nach einer neuen Blockgruppe gesucht. Schlägt auch dies fehl, wird eine langsame lineare Suche durch alle Blockgruppen begonnen.

```
mode scan_mode = (linear, quadratic);

func file_preferred_bg(ext2_inode dir, disk p) gruppen#|e:
    let gruppen# bg = dir.i_block_group,
        ext2_group_desc d = dread(bg, p);

    // Suche nach freien Inodes.
    func group_scan(scan_mode m,int step,
        gruppen# g, disk p) gruppen#|e:

        // Ist eine Inode in der BG frei?
        if (dread(g, p).bg_free_inodes_count > 0) then
            // Abbruchbedingung: Erfolg.
            g;
        else
            // Maximale Schrittweise erreicht?
            if (step >= NGROUPS) then
                // Abbruchbedingung: Fehler.
```

```
                e;
            else
                // Nächster Schritt.
                if (m == quadratic) then
                    group_scan(m, step^2, (g+step) mod NGROUPS, p);
                else
                    group_scan(m, step+1, (g+step) mod NGROUPS, p);
                fi;
            fi;
        fi;

    // In der aktuellen BG noch Inodes frei?
    if (d.bg_free_inodes_count > 0) then
        bg;
    else
        let gruppen#|e bg2 = group_scan(quadratic, 1, bg, p);
        if (bg2 != e) then
            bg2;
        else
            group_scan(linear, 1, bg, p);
        fi;
    fi;
```

Mit Hilfe dieser Funktionen kann `ialloc` direkt notiert werden:

```
func ialloc(ext2_inode dir, ftype new, disk p)
    paar(ext2_inode, disk)|e:

    // Bestimme die Zielblockgruppe.
    let gruppen#|e bg =
        if (new == dir) then dir_preferred_bg(p);
        else file_preferred_bg(dir, p); fi;

    // Keine Inode mehr zu bekommen.
    if (bg == e) e; then
    else
        // Als belegt markieren.
        let abs_block# b = ibmap(bg, p),
            bitmap_block bm =
                alloc_bit(0, (bitmap_block) bread(b, p)),
            disk p2 = bwrite(b, (block) bm, p);

        let disk p3 =
            dwrite(bg, ialloc_desc(new, dread(bg, p2)), p2);

        let disk p4 =
            swrite(ialloc_super(sread(p3)), p3);
```

```
        // Erzeuge neue leere Inode.
        let ext2_inode new_i = (
            new,
            0,
            ... // Weitere Felder der Inode...
        );

        // Endergebnis.
        (new_i, p4);
    fi;
```

Die Funktion bestimmt die Zielblockgruppe mit Hilfe der zum gewünschten Dateityp passenden Funktion. Sobald die Zielblockgruppe bekannt ist und sofern noch eine freie Inode gefunden werden kann, wird die Lage der Inodebelegungsbitmap mit der Funktion `ibmap()` festgestellt. Diese Bitmap wird geladen, und das erste freie Bit in der Bitmap wird belegt. Außerdem wird der Deskriptor für die Blockgruppe auf Stand gebracht.

Die Freigabe von Inodes erfolgt mit Hilfe der Funktion `ifree()` bei umgekehrter Reihenfolge der Arbeitsschritte wie in `ialloc()`: Nachdem aus der Inodenummer die Nummer der Blockgruppe und damit der zu bearbeitende Bitmapblock bestimmt sind, wird das zur Inode gehörende Bit in der Inodebelegungsbitmap freigegeben. Der Deskriptor der Blockgruppe und der Superblock werden auf Stand gebracht und anschließend die veränderten Blöcke zurück auf das Medium geschrieben.

```
func ifree(inode# ino, disk p) disk:
    let super_block   s = sread(p),
        gruppen#     bg = int(ino/I_PER_G),
        bitnr       bit = ino mod I_PER_G,
        ext2_desc     d = dread(bg, p),
        bitmap_block bm = bread(ibmap(bg, p), p);

    let disk p2 = bwrite(ibmap(bg, p),
                  clear_bit(bit, bm), p);
    let disk p3 = dwrite(bg, ifree_desc(d), p);
    let disk p4 = swrite(ifree_super(s), p);

    p4;
```

Die verwendeten Funktionen `ifree_desc()` und `ifree_super()` erzeugen analog zu `ialloc_desc()` und `ialloc_super()` angepaßte Deskriptoren und Superblöcke; ihr Code wird hier nicht gezeigt.

### 4.2.8 Allozierung und Freigabe von Blöcken

Die Belegung von Blöcken erfolgt relativ geradlinig, wenn man einen Zielblock voraussetzt, an dem die Belegung bevorzugt stattfinden soll: Das ext2–Dateisystem versucht, ab dem Zielblock linear und umlaufend einen freien Block zu finden und belegen. Ist nach einem Umlauf kein freier Block gefunden, wird ein Fehler gemeldet.

Dazu werden die folgenden beiden Hilfsfunktionen verwendet:

```
// Laden eines Bitmapblocks.
func bbread(gruppen# g, disk p) bitmap_block:
     (bitmap_block) bread(bbmap(g, p), p);

// Umlaufendes Durchsuchen der Blockbelegungsbitmap.
func block_scan(gruppen #g, gruppen# todo, disk p)
     paar(gruppen#|e, offset|e):

     // Ende der Suche?
     if (todo == 0) (e, e);
     else
          // Freier Block in diesem Bitmapblock?
          o = ffz(0, bbread(g, p));
          if (o != e) (g, o);
          else
               block_scan(g+1 mod NGROUPS, todo-1, p);
          fi;
     fi;
```

Mit Hilfe von `block_scan()`[13] kann eine Funktion formuliert werden, die ausgehend von einem gegebenen Zielblock die Blockbelegungsbitmap durchsucht und versucht, `gruppen#` und `offset` eines freien Blocks zu bestimmen.

[13]In der realen Implementation wird mit Hilfe des Deskriptors für die Blockgruppe geprüft, ob überhaupt freie Blöcke in der Blockgruppe zu finden sind. Falls nicht, braucht die Blockbelegungsbitmap der Blockgruppe nicht geladen und durchsucht zu werden.

```
func find_free_block(disk p, abs_block# goal)
    paar(abs_block#|e, offset|e):

    // Fehlerhafte Goal-Angaben werden stillschweigend
    // ignoriert.
    let abs_block# savegoal =
        if ((goal < FIRSTBLOCK) or (goal > NBLOCKS))
        then FIRSTBLOCK;
        else goal; fi;

    // Aufteilen in (gruppen#, offset).
    let gruppen# g = int(savegoal/BSIZE),
        offset   o = savegoal mod BSIZE;

    // Suche in der aktuellen bg ab dem Goal.
    let offset|e o1 = ffz(o, bbread(g, p));
    if (o1 != e) (g, o1);
    else
        // Durchsuchen der anderen Blockgruppen
        // incl. des Rests von g (ab Offset 0).
        let (gruppen#|e g2, offset|e o2) =
            block_scan(g+1 mod NGROUPS, NGROUPS, p);
        if (g2 != e) (g2, o2) else (e, e); fi;
    fi;
```

Die Blockgruppe, in der sich der Zielblock befindet, wird dabei ab dem Zielblock bevorzugt nach freien Blöcken durchsucht. Läßt sich dort kein freier Block finden, wird linear umlaufend die gesamte Blockbelegungsbitmap abgearbeitet. Da `NGROUPS` Blockgruppen beginnend ab der auf den Zielblock folgenden Blockgruppe gescannt werden, wird die Blockgruppe des Zielblocks selbst zweimal durchgegangen: einmal am Anfang ab dem `offset o` und einmal zum Schluß ab Bit 0.

Die eigentliche Belegungsfunktion `balloc()` gleicht dabei `ialloc()`:

```
func alloc_block(ext2_group_desc d) ext2_group_desc:
    // Anpassen des bg_free_blocks_count.
    ( ...
      d.bg_free_blocks_count-1,
      ...);

func balloc(disk p, abs_block# goal)
    paar(abs_block#|e, disk p):

    // Finde die zu belegende Position.
    let (gruppen# g, offset o) =
        find_free_block(p, goal);
```

```
    // Disk full.
    if (g == e)    (e, p);

    // Markiere den Block als belegt.
    let abs_block# b = bbmap(bg, p),
        bitmap_block bm =
            alloc_bit(o, bbread(g, p)),
        disk p2 = bwrite(b, (block) bm, p);

    let disk p3 =
        dwrite(bg, alloc_block(dread(bg, p2)), p2);

    (g*BGSIZE+o, p3);
```

Das ext2–Dateisystem findet den Zielblock `goal` zum Aufruf von `balloc()`, indem es feststellt, welches die höchste relative Blocknummer vor der relativen Blocknummer ist, für die ein Block zu belegen ist, und als `goal` die diesem Block zugeordnete absolute Blocknummer übergibt. Im Normalfall, bei dicht besetzten Dateien, ist diese relative Blocknummer die Nummer des unmittelbar zuvor belegten Blocks und die Zielblocknummer damit die absolute Blocknummer des unmittelbar zuvor belegten Blocks. Auf diese Weise wird eine Datei zusammenhängend vergrößert, falls möglich.

Ist mit dieser Methode kein Zielblock zu finden (etwa weil der erste Datenblock der Datei geschrieben wird), setzt das ext2–Dateisystem die absolute Blocknummer des Anfangs der Blockgruppe ein, in der die Inode der Datei liegt.

### 4.2.9 Vergrößerung von Dateien

Mit der Funktion `balloc()` kann man aus `bmap()` ein Verfahren zur Vergrößerung von Dateien ableiten. Die Funktion `bgrow()` liefert wie `bmap()` zu einer relativen Blocknummer eine absolute Blocknummer. Ist das Mapping für den betreffenden relativen Block dabei noch nicht vorhanden, wird mit `balloc()` ein freier Block gesucht und die Inode der betreffenden Datei erweitert.

```
func bgrow(ext2_inode i, abs_block# r, disk p)
    tupel(abs_block#|e, ext2_inode, disk):

    if (r<12) then
        // Unbelegter Direct-Block?
        if (i.i_block[r] == 0) then
            // Ja, bestelle einen.
            let (abs_block# new_block, disk pn) =
                balloc(p, search_back_direct(i, r));
            if (new_block == e) then (e, i, p);
            else bgrow_inode(i, r, new_block, pn); fi;
```

```
        else
            // Nein, liefere den alten.
            i.i_block[r];
        fi;
    else
        // Indirekte Blöcke behandeln.
        bgrow_indirektion(p, 1, r-11, i);
    fi;
```

Die Funktion `bgrow()` gleicht im Aufbau `bmap()`. Jedoch wird geprüft, ob das Mapping für den angegebenen relativen Block 0 ist. In diesem der Fall wird mit der noch zu definierenden Funktion `search_back_direct()` ein Zielblock bestimmt und ein neuer Block `new_block` bestellt. Die triviale und deswegen nicht gezeigte Funktion `bgrow_inode()` fügt diesen neuen Block an der Stelle `r` in die Inode `i` ein und liefert das Tripel aus neuer Blocknummer, neuer Inode (mit ggf. angepaßter Größe) und neuem Plattenzustand zurück.

Die Funktion `search_back_direct()` bestimmt dabei den Zielblock für `balloc()`, indem sie ab der Stelle `r` rückwärts die Blockeinträge in der Inode durchgeht und den ersten von 0 verschiedenen Eintrag liefert. Falls kein solcher Eintrag vorhanden ist, liefert die Funktion als Standardwert den Beginn der Blockgruppe, in der die Inode liegt.

```
func search_back_direct(ext2_inode i, abs_block# r)
    abs_block#:

    // Default-Goal.
    if (r == 0) then i.i_block_group * BGSIZE+1;
    else
        // Goal ist der letzte belegte Direct-Block.
        if (i.i_block[r] != 0) then i.i_block[r];
        // Weiter rückwärts suchen.
        else search_back_direct(i, r-1); fi;
    fi;
```

Dieselben Modifikationen werden nun auch an den Hilfsfunktionen von `bmap()` vorgenommen, und es entstehen `bgrow_absteigen()` und `bgrow_indirektion()`. Beide bedienen sich einer Hilfsfunktion, die einen Indirekt–Block rückwärts durchsucht, um einen Zielblock zur Blockbestellung zu finden, und der trivialen Hilfsfunktion `bgrow_block()`, die in einem gegebenen indirekten Block an der richtigen Stelle die neue Blocknummer einfügt und den geänderten Block dann auf das Medium zurückschreibt.

```
// Bestimme einen Goal-Block.
func search_back_indirect(ind_block b,
    abs_block# d, abs_block# r)    abs_block#:

    // Default-Goal ist der indirekte Block.
    if (r == 0) then d;
```

```
    else
        // Liefere einen belegten Eintrag...
        if (b[r] != 0) then b[r];
        // ... oder suche weiter.
        else search_back_indirect(b, d, r-1); fi;
    fi;
```

Wie in `indirektion()` wird in `bgrow_indirektion()` erst einmal ermittelt, wie viele Indirektionsebenen abgestiegen werden muß. Danach wird überprüft, ob der benötigte Eintrag für den indirekten Block der zugehörigen Indirektionsebene schon existiert. Falls nicht, wird ein Block bestellt und die Inode der Datei mit `bgrow_inode()` auf Stand gebracht. Danach beginnt der Abstieg über die verschiedenen Indirektionsebenen.

```
func bgrow_indirektion(disk p, int n, abs_block# ro, ext2_inode i)
    paar(abs_block#|e, ext2_inode, disk):

    // Normalisiere die relative Blocknummer.
    let abs_block# rn = ro - A_PER_B^(n-1);

    // Liegt der gesuchte Block im Zielbereich?
    if (rn < A_PER_B^n) then
        // Ja. Ist er außerdem vorhanden?
        if (i.i_block[11+n] == 0) then
            // Nein, erweitern.
            let (abs_block# new_block, disk p1) =
                balloc(p, search_back_direct(i, 11+n));
            if (new_block == e) then (e, p);
            else
                let (abs_block# d1, ext2_inode i1, disk p2) =
                    bgrow_inode(i, 11+n, new_block, p1);
                bgrow_absteigen(p2, n, rn, new_block, i1);
            fi;
        else
            // Ja, absteigen.
            bgrow_absteigen(p, n, rn, i.i_block[11+n], i);
        fi;
    else
        // Nein, dann Indirektionsebene erhöhen.
        bgrow_indirection(p, n+1, rn, i);
    fi;
```

Die Abstiegsfunktion `bgrow_absteigen()` ermittelt die Blocknummer der nächsttieferen Ebene. Wieder ist zu prüfen, ob diese Blocknummer 0 wäre. Ist dies der Fall, muß ein neuer Block bestellt und in den Block der aktuellen Indirektionsebene eingetragen werden, bevor der Abstieg auf die tiefer gelegene Indirektionsebene möglich ist.

```
func bgrow_absteigen(disk p, int j,
    abs_block# r, abs_block# d,    ext2_inode i)
    paar(abs_block#|e, ext2_inode, disk p):

    let int index       = int(r/A_PER_B^(j-1)) mod A_PER_B;
    let ind_block ind   = (ind_block) bread(d, p);
    let abs_block# next = ind[index];

    // Unterste Indirektionsebene erreicht?
    // Ja
    if (j == 1) then
        // Unbelegter Eintrag?
        if (next == 0) then
            // Ja, bestelle einen Block.
            let (abs_block# new_block, disk pn) =
                balloc(p, search_back_indirect(ind, d, r));
            if (new_block == e) then (e, i, p);
            else bgrow_block(d, r, new_block, i, pn); fi;
        else
            // Nein, liefere den alten.
            (next, i, p);
        fi;
    else
        // Nein, tiefer absteigen. Abstieg möglich?
        if (next == 0) then
            // Nein, erweitern.
            let (abs_block# new_block, disk p1) =
                balloc(p, search_back_indirect(ind, d, r));
            if (new_block == e) then (e, i, p);
            else
                let (abs_block# dummy, ext2_inode i1, disk p2) =
                    bgrow_block(d, r, new_block, i, p1);
                absteigen(p2, j-1, r, new_block, i1);
            fi
        else
            // Ja, absteigen.
            absteigen(p, j-1, r, next, i);
    fi;
```

Eine ähnliche Konstruktion kann zum Freigeben von Blöcken mit der Funktion `bdrop()` verwendet werden. Dabei ist rekursiv so lange abzusteigen, bis entweder die unterste Indirektionsebene erreicht oder der Zeiger auf die nächste Ebene leer ist. Im letzteren Fall entfällt eine weitere Aktion, im ersten Fall ist der entsprechende Block freizugeben. Beim Aufstieg aus der Rekursion ist auf jeder Ebene zu prüfen, ob der betreffende indirekte Block überhaupt noch Zeiger auf nachgeordnete Blöcke enthält. Wenn nicht, kann er ebenfalls freigegeben werden usw. Der Code für diese Funktion wird nicht gezeigt, da er strukturell keine neuen Erkenntnisse bringt.

### 4.2.10 Integritätsbedingungen

#### 4.2.10.1 Überprüfung der Integrität

Im ext2–Dateisystem finden zur Laufzeit kaum Prüfungen auf Integrität des Dateisystems statt. Zwar wird für einzelne Parameter von Kernelfunktionen getestet, ob sie plausible Werte enthalten, aber für eine übergreifende Verifikation von Integritätsbedingungen stehen zur Laufzeit des Dateisystems weder Speicher noch Zeit zur Verfügung.

Statt dessen finden solche Tests beim Ablauf des Dateisystemcheckers `fsck` statt. Dabei unterhält das ext2–Dateisystem im Superblock einen Zähler, der bei jedem Anmelden des Dateisystems hochgezählt wird, und einen Zeitstempel, der den Zeitpunkt der letzten Integritätsprüfung angibt. Nach einer einstellbaren Anzahl von Anmeldungen des Dateisystems oder alternativ nach dem Verstreichen eines einstellbaren Zeitraumes seit dem letzten Check, je nachdem, welches Ereignis zuerst eintritt, wird beim Anmelden des Dateisystems eine Integritätsprüfung durchgeführt, auch wenn das Dateisystem ansonsten als „sauber abgemeldet“ markiert ist.

Ist das Dateisystem nicht als „sauber abgemeldet“ markiert, wird in jedem Fall ein `fsck`–Lauf erzwungen, um einen definierten Zustand herzustellen. Wenn das Wurzeldateisystem in Linux ein ext2–Dateisystem ist, wird es unmittelbar nach dem Systemstart zunächst schreibgeschützt angemeldet, falls notwendig geprüft und dann als beschreibbar umgemeldet. Auf diese Weise ist eine Integritätsprüfung des Wurzeldateisystems mit einem definierten Dateisystemzustand möglich, obwohl der Kernel läuft und mit dem Dateisystem arbeitet.

Das `fsck`–Programm für das ext2–Dateisystem nimmt sehr umfangreiche Prüfungen des Dateisystems vor, die nicht alle für die Betrachtungen dieser Arbeit relevant sind, etwa weil sie sich auf die Integrität der Verzeichnisstruktur beziehen, die in dieser Arbeit nicht behandelt wird. Auf diese Prüfungen wird hier nicht weiter eingegangen.

#### 4.2.10.2 Hilfsfunktionen

Zur Formulierung der Integritätsbedingungen wird eine Hilfsfunktion benötigt: Die Funktion `blocks(i, 0)` liefert die Menge der Datenblöcke, die von der Inode `i` verwaltet werden. Entsprechend liefert `blocks(i, j)` ($1 \leq$ `j` $\leq 3$) die Menge der `j`–fach indirekten Blöcke einer Inode. Die Mengen sind mit Hilfe von `bmap()` bzw. den entsprechenden Mappingfunktionen für indirekte Blöcke leicht zu konstruieren:

$$\mathtt{blocks}(\mathtt{i},0) = \{\mathtt{bmap}(\mathtt{i},\mathtt{j}) | \mathtt{bmap}(\mathtt{i},\mathtt{j}) \neq 0 \wedge 0 \leq \mathtt{j} < 2^{32}\}$$

und `blocks(i, 1..3)` analog mit den Mappingfunktionen für die indirekten Blöcke. Der Index `j` muß tatsächlich von 0 bis $2^{32}$ laufen und nicht von 0 bis `i.i_blocks`, denn es kann nicht vorausgesetzt werden, daß der Blockcount der Datei in der Inode korrekt ist. In der Praxis ist das kein großes Problem, da nicht wirklich alle $2^{32}$ Werte für `j` durchlaufen werden müssen. Statt dessen kann man schon auf den höheren Indirektionsebenen prüfen, ob überhaupt Datenblöcke existieren, so daß sich sehr große Schrittweiten ergeben.

Die Menge aller Blöcke, die zu einer Inode gehören, ist die Vereinigungsmenge der einzelnen Blockmengen:

$$\mathtt{allblocks(i)} = \bigcup_{\mathtt{j}=0}^{3} \mathtt{blocks(i,j)}$$

Die Menge `inodes` ist die Menge aller belegten normalen Inodes, d.h. die Menge aller Inodes `i`, für die $\mathtt{i.i\_dtime} = 0$ ist und deren Inodenummer entweder 2 (Wurzelverzeichnis) oder größer 16 ist.

#### 4.2.10.3 Integritätsbedingungen für einzelne Inodes

Die erste Phase der Dateisystemprüfung geht alle Inodes der Inodetabelle durch und prüft die Einhaltung verschiedener Bedingungen für einzelne Inodes und für Inodes untereinander: Alle Inodes müssen von einem gültigen Typ sein, ihre Länge und ihr Blockcount müssen plausible Werte aufweisen und sie dürfen keine Mappings für Datenblöcke oder indirekten Blöcke in Plattenbereichen haben, in denen keine Datenblöcke liegen können. In Zeichen:

$$\forall \mathtt{i} \in \mathtt{inodes} : \mathtt{i.i\_type} \in \mathtt{ftype} \quad (4.1)$$

$$\forall \mathtt{i} \in \mathtt{inodes} : \mathtt{i.i\_size} \geq |\mathtt{blocks(i,0)}| \quad (4.2)$$

$$\forall \mathtt{i} \in \mathtt{inodes} : \mathtt{i.i\_blocks} = |\mathtt{blocks(i,0)}| \quad (4.3)$$

$$\begin{aligned}
\forall \mathtt{i} \in \mathtt{inodes} \quad : \quad & \forall \mathtt{b} \in \mathtt{allblocks(i)} : \qquad (4.4)\\
& \mathtt{b} \geq \mathtt{FIRSTBLOCK}\\
\wedge \quad & \mathtt{b} < \mathtt{NBLOCKS}\\
\wedge \quad & \mathtt{b} \notin \{\mathtt{smap(g)} | 0 \leq \mathtt{g} < \mathtt{NGROUPS}\}\\
\wedge \quad & \mathtt{b} \notin \{\mathtt{bbmap(g,p)} | \mathtt{diskp}, 0 \leq \mathtt{g} < \mathtt{NGROUPS}\}\\
\wedge \quad & \mathtt{b} \notin \{\mathtt{ibmap(g,p)} | \mathtt{diskp}, 0 \leq \mathtt{g} < \mathtt{NGROUPS}\}\\
\wedge \quad & \mathtt{b} \notin \{\mathtt{kopf(dmap(g,0))} \ldots \mathtt{kopf(dmap(g,NGROUPS))} | 0 \leq \mathtt{g} < \mathtt{NGROUPS}\}\\
\wedge \quad & \mathtt{b} \notin \{\mathtt{kopf(imap(i,p))} | \mathtt{diskp}, 0 \leq \mathtt{i} < \mathtt{NINODES}\}
\end{aligned}$$

Außerdem dürfen sich die von zwei verschiedenen Inodes belegten Blöcke nicht überlappen, d.h.

$$\forall \mathtt{i} \in \mathtt{inodes} : \forall \mathtt{j} \in \mathtt{inodes} : \mathtt{allblocks(i)} \cap \mathtt{allblocks(j)} = \emptyset \quad (4.5)$$

Die Bedingungen 4.4 und 4.5 entsprechen einander, wenn man die Mappings von Superblöcken, Deskriptoren usw. als degenerierte, nicht durch Inodes beschriebene Form von Dateien ansieht:

Beide Bedingungen besagen nichts weiter, als daß ein Block des Mediums nur eine Funktion haben, d.h. nur von einem Mapping verwendet werden kann.

Die Verschärfung, daß jeder Block von genau einem Mapping verwendet werden muß, wird in der Dateisystemprüfung nicht explizit formuliert. Trotzdem können keine Blöcke verloren gehen, da in der letzten Phase der Dateisystemprüfung die Blockbelegungsbitmap regeneriert wird, die allen Blöcken, die keine andere Funktion haben, die Funktion „freier Block" zuweist.

#### 4.2.10.4 Integritätsbedingungen für Verzeichnisse und Inodes

In der zweiten Phase der Integritätsprüfung werden die einzelnen Verzeichnisse eines Dateisystems untersucht: Alle Verzeichnisseinträge müssen mindestens acht Bytes[14] lang sein, dürfen aber nicht länger sein als der verbleibende Rest des Blocks. Außerdem kann der in einem Verzeichniseintrag enthaltene Name nicht länger sein als der Verzeichniseintrag selbst weniger acht Bytes.

Die im Verzeichniseintrag enthaltene Inodenummer muß eine gültige, nicht gelöschte Datei bezeichnen. Der erste Name in jedem Verzeichnis muß `.` (Punkt) lauten, und die Inodenummer in diesem Verzeichniseintrag muß die Inodenummer des Verzeichnisses selbst sein. Der zweite Namenseintrag in jedem Verzeichnis ist zwingend der Eintrag für das übergeordnete Verzeichnis `..` (Punkt Punkt).

Der Datentyp eines Verzeichniseintrages ist in der gewählten strikten Notation schlecht darzustellen, denn seine Größe ist variabel. Entsprechend ist die Anzahl der Verzeichniseinträge in einem Block auch nicht konstant, sondern abhängig von der Größe der einzelnen Einträge. Die Verzeichniseinträge in einem Block einer Verzeichnisdatei füllen diesen Block jedoch immer ganz aus, und ein einzelner Verzeichniseintrag erstreckt sich niemals über eine Blockgrenze.

Dehnt man die Grenzen der Notation ein wenig, kann man einen Verzeichniseintrag wie nachstehend angeben und einen Block einer Verzeichnisdatei dann als Tupel und nicht als Feld von Verzeichniseinträgen interpretieren.

```
mode dirent = (
    inode# d_ino;      // Inodenummer.
    offset d_off;      // Länge des Verzeichniseintrages.
    offset d_reclen;   // Länge des Namens im Eintrag.
    char   d_name[d_reclen]; // Feld variabler Größe.
  );
```

[14]Vier Bytes für die Inodenummer, zwei Bytes für die Länge des Verzeichniseintrages und weitere zwei Bytes für die Länge des darin enthaltenen Namens.

Damit lassen sich die oben angegebenen Bedingungen[15] in Zeichen wie folgt notieren, wobei `p` die zu prüfende `disk` ist:

$$\begin{aligned}
\forall \mathtt{i} \in \mathtt{inodes} \quad &: \quad \mathtt{i.i_type} = \mathtt{dir} \Rightarrow \\
& \quad\ \mathtt{kopf((seq\ dirent)bread(bmap(i,0,p))).d_name} = \text{“.“} \\
& \wedge \quad \mathtt{iread(kopf((seq\ dirent)bread(bmap(i,0,p))).d_ino,p)} = \mathtt{i} \\
& \wedge \quad \mathtt{kopf(rest((seq\ dirent)bread(bmap(i,0,p)))).d_name} = \text{“..“} \\
\wedge \forall \mathtt{b} \in \mathtt{blocks(i,0)} \quad &: \quad \sum_{\mathtt{d} \in \mathtt{(seq\ dirent)b}} \mathtt{d.d_off} = \mathtt{BSIZE} \\
\wedge \forall \mathtt{d} \in \mathtt{(tupel\ dirent)b} \quad &: \quad \mathtt{d.d_off} \geq \mathtt{d.d_reclen} + 8 \\
& \wedge \quad \mathtt{iread(d.d_ino,p)} \in (\mathtt{inodes} \cup \{0\})
\end{aligned} \tag{4.6}$$

Die Phasen 3 und 4 der Dateisystemprüfung beschäftigen sich anschließend damit, den Zusammenhang und die Gültigkeit der Verzeichnisstruktur sicherzustellen und die Referenzzähler `i_nlink` aller Inodes zu regenerieren. Phase 5 erzeugt aus den zuvor gesammelten Informationen neue Block– und Inodebelegungsbitmaps und neue Deskriptoren für die Blockgruppen.

## 4.3 Modellierung der Änderungen

Ausgehend vom Modell des ext2–Dateisystems wurden Modifikationen vorgenommen, die die Installation einer Inodetabelle in Form einer Datei erlauben. Die Änderungen lassen sich in die folgenden Kategorien aufteilen:

- Aufteilung der Informationen aus den Deskriptoren der Blockgruppen in blockbezogene und inodebezogene Informationen. Reduzierung der Deskriptoren auf blockbezogene Informationen.
- Änderung der Accessorfunktionen `iread` und `iwrite`, so daß sie eine Inodetabelle und eine Belegungsbitmap in Form von Dateien akzeptieren.
- Erweiterung der Accessorfunktion `iread` um eine Hilfsfunktion `igrow`, die die Inodetabelle wachsen läßt, und Erweiterung der Accessorfunktion `iwrite` um eine Hilfsfunktion `idrop`, die vollständig freie Blöcke der Inodetabelle freigibt.

[15] Es fehlt die Bedingung, daß ein Verzeichniseintrag maximal so lang sein kann wie der verbleibende Rest des Blocks. Sie kann angegeben werden, indem man Zeiger und Zeigerarithmetik einführt oder eine Funktion `offset()` voraussetzt, die die Mechanik der Zeigerarithmetik verbirgt.

### 4.3.1 Konstanten und elementare Funktionen

Im modifizierten Dateisystem enthalten die Deskriptoren der Blockgruppen nur noch Informationen über die Blöcke der Blockgruppe, jedoch nicht mehr über die Inodes, da Inodes nicht mehr strikt einer Blockgruppe zugeordnet werden können. Die neue Deskriptorstruktur ist wesentlich kleiner als zuvor: Sie belegt nur noch acht Bytes.

```
let DSIZE = 8; // Größe eines Deskriptors in Bytes.
```

Die aus dieser Konstante abgeleiteten Werte werden wie bisher berechnet. Es passen jetzt allerdings mehr Deskriptoren als zuvor in einen Datenblock.

### 4.3.2 Blockgruppen

Die maximale Größe einer Blockgruppe wird weiterhin unverändert durch die Anzahl der Bits in einem Datenblock vorgegeben: Eine Blockgruppe muß vollständig durch ihre Blockbelegungsbitmap beschrieben werden, und diese Belegungsbitmap muß selbst wiederum in einem einzigen Datenblock Platz finden. Daher gilt wie zuvor:

```
let int BGSIZE = BSIZE * BITS_PER_B;
mode bitnr = {0 .. BGSIZE-1};
```

Die Konstanten `IB_PER_G` (Anzahl der Datenblöcke pro Blockgruppe, die Inodes enthalten) und `I_PER_G` (Anzahl der Inodes pro Blockgruppe) werden nicht mehr berechnet, da Inodes nicht mehr an Blockgruppen gebunden sind.

Der Verwaltungsoverhead, der fest einer Blockgruppe zugeordnet werden kann, sinkt um einen Block, weil nicht mehr statisch ein Block pro Blockgruppe für eine Inodebelegungsbitmap zugewiesen werden muß. Die Berechnung der Anzahl der Blockgruppen pro Dateisystem muß dies natürlich berücksichtigen:

```
// GBLOCKS: Anzahl der Blocks zur Speicherung der Deskriptoren
// Overhead: Superblock + Blockbitmapblock + Deskriptoren.
let int GBLOCKS    = int((int(NBLOCKS/BGSIZE)+1)/D_PER_B),
    int G_OVERHEAD = GBLOCKS+FIRSTBLOCK+2;
```

Die weitere Berechnung der Anzahl der Blockgruppen folgt dann dem Modell für das unmodifizierte Dateisystem. Die Funktion `gmap` wird unverändert übernommen.

Die Konstante `NINODES` steht nicht mehr zur Verfügung. Statt dessen existieren Felder mit statistischen Informationen über die Anzahl der Inodes im Dateisystem im Superblock.

### 4.3.3 Superblöcke

Die Funktionen zum Mapping des Superblocks und seiner Kopien sind unverändert. Ebenso sind die Accessorfunktionen von Modifikationen nicht betroffen.

Die Struktur des Superblocks ist jedoch gegenüber dem unmodifizierten Dateisystem leicht erweitert. Die folgenden Felder sind hinzu gekommen:

```
// Zum Auffinden der Inode der Inodetabellendatei.
abs_block# s_special_inodes;

// Zählt die Anzahl der freien Inodes pro Block
// der Inodebelegungsbitmap.
short      s_free_inodes_count[2048];
// Anzahl der Zähler.
short      s_free_inodes_max;
```

### 4.3.4 Deskriptoren und Bitmaps

Die Lage der Deskriptoren hat sich gegenüber dem unmodifizierten Dateisystem nicht geändert. Entsprechend bleibt die Funktion `dmap` unverändert.

Ein Deskriptor für eine Blockgruppe enthält beim modifizierten Dateisystem jedoch nur noch Informationen über die Blöcke der Blockgruppe.

```
mode ext3_group_desc = (
    abs_block# bg_block_bitmap;
    int        bg_free_blocks_count;
    int        bg_used_dirs_count;
  );
```

Die Accessorfunktionen `dread` und `dwrite` werden unverändert (d.h. für den neuen Datentyp `ext3_group_desc` angepaßt) übernommen. Auch die Funktion `bbmap`, die die Lage der Belegungsbitmap für Blöcke auf dem Medium beschreibt, kann unverändert übernommen werden. Die Funktion `ibmap` (Positionierung der Belegungsbitmap für Inodes) macht statt dessen wie in den Vorüberlegungen beschrieben von einer Inode Gebrauch.

```
let inode# EXT3_IBM_NR = 6; // Inode Nummer der InodebitMap.

func ibmap_b(rel_block# b, disk p) abs_block#:
    let ext3_inode i = iread(EXT3_IBM_NR, p);

    bmap(i, b, p);
```

Dabei ist zu beachten, daß sich die Signatur der Funktion geändert hat: Im alten Dateisystem war *ibmap* eine Funktion, die eine `gruppen#` und eine `disk` auf eine absolute Blocknummer abgebildet hat. Im modifizierten Dateisystem wird statt einer Blockgruppennummer eine `rel_block#` abgebildet.

Diese Änderung der Funktionssignatur entspricht einer Änderung in der Bedeutung der Belegungsbitmap: Im alten Dateisystem war jeder Blockgruppe ein Block mit einer Belegungsbitmap für die Inodes dieser Blockgruppe zugeordnet. In den meisten Fällen war dieser Block nicht voll ausgenutzt, denn eine Blockgruppe kann aus maximal `BGSIZE` Blöcken bestehen, hatte aber nur `BGSIZE/IRATIO` Inodes. Die Berechnung der Größe einer Blockgruppe erfolgt nun so, daß die Belegungsbitmap für Blöcke möglichst ausgenutzt ist; folglich ist der Block mit der Belegungsbitmap für Inodes im alten Dateisystem nur zu einem `IRATIO`–Bruchteil ausgenutzt.

Im modifizierten Dateisystem muß auf die Bindung von Inodes zu Blockgruppen keine Rücksicht mehr genommen werden. Es wird statt dessen angenommen, daß die einzelnen Blöcke der Belegungsbitmap für Inodes immer voll ausgenutzt werden. Das `i`-te Bit der Belegungsbitmapdatei gibt also den Zustand der Inode mit der Nummer `i` an. Die Funktion `ibmap` kann also auch so angegeben werden, daß sie direkt Inodenummern akzeptiert[16]:

```
func ibmap_i(inode# i, disk p) paar(abs_block#, bitnr):
    let rel_block# r = int(i/BITS_PER_B),
        bitnr      s = i mod BITS_PER_B;

    (ibmap_b(r, p), s);
```

### 4.3.5 Inodetabelle

Im herkömmlichen Dateisystem ist die Inodenummer beschränkt auf den Bereich {`0 .. NINODES-1`}. Die Konstante `NINODES` existiert im neuen Dateisystem nicht mehr, es ist (von Beschränkungen der Implementation abgesehen) möglich, alle Datenblöcke des Dateisystems in Inodeblöcke umzuwandeln. Das modifizierte Dateisystem könnte also ohne Berücksichtigung des Verwaltungsoverheads aus bis zu `NBLOCKS*I_PER_B` Inodes bestehen.

```
mode inode# = {0 .. NBLOCKS*I_PER_B-1};
```

Die Hilfsfunktion `inmap` aus dem alten Dateisystem wird nicht mehr benötigt. Das modifizierte Dateisystem definiert `imap` statt dessen unter Zuhilfenahme von `bmap` zunächst einmal naiv so:

[16]Die Definition von `inode#` für das neue Dateisystem steht noch aus. Sie wird im folgenden Abschnitt gegeben.

```
let inode# EXT3_INT_NR = 5; // Inodenummer der Inodetabellendatei

// Naive Version
func imap(inode# i, disk p) paar(abs_block#, offset):
    let ext3_inode i = iread(EXT3_INT_NR, p),
        rel_block# b = int(i/I_PER_B),
        offset o     = (i mod I_PER_B) * ISIZE/BSIZE;

    (bmap(i, b, p), o);
```

Diese Funktionsdefinition enthält einen ungeschützten Aufruf der Funktion `iread` zum Laden der Inode der Inodetabellendatei. `iread` verwendet aber wiederum `imap` zur Bestimmung der Lage der Inode im Dateisystem (die Definition von `ext3_inode` wird weiter unten gegeben):

```
mode inr = {0 .. I_PER_B-1};
mode inode_block = ext3_inode[inr];

// Naive Version
func iread(inode# i, disk p) ext3_inode:
    let (abs_block# b, offset o) = imap(i, p),
        inr i_in_b = o/ISIZE;

    (inode_block) bread(b, p).[i_in_b];
```

Dadurch entsteht die Aufrufsequenz `iread(any, p)` → `imap(any, p)` → `iread(5, p)` → `imap(5, p)` → ... In `imap` müssen besondere Vorkehrungen getroffen werden für den Fall, daß die Position einer ausgezeichneten Inode bestimmt werden soll. Wie in 4.1.6 geschildert gibt es dazu zwei verschiedene Möglichkeiten: Entweder man zieht die gesamte Inode der Inodetabellendatei aus der Inodetabelle heraus, so daß `iread(5, p)` niemals `imap` aufruft, oder man hinterlegt nur die Adresse des einen Blocks, der diese Inode enthält, an einer bekannten Stelle, so daß das Funktionsergebnis für `imap(5, p)` ohne Aufruf von `iread` berechnet werden kann, weil es direkt bekannt ist.

Für das modifizierte Dateisystem wurde die zweite Methode gewählt, weil sie leichter zu implementieren war: Im Superblock wurde ein Feld `abs_block# s_special_inodes` eingefügt, das die Blocknummer des ersten Blockes der Inode `EXT3_INT_NR` enthält. `imap` fragt auf diese spezielle Bedingung ab und liefert in diesem Fall ein Ergebnis ohne den Aufruf von `imap`:

```
// Version mit aufgelöster Selbstbezüglichkeit.
func imap(inode# i, disk p) paar(abs_block#, offset):
    let super_block s = sread(p);

    // Ermittle die relative Blocknummer.
    let rel_block# b = int(i/I_PER_B);

    // Teste auf den Sonderfall.
```

```
    if ((i == EXT3_INT_NR) and (b == 0)) then
        // Sonderfall.
        s.s_special_inodes;
    else
        // Normallfall.
        let ext3_inode i = iread(EXT3_INT_NR, p),
            offset o     = (i mod I_PER_B) * ISIZE/BSIZE;

        (bmap(i, b, p), o);
    fi
```

Die Implementierung der ersten Methode würde statt dessen in `iread` ansetzen: Die Position des Datenblocks für die besondere Inode wird nicht mittels `imap` bestimmt, sondern im Feld `s_special_inodes` im Superblock nachgeschlagen.

```
// Version mit aufgelöster Selbstbezüglichkeit.
func iread(inode #i, disk p) ext3_inode:
    let super_block s = sread(p);
    // Teste auf Sonderfall.
    if (i == EXT3_INT_NR)
        // Sonderfall
        let abs_block# b = s.sb_special_inodes,
            i_in_b       = i mod I_PER_B;
    else
        // Normalfall.
        let (abs_block# b, offset o) = imap(i, p),
            inr i_in_b = o/ISIZE;
    fi;

    (inode_block) bread(b, p).[i_in_b];
```

In beiden Fällen muß der Funktion die fehlende Information über eine zusätzliche Informationsquelle zugeführt werden. Der Superblock bietet sich dafür an, weil seine Position bekannt ist[17].

Die Datenstruktur der `ext3_inode` entspricht der `ext2_inode` weitgehend; sie ist um ein zusätzliches Feld `i_block_group` vom Typ `gruppen#` erweitert, das die Nummer der Blockgruppe angibt, aus der die Datenblöcke für diese Datei bevorzugt belegt werden sollen.

[17]Die Implementierung des Dateisystems hält den Superblock eines angemeldeten Dateisystem ständig im Speicher. Zugriffe auf hier gespeicherte dateisystemspezifische Informationen sind also extrem schnell möglich.

### 4.3.6 Allozierung und Freigabe von Inodes und Blöcken

Der modifizierte Algorithmus `ialloc` unterscheidet sich bedingt durch die Dynamisierung sehr stark von dem in Abschnitt 4.2.7 beschriebenen Verfahren. Im modifizierten Dateisystem ist die Inodetabelle immer nur so groß wie benötigt. Wächst die Anzahl der Inodes, werden neue Inodes erzeugt. Der Algorithmus muß also im Gegensatz zu dem in 4.2.7 beschriebenen Verfahren davon ausgehen, daß die Inodetabelle fast immer fast vollständig belegt ist.

Um schnell freie Inodes in der Inodetabelle aufzufinden, wird gemäß Abbildung 4.5 der Superblock wird um ein Array `short s_inodes_count[]` erweitert. Jedes Arrayelement `i` gibt an, wie viele Inodes im `i`–ten Block der Inodebelegungsbitmap als frei gekennzeichnet sind. Das Array wird beim Anmelden des Dateisystems dynamisch erzeugt, indem alle Blöcke der Inodebitmapdatei durchgelesen werden und die als frei markierten Inodes gezählt werden. Da das Array im Hauptspeicher gehalten wird, können Bitmapblöcke mit freien Inodes relativ schnell identifiziert werden.

Eine Inode wird mit den folgenden Arbeitsschritten belegt:

1. Bestimme die Blockgruppe, in der die Datenblöcke für die neue Inodes bevorzugt gelegt werden sollen.
2. Finde in `sb.s_free_inodes_count[]` einen Block der Belegungsbitmap für Inodes mit einem Zählerstand größer als 0. Kann kein solcher Block gefunden werden, wende den Algorithmus `igrow` an.
3. Finde in dieser Bitmap ein Bit, das eine freie Inode markiert.
4. Setze das Bit und korrigiere `sb.s_free_inodes_count[]` sowie den globalen Zähler für die Anzahl der Inodes und der freien Inodes im Dateisystem.
5. Wenn ein Verzeichnis angelegt werden soll, korrigiere außerdem den Zähler `bg_used_dirs_count` der betreffenden Blockgruppe.
6. Erzeuge die leere Inode und gib sie zurück.

Die Bestimmung der Blockgruppe für die Datenblöcke erfolgt für Verzeichnisse in gleicher Weise wie im ext2–Dateisystem, jedoch mit einer geänderten Bewertungsfunktion: Im ext2–Dateisystem geht die Anzahl der noch verfügbaren Inodes bei der Auswahl der Zielblockgruppe ein. Dieser Wert steht im ext3–Dateisystem nicht mehr zur Verfügung, so daß hier mit der Anzahl der freien Blöcke und der Anzahl der Verzeichnisse in der Blockgruppe gearbeitet wird:

```
func dir_preferred_bg(disk p) gruppen#|e:
    let super_block sb = sread(p),
        int avefree = sb.free_blocks_count/NGROUPS;

    func group_scan(gruppen#|-1 i, ext2_group_desc bestd,
        int best, disk p) gruppen#|e:
```

```
        // Abbruchbedingung.
        if (i == -1)
            // Noch freie Inodes vorhanden?
            if (bestd.bg_free_blocks_count == 0)
                // Abbruch mit Fehler.
                e;
            else
                // Abbruch mit Erfolg.
                best;
        else
            let ext2_group_desc d = dread(i, p);
            if  (d.bg_free_blocks_count >= avefree
            and d.bg_used_dirs_count < bestd.used_dirs_count)
                // Schritt mit neuem Favoriten.
                group_scan(i-1, d, i, p);
            else
                // Schritt ohne neuen Favoriten.
                group_scan(i-1, bestd, best, p);
            fi;
        fi;

    // Start.
    group_scan(NGROUPS, dread(NGROUPS, p), NGROUPS, p);
```

Die Funktion `file_preferred_bg()` gibt im ext3–Dateisystem nur den Inhalt des Feldes `i_block_group` des übergeordneten Verzeichnisses zurück:

```
func file_preferred_bg(ext3_inode dir, disk p) gruppen#|e:
    dir.i_block_group;
```

Im nächsten Schritt muß ein Block in der Inodebelegungsbitmap gefunden werden, der noch Nullbits, also freie Inodes, enthält. Kann kein solcher Block gefunden werden, muß die Inodetabellendatei vergrößert werden.

```
func find_ibm_block(disk p) rel_block#|e:
    let super_block s = sread(p);

    func scan_bitmap(super_block s, short counter) rel_block#|e:
        if (counter == s.s_free_inodes_max)
            e;
        else
            if (s.s_free_inodes_count[counter] > 0)
                counter;
            else
                scan_bitmap(s, counter+1);
```

```
            fi;
        fi;

    scan_bitmap(s, 0);
```

Der entsprechende Block der Inodebelegungsbitmap muß geladen werden, um dort eine Inode belegen zu können:

```
func mark_inode(rel_block# r, disk p)
    let abs_block#   b  = bmap(iread(EXT3_IBM_NR), p), r, p);
        bitmap_block bm = (bitmap_block) bread(b, p);

    alloc_bit(0, bm);
```

Die passenden Funktionen `ialloc_desc()` und `ialloc_super()` (siehe Abschnitt 4.2.7) werden hier nicht mehr explizit angegeben. Sie erzeugen aus einem gegebenen Deskriptor bzw. Superblock einen neuen Deskriptor bzw. Superblock, in dem die entsprechenden Zähler zur Buchhaltung angepaßt worden sind.

```
func ialloc_desc(ftype t, ext3_group_desc d) ext3_group_desc:
    ...
    // Anpassen von bg_used_dirs_count, falls notwendig.

func ialloc_super(super_block s, rel_block# r, bit more_counters)
    super_block:
    ...
    // Anpassen der Zähler in s_free_inodes_count[r],
    // ggf. anpassen von s_free_inodes_max.
```

Mit diesen Funktionen kann man nun eine Funktion `ialloc()` angeben, die im ext3–Dateisystem eine Inode belegt und dabei die Inodetabelle und Inodebelegungsbitmap erweitert, falls notwendig.

```
func update_all(rel_block# r, super_block s,
    bit more_counters, disk p)
    paar(ext3_inode, disk):

    // Inode in der Bitmap belegen.
    let (abs_block# b, bitmap_block bm) = mark_inode(r, p);
        // Bitmap, Deskriptor und Superblock aktualisieren.
        disk p2 = bwrite(b, bm, p);
        disk p3 = dwrite(bg,
                          ialloc_desc(new, dread(bg, p2)), p2);
        disk p4 = swrite(ialloc_super(s, r, more_counters), p3);
```

```
    let ext3_inode new_i = (
        new,
        0,
        ...
        bg;  // An der Stelle i_block_group.
    );

    (new_i, p4);

func ialloc(ext3_inode dir, ftype new, disk p)
    paar(ext3_inode|e, disk):

    // Bestimmung der Zielblockgruppe.
    let gruppen#|e bg =
        if (new = dir) dir_preferred_bg(p);
        else file_preferred_bg(dir, p); fi;

    if (bg == e) (e, p);
    else
        // Sind Bitmap-Blöcke mit freien Inodes zu finden?
        let rel_block# r = find_ibm_block(p);

        // Die Inodetabelle war voll?
        if (r == e)
            // Ja, sie muß erweitert werden.
            let (rel_block# r2|e, disk p2) = igrow(p);
            // Hat das Erweitern funktioniert?
            if (r2 == e)
                (e, p);
            else
                update_all(r2, s, 1, p2);
            fi;
        else
            update_all(r, s, 0, p);
        fi;
    fi;
```

Die Funktion sucht zunächst einen Block der Inodebelegungsbitmap, der freie Inodes aufweist. Existiert kein solcher Block, wird versucht, die Inodetabelle mit `igrow()` zu erweitern. Gelingt dies, kann mit Sicherheit eine Inode belegt werden, anderenfalls wird ein Fehler signalisiert. Falls von Anfang an noch freie Inodes zu finden sind, wird eine von diesen belegt.

In der neuen Inode wird an der Stelle `i_block_group` die Nummer der Blockgruppe hinterlegt, aus der bevorzugt Datenblöcke zu belegen sind. Die Funktion `balloc()` kann dann unverändert aus dem ext2–Dateisystem übernommen werden. Der Inhalt von `i_block_group` geht im ext3–Dateisystem in die Berechnung des Zielblocks `goal` ein: Falls der erste Datenblock der Datei belegt werden soll, wird er bevorzugt aus der hier angegebenen Blockgruppe gewählt.

Die Freigabe von Inodes erfolgt analog anders herum: Aus der angegeben Inodenummer wird eine relative Blocknummer der Inodebelegungsbitmap und eine Bitnummer bestimmt. Der entsprechende Block der Belegungsbitmap wird über `bmap()` bestimmt, geladen, modifiziert und geschrieben. Ebenso werden die Deskriptoren und der Superblock angepaßt.

### 4.3.7 Dynamische Inodetabelle

Die Erweiterung der Inodetabelle erfolgt auf Anforderung aus der Funktion `ialloc()` heraus mit Hilfe der Funktion `igrow()`. Dabei wird mittels `bgrow()` ein neuer Block für die Inodetabellendatei angefordert. Wächst die Inodetabelle dabei so, daß die Bitmap auch vergrößert werden muß, wird zusätzlich ein weiterer Block für die Inodebelegungsbitmap angefordert. In diesem Fall ist auch das Zählerfeld `s_free_inodes_count[]` im Superblock zu vergrößern, und der neue Zähler muß initialisiert werden.

In jedem Fall müssen in der Inodebelegungsbitmap Bits für die neuen Inodes freigegeben, und die Zähler für die Gesamtzahl der Inodes und die Anzahl der freien Inodes angepaßt werden.

```
func igrow(disk p) paar(rel_block# r|e, disk p):
    let super_block       s  = sread(p),
        ext3_inode       e3i = iread(EXT3_INT_NR, p),
        ext3_ibitmap     e3b = iread(EXT3_IBM_NR, p);

    let int inodes  = e3i.i_size/ISIZE,   // Anzahl der Inodes.
        int ino_rel = int(inodes/iperb);  // zu bestellender Block.
        int ibm_rel = int(inodes/BGSIZE); // zu bestellender Block.
        int ibm_off = inodes % BGSIZE;    // Position in der Bitmap.

    // Vergrößere die Inodetabelle.
    let (abs_block# it, ext3_inode e3i2, disk p2) =
        bgrow(e3i, ino_rel, p);
    if (it == e) then (e, p);
    else
        // Muß die Bitmap auch wachsen?
        if (ibm_rel >= int(e3b.i_size/BSIZE)) then
            // Ja, erweitere die Bitmap.
            let (abs_block# ib, ext3_inode e3b2, disk p3) =
                igrow_bitmap_init(bgrow(e3b, ibm_rel, p2));
            if (ib == e) then (e, p);
            else
                // Passe die Zähler im Superblock an.
                let super_block s2 = igrow_super2(s),
            fi;
        else
            // Nein, passe nur Zahl der Inodes an.
            let super_block      s2 = igrow_super(s),
                disk             p3 = p2;
```

```
        fi;
        // Schreibe Superblock und Inodes auf das Medium zurück.
        let disk p4 = igrow_update_all(s2, e3i2, e3b2, p3);
        (r, p4);
    fi;
```

Die beiden Varianten der Funktion `igrow_super()` sind trivial. Beide Versionen erhöhen die gesamte Anzahl der Inodes im Superblock und die Anzahl der freien Inodes jeweils um den Wert `I_PER_B`. Die eine Variante erhöht außerdem die Anzahl der Zähler für die einzelnen Blockgruppen, `s_free_inodes_max`, um eins und setzt den Wert des neuen Zählers auf `I_PER_B`. `igrow_update_all()` ist schließlich eine Ansammlung von `swrite()`– und `iwrite()`–Aufrufen zur Erzeugung eines neuen Disk–Zustands.

### 4.3.8 Integritätsbedingungen

Die Integritätsbedingungen aus Abschnitt 4.2.10 gelten unverändert auch im ext3–Dateisystem weiter, da sie entsprechend formuliert wurden[18]: In den Integritätsbedingungen wurden die Funktionen `ibmap()` und `imap()` verwendet, um die Datenblöcke zu lokalisieren, die Inodebelegungsbitmap und Inodetabelle darstellen. Mit den veränderten Funktionen des ext3–Dateisystems werden ohne Änderung der Formulierung die richtigen Datenblöcke gefunden.

In den Formulierungen aus Abschnitt 4.2.10 wird jedoch auf die nicht mehr existierende Konstante `NINODES` Bezug genommen. An ihrer Stelle ist die aktuelle Anzahl von Inodes im Dateisystem einzusetzen, die im Superblock gefunden werden kann (`sread(p).s_inodes_count`).

Da Inodebelegungsbitmap, Blockbelegungsbitmap[19] und Inodetabelle nun jedoch durch reguläre Dateien repräsentiert werden können, greift die fünfte Integritätsbedingung für das ext2–Dateisystem („Dateien dürfen einander nicht überlappen“) auch für diese Komponenten des Dateisystems. Indem man die Menge `inodes` also um diese Dateien erweitert, kann man die Integritätsbedingung 4.4 vereinfachen, die die nicht durch Inodes repräsentierten Komponenten des Dateisystems aufzählt. Sie lautet dann noch:

$$\begin{aligned} \forall \mathtt{i} \in \mathtt{inodes} \quad : \quad & \forall \mathtt{b} \in \mathtt{allblocks(i)} : \\ & \mathtt{b} \geq \mathtt{FIRSTBLOCK} \\ \wedge \quad & \mathtt{b} < \mathtt{NBLOCKS} \\ \wedge \quad & \mathtt{b} \notin \{\mathtt{smap(g)} | 0 \leq \mathtt{g} < \mathtt{NGROUPS}\} \\ \wedge \quad & \mathtt{b} \notin \{\mathtt{kopf(dmap(g,0))} \ldots \mathtt{kopf(dmap(g,NGROUPS))} | 0 \leq \mathtt{g} < \mathtt{NGROUPS}\} \end{aligned} \tag{4.7}$$

Die Integritätsbedingungen für Inodes und Verzeichnisse bleiben unverändert.

[18] Leider wurden die Integritätsbedingungen im Code des `fsck`–Programms des ext2–Dateisystems nicht so formuliert, sodaß alle Bezugnahmen auf geänderte Datenstrukturen einzeln angepaßt werden mußten.

[19] Es wird eine korrekte Inode für die Blockbelegungsbitmap beim Anlegen des Dateisystems konstruiert, aber später niemals mehr verwendet oder verändert. Der Zugriff auf die Blöcke der Blockbelegungsbitmap erfolgt weiterhin über die Zeiger in den Deskriptoren.

# Kapitel 5

# Implementierung eines Dateisystems mit variabler Anzahl Inodes

## 5.1 Entwicklungsumgebung

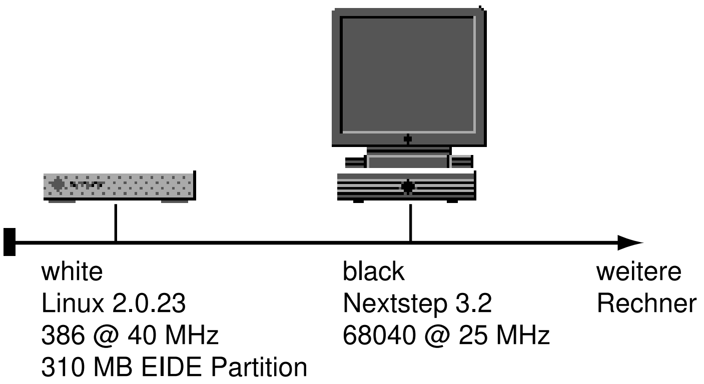



Abbildung 5.1: Verwendete Konfiguration: Test und Entwicklung fanden auf dem Linux–System `white` statt. Zur Absicherung gegen Datenverluste wurden alle wichtigen Daten auf der Workstation `black` gesichert. Von dieser Maschine aus erfolgte auch das Debugging.

Die Entwicklung des neuen Dateisystems erfolgte auf der Basis der Quelltexte für die Linux–Variante des ext2–Dateisystem von Rémy Card [Ca93]. Zum Einsatz kam ein Standard–PC mit Intel 80386DX/40 Prozessor, 16 MB RAM und einer 800 MB EIDE Harddisk unter Linux 1.2.13 (Maschine `white`). Im Verlauf der Entwicklung wurde die Betriebssystemversion über mehrere Zwischenstände auf Linux 2.0.23 erhöht, um mit der Fortentwicklung des Linux–Betriebssystems Schritt zu halten. Da die Entwicklung eines Dateisystems wie jede Arbeit am Kernel immer die Möglichkeit eines Systemverlustes nach sich zieht, erfolgt das Debugging und die Speicherung aller wichtigen Daten auf einer über TCP/IP und Ethernet angeschlossenen NeXT–Workstation `black` unter Nextstep 3.2 (Abbildung 5.1). Auf dem Linux–Rechner wird die normale GNU Entwicklungsumgebung bestehend aus den GNU–Programmen gcc, gas, gld, gdb usw. eingesetzt.

```
root@white:~# df --print-type
Filesystem Type  1024-blocks  Used Available Capacity Mounted on
/dev/hda1  umsdos    50756   35310    15446     70%   /dosc
/dev/hda3  ext2     247871  191265    43805     81%   /
/dev/hda5  ext2     179895  138394    31902     81%   /var/spool/news
/dev/hda6  ext3     307851      13   291939      0%   /scratch
black:/    nfs     1518733 1092951   273908     80%   /Net/black
black:/v   nfs     1023901  590702   382003     61%   /Net/black/v
```

Abbildung 5.2: Aufteilung der verwendeten Dateisysteme: Entwicklung und Test des neuen `ext3`–Dateisystems erfolgten auf der etwas über 300 MB großen Partition `/dev/hda6`.

Auf dem Linux–Rechner wird als Produktions–Dateisystem das unmodifizierte ext2–Dateisystem eingesetzt. Die Modifikation dieses Dateisystems zu einem Dateisystem mit variabler Anzahl von Inodes erfolgt unter dem Arbeitstitel `ext3`. Zur Entwicklung und zum Testen steht eine rund 300 MB große Partition zur Verfuegung (In Abbildung 5.2 als `/scratch` angemeldet). Das Entwicklungssystem verfügt außerdem noch über eine Boot– und Rootpartition, auf der sich das Betriebssystem befindet, und einen rund 170 MB großen USENET–Newsspool. Die Arbeits– und Homeverzeichnisse aller Benutzer werden über NFS von der Nextstation importiert, denn der Entwicklungsrechner verfügt über keine eigenen Einrichtungen zur Datensicherung.

Auf der Maschine befindet sich aus historischen Gründen noch eine Partition mit MS-DOS. Die in Abbildung 5.2 nicht gelistete Partition `/dev/hda2` (16 MB) wird als Swapspace verwendet, die ebenfalls nicht gelistete Partition `/dev/hda4` ist die Containerpartition für die sogenannten *erweiterten* Partitionen `/dev/hda5` und `/dev/hda6`, eine aus historischen Gründen notwendige Eigenheit des auf PCs gebräuchlichen Partitionssystems.

## 5.2 Vorgehensweise

Ziel dieser Arbeit war es, ein Dateisystem mit einer dynamischen Anzahl von Inodes zu entwickeln. Zwei alternative Vorgehensweisen kamen in Frage: Simulation eines Dateisystems mit Hilfe eines NFS–Servers oder die Entwicklung und Integration des Dateisystems in den Kernel.

### 5.2.1 Dateisystemsimulator als NFS–Server

Es ließe sich ein NFS–Server programmieren, der mit Hilfe von mehreren in einem existierenden Dateisystem angelegten Dateien Festplatten simulieren und auf diesen ein Dateisystem betreiben könnte. Ein möglicher Ausgangspunkt für diese Entwicklung wäre dabei der NFS–Server von Linux, der als Usermode–Programm mit Systemverwalterrechten implementiert ist.

Zur Realisierung müßte jedoch der gesamte Komplex von Gerätetreiber, Buffer Cache und Dateisystem (Abbildung 5.3) im Usermode simuliert werden. Gerätetreiber (*device driver*),

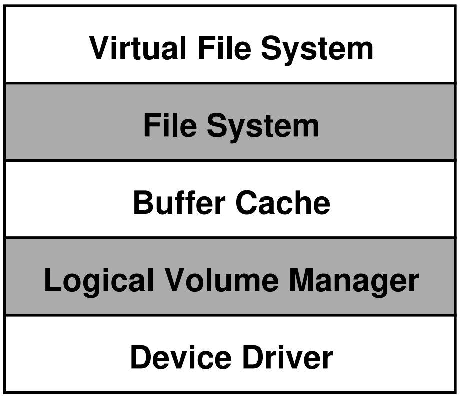



Abbildung 5.3: Gliederung der Abhängigkeiten zwischen den verschiedenen Treiberschichten für ein Dateisystem. An den grau unterlegten Schichten waren Arbeiten geplant.

Blockdepot (*buffer cache*) und Dateisystemumschaltung (*virtual file system*) wären konventionell zu implementieren, während im Bereich Dateisystem (*file system*) und im neu einzuführenden Speicherplatzmanagment (*logical volume manager*) neue Ideen verwirklicht werden könnten.

Die großen Vorteile dieses Ansatzes wären auf den ersten Blick eine leichtere Implementierung und eine vereinfachte Fehlersuche. Außerdem würde man erwarten, daß die Entwicklungszyklen für einen solchen Server kürzer sind als beim Übersetzen und Binden eines neuen Kernels und daß Programmierfehler während der Entwicklung keine Systemabstürze nach sich ziehen.

Eine weitergehende Analyse zeigt jedoch schnell, daß dieser Ansatz eine Reihe von Problemen mit sich bringt:

- Um eine realistische Simulation eines Dateisystems zu erreichen, müßte im Usermode ein sehr großer Teil der Kernelumgebung nachprogrammiert werden, d.h. die in Abbildung 5.1 weiß unterlegten Teile des Kernels müßten ebenfalls realisiert werden, ohne neue Aspekte zu Tage zu fördern.

- Im Usermode sind I/O–Operationen synchron, so daß viele der Race–Conditions, wie sie im Kernelmode auftreten können, hier nicht erzeugt werden. Um realistische Bedingungen herzustellen, müßten die Ein– und Ausgabeoperationen asynchron ablaufen, was den Code stark verkomplizieren würde.

- Nur ein vergleichsweise kleiner Teil des Arbeitsaufwandes liegt in der Entwicklung des Dateisystems selbst. Der größte Teil des Zeitaufwandes ist für die Entwicklung von Usermode–Werkzeugen zum Anlegen, Debuggen und Pflegen von Dateiensystemen zu veranschlagen. Diese Werkzeuge müßten bei einer vollkommenen Neuentwicklung ebenfalls neu geschaffen werden.

Daraus ergibt sich, daß die Entwicklung eines regulären Dateisystems im Kernelkontext effektiver ist.

### 5.2.2 Dateisystem als Kernelmodul

Der andere Ansatz ein Dateisystem im Kernelkontext zu entwickeln, wirkt auf den ersten Blick aufwendiger als ein Programmierprojekt im Usermode. Es ist jedoch so, daß die Entwicklung von Kernelcode für Linux weniger aufwendig ist als für herkömmliche UNIX–Betriebssysteme: Der Linux–Kernel erlaubt das Laden und Entladen von Modulen zur Laufzeit. Module können unabhängig vom laufenden Kernel übersetzt werden, und sie lassen sich ohne Systemneustart laden und testen, was zu angenehm kurzen Test– und Änderungszyklen führt. Module laufen zwar ohne Speicherschutz im Kernelmode und können so bei Programmierfehlern Systemdatenstrukturen zerstören, aber der weitaus häufigste Programmierfehler (Dereferenzierung von Nullzeigern) wird abgefangen: Linux bildet auch im Kernelmode die Speicherseite 0 nicht ab und ist in der Lage, das Modul zu identifizieren, das diesen Seitenfehler verursacht hat. Greift ein Modul also auf Speicher in der Nullseite zu, bleibt Linux nicht wie ein konventionelles UNIX mit einer Panikmeldung stehen, sondern der Kernel trennt sich vom verursachenden Modul und läuft weiter. Die Analyse der so erzeugten Fehlermeldungen wird dadurch vereinfacht, daß die Module Symboltabellen exportieren[1], die über ein spezielles Kernelinterface zur Verfügung stehen und bei der Lokalisierung der fehlerhaften Funktion sehr nützlich sind.

Das Dateisystem wurde daher auf der Basis des ext2–Dateisystemcodes im Kernelmode entwickelt. Der ext2–Dateisystemcode wurde analysiert (siehe Abschnitt 5.4) und seine Einbettung in die Linux–Kernelumgebung dokumentiert (siehe Abschnitt 5.3). Aus dem Code für das ext2–Dateisystem wurde ein zunächst funktionsgleiches Dateisystem mit der Kennung `ext3` geschaffen, das durchgehend mit Debugcode versehen wurde. Diese Ausgangsbasis konnte dann schrittweise auf der Grundlage der in Kapitel 4 vorgestellten Überlegungen modifiziert werden (siehe Abschnitt 5.5). Parallel dazu wurde der Code der ext2–Werkzeuge analysiert und dokumentiert (siehe Abschnitt 5.6) und für die Zwecke des neuen Dateisystems verändert.

## 5.3 Die Kernelumgebung für Dateisysteme in Linux

Der Linux–Kernel ist in der Lage, mit einer Vielzahl von Dateisystemtypen umzugehen (Abbildung 5.4) und so Daten anderer Betriebssysteme mitzunutzen. Alle diese Dateisysteme werden im Kernel durch einheitliche Datenstrukturen dargestellt, die eine Verfeinerung des *Vnode Interface* darstellen, wie es von Kleiman in [Kl86] beschreiben worden ist.

Die Idee, die dem *virtual file system* (VFS) nach Kleiman zugrunde liegt, besteht darin, dateisystemunabhängige Datenstrukturen zur Repräsentation von Dateisystemtypen, Superblöcken, Inodes und Dateien zu schaffen. Diese unspezialisierten Datenstrukturen können mittels einer Union um dateisystemspezifische Daten erweitert werden, die nur für einen bestimmten Dateisystemtyp Bedeutung haben.

Jede der Datenstrukturen der VFS–Layer enthält zusätzlich einen Zeiger auf eine Struktur, die nur aus Funktionszeigern besteht. Diese Funktionszeiger definieren den dateisystemspezifischen Satz von Operationen, die auf die betreffenden Datenstruktur anwendbar sind. Das VFS enthält

[1] Dies ist in jedem Fall notwendig, damit der Kernel das Modul zur Laufzeit laden und binden kann.

| Bezeichnung | Erläuterung |
|---|---|
| `msdos` | MS–DOS Dateisystem 1.0 bis 6.2 (8.3 Namenskonvention). |
| `vfat` | MS–DOS Dateisystem 7.0 (lange Dateinamen). |
| `umsdos` | MS–DOS Dateisystem mit simulierten UNIX–Zugriffsrechten für individuelle Dateien. |
| `sysv` | S5FS mit 1 KB Blockgröße (System V, SCO Xenix, Coherent). |
| `isofs` | ISO-9660 CD–ROM Dateisystem mit Rockridge–Extensions für lange Dateinamen und UNIX–Zugriffsrechte. |
| `minix` | Standarddateisystem von Andrew Tanenbaums Minix. |
| `xiafs` | Erweiterung des `Minix`–Dateisystems auf Partitionen größer 64 MB und längere Dateinamen. |
| `ext` | Extended Filesystem, der Vorläufer des `ext2`–Dateisystems. |
| `ext2` | Das Linux–Standarddateisystem. |
| `ufs` | Reimplementation des BSD FFS. |
| `affs` | Amiga Fast Filing System (read only). |
| `hpfs` | OS/2 High Performance File Systems (read only). |
| `ntfs` | Windows NT File System (read only). |
| `proc` | Pseudodateisystem, daß das `/dev/kmem`–Interface zur Abfrage von Systemstatusinformationen ersetzt. |
| `nfs` | Network File System, Version 2. |
| `smbfs` | Server Message Block (DEC Pathworks, OS/2 LanManager, Windows für Workgroups, Windows NT Server). |
| `ncpfs` | Novell Core Protocol (Novell Fileserver 2.x, 3.x). |

Abbildung 5.4: Von Linux 2.0.x unterstützte Dateisystemtypen.

den allgemeinen Code, der mit Superblöcken, Inodes und Dateien umgeht. Wann immer in diesem Kontext dateisystemspezifische Operationen notwendig sind, wird über die mit der Datenstruktur assoziierten Funktionszeiger die zugehörige spezialisierte Funktion aufgerufen. Dieser dateisystemspezifische Code kann sich wiederum der vom Buffer Cache bereitgestellten Funktionen und gemeinsamer Primitive zum Locking und zur Synchronisation bedienen.

Die folgenden beiden Abschnitte geben einen Überblick über den Aufbau der Linux VFS–Layer und die von Dateisystemen genutzte Funktionalität des Buffer Cache, soweit diese für die vorliegende Arbeit von Bedeutung sind. Eine umfassende Darstellung des Linux–Kernels findet sich in [Be94].

### 5.3.1 Das Interface zum VFS

Das Linux *virtual file system* (*VFS*) Interface ist eine Indirektionsebene, die von allen Systemaufrufen verwendet wird, die mit Dateien oder Inodes arbeiten. Der Teil der Dateioperation, der vom verwendeten Dateisystemtyp unabhängig ist, wird in der VFS–Layer selbst abgehandelt. Für typabhängige Operationen wird je nach Dateisystemtyp, auf dem die benutzte Datei oder Inode abgelegt ist, der passende Code des Zieldateisystems aufgerufen.

Das VFS definiert eine `struct inode`, die eine vom Dateisystemtyp unabhängige Repräsentation einer Inode im Speicher ist. Diese Datenstruktur enthält ein Zeigerfeld `i_op` auf eine

```
struct inode {
    ...
    struct inode_operations *i_op;
    struct super_block *i_sb;
    ...
    union {
        struct pipe_inode_info pipe_i;
        ...
        struct ext2_inode_info ext2_i;
        struct msdos_inode_info msdos_i;
        ...
        struct socket socket_i;
    } u;
};

struct file {
    ...
    loff_t f_pos;
    ...
    struct inode *f_inode;
    struct file_operations *f_op;
};
```

Abbildung 5.5: Gekürzte Darstellung einer VFS–Inode und eines VFS–File aus der Include–Datei `<linux/fs.h>`. Die Inode–Struktur enthält einen Zeiger auf die `inode_operations` sowie spezifische Datenfelder für den jeweiligen Dateisystemtyp. Die File–Struktur enthält ebenfalls einen Zeiger auf die Menge der auf diese Struktur anwendbaren Operationen und einen Rückwärtsverweis auf die zur Datei gehörende Inode.

`struct inode_operations` sowie eine `union` aller Datenstrukturen, die die jeweils typabhängigen Daten der Inode enthalten (Abbildung 5.5). Die Struktur `inode_operations` besteht aus Zeigern auf alle Funktionen, die `struct inode` manipulieren. Je nachdem, auf welchem Dateisystemtyp die Inode abgelegt ist, zeigt der `i_op`–Zeiger in der Inode auf eine andere `inode_operations`–Struktur, die für den Dateisystemtyp und den Dateityp (regular, directory, . . . ) charakteristisch ist.

Alle wichtigen Strukturen der VFS–Schicht verfügen über derartige Zeiger auf Strukturen mit Funktionszeigern: `struct file` (Abbildung 5.5) verweist über den Zeiger `f_op` auf eine Struktur `file_operations` mit Zeigern auf Funktionen zur Manipulation der `file`–Struktur; in der Superblock–Struktur (Abbildung 5.6) wird auf `super_operations` verwiesen. Da in Zusammenhang mit einem `struct file_system_type` nur eine einzige Funktion (`read_super`) typabhängig ist (ebenfalls Abbildung 5.6), existiert hier keine spezielle Datenstruktur mit Zeigern, sondern es wird direkt auf diese eine Funktion verwiesen.

Die verschiedenen Typen der geladenen Dateisysteme werden in einer linearen Liste von `file_system_type`–Strukturen verwaltet. Die VFS–Schicht ist über die zwei Funktionen

```
struct super_block {
    ...
    struct file_system_type *s_type;
    struct super_operations *s_op;
    union {
        struct minix_sb_info minix_sb;
        struct ext_sb_info ext_sb;
        ...
        struct ufs_sb_info ufs_sb;
    } u;
};

struct file_system_type {
        struct super_block *(*read_super) (struct super_block *,
            void *, int);
        const char *name;
        int requires_dev;
        struct file_system_type *next;
};
```

Abbildung 5.6: Gekürzte Darstellung eines VFS–Superblocks und eines VFS–Filesystemtyps aus der Include–Datei `<linux/fs.h>`. Der Filesystemtyp verfügt mit dem Funktionszeiger `read_super` über die Möglichkeit, neue Instanzen eines Dateisystems seines Typs zu erzeugen. Diese Instanzen verweisen über den Zeiger `s_type` zurück zu ihm.

`register_filesystem()` und `unregister_filesystem()` in der Lage, diese Liste zur Kernel–Laufzeit zu vergrößern oder zu verkleinern[2]. Mit Hilfe weiterer nicht vom Typ des Dateisystems abhängiger Funktionen ist es möglich, die dem Kern bekannten Dateisystemtypen aufzuzählen oder nach bestimmten Dateisystemtypen suchen zu lassen.

Wenn dem Kernel ein Dateisystemtyp bekannt ist, kann ein Dateisystem dieses Typs aktiviert werden, indem sein Superblock mit Hilfe der durch den Funktionszeiger `read_super` bezeichneten Funktion eingelesen wird. Das Ergebnis dieses Funktionsaufrufes ist ein initialisierter Superblock, der neben typunabhängigen Informationen auch die für diesen Dateisystemtyp spezifischen Informationen enthält. Linux verwaltet die Superblöcke aller angemeldeten Dateisysteme unabhängig von ihrem Typ in einem Feld `super_block`. Der Dateisystemtyp eines Superblocks ist über den in ihm enthaltenen Rückwärtsverweis `s_type` leicht feststellbar. Ein Dateisystem kann deaktiviert werden, indem man über den Zeiger `put_super` die spezifische Deaktivierungsfunktion aufruft.

In einem Dateisystem können existierende Inodes aktiviert und deaktiviert werden. In der Operationsstruktur des Superblocks sind zu diesem Zweck die Zeiger auf zwei Funktionen

[2] Für statisch gebundene Kernel wäre dies unsinnig, aber Linux ist in der Lage, zur Laufzeit Kernelmodule zu laden und zu binden. Auf diese Weise kann sich die Liste der bekannten Dateisysteme zur Laufzeit ändern.

read_inode und put_inode vorhanden, die Inodes laden bzw. in das Dateisystem zurückschreiben. Neue Inodes können je nach gewünschtem Inodetyp entweder mit create, mit mkdir oder mit mknod erschaffen werden. Die Zeiger auf die Funktionen zur Erzeugung neuer Inodes gehören allerdings nicht zum Superblock, sondern stehen in der Operationsstruktur der Inode: Beim Anlegen neuer Inodes wird über die Inode des Elternverzeichnisses die Funktion ausgewählt, mit der die neue Inode angelegt werden soll, d.h. beim Anlegen einer neuen Datei wird dir->i_op->create usw. aktiviert.

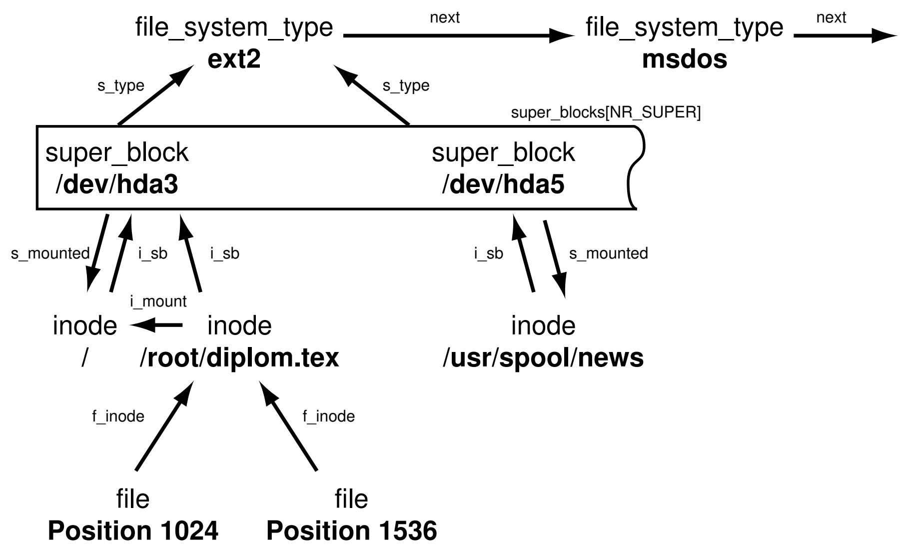



Abbildung 5.7: Verkettung der VFS–Strukturen: Die Datenstrukturen der VFS–Schicht verweisen „stromaufwärts“ aufeinander. Die Anzahl der Inode– und File–Strukturen ist dabei in Linux nicht statisch, die Anzahl der Superblocks wird jedoch durch die Größe des Feldes super_blocks begrenzt. Die Verkettung der Inode–Strukuren bzw. File–Strukturen untereinander ist aus Platzgründen nicht dargestellt.

Wiederum kann über einen Rückwärtsverweis von einer Inode aus der zu dieser Inode gehörende Superblock erreicht werden (Zeiger i_sb in der Inode–Struktur). Jede Inode enthält außerdem einen Verweis auf die Inode des Mountpoints des Dateisystems (Zeiger i_mount in der Inode–Struktur), d.h. auf die Wurzel des Teildateisystems.

Die „Nutzlast“ des struct file besteht in der Hauptsache aus dem Datenfeld f_pos, das den Dateizeiger in einer Datei realisiert. Jede Datei verweist außerdem über den Rückwärtszeiger f_inode auf ihre zugehörige Inode und über den Zeiger f_op auf ihre Dateioperationen. Für diese Arbeit ist struct file nicht weiter von Bedeutung.

Die Verzeigerung der Strukturen untereinander ist in 5.7 graphisch dargestellt. Aus Gründen der Übersichtlichkeit wurden jedoch die Zeiger zur Verwaltung der Inode– und Dateitabelle[3] nicht eingezeichnet.

[3] In Linux keine Tabellen, sondern dynamische Strukturen.

### 5.3.2 Von Dateisystemen verwendbare Kernelfunktionen

#### 5.3.2.1 Buffer Cache

Linux verfügt über eine Reihe von generischen Funktionen zur Verwaltung des Buffer Cache. Diese Funktionen müssen von einem Dateisystem verwendet werden, um Datenblöcke zu lesen und zu schreiben. Der Aufbau und die Funktionsweise dieser Funktionen ähnelt naturgemäß denen anderer UNIX–Versionen, speziell dem Cache–Managment von BSD [BSD89]. Da Linux jedoch nicht auf offiziellen UNIX–Quelltexten aufbaut, bestehen in einigen Details Unterschiede in Namen oder Wirkung dieser Funktionen.

Puffer werden in Linux durch einen `struct buffer_head` beschrieben, der Felder für die Gerätenummer, die Blocknummer und die Größe des im Puffer enthaltenen Datenblocks enthält. Eine Reihe von Statusbits informiert über den aktuellen Status des Puffers, und Linux stellt Funktionen zur Manipulation und Prädikate zur Abfrage dieser Statusbits bereit. Neben einem Zeiger `b_data` auf die vom `buffer_head` verwalteten Daten existieren noch eine Reihe weiterer Zeiger zur Verwaltung des Buffer Cache, wie sie in [Ba86] und [BSD89] beschrieben ist.

Die für die Entwicklung von Dateisystemen wesentlichen Zustände eines Puffers sind *locked* (gesperrt), *dirty* (verändert) und *uptodate* (aktualisiert). Ein gesperrter Puffer wird gerade von einem anderen Prozess manipuliert; auf ihn kann nicht zugegriffen werden. Ein veränderter Puffer muß vor der Freigabe seiner Daten auf das Medium zurückgeschrieben werden, und ein Puffer ist aktualisiert, wenn seine Daten im Speicher mit denen auf der Platte synchronisiert sind.

Mit Hilfe der Funktionen `bread()` und `breada()` ist es möglich, unter Angabe eines Paares von Gerätenummern und einer Blocknummer einen Puffer mit Daten anzufordern. Ist der geforderte Puffer noch im Cache vorhanden und aktuell, wird diese Version verwendet. Andernfalls wird eine Leseanforderung zum Medium generiert. Das Resultat ist ein aktualisierter, entsperrter, unveränderter Puffer. `breada()` hat außerdem noch den Seiteneffekt, Leseanforderungen für bis zu maximal 16[4] weitere Leseanforderungen für Puffer in die Warteschlange zu stellen. Nicht mehr benötigte Puffer können mit `brelse()` freigegeben werden. Ist der Puffer dabei als verändert markiert, sorgt das Betriebssystem für ein asynchrones, verzögertes Zurückschreiben der Daten auf das Medium, bevor der mit dem Puffer assoziierte Speicher freigegeben wird.

Mit der Funktion `mark_buffer_uptodate()` kann der gleichnamige Zustand eines Puffers manipuliert werden, mit dem Prädikat `buffer_uptodate()` ist der Zustand abfragbar.

Der *dirty*–Zustand wird über einen etwas unterschiedlich aufgebauten Satz von Funktionen gesteuert: `mark_buffer_dirty()` markiert einen Puffer als verändert und stellt ihn in die entsprechende Warteschlange für zu schreibende Puffer. Der anzugebende Parameter ist die Schreibverzögerungszeit für diesen Puffer. Die inverse Operation `mark_buffer_clean()` wird im Dateisystem nicht benötigt. Mit Hilfe des Prädikates `buffer_dirty()` kann der Zustand des Puffers abgefragt werden.

[4]Ein einstellbarer Kernelparameter mit dem Standardwert 16.

Puffer können mit Hilfe des `locked`–Statusbits für exklusiven Gebrauch gesperrt werden. Die Funktion `lock_buffer()` versucht, das Lock für einen Puffer zu erhalten, kann dabei aber den anfordernden Prozeß schlafen legen. `unlock_buffer()` gibt ein so erhaltenes Lock wieder auf und weckt alle anderen Prozesse wieder auf, die auf diesen Puffer warten. Der Zustand eines Puffers läßt sich mit `buffer_locked()` abfragen. `wait_on_buffer()` legt einen Prozeß schlafen, solange ein Puffer gesperrt ist; diese Funktion wird im Dateisystem verwendet, um `lock_buffer()` und um synchrone Schreibzugriffe auf das Medium zu realisieren.

Um solche synchronen Schreibzugriffe zu erzeugen, ist es notwendig, mit der Funktion `ll_rw_block()` eine Ein-/Ausgabeanforderung an den Gerätetreiber zu generieren und mit `wait_on_buffer()` auf das Entsperren des betroffenen Puffers zu warten.

#### 5.3.2.2 Inode Cache und Superblock

Analog zu den Sperrfunktionen für Puffer existieren Sperrfunktionen für Inodes mit entsprechender Bedeutung: `lock_inode()`, `unlock_inode()` und `wait_on_inode()`. Alle diese Funktionen bearbeiten `struct inode`–Strukturen, also die dateisystemunabhängige Repräsentation von Inodes im Speicher. Solche Strukturen werden von der parameterlosen Funktion `get_empty_inode()` geliefert. Bereits existierende Inodes können mit der Funktion `iget()` eingelesen werden, die von der Hilfsfunktion `read_inode()` Gebrauch macht (siehe Abschnitt 5.4.2). Inodes werden mit `iput()` wieder freigegeben.

Die Attribute einer Datei, die in einer Inode gespeichert sind, können mit `inode_setattr()` verändert werden. Die Funktion sorgt dafür, daß nur die gewünschten Attribute geändert werden und daß auch die Kopie der Inode auf dem Medium angepaßt wird. Eine Überprüfung, ob die zur Änderung benötigten Rechte vorhanden sind, erfolgt nicht in dieser Funktion, sondern gesondert mit dem Prädikat `inode_change_ok()`, das explizit vorher aufgerufen werden muß.

Schließlich existiert auch für den Superblock ein Trio von Sperrfunktionen `lock_super()`, `unlock_super()` sowie `wait_on_super()` mit den erwarteten Eigenschaften.

## 5.4 Aufbau und Funktionsweise des ext2–Dateisystems

### 5.4.1 Aktivierung und Deaktivierung des Dateisystems

Das ext2–Dateisystem kann entweder statisch in den Kernel eingebunden oder zur Laufzeit als Modul geladen werden. Die Modulversion unterscheidet sich von der statisch gebundenen Version nur durch zwei zusätzliche parameterlose Funktionen `init_module` und `cleanup_module`, die jeweils beim Laden und Entladen des Moduls aktiviert werden, sowie durch die Verwendung eines Zählers, der die Anzahl der aktiven Verwendungen des Moduls mitzählt. In jedem Fall registriert das Dateisystem entweder beim Start des Kernels oder beim Laden des Moduls den Dateisystemtyp `ext2` und die zugehörige Funktion `read_super` zum Laden eines Superblocks beim Kernel.

Diese Funktion wird dann aktiviert, wenn ein `mount`–Systemaufruf versucht, ein Dateisystem dieses Typs zu mounten. Sie erhält eine Superblock–Struktur als Parameter, in der praktisch nur die Gerätenummern und Mount–Flags ausgefüllt sind. Ihre Aufgabe ist es, die dateisystemspezifischen Optionen des `mount`–Systemaufrufes zu parsen, die Kopie eines Superblocks von der Festplatte zu lesen und anhand der vorgefundenen Informationen die übergebene Superblock–Struktur des Dateisystems im Speicher auszufüllen. Zusammen mit dem Superblock werden die in den unmittelbar folgenden Blöcken gespeicherten Blockgruppendeskriptoren eingelesen. Die Buffer Cache–Blöcke, die den Superblock und die Deskriptoren enthalten, bleiben so lange aktiv, wie das Dateisystem angemeldet ist.

Wenn die Mountoptionen dies vorsehen, werden Plausibilitätsprüfungen am Dateisystem vorgenommen: Die Belegungsbitmaps für Inodes und Blöcke werden mit den statistischen Informationen abgeglichen, und es wird sichergestellt, daß in jedem Blockgruppendeskriptor die Zeiger auf die Bitmapblocks und die Inodetabelle plausible Werte haben. Während der Initialisierung ist der Superblock gesperrt, aber die Funktion hinterläßt den fertig initialisierten Superblock entsperrt und mit aktiver Root-Inode. Der Gebrauchszähler für das ext2–Dateisystem wird durch diese Funktion um Eins erhöht.

Die Deaktivierung eines Dateisystem erfolgt über den Aufruf der Funktion `put_super` aus der VFS-Layer heraus. Dazu wird die aktiv gehaltene Festplattenkopie des Superblocks auf den aktuellen Stand gebracht und der Block als *dirty* markiert, damit er baldmöglichst zurückgeschrieben wird. Die Buffer Cache–Blöcke, die die Deskriptoren, möglicherweise noch aktive Bitmap–Blöcke und den Superblock enthalten, werden freigegeben, und der Gebrauchszähler wird wieder erniedrigt. Die aktiv gehaltene Inode des Root–Verzeichnisses braucht von `put_super` nicht freigegeben zu werden: Die VFS–Schicht deaktiviert diese Inode mit `iput()` schon vor dem Eintritt in die Funktion.

Es ist möglich, den `mount`–Systemaufruf mit der Option `REMOUNT` aufzurufen. In diesem speziellen Fall aktiviert die VFS–Layer die dateisystemspezifische Remount–Funktion `remount_fs`. Ihre Hauptaufgabe besteht darin, die Mountoptionen neu zu parsen und in den aktiven Superblock zu übertragen. Dabei kann der Status des Dateisystems von „beschreibbar“ auf „nur lesen“ oder umgekehrt wechseln. Da in der Festplattenkopie des Superblocks vermerkt wird, ob das Dateisystem sauber abgemeldet wurde, müssen diese Sonderfälle abgefangen und die Festplattenkopie des Superblocks mit den korrekten Flags auf das Medium zurückgeschrieben werden.

Die statistischen Informationen über das Dateisystem werden von `ext2_statfs` zurückgeliefert, wenn diese Funktion über den Systemaufruf `statfs` aktiviert wird.

### 5.4.2 Aktivierung und Deaktivierung von Inodes

Die Funktion `iget` der VFS–Schicht ruft die dateisystemspezifische Funktion `read_inode` zum Ausfüllen einer Inode–Struktur auf. Die Inode ist dabei beim Eintritt in `read_inode` schon gesperrt. Zum Einlesen der Inode muß die Lage der Inode im Dateisystem bestimmt (`imap`) und der Block, der die Inode enthält, geladen werden. Danach können die Informationen aus der On–Disk–Kopie der Inode in den dateisystemunabhängigen `struct inode` umkopiert und der Block wieder freigegeben werden. Zugleich wird das Feld mit den `inode_operations` der Inode abhängig vom Dateityp korrekt belegt. Das ext2–Dateisystem unterscheidet dabei die folgenden Dateitypen:

| Typ | Operationen |
|---|---|
| regular | `ext2_file_inode_operations` |
| directory | `ext2_dir_inode_operations` |
| symlink | `ext2_symlink_inode_operations` |
| char special | `chrdev_inode_operations` |
| block special | `blkdev_inode_operations` |
| fifo | `fifo_inode_operations` |

Das Dateisystem muß selbst also nur die Dateitypen für gewöhnliche Dateien, Verzeichnisse und Verweise implementieren, kann jedoch die Funktionalität für Gerätedateien und Pipelines aus der VFS–Layer übernehmen, ohne eigenen Code zur Verfügung zu stellen.

Die Deaktivierung einer Inode erfolgt aus der Funktion `iput` der VFS–Schicht heraus, sobald der Gebrauchszähler `i_count` der Inode Null erreicht. Die Funktion ruft bei gesperrter Inode die dateisystemspezifischen Funktionen `put_inode` und `write_inode` in dieser Reihenfolge auf.

`ext2_put_inode` wird in den meisten Fällen nur die vorbestellten Blöcke freigeben (siehe Abschnitt 5.4.4). Falls der Namenszähler `i_nlink` der Inode jedoch Null erreicht, gilt die Datei als gelöscht und muß vollständig freigegeben werden. Die aktuelle Zeit wird als Löschzeit `i_dtime` in die Inode eingetragen, und sie wird auf das Medium zurückgeschrieben[5]. Über `ext2_truncate` werden die Datenblöcke der Inode in der Belegungsbitmap und in den Statistiken der Deskriptoren freigegeben und mit `ext2_free_inode` wird die Inode selbst ebenfalls in Bitmaps und Statistiken als frei markiert.

`ext2_write_inode` ist die zu `ext2_read_inode` symmetrische Funktion. Sie bestimmt ebenfalls die Lage der Inode im Dateisystem (`imap`) und lädt den Block, der die Inode enthält. Nachdem sie die Daten aus der Inode–Struktur in die für das Medium passende Form gebracht hat, markiert sie den Block als `dirty` und sorgt bei entsprechend gesetzten Mount–Optionen für ein synchrones Zurückschreiben der Änderungen.

[5] Die Größe der Datei wird *nicht* auf Null gesetzt, und die Zeiger auf die Datenblöcke der Inode werden *nicht* angetastet, obwohl die Datenblöcke als frei markiert werden! Auf diese Weise ist es möglich, die Datei wiederherzustellen, solange die Datenblöcke der Datei nicht anderweitig belegt werden.

### 5.4.3 Belegung und Freigabe von Inodes

Die Belegung neuer Inodes ist notwendig, wenn in einem Dateisystem neue Dateien angelegt werden. Von allen Inode–Operationen haben nur vier Operationen das Potential, Inodes zu verbrauchen:

- `create`, implementiert durch `ext2_create` zum Anlegen von regulären Dateien. Die Funktion wird im Zuge der Ausführung von `open` und `namei` aufgerufen, wenn eine Datei mit dem Modus `O_CREAT` geöffnet werden soll.
- `mknod`, implementiert durch `ext2_mknod` zum Anlegen von generischen Inodes, darunter auch Pipelines und Gerätedateien, im Zuge der Ausführung des `mknod`–Systemaufrufs.
- `mkdir`, implementiert durch `ext2_mkdir` zum Anlegen von neuen Verzeichnissen.
- `symlink`, implementiert durch `ext2_symlink` zum Anlegen von symbolischen Verweisen.

Das Vorgehen ist in allen vier Fällen relativ ähnlich: Für die neu zu erzeugende Datei wird mittels der Hilfsfunktion `ext2_new_inode` eine neue Inode auf dem Medium alloziert und die zugehörige Inode–Struktur im Speicher erzeugt. Die folgenden Anweisungen manipulieren dann die Verzeichniseinträge, um einen Eintrag für die neue Datei in einem Verzeichnis zu schaffen. Für die Zwecke dieser Arbeit genügt es, die Funktionsweise von `ext2_new_inode` zu diskutieren.

Die Freigabe von Inodes kann durch drei Inode–Operationen erfolgen:

- `rmdir`, implementiert durch `ext2_rmdir` zum Löschen von Verzeichnissen, im Zuge der Ausführung des `rmdir`–Systemaufrufes. Die Inode des zu entfernenden Verzeichnisses wird freigegeben.
- `unlink`, implementiert durch `ext2_unlink` zum Löschen von Verzeichniseinträgen, im Zuge der Ausführung des `unlink`–Systemaufrufs.
- `rename`, implementiert durch `ext2_rename` zum Umbenennen von Dateien, im Zuge der Ausführung des `rename`–Systemaufrufs. Diese Operation löscht nur dann eine Datei, wenn das Ziel der Operation bereits existiert und kein Verzeichnis ist.

Alle diese Operationen verändern lediglich Verzeichniseinträge und passen den Namenszähler `i_nlink` der Inode an. Die Freigabe der Inode erfolgt im Zuge der bereits diskutierten Funktion `iput` jeweils am Ende dieser Systemaufrufe.

Die Funktion `ext2_new_inode` implementiert den Algorithmus `ialloc`. Zunächst werden die Voraussetzungen für die Suche nach einer freien Inode geschaffen: Der Kernel besorgt eine leere Inode–Struktur und sperrt den Superblock. Entsprechend der in Abschnitt 4.2.7 besprochenen Strategie wird nun anhand der Deskriptoren eine Blockgruppe bestimmt, in der die

Inode bevorzugt angelegt werden soll. Mit Hilfe der Belegungsbitmap für Inodes dieser Blockgruppe wird eine freie Inode als belegt markiert, und die Statistikinformationen im Deskriptor der Blockgruppe und im Superblock werden korrigiert. Danach wird die Inode initialisiert und der Superblock wieder entsperrt. Die dazu symmetrische Funktion `ext2_free_inode` wurde bereits diskutiert.

### 5.4.4 Belegung und Freigabe von Blöcken

Neue, leere Blöcke werden benötigt, wenn eine Datei wächst, d.h. in der `write`–Funktion des Dateisystems. In `ext2_file_write()` werden nach Prüfung diverser Vorbedingungen die zu beschreibenden Blöcke in einer Schleife durch den wiederholten Aufruf von `ext2_getblk()` in Puffer geholt. `ext2_getblk()` erfüllt diese Aufgabe durch Anwendung des Algorithmus `bmap` gefolgt von einem `bread()`, sofern ein Mapping bereits existiert. Abhängig von einem Flag *create*, das beim Aufruf von `ext2_getblk()` mitgegeben wird, erweitert diese Funktion das Mapping einer Inode, falls notwendig. Dazu wird ein Zielblock (*goal*) bestimmt, in dessen Nähe vorzugsweise Platz belegt werden sollte und die Funktion *ext2_alloc_block()* mit der Blocknummer dieses Zielblocks aufgerufen.

*ext2_alloc_block()* sieht nach, ob ein vorheriger Aufruf der Funktion eine Vorbestellung von Blöcken bewirkt hat und ob diese Vorbestellung für die angegebene Zielblocknummer richtig liegt (Das Ziel muß unmittelbar vor dem Beginn oder auf dem Beginn der Vorbestellung liegen). Ist dies der Fall, wird kein weiterer Block bestellt, sondern die Anfrage nach einem neuen Block aus der Vorbestellung befriedigt.

Kann die Anforderung eines neuen Blocks dagegen nicht aus der Vorbestellung befriedigt werden, wird eine eventuell existierende Vorbestellung aufgegeben und mit Hilfe der Funktion *ext2_new_block()* ein neuer Block bestellt. Sollte die wachsende Datei eine reguläre Datei sein, wird von `ext2_new_block()` zugleich eine neue Vorbestellung für die auf die Zielblocknummer folgenden Blöcke aufgebaut.

`ext2_new_block()` versucht, mit einer linearen Suche beginnend am Zielblock einen freien Block zu belegen. Zusätzlich versucht die Funktion, weitere zusammenhängende Blöcke (bis zu maximal sieben folgende Blöcke) hinter dem soeben belegten Ziel zu belegen. Diese Blöcke werden als Extent (Startblock, Länge) in der Speicherrepräsentation der Inode als *vorbestellt* vermerkt.

Vorbestellte Blöcke werden freigegeben, wenn die Datei geschlossen wird oder der Zielblock einer Dateierweiterung nicht mehr linear an den vorherigen Aufruf anschließt.

Der Vorbestellungsmechanismus bewirkt, daß `ext2_new_block()` versucht, für reguläre Dateien bis zu acht zusammenhängende Blöcke am Stück zu allozieren. Wachsende Dateien haben auf diese Weise bessere Chancen, in größeren zusammenhängenden Stücken zu wachsen und Fragmentierung zu vermeiden.

Die zu `ext2_new_block()` komplementäre Funktion `ext2_free_blocks()` gibt Blöcke wieder frei. Beide Funktionen versehen dieselben buchhalterischen Aufgaben: Bei gesperrtem Superblock werden die Blockbelegungsbitmaps verändert, die Statistiken in den Deskriptoren auf Stand gebracht und die globale Anzahl freier Blöcke im Superblock korrigiert.

### 5.4.5 Verwaltung der Bitmaps

Blockverwaltung und Inodeverwaltung machen Gebrauch von einer Reihe primitiver Funktionen zur Verwaltung von Bitmaps: `get_group_desc()` zum Auffinden eines Deskriptors, `read_x_bitmap()` zum physikalischen Lesen einer Bitmap und `load_x_bitmap()` zum Unterhalten eines LRU–Caches für `n` Bitmaps[6]. Es existiert weiterhin eine Reihe von primitiven Funktionen zur Arbeiten mit Bitmaps: `set_bit()`, `clear_bit()`, `test_bit()`, `find_first_zero_bit()` sowie `ext2_count_free()` zum Zählen von Freimarkierungen (Nullbits) in einer Bitmap.

## 5.5 Von `ext2` zu `ext3`

### 5.5.1 Erzeugung eines Entwicklungsumfeldes

#### 5.5.1.1 Ausgangsbasis

Der Ausgangspunkt der Entwicklung war der Code des Linux ext2–Dateisystems, wie er in `/usr/src/linux/fs/ext2` vorliegt. Aus diesem Quelltext wurde durch einfache Ersetzung aller Vorkommen von `ext2` durch `ext3` und von `EXT2` durch `EXT3` ein funktionsgleiches Dateisystem erzeugt, das unabhängig vom ext2–Dateisystem übersetzt und geladen werden kann. Zusätzlich waren die für das ext2–Dateisystem spezifischen Includedateien `linux/ext2_fs.h` (alle ext2–spezifischen Informationen), `linux/ext2_fs_i.h` (dateisystemspezifische Erweiterung der Inodestruktur) und `linux/ext2_fs_sb.h` (dateisystemspezifische Erweiterung der Superblockstruktur) zu übernehmen und anzupassen. Auf diese Weise war es möglich, am Dateisystemcode zu entwickeln und parallel ein funktionierendes Dateisystem für die Systempartitionen zu verwenden.

Die VFS–Strukturen zur Darstellung von Inodes und Superblöcken in Linux enthalten dateisystemspezifische Informationen in Form von `unions` (Abbildungen 5.5 und 5.6). Zur Erzeugung eines neuen Dateisystemtyps ist es daher notwendig, die in `linux/fs.h` deklarierten `unions` um den neuen Typ zu erweitern.

Der Linux–Kernel wird normalerweise mit einem Satz hierarchischer Makefiles übersetzt, die den gesamten Kernel und alle Kernelmodule bauen und installieren. Auf diese Weise ist sichergestellt, daß alle am Kernel beteiligten Objektmodule und Bibliotheken konsistent sind und in aktueller Form vorliegen. Für die Entwicklung hat sich dieses Verfahren als zu langsam erwiesen, denn ein solches hierarchisches `make` prüft immer den gesamten Dateibaum der Kernelquellcodes auf Veränderungen durch, auch wenn sichergestellt ist, daß nur an Dateien in einem einzigen Verzeichnis Veränderungen vorgenommen wurden.

Eine unabhängige Übersetzung eines Kernel–Subsystems als zur Laufzeit ladbare Erweiterung ist in Linux normalerweise nicht vorgesehen. Mit einem kleinen Shellscript (Abbildung 5.8), das

[6] `n` ist standardmäßig 16.

```
root@white:~/linux/fs/ext3# cat m
TOPDIR=/usr/src/linux
CFLAGS="-g -D__KERNEL__ -I/usr/src/linux/include -Wall \
     -Wstrict-prototypes -O2 -fomit-frame-pointer \
     -fno-strength-reduce -pipe -m386 -DCPU=386 -DMODULE"
export TOPDIR CFLAGS

make -k $*
```

Abbildung 5.8: Das gezeigte Shellscript erzeugt die notwendige Umgebung zum Aufruf von `make`, wenn man ein Kernelmodul unabhängig vom hierarchischen Übersetzungssystem des Kernels erzeugen möchte. Die Definition des Symbols `MODULE` bewirkt die Erzeugung einer zur Laufzeit ladbaren Kernelerweiterung im Gegensatz zu einem statisch gebundenen Subsystem des Kernels.

die zur Erzeugung von Kernelmodulen notwendigen Umgebungsvariablen und Compileroptionen definiert, ist es trotzdem möglich, Subsysteme unabhängig vom Gesamtkernel zu übersetzen. Eine solche unabhängige Übersetzung macht nur dann Sinn, wenn das Subsystem nicht fester Bestandteil des Kerns sein soll, sondern als Modul geladen werden kann. Die Übersetzungsoption `-DMODULE` bewirkt, daß der für ladbare Module notwendige Code erzeugt wird.

Der Übersetzungsvorgang hinterläßt bei Gelingen die Datei `ext3.o`, die mit dem Linux–Kommando `insmod ./ext3.o` zum laufenden Kernel gebunden werden kann. Ein Übersetzerlauf für das gesamte ext3–Dateisystem nach einem vorhergehenden Aufruf von `make clean` dauert auf der verwendeten Hardware etwas über fünf Minuten im Gegensatz zu einer Durchlaufzeit von mehr als fünfzehn Minuten für einen Aufruf des hierarchischen `Makefile` in `/usr/src/linux` und mehr als drei Stunden für eine komplette Generierung des vollständigen Kernels aus den Quelltexten nach einem `make clean` auf dem gesamten Dateibaum der Kernelquelltexte.

Das Einfügen des Kernelmoduls in den Kernel mit `insmod` bewirkt einen Aufruf der Funktion `init_module()` des Moduls, wo mit `register_filesystem()` der neue Dateisystemtyp `ext3` beim Kernel registriert wird. Die Registrierung des Dateisystemtyps kann durch Ausgabe der Datei `/proc/filesystems`[7] verifiziert werden.

Das so erzeugte Dateisystem unterscheidet sich nur im Namen vom ursprünglichen ext2–Dateisystem. Ein Test, bei dem ein ext2–Dateisystem mit `mount -t ext3` an- und abgemeldet wurde, verlief erfolgreich.

[7] Das `/proc`–Dateisystem ist Linux–spezifisch und ersetzt das bei anderen UNIX–Varianten verwendete `nlist()`/`/dev/kmem`–Interface durch ein Dateisystem mit simulierten Dateien, die Informationen über den Kernelzustand im Klartext enthalten. Im Gegensatz zum Auslesen von `/dev/kmem` brauchen Programme, die `/proc` auslesen, keine Privilegien, und Zugriffskontrolle kann über normale UNIX–Zugriffsrechte an den simulierten Dateien ausgeübt werden.

```
root@white:~/linux/fs/ext3# insmod ./ext3.o
root@white:~/linux/fs/ext3# ls -l /proc/filesystems
-r--r--r--   1 root  root  0 Nov 29 19:16 /proc/filesystems
root@white:~/linux/fs/ext3# cat /proc/filesystems
        ext2
nodev   proc
nodev   nfs
        msdos
        vfat
        umsdos
nodev   smbfs
        ext3
```

Abbildung 5.9: Nach der Registrierung des `ext3`–Kernelmoduls ist der neue Dateisystemtyp dem Kernel bekannt. Durch Ausgabe der simulierten Datei `/proc/filesystems` kann dies überprüft werden.

#### 5.5.1.2 Hilfsmittel zur Fehlersuche

Da Linux über keinen Kerneldebugger verfügt, ist es notwendig, den zu testenden Code zur Fehlersuche mit Kontrollausgaben zu versehen. Linux stellt für diese Zwecke die Funktion `printk()` bereit, die analog zum `printf()` der C–Standardbibliothek Ausgaben vornehmen kann. Um selektiv Kontrollausgaben zuschalten zu können, ohne für jeden Test das Kernelmodul neu übersetzen und laden zu müssen, wurde der Dateisystemcode um ein kleines Debuggingsystem erweitert, für das automatisch Code erzeugt wird, wenn das Symbol `EXT3_DEBUG` bei der Übersetzung definiert ist.

Dieses Debuggingsystem erweitert die dateisystemspezifische Superblockstruktur für das ext3–Dateisystem um einen vier Einträge umfassenden Stack, auf dem die Namen von zu überwachenden Funktionen oder Quelltextdateien abgelegt werden können. Eintrag und Löschung von solchen Namen erfolgen mit Hilfe zusätzlicher `ioctl()`–Aufrufe auf eine beliebige Datei im zu untersuchenden Dateisystem. Ein Hilfsprogramm `ext3_debug` erlaubt es, zur Laufzeit des ext3–Dateisystems Namen von zu überwachenden Funktionen oder Quelltextdateien auf dem Debugstack abzulegen oder zu entfernen.

Im Kernelquelltext des ext3–Dateisystems dient das C–Macro `ext3_debug()` zum Aufruf der Funktion `ext3_do_debug()`. Dabei werden dem Aufruf automatisch der Name der aktuellen Quelltextdatei, die Zeilennummer des Aufrufs und der Name der Funktion, in der der Funktionsaufruf steht, hinzugefügt. Mit Hilfe dieser Informationen und der Namen auf dem Debugstack kann die Funktion entscheiden, ob Kontrollausgaben gewünscht sind oder nicht. In der Folge wurden die Aufrufe aller Kernelfunktionen des ext3–Dateisystems mit Aufrufen von `ext3_debug()` versehen, in denen die Werte aller wichtigen Eingabeparameter ausgegeben werden. Auf diese Weise ist eine selektive Überwachung des laufenden Kernelmoduls möglich. Abbildung 5.11 zeigt Ausgaben des Debugmoduls, wie sie beim Anmelden eines Dateisystems entstehen (gekürzt und umgebrochen). Man erkennt, wie versucht wird, die Inode des Wurzelverzeichnisses (Inodenummer 2) zu lesen. Für diese Inode ergibt sich die relative Blocknummer

```
/* Abfrage: Ist das Debugging eingeschaltet? */
#define EXT3_IOC_GETDEBUG    _IOR('D', 1, long)
/* Einschalten/Ausschalten des Debugsystems. */
#define EXT3_IOC_SETDEBUG    _IOW('D', 2, long)

/* Zu untersuchende Dateien und Funktionen
 * pushen/poppen. */
#define EXT3_IOC_ADDFUNCTION  _IOW('D', 3, char *)
#define EXT3_IOC_DELFUNCTION  _IOW('D', 4, char *)
#define EXT3_IOC_ADDFILE      _IOW('D', 5, char *)
#define EXT3_IOC_DELFILE      _IOW('D', 6, char *)

/* Anwendung:
 * #include <sys/ioctl.h>
 * #include <linux/ext3_fs.h>
 *
 * res = ioctl(fd, EXT3_IOC_SETDEBUG, debuglevel);
 */
```

Abbildung 5.10: Die Steuerung des Debugsystems erfolgt mit Hilfe von `ioctl()`–Aufrufen auf eine beliebige geöffnete Datei im zu untersuchenden Dateisystem. Diese Art der Steuerung war am einfachsten zu implementieren, da das neue Dateisystem ohnehin eine `ioctl()`–Funktion implementieren muß und dies einfacher zu integrieren war als ein neuer Systemaufruf.

0 mit einem Offset von 256 im Block. Zur Umwandlung dieser relativen in eine absolute Blocknummer wird die Inode der Inodetabellendatei (Inodenummer 5) bemüht. Die resultierende absolute Blocknummer ist 4.

### 5.5.2 Aggressive Vorbestellung von Blöcken

Zum Test der Entwicklungsumgebung und zum besseren Verständnis der Funktionsweise des ext2–Dateisystems wurden zwei kleinere Modifikationen am ext2–Dateisystem vorgenommen. Die eine Änderung bezieht sich auf die Strategie bei der Vorbestellung von Blöcken bei wachsenden Dateien; sie wurde für den ext3–Code als sinnvoll angesehen und beibehalten. Die andere Änderung ist die Implementierung der vom BSD FFS bekannten *long seeks*. Beim ext3–Dateisystem ist jedoch nicht garantiert, daß Inodes in der Nähe ihrer Datenblöcke gespeichert sind (siehe Abschnitt 4.1.3). So wurde der Mechanismus zum Wechsel der Blockgruppe für ext3 wieder fallengelassen.

Das ext2–Dateisystem versucht, bei Bestellungen von Blöcken für reguläre Dateien für jeden bestellten Block bis zu sieben weitere Blöcke in Folge vorzubestellen. Weitere Anforderungen von Blöcken werden dann aus dem Pool der vorbestellten Blöcke befriedigt, falls die Bedingungen dafür gegeben sind.

```
root@white:~# mount -t ext3 /dev/hda6 /mnt
root@white:~# dmesg | tail
21:34:52 white kernel: (inode.c, 543):
         ext3_read_inode: inode=2
21:34:53 white kernel: (inode.c, 577):
         ext3_read_inode: iperb=8  rel_block=0 offset=256
21:34:53 white kernel: (inode.c, 143):
         ext3_bmap: inode=5 block=0
21:34:53 white kernel: (inode.c, 581):
         ext3_read_inode: abs_block=4
```

Abbildung 5.11: Zur Vereinfachung der Fehlersuche wurde das ext3–Dateisystem mit einem Debuggingsystem ausgestattet. Das Beispiel zeigt einen Teil des Protokolls der Funktionsaufrufe, die beim Anmelden eines Dateisystems entstehen: Die Lage der Root–Inode (2) in der Inodetabellendatei wird bestimmt (relativer Block 0, Offset 256), und die Inode wird geladen.

Abbildung 5.12 zeigt die Wirkungsweise der Vorbestellung: Auf einem ansonsten leeren Dateisystem wird eine 8 KB große Datei angelegt. Das System belegt die erste frei verfügbare Inode (Inodenummer 17) und sucht dann einen freien Block, um ihn der Datei als Datenblock zuzuweisen. Für die Datei existiert weder eine Vorbestellung (`preallocation miss`), noch sind Datenblöcke vorhanden, die für die Berechnung des Zielblockes der Blockbelegung hilfreich sein könnten. Daher wird als Zielblock der erste Block der Blockgruppe angenommen (`goal=1`), in der die Inode der Datei liegt.

Da es sich bei dem gezeigten Beispiel noch um ein weitgehend unmodifiziertes ext2–Dateisystem handelt, sind die vorderen Blöcke jeder Blockgruppe durch die Inodetabelle der Blockgruppe belegt. Erst Bit 274 der Blockbelegungsbitmap ist frei und wird von `ext3_new_block()` als belegt gekennzeichnet. Da die Belegung für eine reguläre Datei erfolgt, werden zugleich die folgenden sieben Bits als Vorbestellung belegt[8]. Das weitere lineare Wachstum der Datei kann dann zunächst aus der Vorbestellung der Inode bestritten werden, ohne daß weitere Aufrufe von `ext3_new_block()` notwendig sind.

Der Code für die Vorbestellung von Blöcken wurde im ext3–Dateisystem so abgeändert, daß er sehr viel mehr Blöcke zu reservieren versucht und die Größe der Vorbestellung der Größe der Datei anpaßt. Dem liegt die Annahme zugrunde, daß große Dateien, die weiter wachsen, noch sehr viel größer werden. In ext3 gelten für die Vorbestellung folgende Regeln:

- Die Vorbestellung für eine Inode beträgt maximal so viele Blöcke, wie die Datei zum Zeitpunkt der Vorbestellung schon belegt.
- Die Vorbestellung in Blöcken für eine Inode wird immer auf die nächstkleinere Achtergrenze abgerundet.

[8] Es entsteht also ein Extent von acht zusammenhängenden Blöcken, der nicht auf einer Achtergrenze beginnt. In BSD FFS wäre dies nicht möglich gewesen.

```
root@white:~# mount -t ext3 /dev/hda6 /mnt
root@white:~# dd if=/dev/zero of=/mnt/x bs=1k count=8
root@white:~# dmesg | head
20:42:08 white kernel: (ialloc.c, 444): ext3_new_inode:
    allocating inode 17
20:42:08 white kernel: (inode.c, 215): inode_getblk:
    hint = 0, goal = 1.
20:42:08 white kernel: (inode.c, 116): ext3_alloc_block:
    preallocation miss (7/9).
20:42:08 white kernel: (balloc.c, 287): ext3_new_block:
    goal=1.
20:42:09 white kernel: (balloc.c, 431): ext3_new_block:
    found bit 274
20:42:09 white kernel: (balloc.c, 455): ext3_new_block:
    Preallocated a further 7 bits.
20:42:09 white kernel: (inode.c, 215): inode_getblk:
    hint = 275, goal = 275.
20:42:09 white kernel: (inode.c, 98): ext3_alloc_block:
    preallocation hit (8/10).
20:42:09 white kernel (inode.c, 215): inode_getblk:
    hint = 276, goal = 276.
20:42:09 white kernel: (inode.c, 98): ext3_alloc_block:
    preallocation hit (9/11).
```

Abbildung 5.12: Wächst eine reguläre Datei, versucht die Funktion `ext3_new_block` in Erwartung weiteren Dateiwachstums, zusätzliche Blöcke in Folge zu belegen. Weitere Anforderungen von neuen Blöcken für diese Inode werden dann aus dieser Vorbestellung befriedigt, falls die lineare Folge der Anforderungen nicht durchbrochen wird.

- Die Vorbestellung für eine Inode beträgt maximal 256 Blöcke und minimal 8 Blöcke.
- Die Vorbestellung für eine Inode muß zusammenhängend sein.

Die ersten drei Regeln werden durch den nachstehenden Code realisiert, die vierte Bedingung ist auch im ext2–Dateisystem schon erfüllt.

```
prealloc_limit = ((inode->i_blocks > 256)?
                  /* Maximalgrenze festlegen. */
                  256:
                  /* Abrunden auf nächstgelegene Achtergrenze. */
                  inode->i_blocks)&0xfff8;
             /* Minimalgrenze festlegen. */
             if (prealloc_limit<8)
                 prealloc_limit = 8;
```

Diese Änderung bewirkt, daß bei der Verlängerung von großen Dateien größere zusammenhängende Blöcke vorbestellt werden. Werden zwei Dateien innerhalb derselben Blockgruppe konkurrierend verlängert, sind die durch die größere Vorbestellung entstehenden zusammenhängenden Abschnitte länger als beim ext2–Dateisystem. Da Vorbestellungen aber sowohl im Dateisystem als auch in der Quota–Berechnung eines Benutzers Platz belegen, kann es bei sehr vollen Dateisystemen oder Operationen nahe dem Hardlimit der Quota eines Benutzers zu *Starvation*–Effekten kommen: Werden in Umgebungen mit einer geringen Anzahl freier Blöcke viele Dateien gleichzeitig zum Verlängern geöffnet, kann die Gesamtsumme der vom Dateisystem herausgegebenen Vorbestellungen einen beträchtlichen Anteil der freien Blöcke ausmachen. Das Dateisystem erscheint voll, ist es aber nicht, denn die freien Blöcke sind zwar alle als belegt markiert, aber noch nicht beschrieben.

Eine ähnliche Betriebssituation kann auch im BSD FFS eintreten (siehe Abschnitt 3.2.1.3): Im Normalbetrieb versucht das Dateisystem, für wachsende Dateien ganze Blöcke statt einzelner Fragmente zu reservieren, um ein Umkopieren der Fragmente für wachsende Dateien zu vermeiden. Sinkt der frei verfügbare Platz im BSD–Dateisystem unter eine bestimmte Grenze, verzichtet das Dateisystem auf diese Optimierung und zeigt dies durch eine Kernelmeldung (`optimization changed from time to space`) an [BSD89, 7.7]. Dies behebt das Problem der *Starvation* für volle Dateisysteme. Für einzelne Benutzer, die am Rande ihres Quota–Limits operieren, schaltet BSD schaltet die Optimierung ab.

Die Modifikation am ext2–Dateisystem folgt hier der Vorgehensweise in BSD: Steigt der Füllstand des Mediums über einen Schwellenwert, wird die aggressive Vorbestellung von Blöcken abgeschaltet, und das Dateisystem verhält sich wie ein unmodifiziertes ext2–Dateisystem. Auf Benutzerquotas wird jedoch keine Rücksicht genommen.

### 5.5.3 Änderungen an den Datenstrukturen

Schon das ext2–Dateisystem reserviert die vorderen Inodes des Dateisystems für seine internen Zwecke. Erst die Inodes mit den Nummern 11 und höher stehen für normale Dateien zur Verfügung[9].

Im ext3–Dateisystem werden zusätzliche Inodes mit besonderer Bedeutung benötigt: Eine Inode wird zur Verwaltung der Inodetabellendatei verbraucht, eine weitere zur Speicherung der Inodebelegungsbitmap (siehe Abschnitt 4.1.4). Zusätzlich wurde eine weitere Inode zur Beschreibung der Blockbelegungsbitmap eingeführt, auch wenn die Information in dieser Inode statisch ist und das Auffinden der Datenblöcke der Blockbelegungsbitmap vom Dateisystem (noch) nicht über die Inode erledigt wird. Abbildung 5.13 beschreibt die reservierten Inodes und ihre Bedeutung im ext3–Dateisystem.

Die Datenstruktur einer Inode auf der Festplatte mußte gegenüber dem ext2–Dateisystem kaum geändert werden. Einzig zuvor unbenutztes 32 Bit großes Feld wird nun zur Speicherung der

[9]In ext2 haben nur die Inodes 1 bis 6 eine besondere Bedeutung, der Rest ist lediglich reserviert.

```
/* Special inodes numbers */

/* Unused. */
#define EXT3_DUMMY_INO       0
/* Bad blocks file. */
#define EXT3_BAD_INO         1
/* Root directory inode. */
#define EXT3_ROOT_INO        2
/* Boot loader program (unused). */
#define EXT3_BOOT_LOADER_INO 3
/* Undelete directory (unused). */
#define EXT3_UNDEL_DIR_INO   4
/* Inode table file inode. */
#define EXT3_INO_INO         5
/* Inode bitmap file inode. */
#define EXT3_IMAP_INO        6
/* Block bitmap file inode. */
#define EXT3_BMAP_INO        7
/* Posix access control lists (unused). */
#define EXT3_ACL_IDX_INO     8
#define EXT3_ACL_DATA_INO    9

/* First non-reserved inode for old ext3 filesystems. */
#define EXT3_GOOD_OLD_FIRST_INO 16
```

Abbildung 5.13: Die ersten 16 Inodes sind im ext3–Dateisystem für interne Zwecke des Dateisystems selbst reserviert. Neu hinzugekommen gegenüber dem ext2–Dateisystem sind die Inode für die Inodetabelle, die Inodebelegungsbitmap und die Blockbelegungsbitmap.

„gedachten Lage" auf dem Medium verwendet (siehe Abschnitt 4.1.3.1): Das neue Feld `i_bg`[10] im `struct inode` gibt die Nummer der Blockgruppe an, aus der die Datenblöcke für die Datei bevorzugt reserviert werden sollen, d.h. es bezeichnet die Nummer der Blockgruppe, in der die Inode reserviert worden wäre, wären Inodes noch an Blockgruppen gebunden.

Die Änderungen an den Blockgruppendeskriptoren sind dagegen ziemlich radikal ausgefallen, obwohl es sich nur um Streichungen handelt (siehe Abschnitt 4.3.4). Der neue `struct ext3_group_desc` enthält keine statistischen Informationen über Inodes mehr und belegt deswegen nur noch 8 statt bisher 32 Bytes.

Die umfangreichsten Änderungen wurden am Superblock vorgenommen und zwar sowohl an seiner Repräsentation auf Disk als auch am erweiterten Superblock im Speicher. Die Unterschiede zwischen einem Superblock im ext2–Dateisystem und ext3–Dateisystem lassen sich wie folgt charakterisieren:

[10] Ein Alias für `osd1.linux1.l_i_bg`. `osd1` ist eine `union`, die je nach verwendetem Betriebssystem andere Bedeutungen hat.

- Im ext3–Superblock ist zusätzlich ein Feld `s_special_inodes` enthalten, das die absolute Blocknummer des Datenblocks der Inodetabellendatei enthält. Da eine Inode 128 Byte groß ist und die kleinste unterstützte Blockgröße beim ext3–Dateisystem 1024 Byte ist, sind die ersten 8 Inodes des ext3–Dateisystem so über den Superblock statt über die Inodetabellendatei lokalisierbar. Auf diese Weise wird das in Abschnitt 4.1.6 beschriebene Problem der Selbstbezüglichkeit der Inodetabellendatei gelöst, denn die Inodenummern für die Inodetabellendatei sowie die zugehörige Inodebelegungsbitmap sind kleiner als 8.
- Das Feld `s_inodes_per_group` aus dem Superblock des ext2–Dateisystems hat im ext3–Dateisystem keine Bedeutung mehr und entfällt. Verschiedene Berechnungen, die auf der Basis dieser Information arbeiten, müssen auf eine andere Grundlage gestellt werden.
- Ebenso entfallen in der Speicherrepräsentation des Superblocks von ext3 die Felder `s_inodes_per_group` und `s_itb_per_group` (*Inode Table Blocks per Group*, Anzahl der Blöcke der Inodetabelle pro Blockgruppe) wegen Bedeutungslosigkeit.
- Dafür enthält die Speicherrepräsentation eines ext3–Superblocks die neuen Felder `s_ino_ino` und `s_ibitmap_ino`. Es handelt sich um Zeiger auf die mit `iget()` aktivierten Speicherrepräsentationen der Inodetabellendatei und der zugehörigen Datei mit der Inodebelegungsbitmap.
- Neu ist auch `s_free_inodes_count`, ein Zeiger auf ein Feld von `short`–Werten in der Größe einer Speicherseite (2048 Einträge). Dieses Feld enthält für jeden Block der Inodebelegungsbitmap einen Zähler, der die Anzahl der freien Inodes für diesen Bitmapblock angibt. Mit Hilfe dieser Tabelle ist es möglich, schnell freie Inodes zu finden (siehe Abschnitt 4.1.4) Zu diesem Mechanismus gehört auch noch die Variable `s_free_inodes_max`, die die Anzahl der benutzten Zähler im Feld `s_free_inodes_count` angibt, d.h. beim Anmelden des Dateisystems wird eine ganze Speicherseite für diese Zähler angefordert, aber nur die ersten `s_free_inodes_max` dieser Zähler werden verwendet.

### 5.5.4 Aktivierung und Deaktivierung des Dateisystems

Wenn das Dateisystemmodul `ext3.o` mit dem Kommando `insmod ext3` geladen und in den Kernel eingebunden wird, wird seine Funktion `init_module` aufgerufen, die mittels `register_filesystem` den Dateisystemtyp `ext3` beim Kernel registrieren läßt. Wird nun ein Dateisystem vom Typ `ext3` mittels `mount()` beim Kernel angemeldet, erkennt dieser den entsprechenden Dateisystemtyp und ruft die beim Kernel registrierte Funktion `ext3_read_super` des Moduls auf, um den Superblock des betreffenden Dateisystems einzulesen und die im Speicher befindlichen Strukturen für dieses Dateisystem zu initialisieren. Zusätzlich zu den normalen Tätigkeiten der Dateisysteminitialisierung wie sie beim ext2–Dateisystem stattgefunden hat, muß der Code für das ext3–Dateisystem die folgenden Dinge erledigen:

- Er muß mittels `kmalloc()` eine Speicherseite beim Kernel reservieren lassen, in der er seine Zähler `s_free_inodes_count` unterhält.

- Nachdem die numerischen Inhalte aller Datenfelder des Superblocks initialisiert sind und der Zeiger auf die Operationsstruktur des Superblocks für das ext3–Dateisystem vorbelegt ist, müssen die Inodes, die die Inodetabellendatei und die Inodebelegungsbitmap beschreiben, in den Speicher gelesen werden. Da das Dateisystem noch nicht so weit initialisiert ist, daß man `iget()` zum Lesen dieser Inodes anwenden könnte, muß die entsprechende Operation mittels `bread()`, `get_empty_inode()` und manuellen Initialisieren der entsprechenden Datenstrukturen gefälscht werden. Das Ergebnis sind aktivierte Inodes für diese beiden Dateien.

- Die Inode des Root–Verzeichnisses wird mittels eines normalen `iget()` geladen.

- Schließlich wird die Inodebelegungsbitmap durchgelesen und die Anzahl der freien Inodes in jedem Block dieser Bitmap bestimmt. Mit diesen Werten wird das Feld `s_free_inodes_count` belegt.

Das Dateisystem ist so fertig aufgesetzt und betriebsbereit. Im Gegensatz zum ext2–Dateisystem hält es unmittelbar nach der Anmeldung nicht nur eine Inode (das Root–Verzeichnis, Inode 2), sondern drei Inodes dauerhaft aktiv (zusätzlich die Inodetabellendatei, Inode 5 und die Inodebelegungsbitmap, Inode 6).

Das wird leider beim Abmelden des Dateisystems zu einem Problem, da die VFS–Layer davon ausgeht, daß nach dem Deaktivieren der Root–Inode keine weiteren Inodes auf einem Dateisystem mehr aktiv sein dürfen. Die VFS–Layer kennt den Spezialfall des Quota–Subsystems, das ebenfalls dauerhaft Dateien auf Dateisystemen geöffnet hält, und schaltet die Quota für ein abzumeldendes Dateisystem selbständig aus, bevor sie überprüft, ob das Dateisystem zum Abmelden bereit ist. Es existiert aber kein Mechanismus, mit dem ein Dateisystem selbst entscheiden könnte, ob es zum Abmelden bereit ist oder mit dem es *vor* dem Abmelden über den Abmeldewunsch benachrichtigt werden könnte. Ohne eine spezielle Änderung im eigentlich dateisystemunabhängigen Code der VFS–Layer ist es nicht möglich, ein ext3–Dateisystem wieder abzumelden und da diese Änderung in der VFS–Layer und nicht im dateisystemspezifischen Modulcode vorzunehmen ist, ist das Problem mit einem Kernelmodul alleine nicht lösbar.

### 5.5.5 Aktivierung und Deaktivierung von Inodes

Die Funktion `read_inode()` des ext2–Dateisystems hat die einzulesende Inode mit dem folgenden Code bestimmt und geladen, der Vorbild für das Modell in Abschnitt 4.2.5 ist. Der Anschaulichkeit wegen wurden Fehlerprüfungen und andere Details weggelassen.

```
/* Bestimme die Blockgruppe. */
block_group = (inode->i_ino-1)/EXT2_INODES_PER_GROUP(inode->i_sb);

/* dmap(). */
group_desc  = block_group>>EXT2_DESC_PER_BLOCK_BITS(inode->i_sb);
desc        = block_group&(EXT2_DESC_PER_BLOCK(inode->i_sb)-1);
```

```
/* dread(). */
bh          = inode->i_sb->u.ext2_sb.s_group_desc[group_desc];
gdp         = (struct ext2_group_desc *) bh->b_data;
offset      = ((inode->i_ino - 1)
              % EXT2_INODES_PER_GROUP(inode->i_sb))
              * EXT2_INODE_SIZE(inode->i_sb);
/* inmap(). */
block       = gdp[desc].bg_inode_table +
              (offset >> EXT2_BLOCK_SIZE_BITS(inode->i_sb));

/* imap(). */
offset     &= (EXT2_BLOCK_SIZE(inode->i_sb) - 1);

/* iread(). */
bh          = bread (inode->i_dev, block, inode->i_sb->s_blocksize);
raw_inode   = (struct ext2_inode *) (bh->b_data + offset);
```

Ganz analog ist diese Funktion im ext3–Dateisystem eine Umsetzung des Modellcodes aus Abschnitt 4.3.5:

```
ino       = inode->i_ino;
iperb     = inode->i_sb->u.ext3_sb.s_inodes_per_block;

/* imap(). */
rel_block = ino/iperb;
offset    = ino%iperb * EXT3_INODE_SIZE(inode->i_sb);
/* Selbstbezüglichkeit auflösen. */
abs_block = (ino == EXT3_INO_INO)?
            /* Spezialfall. */
            inode->i_sb->u.ext3_sb.s_es->s_special_inodes:
            /* Allgemeiner Fall. */
            ext3_bmap(EXT3_INT(inode->i_sb), rel_block);

/* iread(). */
bh        = bread(inode->i_dev, abs_block, inode->i_sb->s_blocksize);
raw_inode = (struct ext3_inode *) (bh->b_data + offset);
```

Man sieht, daß der Code durch die Gestaltung der Inodetabelle als Datei an dieser Stelle etwas vereinfacht wird. `write_inode` ist ein Spiegelbild dieser Funktion.

### 5.5.6 Belegung und Freigabe von Inodes

Die Belegung einer neuen Inode erfolgt im ext3–Dateisystem in zwei voneinander relativ unabhängigen Schritten. Der eine Codeteil bestimmt die Zielblockgruppe, in der die Datenblöcke der Inode bevorzugt belegt werden sollen. Der Code zum Finden dieser Zielblockgruppe gleicht weitgehend dem Code im ext2–Dateisystem; einzig die Bedingung zum Auffinden der Zielblockgruppe ist etwas verändert, denn die Anzahl der freien Inodes einer Blockgruppe ist in ext3 keine verwertbare Größe mehr. Im ext2–Dateisystem wird diese Zielblockgruppe wie in Abschnitt 4.2.7 beschrieben bestimmt; hier ist das entsprechende reale Codefragment in verkürzter Darstellung:

```
if (S_ISDIR(mode)) {
  /* dir_preferred_bg(). */

  /* Durchschnittliche Anzahl Inodes pro BG. */
  avefreei = es->s_free_inodes_count /
             sb->u.ext2_sb.s_groups_count;

  /* Wähle von allen BGs eine, die ... */
  for (j = 0; j < sb->u.ext2_sb.s_groups_count; j++) {
    tmp = get_group_desc (sb, j, &bh2);
    /* ... überdurchschnittlich viele freie Inodes hat ... */
    if (tmp->bg_free_inodes_count &&
        tmp->bg_free_inodes_count >= avefreei) {
      /* ... und unter diesen jene mit maximaler freier
       * Blockzahl. */
      if (!gdp ||
          (tmp->bg_free_blocks_count >
                gdp->bg_free_blocks_count)) {
        i = j;
        gdp = tmp;
      }
    }
  }
} else {
  /* file_preferred_bg(). */(
  i = dir->u.ext2_i.i_block_group;
  /* Weiterer Code ausgelassen... */
}
```

Dieser Code läuft so auch im ext3–Dateisystem ab, nur daß die entscheidende Bedingung jetzt

```
if (tmp->bg_free_blocks_count &&
    tmp->bg_free_blocks_count >= avefree) {
  if (!gdp ||
      (tmp->bg_used_dirs_count <
            gdp->bg_used_dirs_count)) {
    target = j;
    gdp = tmp;
  }
}
```

lautet. `avefree` ist die durchschnittliche Anzahl von freien Datenblöcken in einer Blockgruppe, so daß ext3 also unter den Blockgruppen mit überdurchschnittlich vielen freien Datenblöcken diejenige mit einer minimalen Anzahl von Verzeichnissen auswählt.

Im zweiten Belegungsschritt werden die statistischen Informationen über das Dateisystem nach einer freien Inode durchsucht. Bei erfolgreicher Suche wird versucht, diese zu belegen. Dies bestimmt unabhängig von der Positionierung der Datenblöcke die Lage der Inode im Dateisystem. Existiert keine freie Inode mehr, wird versucht, die Inodetabelle zu vergrößern.

Die Suche nach einer freien Inode beginnt auf der höchsten Ebene der in Abbildung 4.5 dargestellten Hierarchie von Statistikdaten, in der versucht wird, einen Bitmapblock zu finden, der noch freie Inodes verzeichnet:

```
for (target=0;
     target<sb->u.ext3_sb.s_free_inodes_max;
     target++) {
  if (sb->u.ext3_sb.s_free_inodes_count[target])
    break;
}
```

Falls diese Suche keine freien Inodes in der Belegungsbitmap liefert, ist die Inodetabelle voll und muß vergrößert werden. Gelingt dies nicht, ist das Medium voll, und das Anlegen der Inode scheitert. Im Code:

```
/* I-Tabelle vergrößern. */
if ((target >= sb->u.ext3_sb.s_free_inodes_max)
 && !give_up) {
    give_up = ext3_grow_inode_table(dir->i_sb);
    if (!give_up)
        goto repeat;

    /* Fehlschlag. */
    unlock_super (sb);
    iput(inode);
    return NULL;
}
```

Dabei wird die Variable `give_up` zu Beginn der Funktion mit 0 belegt; unmittelbar danach steht das Label `repeat:`. Falls der Aufruf von `ext3_grow_inode_table()` erfolgreich war, wird der gesamte Prozeß zum Belegen einer Inode also von vorne durchlaufen. Scheitert das Vergrößern der Inodetabelle, werden die gehaltenen Ressourcen freigegeben, und dem Aufrufer wird ein Fehler signalisiert.

Wurde dagegen ein Bitmapblock gefunden, der noch freie Inodes enthält, wird nach dem Laden dieses Blocks das erste eine freie Inode bezeichnende Bit im Block gesucht und die entsprechende Inode als belegt gekennzeichnet. Es kann jedoch vorkommen, daß in dem soeben geladenen

Bitmapblock kein freies Bit mehr aufzufinden ist, obwohl die Statistikdaten für diesen Block freie Inodes anzeigen, denn es finden zum Laden des Bitmapblocks unter Umständen Festplattenzugriffe statt. Durch diese Zugriffe wird der anfordernder Prozeß in schlafenden Zustand versetzt: Die Anforderung einer neuen Inode ist nicht atomar, und es kann vorkommen, daß ein anderer Prozeß gerade die Inode belegt, die auch der erste Prozeß anfordern wollte. In diesem Fall muß der gesamte Anforderungsvorgang wiederum erneut durchlaufen werden.

```
/* Laden des entsprechenden Bitmapblocks
 * (ggf. Verlust der CPU). */
bitmap_nr =
  load_inode_bitmap(sb, target*EXT3_BITS_PER_BLOCK(sb)+1);
bh = sb->u.ext3_sb.s_inode_bitmap[bitmap_nr];

/* Inode suchen und belegen. */
j = find_first_zero_bit (bh->b_data,
      EXT3_BITS_PER_BLOCK(sb));
if (j < EXT3_BITS_PER_BLOCK(sb)) {
  /* Das Belegen kann fehlschlagen, denn
   * ein konkurrierender Prozeß kann
   * die letzte freie Inode in der Blockgruppe
   * belegen. Der Belegungsprozeß muß dann
   * erneut gestartet werden. */
  if (set_bit (j, bh->b_data)) {
    goto repeat;
  }

  /* Buchhaltungscode ausgelassen. */
} else {
  /* Hier ebenso. */
  goto repeat;
}
j += target * EXT3_BITS_PER_BLOCK(sb);
```

Der Rest der Funktion beschäftigt sich mit dem Anpassen der verschiedenen Zähler im Blockgruppendeskriptor und im Superblock, den Deskriptoren und dem Initialisieren der neuen Inode–Datenstruktur. Verkürzt dargestellt:

```
  /* Zählerfeld im Superblock */
  sb->u.ext3_sb.s_free_inodes_count[target]--;
  /* Globaler Zähler im Superblock-on-disk */
  es->s_free_inodes_count--;
  /* Wenn ein Verzeichnis angelegt wurde,
   * auch den Deskriptor mit anpassen. */
  if (S_ISDIR(mode))
      gdp->bg_used_dirs_count++;
```

### 5.5.7 Erweitern der Inodetabelle

Beim Erweitern der Inodetabelle kann man direkt so vorgehen, wie es in Abschnitt 4.3.7 beschrieben ist. Lediglich an einer Stelle ist das Potential für einen Deadlock gegeben: Sowohl `ext3_new_inode()` als auch `ext3_new_block()` verändern Datenstrukturen im Superblock nicht atomar und halten daher ein Lock auf den Superblock. Muß die Inodetabelle im Kontext von `ext3_new_inode()` vergrößert werden, ist der Superblock schon gesperrt, und ein Aufruf von `ext3_getblk()` zur Beschaffung eines zusätzlichen Blocks für die Inodetabelle kann niemals terminieren, denn er mündet in `ext3_new_block()` und diese Funktion versucht ein zweites Mal den schon gesperrten Superblock zu sperren.

Daher muß unmittelbar vor dem Übergang aus der Inodeverwaltung in die Blockverwaltung das Lock auf den Superblock aufgegeben werden und danach wieder neu beschafft werden. Da innerhalb von `ext3_getblk()` Ein–/Ausgabeoperationen stattfinden, bevor `ext3_new_block()` aufgerufen und der Superblock erneut gesperrt wird, muß der Zustand der Verwaltungsdaten im Superblock vor der Aufgabe des Locks konsistent sein.

Um das sicherzustellen, beschäftigt sich `ext3_grow_inode_table()` zunächst mit der Erweiterung der Bitmapdatei.

```
// Bequemer Zugriff auf Konstanten.
iperb   = sb->u.ext3_sb.s_inodes_per_block;
bsize   = sb->s_blocksize;

// Wie in Kapitel 4.
ino     = EXT3_INT(sb)->i_size/EXT3_INODE_SIZE(sb);
ino_rel = ino/iperb;
ibm_rel = ino/EXT3_BITS_PER_BLOCK(sb);
ibm_off = ino%EXT3_BITS_PER_BLOCK(sb);

// Bitmap muß erweitert werden?
if (ibm_rel >= EXT3_IBM(sb)->i_size/bsize) {
    // Ja.
    unlock_super(sb); /* or deadlock */
    ibm_new = ext3_getblk(EXT3_IBM(sb), ibm_rel, 1, &err);
    lock_super(sb);
    if (!ibm_new) {
        goto fail;
    }
    EXT3_IBM(sb)->i_size += bsize;
    EXT3_IBM(sb)->i_dirt  = 1;

    /* Markiere alle Inodes als belegt. */
    memset(ibm_new->b_data, 0xff, bsize);
    mark_buffer_dirty(ibm_new, 1);
    brelse(ibm_new);
```

```
        /* Initialisiere einen neuen Zähler. */
        sb->u.ext3_sb.s_free_inodes_count[ibm_rel] = 0;
        sb->u.ext3_sb.s_free_inodes_max++;
    }
```

Die Belegungsbitmap für Inodes hat garantiert die erforderliche Größe, die für eine Erweiterung der Inodetabelle notwendig ist. Der Bitmapblock ist außerdem so initialisiert, daß alle von ihm verwalteten Inodes als belegt erscheinen. Der zugehörige Zähler für die Anzahl der freien Inodes steht entsprechend auf 0.

Im nächsten Schritt wird die Inodetabelle erweitert. Dabei sind alle bisher gemachten Änderungen rückgängig zu machen, falls diese Erweiterung nicht gelingt.

```
    /* Erweiterung der Inodetabelle. */
    unlock_super(sb); /* or deadlock */
    ino_new = ext3_getblk(EXT3_INT(sb), ino_rel, 1, &err);
    lock_super(sb);
    if (!ino_new) {
        /* Rückgängig machen. */
        unlock_super(sb); /* or deadlock */
        ext3_free_blocks(EXT3_IBM(sb), ibm_rel, 1);
        lock_super(sb);
        sb->u.ext3_sb.s_free_inodes_max--;
        goto fail;
    }

    EXT3_INT(sb)->i_size += bsize;
    EXT3_INT(sb)->i_dirt  = 1;
    mark_buffer_dirty(ino_new, 1);
    brelse(ino_new);
```

Die Erweiterung der Inodetabelle ist nun noch in den Verwaltungsinformationen einzutragen und die neuen Inodes müssen für den Gebrauch freigegeben werden.

```
    /* Gesamtzahl und freie Inodes erhöhen. */
    es->s_inodes_count      += iperb;
    es->s_free_inodes_count += iperb;
    /* Inodebelegungsbitmap laden. */
    i = load_inode_bitmap(sb, ino);
    ibm_new = sb->u.ext3_sb.s_inode_bitmap[i];
    for (i=0; i<iperb; i++) {
        /* Inode freigeben und Buchhaltung. */
        clear_bit(i+ibm_off, ibm_new->b_data);
        sb->u.ext3_sb.s_free_inodes_count[ibm_rel]++;
    }
    mark_buffer_dirty(ibm_new, 1);
    /* brelse(ibm_new) automatisch durch load_inode_bitmap(). */
    mark_buffer_dirty(sb->u.ext3_sb.s_sbh, 1);
    /* brelse(s_sbh) beim umount(). */
```

### 5.5.8 Verwaltung der Bitmaps

Da die Belegungsbitmap für Inodes nicht mehr wie im ext2–Dateisystem über die Deskriptoren der Blockgruppen verwaltet wird, sondern ebenfalls auf eine dynamisch wachsende Datei umgestellt wurde, mußten die Funktionen `read_inode_bitmap()` zum Lesen eines Bitmapblocks und `load_inode_bitmap()` zur Verwaltung eines LRU–Caches für gelesene Bitmapblöcke angepaßt werden.

`read_inode_bitmap()` spiegelt dabei die in Abschnitt 4.3.4 dargestellten Änderungen wider: Die Funktion bestimmt die Blocknummer des gewünschten Bitmapblocks mit

```
abs_block=bmap(EXT3_IBM(sb), rel_block);
```

statt sie wie im ext2–Dateisystem im Deskriptor der Blockgruppe nachzuschlagen. Anschließend lädt sie den Block in den vom Aufrufer vorgegebenen Speicherplatz des Cache. Der Aufrufer, `load_inode_bitmap()`, muß dabei der veränderten Semantik von `read_inode_bitmap()` angepaßt werden: Statt einer Blockgruppennummer wird beim Aufruf eine relative Blocknummer in der Inodebelegungsbitmap angegeben.

`load_inode_bitmap()` unterhält im Superblock einen Cache mit Einträgen für bis zu `EXT3_MAX_GROUP_LOADED`[11] häufig verwendete Bitmapblöcke, um die Anzahl der Zugriffe auf das Medium zu minimieren. Solange die Anzahl der Inodes im Dateisystem klein genug ist[12], werden Bitmapblöcke direkt in den Cache abgebildet. Für größere Inodebelegungsbitmaps wird der Cache nach einem LRU–Schema verwaltet.

[11] Standardwert: 16.

[12] `EXT3_MAX_GROUP_LOADED*EXT3_BITS_PER_BLOCK = 131072.`

## 5.6 Aufbau und Funktionsweise der Hilfsprogramme

Ein Dateisystem umfaßt nicht nur den Kernelcode für das eigentliche Dateisystem, sondern auch eine Reihe von Hilfsprogrammen, die im Usermode ablaufen und die Wartung von Dateisystemen ermöglichen. Typischerweise machen diese Programme den weitaus größeren Teil[13] des Codierungs- und Anpassungsaufwands aus. Sie sind deswegen größer als das eigentliche Dateisystem, weil sie direkt mit dem Benutzer interagieren und weil sie auch auf defekte Dateisysteme anwendbar sein sollen. In beiden Fällen muß der Programmcode mit falschen Eingabedaten rechnen und versuchen, diese Situation in irgendeiner Weise sinnvoll zu handhaben.

Zum Dateisystem gehören die folgenden Hilfsprogramme:

**badblocks** Die ext2– und ext3–Dateisysteme kennen jeweils eine spezielle Inode, in der Zeiger auf defekte Blöcke des Mediums untergebracht werden. Entwickelt das Medium im laufenden Betrieb neue Defekte, müssen die Blocknummern der defekten Blöcke in dieser Inode eingetragen werden. Dies geschieht normalerweise durch einen Lauf des `fsck`–Programms, das sich `badblocks` zur Ermittlung der Blocknummern der defekten Blöcke bedient.

Das Programm arbeitet ausschließlich auf Blockebene und mußte nicht angepaßt werden.

**chattr, lsattr** Die ext2– und ext3–Dateisysteme kennen eine Reihe von zusätzlichen Dateiattributen, die über die normale UNIX-Dateisemantik hinausgehen (siehe Abschnitt 3.3.1.3). Die Programme `lsattr` und `chattr` erlauben den lesenden bzw. schreibenden Zugriff auf diese Attribute.

Die Programme kommunizieren über spezielle `ioctl(2)`–Systemaufrufe mit dem Kernelcode des Dateisystems und brauchten nicht angepaßt zu werden.

**debugfs** Mit Hilfe des Programms `debugfs` lassen sich beschädigte Dateisysteme reparieren und manuell Korrekturen an einem abgemeldeten Dateisystem vornehmen. `debugfs` verfügt über ein einfaches Kommandointerface und kann eine Reihe von primitiven Manipulationen an einem Dateisystem vornehmen (Erzeugen und Löschen von Verzeichniseinträgen, Manipulation von Attributen des Superblocks oder beliebiger Inodes usw.).

Das Programm mußte an die Struktur des neuen Dateisystems angepaßt werden.

**dumpe3fs** Mit Hilfe des Programms `dumpe3fs` lassen sich die im Superblock und in den Deskriptoren der Blockgruppen enthaltenen Statistikdaten auslesen und darstellen. Das Programm zeigt neben allen Feldern des Superblocks auch die Länge und die Lage der Blöcke der Inodetabellendatei und der Inodebitmapdatei an. Von allen Deskriptoren werden die Anzahl und die Lage der freien Blöcke sowie die Anzahl der Verzeichnisse, die der Blockgruppe zugeordnet sind, ausgegeben.

Das Programm mußte an die Struktur des neuen Dateisystems angepaßt werden.

[13] Im Fall des ext3–Dateisystems hat der Kernelcode einen Umfang von etwa 200 KB Quelltext, während den zugehörigen Benutzerprogrammen mehr als 1 MB Quelltext zugrunde liegt.

**fsck, e3fsck** Das Programm `fsck` ist ein Treiberprogramm für unterschiedliche, dateisystemspezifische Prüfprogramme. Es versucht, den Typ eines Dateisystems festzustellen und dann das zum Dateisystemtyp passende Prüfprogramm zu starten.

`e3fsck` ist das zum ext3–Dateisystem passende Prüfprogramm. Es prüft die für das Dateisystem geltenden Integritätsbedingungen und versucht, beschädigte Dateisysteme zu reparieren, indem es je nach Betriebsart Annahmen über die Art des Fehlers macht oder den Systemverwalter um Rat fragt.

Das Programm `e3fsck` mußte an die Struktur des neuen Dateisystems angepaßt werden.

**mke3fs** Zum Anlegen neuer, leerer Dateisysteme dient das Hilfsprogramm `mke3fs`. Es erzeugt Superblöcke, Deskriptoren für die Blockgruppen sowie eine Blockbelegungsbitmap. Außerdem werden ein Hauptverzeichnis, ein darin enthaltenes `lost+found`–Verzeichnis und eine minimale Inodetabelle sowie die Inodebelegungsbitmap geschrieben.

Das Programm mußte an die Struktur des neuen Dateisystems angepaßt werden.

**mklost+found** Das Reparaturprogramm `e3fsck` ist auf das Vorhandensein eines Verzeichnisses `/lost+found` mit vorbestellten, leeren Verzeichnisblöcken angewiesen. Das Programm `mklost+found` erzeugt ein solches Verzeichnis, falls das originale, bei der Erstellung des Dateisystems erzeugte Verzeichnis nicht mehr existiert.

Das Programm verwendet normale Systemaufrufe für Dateisysteme (`mkdir(2)`, `creat(2)`) und brauchte nicht angepaßt zu werden.

**tune3fs** Im Superblock des Dateisystems sind einige einstellbare Parameter enthalten. Darunter befinden sich auch die Intervalle, innerhalb derer das Dateisystem spätestens einer Integritätsprüfung unterzogen wird. Mit dem Programm `tune3fs` können alle einstellbaren Parameter konfiguriert werden.

Das Programm macht Annahmen über die Struktur des Superblocks und mußte daher angepaßt werden.

Viele dieser Hilfsprogramme müssen direkt mit Datenstrukturen des Dateisystems arbeiten, d.h. sie müssen Superblöcke, Deskriptoren, Inodes und Verzeichnisse eines ext3–Dateisystems direkt auf einer Gerätedatei und ohne die Unterstützung des Kernels interpretieren können. Die dazu notwendigen Funktionen existieren in einer externen Funktionsbibliothek, die entweder als statisch oder dynamisch gebundene Library gebaut werden kann.

Das Programmierinterface des Kernels unterscheidet sich zu sehr von dem Interface für Usermode–Programme, als daß sich Funktionen aus dem Kernelcode direkt in dieser Bibliothek wiederverwenden ließen. Außerdem müssen die Funktionen zum Dateisystemzugriff im Kernel auf Geschwindigkeit optimiert sein, während es bei den identischen Funktionen der Usermode–Bibliothek mehr auf ein robustes Fehlerverhalten ankommt. Trotzdem müssen beide Funktionsgruppen identisches Verhalten zeigen, was darauf hinausläuft, jede strukturelle Änderung im Kernelcode auch in der Usermode–Bibliothek zu reproduzieren, getrennt zu testen und neben der korrekten Funktionalität auch Funktionsgleichheit mit dem Kernelcode zu prüfen.

Die Bibliothek stellt Iteratoren zur Aufzählung aller Blöcke eines Verzeichnisses, aller Verzeichniseinträge in einem Block oder aller Inodes eines Dateisystems zur Verfügung. Ein Iterator für eine bestimmte Datenstruktur wird dabei mit einem Zeiger auf eine anzuwendende Funktion und einem Zeiger auf einen Zustand wiederholt aufgerufen. In jedem Aufruf wendet der Iterator die bezeichnete Funktion auf eine einzelne Datenstruktur an und gibt dieser Funktion dabei einen Zeiger auf den aktuellen Zustand mit. Die aufgerufene Funktion kann nun die Datenstruktur bearbeiten und über bestimmte Rückgabewerte mit dem Iterator kommunizieren.

Weiterhin stellt die Bibliothek Primitive zur Verwaltung von Bitmaps, zur Belegung und Freigabe von freien Inodes und Blöcken, zum Arbeiten mit Deskriptoren und Superblöcken und zur Manipulation von Verzeichnissen zur Verfügung, d.h. die primitiven Funktionen des Kernelcodes des Dateisystems werden hier noch einmal nachgebildet. Die Codegröße des Bibliotheksquelltextes liegt bei etwa zwei Dritteln der Größe des Kernelmoduls.

Zwei weitere Bibliotheken dienen der Handhabung von Programmfehlermeldungen (`libcomm_err.a`) und dem Einlesen und Parsen von Benutzerkommandos (`libss.a`). Diese beiden Bibliotheken werden ausschließlich von `debugfs` und `fsck` verwendet.

### 5.6.1 Anlegen eines Dateisystems

Beim Anlegen eines Dateisystems mit `mke3fs` kommt es zu einem Konflikt wegen der Selbstbezüglichkeit der Inodetabelle, der dem in Abschnitt 4.3.5 beschriebenen Problem ähnelt: Während der Erzeugung des Dateisystems müssen zu einem möglichst frühen Zeitpunkt die Blockbelegungsbitmap, die Inodebelegungsbitmap und die Inodetabelle angelegt werden. Dabei ist es nicht möglich, die Inode der Inodetabelle anzulegen, solange noch keine Inodetabelle existiert.

Man hilft sich aus diesem „Henne–Ei–Problem“, indem man eine Inode für die Inodetabelle im Speicher anlegt und die zur Inodetabelle gehörenden Blöcke dort zuweist. Mit Hilfe dieser Pseudo–Inode ist es möglich, eine Inodetabelle aufzubauen, die jedoch die sich selbst beschreibende Inode noch nicht enthält. Frühestens nachdem der Inodetabelle genug Speicher zur Selbstbeschreibung zugewiesen ist, ist es möglich, die Inode der Inodetabelle auf das Medium herauszuschreiben — tatsächlich geschieht dies aber erst ganz zum Schluß.

Dabei darf der Inodetabelle kein Block zugewiesen werden, der später einmal für den Superblock, einen Deskriptor oder die Blockbelegungsbitmap verwendet wird, denn diese Teile des Dateisystems werden an festen Stellen[14] erwartet. Daher kann der Inodetabelle erst dann Speicher zugewiesen werden, nachdem Platz für die genannten statischen Teile des Dateisystems reserviert worden ist. Da zu diesem Zeitpunkt jedoch noch keine Belegungsbitmaps existieren, kann diese Reservierung nicht auf dem Medium stattfinden. Wieder behilft man sich, indem man die Bitmaps zur Verwaltung des zukünftigen Dateisystems zunächst im Speicher konstruiert und erst zum Abschluß auf das Medium schreibt.

[14] Es ist zwar möglich, die Blöcke der Blockbelegungsbitmap zu verschieben, aber mit Kenntnis der Position der Blockbelegungsbitmap läßt sich das Dateisystem im Fehlerfall auch ohne Superblock–Kopie wieder rekonstruieren.

Sobald eine Inodetabelle existiert, ist es möglich, eine Inode für die Blockbelegungsbitmap zu konstruieren, indem die Informationen über die Lage der Bitmapblöcke aus den Deskriptoren der Blockgruppen in die Inode kopiert wird. Ebenso kann eine minimale Inode für die Inodebelegungsbitmap angelegt werden.

Ab diesem Zeitpunkt ist das Dateisystem ausreichend funktionsfähig, um ein Wurzelverzeichnis und ein `lost+found`–Verzeichnis in diesem zu enthalten.

### 5.6.2 Dateisystemprüfung

Da sich die Integritätsbedingungen beim Übergang vom ext2– auf das ext3–Dateisystem nur wenig geändert haben, sollte man erwarten, daß der `fsck` relativ leicht auf die Verhältnisse im neuen Dateisystem anzupassen ist. Unglücklicherweise sind die vom Programm geprüften Integritätsbedingungen nicht so formuliert, daß sie von Mappingfunktionen Gebrauch machen. Statt dessen sind alle Vorkommen von Nachschlageoperationen in den Deskriptoren von Blockgruppen und alle anderen Bezüge auf eine statische Inodetabelle einzeln zu finden und anzupassen.

Trotzdem bleibt die grundsätzliche Funktionsweise des Programms unverändert: `fsck` liest die Inodetabelle des Dateisystems durch und rekonstruiert aus den dort vorgefundenen Informationen die Gesamtstruktur des Dateisystems, nachdem die Informationen auf Plausibilität und Einhaltung der Integritätsbedingungen geprüft worden sind. Dabei ist das Programm vollkommen auf die Informationen in der Inode der Inodetabellendatei angewiesen, um die Blöcke der Inodetabelle zu finden. Anders als im ext2–Dateisystem besteht keine Möglichkeit mehr, die Lage der Inodetabelle auf der Basis einiger wahrscheinlicher Annahmen zu erraten. Bei Verlust der Inode für die Inodetabellendatei ist das gesamte Dateisystem zerstört.

# Kapitel 6

# Konventionelles und dynamisches Dateisystem im Vergleich

Das Ziel bei der Entwicklung des dynamischen Dateisystems war es, ohne wesentliche Leistungseinschränkung eine variable Anzahl von Inodes ermöglichen. Im folgenden werden die Geschwindigkeit des alten und des neuen Dateisystems miteinander verglichen und verschiedene strukturelle Eigenschaften der Dateisysteme gegenübergestellt.

## 6.1 Meßverfahren

Die Änderungen am ext3–Dateisystem gegenüber dem ext2–Dateisystem haben Auswirkungen auf das Anlegen von Inodes, d.h. auf die Erzeugung von Dateien und Verzeichnissen. Daher mußte der Algorithmus zur Positionierung von Datenblöcken in Blockgruppen gegenüber dem ext2–Dateisystem verändert werden. In diesem Zusammenhang wurde das Verhalten des ext3–Dateisystems beim Erweitern von Dateien verändert: Das ext3–Dateisystem arbeitet im Gegensatz zum ext2–Dateisystem mit aggressiver Vorbestellung von Blöcken.

Zum Messen der Leistungsdaten des ext3–Dateisystems wurden vier Benchmarkprogramme entwickelt, die verschiedene Aspekte dieser Veränderungen hervorheben sollen:

**Test 1** Die erste Messung bezieht sich auf die Zeit, die zum Anlegen des Dateisystems gebraucht wird.

Da für das ext3–Dateisystem keine großen Inodetabellen und Inodebelegungsbitmaps geschrieben werden müssen, ist zu erwarten, daß ext3–Dateisysteme sehr viel schneller angelegt werden können als ext2–Dateisysteme.

**Test 2** Um das Verhalten des ext3–Dateisystem bei der Erweiterung von Inodetabellen zu untersuchen, wurde ein Programm geschrieben, das einen drei Ebenen tiefen Verzeichnisbaum mit jeweils zehn Ästen auf jeder Ebene anlegt. In der untersten Ebene werden in jedem Verzeichnis 100 leere Dateien angelegt. Auf diese Weise entstehen 1110 Verzeichnisse und 100000 Dateien.

Der Test überprüft das Verhalten des Dateisystems bei der Erweiterung der Inodetabelle. Über den Geschwindigkeitsvergleich mit dem ext2–Dateisystem läßt sich der Aufwand dieser Operation abschätzen.

**Test 3** Da die Regeln zur Bestimmung der Blockgruppe, aus der die Datenblöcke für eine Datei bevorzugt alloziert werden, im ext3–Dateisystem geändert sind, wurde das Testprogramm aus Test 2 so modifiziert, daß es einen zwei Ebenen tiefen Verzeichnisbaum von jeweils 10 KB großen Dateien erzeugt. Auf diese Weise entstehen 110 Verzeichnisse und 10000 Dateien, die zusammen ca. 100 MB Plattenplatz belegen.

Der Test zeigt, ob die veränderten Algorithmen zur Positionierung von Datenblöcken und Verzeichnissen wirksam sind und die Blockgruppen gleichmäßig mit Datenblöcken belegen.

**Test 4** Der letzte Test wurde als pathologisches Beispiel für die Auswirkungen des geänderten Verfahrens zur Vorbestellung von Blöcken entwickelt. Das Testprogramm öffnet simultan 100 Dateien, in die reihum in Schritten von jeweils 1 KB bis zu einer Dateigröße von jeweils 500 KB Daten geschrieben werden. Da die Dateien alle im selben Verzeichnis liegen und sowohl das ext2– als auch das ext3–Dateisystem Dateien im selben Verzeichnis derselben Blockgruppe zur Allokation von Datenblöcken zuordnet, entsteht so eine konkurrierende Erweiterung von 100 Dateien.

Dieser Test zeigt vor allen Dingen die Auswirkungen der aggressiven Vorbestellung von Blöcken.

Die Testprogramme wurden jeweils auf ein leeres, frisch angelegtes Dateisystem angewendet. Die Zeitmessung erfolgte durch den Betriebssystemkern mit Hilfe des UNIX–Kommandos `time(1)`, das die Systemfunktion `getrusage(2)` verwendet, um die Zeitstatistiken eines Sohnprozesses abzufragen. Das System befand sich zum Zeitpunkt der Messung im normalen Mehrbenutzerbetrieb, jedoch war der Meßbenutzer der einzige Benutzer auf der Maschine, und der `cron(8)`–Jobscheduler war abgeschaltet, um Verfälschungen der Meßwerte zu verhindern.

Als Datenquelle wurde bei allen Benchmarks, die Daten schreiben, ein mit `malloc(3)` vorbestelltes Speicherstück passender Größe verwendet. Alle Dateioperationen erfolgten mit direkten Betriebssystemaufrufen (`open(2)`, `write(2)`, `close(2)`) und nicht mit Funktionen der `libc.a`. Die C–Bibliothek puffert ihre Schreib– und Lesezugriffe in eigenen internen Puffern von dateisystemabhängiger Größe. Um zu verhindern, daß diese Puffer mit den Messungen interferieren, wurde auf die Verwendung von `libc`–Funktionen verzichtet. Am Ende der Testprogramme 2, 3 und 4 wurde jeweils ein Zurückschreiben des Buffer Cache vor dem Programmende erzwungen.

Das verwendete Beispieldateisystem wurde jeweils auf der nachstehenden Partition einer Maxtor 7850 AR Festplatte (Revision UA7X6059) angelegt:

```
Disk /dev/hda: 16 heads, 63 sectors, 1654 cylinders
Units = cylinders of 1008 * 512 bytes

   Device Boot  Begin   Start    End  Blocks   Id  System
/dev/hda6        1024    1024   1654  317992+  83  Linux native
```

Es handelt sich um die innerste, langsamste Partition einer EIDE–Festplatte mit ZBR. Die Partition wird sowohl vom ext2– als auch vom ext3–Dateisystem in 38 Blockgruppen mit jeweils 8192 Blöcken zu 1024 Bytes und eine kleinere Restblockgruppe aufgeteilt.

## 6.2 Meßergebnisse

### 6.2.1 Gemessene Konfigurationen

Es wurden vier unterschiedliche Dateisystemkonfigurationen ausgemessen:

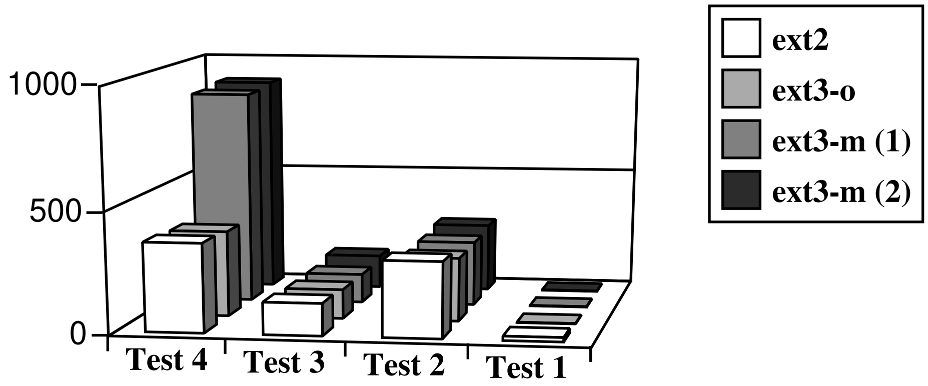
**ext2** Unter der Bezeichnung `ext2` wurde ein normales, unverändertes ext2–Dateisystem von Linux 2.0.23 gemessen. Das Dateisystem wurde mit einer Inode für jeweils 2048 Bytes Festplattenplatz angelegt (`mke2fs -i 2048 /dev/hda6`), um auf die erforderliche Anzahl von Inodes für Test 2 zu kommen.

**ext3-o** Unter der Bezeichnung `ext3-o` wurde ein ext3–Dateisystem mit dynamischer Anzahl von Inodes, aber ohne aggressive Vorbestellung von Blöcken gemessen. Die Tests erfolgten mit einer initialen Inodetabelle von 17 belegten Inodes und 24 Inodes insgesamt[1], die Inodetabelle wurde im Laufe des Tests vom Dateisystem erweitert.

**ext3-m (1) und ext3-m (2)** Unter der Bezeichnung `ext3-m` wurde ein ext3–Dateisystem mit dynamischer Anzahl von Inodes und mit aggressiver Vorbestellung von Blöcken gemessen. Die Zahl in Klammern gibt die Nummer des Meßdurchlaufes an: Im ersten Meßdurchlauf begann das Dateisystem mit 17 belegten Inodes und 24 Inodes insgesamt. Im zweiten Meßdurchlauf fand es eine Inodetabelle passender Größe aus dem ersten Meßdurchlauf vor.

Abbildung 6.1 zeigt alle vier Dateisysteme und alle vier Meßläufe im Überblick. Die angegebenen Zeiten sind in der oberen Grafik Gesamtzeiten (*wall clock time*), während die untere Grafik den Verbrauch an CPU–Zeit im Kernel (*system time*) für dieselben Messungen zeigt. Die genauen Meßwerte finden sich in Abbildung 6.2.

[1] Dies ist die kleinste Inodetabelle, die das ext3–Dateisystem anlegen kann: 16 Inodes werden vom Dateisystem für interne Zwecke reserviert, und eine Inode wird für das Verzeichnis `/lost+found` verbraucht.



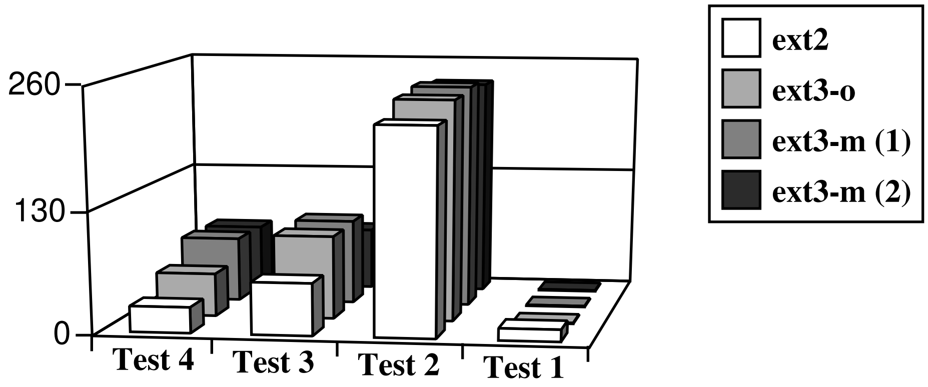




Abbildung 6.1: Vergleich der Arbeitsgeschwindigkeiten zwischen dem ext2– und ext3–Dateisystem. Die obere Grafik zeigt Gesamtzeiten, d.h. vom Benutzer wahrgenommene Zeitdauern für die Operation. Die untere Grafik zeigt Systemzeiten, d.h. die vom Kernel zur Durchführung der Operation aufgewendete Zeit. Alle Zeiten werden in Sekunden abgegeben.

| | ext2 | | | ext3-o | | |
|---|---|---|---|---|---|---|
| | Total | System | User | Total | System | User |
| Test 1 | 18,34 | 13,03 | 0,37 | 1,55 | 1,02 | 0,07 |
| Test 2 | 315,69 | 222,34 | 17,69 | 265,29 | 238,11 | 19,61 |
| Test 3 | 134,61 | 56,27 | 1,96 | 122,40 | 90,09 | 1,93 |
| Test 4 | 370,13 | 27,85 | 0,61 | 357,98 | 46,53 | 0,81 |
| | **ext3-m (1)** | | | **ext3-m (2)** | | |
| | Total | System | User | Total | System | User |
| Test 1 | 1,55 | 1,02 | 0,07 | 1,55 | 1,02 | 0,07 |
| Test 2 | 270,53 | 242,10 | 17,09 | 286,35 | 235,55 | 17,45 |
| Test 3 | 121,10 | 91,77 | 2,10 | 139,85 | 66,78 | 2,00 |
| Test 4 | 888,12 | 69,72 | 0,55 | 903,50 | 67,84 | 0,53 |

Abbildung 6.2: Vergleich der Arbeitsgeschwindigkeiten zwischen dem ext2– und ext3–Dateisystem, Tabelle der Meßwerte. Alle Zeiten werden in Sekunden angegeben.

### 6.2.2 Ergebnisse von Test 1

Erwartungsgemäß erfolgt das Anlegen eines ext3–Dateisystems sehr viel schneller als das Anlegen eines ext2–Dateisystems. Um ein ext3–Dateisystem zu erzeugen, müssen eine Inodetabelle und eine Inodebitmap konstanter Größe geschrieben werden. Unabhängig vom Dateisystemtyp sind mehrere Superblock– und Deskriptorkopien sowie Bitmapblöcke für die Blockbelegungsbitmap zu schreiben. Die Anzahl der zu schreibenden Kopien hängt von der Größe des Dateisystems ab.

Zur Erzeugung eines ext2–Dateisystems müssen dagegen zusätzlich die vollständigen Inodetabellen und Inodebelegungsbitmaps geschrieben werden. Im verwendeten Beispieldateisystem werden bei einer Inode für jeweils 4 KB Festplattenplatz 2048 Inodes pro Blockgruppe angelegt, d.h. pro Blockgruppe müssen 256 Blöcke Inodetabelle angelegt werden. Werden mehr Inodes für das Dateisystem verlangt, verlängert sich die Zeit zur Generierung des Dateisystems: 11,11 Sekunden bei einer Inode für 4096 Bytes Festplattenplatz, 18,34 Sekunden bei einer Inode für 2048 Bytes Festplattenplatz und 32,89 Sekunden bei einer Inode für 1024 Bytes Festplattenplatz. Die Zeitzunahme ist nicht linear: Während das Anlegen der ersten beiden Dateisysteme noch vom Buffer Cache abgepuffert werden kann, ist die Menge der zu schreibenden Inodetabellenblöcke im letzten Fall zu groß für den Cache des verwendeten Systems, so daß `mke2fs` auf die Festplatte warten muß.

### 6.2.3 Ergebnisse von Test 2

Der zweite Test mißt fast ausschließlich die Systemleistung beim Anlegen von Dateien und Verzeichnissen. Das ext2–Dateisystem kommt hierbei auf eine Leistung von etwa 454 Inodes pro Sekunde Systemzeit, während ext3-o nur 424 Inodes pro Sekunde erreicht, also etwa 7%

langsamer ist. Die ext3-m–Dateisysteme verhalten sich in diesem Test kaum anders, da die Vorbestellung von Blöcken nur bei regulären Dateien greift. Obwohl die ext3–Dateisysteme mehr Systemzeit verbrauchen als das ext2–Dateisystem, sind sie in der Messung der Gesamtzeit durchgehend schneller als das ext2–Dateisystem, denn ihre Inodes liegen in einer vergleichsweise zusammenhängenden Inodedatei dicht beisammen, während sich die Inodes des ext2–Dateisystems über das gesamte Medium verteilen.

```
Filesystem     Type   Inodes   IUsed   IFree  %IUsed Mounted on
/dev/hda3      ext2    64256   14855   49401    23%  /
/dev/hda5      ext2    96000   37247   58753    39%  /var/spool/news
/dev/hda6      ext2   159120  101121   57999    64%  /mnt

Filesystem     Type   Inodes   IUsed   IFree  %IUsed Mounted on
/dev/hda6      ext3   101128  101127       1   100%  /mnt
```

Abbildung 6.3: Größe der Inodetabelle nach Test 2: Beide Dateisysteme belegen 101110 Inodes, d.h. etwa 12,3 MB Festplattenplatz werden durch Inodes benutzt. Während das ext3–Dateisystem wirklich nur 12,3 MB Platz für die Inodes verbraucht, benötigt das ext2–Dateisystem 19,4 MB. Zum Vergleich sind die Statistiken für zwei seit längerer Zeit in Benutzung befindliche ext2–Dateisysteme angegeben.

Abbildung 6.3 zeigt die Inodestatistiken nach einem Durchlauf von Test 2. Beide Dateisysteme belegen im Laufe des Tests 101110 Inodes, dazu kommen die beim Anlegen des Dateisystems initial belegten Inodes. Während die Inodetabelle beim ext3–Dateisystem nur gerade die benötigte Größe hat, sind beim ext2–Dateisystem etwa 7,1 MB Plattenplatz in ungenutzten Inodetabellenblöcken festgelegt. Es ist zu beachten, daß die Anzahl der durch den Testlauf angelegten Inodes extrem hoch ist. Zum Vergleich sind die Statistiken für die zwei anderen ext2–Dateisysteme derselben Maschine angegeben, die im Produktionsbetrieb genutzt werden. Obwohl die Gesamtzahl der Inodes hier geringer ist, ist größenordnungsmäßig dieselbe Menge Festplattenplatz in ungenutzten Inodes festgelegt (6,0 und 7,2 MB).

### 6.2.4 Ergebnisse von Test 3

Der dritte Test erzeugt sequentiell sehr viele Dateien von geringer Größe. Die Dateien verteilen sich auf unterschiedliche Verzeichnisse, die nacheinander angelegt und gefüllt werden. Damit sind optimale Verhältnisse gegeben, um die Dateien auf die verschiedenen Blockgruppen zu verteilen: Wenn eine Verzeichnis–Inode neu angelegt wird, berechnet die `new_inode()`–Funktion des Dateisystems die Position der neuen Inode im Dateisystem aufgrund der Platzverhältnisse in den einzelnen Blockgruppen zu diesem Zeitpunkt. In Test 3 ist durch den sequentiellen Ablauf sichergestellt, daß zum Zeitpunkt dieser Berechnung jeweils keine der bereits existierenden Dateien und keines der bereits existierenden Verzeichnisse noch weiter wächst. Ebenso werden die Blöcke der einzelnen Dateien strikt sequentiell geschrieben, so daß niemals konkurrierende Schreibzugriffe in einer Blockgruppe stattfinden.

Die geschriebenen Dateien sind jedoch zu klein, als daß die Vorbestellung von Blöcken eine große Rolle spielen könnte: Beide Arten der Vorbestellung reservieren zunächst acht aufeinanderfolgende Blöcke für die ersten 8 KB Dateiinhalt. Im zweiten Zug werden entweder acht (ext2, ext3-o) oder sechzehn (ext3-m) Blöcke vorbestellt, von denen jedoch nur zwei verwendet werden können, während der Rest wieder aufgegeben werden muß, wenn die Datei geschlossen wird.

```
root@white:/mnt# ~/e3fsprogs/debugfs/debugfs /dev/hda6
debugfs:  stats
...
 Group  0: 6122 free blocks,    3 used directories
 Group  1: 5928 free blocks,    3 used directories
 Group  2: 5184 free blocks,    3 used directories
...
 Group 12: 6184 free blocks,    3 used directories
 Group 13: 5183 free blocks,    3 used directories
 Group 14: 5183 free blocks,    3 used directories
...
 Group 35: 6185 free blocks,    2 used directories
 Group 36: 6185 free blocks,    2 used directories
 Group 37: 6185 free blocks,    2 used directories
 Group 38: 4688 free blocks,    2 used directories
```

Abbildung 6.4: Auslastung der Blockgruppen eines ext3-m–Dateisystems nach einem Durchlauf von Test 3 (gekürzt). Alle Blockgruppen bis auf die letzte enthalten entweder etwas mehr als 6100 oder 5100 freie Blöcke, denn jedes erzeugte Blatterzeichnis des Dateibaums umfaßt 100 Dateien zu jeweils 10 Blöcken. Die Anzahl der Verzeichnisse pro Blockgruppe ist ausbalanciert.

Der Test zeigt, daß es gelingt, die Auslastung der einzelnen Blockgruppen auszubalancieren (siehe Abbildung 6.4): Beinahe alle Blockgruppen haben entweder ca. 6100 oder 5100 freie Blöcke, und die angelegten Verzeichnisse verteilen sich über alle Blockgruppen. Das Testprogramm legt in den Blattverzeichnissen des erzeugten Dateibaums jeweils 100 Dateien zu jeweils 10 Blöcken an. Je nachdem, wie viele Blattverzeichnisse in einer Blockgruppe enthalten sind, hat diese entweder 1000 Blöcke mehr oder weniger frei. Die letzte Blockgruppe, die etwas kleiner als die anderen Blockgruppen ist, weist eine etwas geringere Zahl von freien Blöcken auf.

Die Inodetabellendatei ist wegen der aggressiven Vorbestellung von Blöcken sehr stark zusammenhängend (siehe Abbildung 6.5): Sie besteht aus fünf Extents, von denen die ersten zwei schon beim Anlegen des Dateisystems durch die Bibliothek der Hilfsprogramme erzeugt werden. Derselbe Test erzeugt in einem ext3-o–Dateisystem eine Inodetabellendatei aus neun Extents.

Das ext3–Dateisystem ist im Test 3 in der Gesamtzeit etwa 9% schneller als das ext2–Dateisystem, obwohl es fast 60% mehr Systemzeit verbraucht.

```
Daten für ext3-m:

debugfs: stat <5>
Inode: 5   Type: regular    Mode:  0644   Flags: 0x0   Version: 0
User:     0   Group:     0   Size: 1296384
File ACL: 0    Directory ACL: 0
Links: 1   Blockcount: 2544
Fragment:  Address: 0    Number: 0    Size: 0
ctime: 0x32c64618 -- Sun Dec 08 11:21:12 1996
atime: 0x32c643b4 -- Sun Dec 08 11:11:00 1996
mtime: 0x32c643b4 -- Sun Dec 08 11:11:00 1996

BLOCKS:
4-5, 8, 22-1005, 2008-2061, 10201-10431
TOTAL: 1272

Vergleichsdaten für ext3-o:
BLOCKS:
4-5, 8, 22-37, 39-925, 1566-1573, 1905-1984,
2616-2623, 2995-3072, 10201-10431
TOTAL: 1272
```

Abbildung 6.5: Zusammenhang der Inodetabellendatei nach Test 3: Die Ausgabe von `debugfs` für die Inode 5 zeigt, daß die Inodetabellendatei im ext3–Dateisystem mit aggressiver Vorbestellung von Blöcken geringer fragmentiert ist als im ext3–Dateisystem ohne diese Veränderung.

### 6.2.5 Ergebnisse von Test 4

Der vierte Test ist bewußt als pathologisches Beispiel entwickelt worden, um die Auswirkungen der aggressiven Vorbestellung von Blöcken zu testen. In diesem Test werden 100 Dateien im selben Verzeichnis simultan in kleinen Schritten vergrößert. Die Auswirkungen auf die Systemleistung sind katastrophal: Der Test benötigt im ext3-m–Dateisystem fast zweieinhalbmal soviel Zeit wie im ext3-o–Dateisystem. Während dieser Zeit ist das Arbeitsgeräusch der Festplatte deutlich zu hören, d.h. die Festplattenhardware ist vornehmlich mit Kopfpositionierungen beschäftigt.

Dadurch, daß das ext3-m–Dateisystem am Ende einer zum Schreiben geöffneten großen Datei große Mengen Platz vorbestellt, sind die Zonen, in denen der Test 4 jeweils schreiben muß, vergleichsweise weit voneinander entfernt. Die vom Schreibkopf der Festplatte zurückzulegenden Entfernungen sind wesentlich länger als im ext3-o–Dateisystem. Da die geschriebene Datenmenge außerdem größer ist als der Buffer Cache des Dateisystems, muß die nahezu die gesamte Ausdehnung des beschriebenen Plattenbereichs mehrmals vom Schreibkopf angefahren werden, um den Buffer Cache zu leeren.

Die resultierende Datei ist jedoch deutlich besser zusammenhängend als eine unter ext3-o vom selben Test erzeugte Datei (siehe Abbildung 6.6). Sie enthält im Gegensatz zur von ext3-o

```
In Test 4 auf ext3-m erzeugte Datei:

debugfs:  stat 050
Inode: 67   Type: regular    Mode:  0644   Flags: 0x0   Version: 1
User:     0   Group:     0   Size: 512000
File ACL: 0    Directory ACL: 0
Links: 1   Blockcount: 1006
Fragment:  Address: 0    Number: 0    Size: 0
ctime: 0x32c585d0 -- Sat Dec 28 21:40:48 1996
atime: 0x32c582fa -- Sat Dec 28 21:28:42 1996
mtime: 0x32c585d0 -- Sat Dec 28 21:40:48 1996
BLOCKS:
1407-1414 2688-2703 5792-5839 15497-15640 35588-35843 61444-61474
TOTAL: 503

Vergleichsdaten für ext3-o:

BLOCKS:
  439-  446  1239- 1246  2039- 2046  2839- 2846  3639- 3646
 4439- 4446  5239- 5246  6039- 6046  6839- 6846  7639- 7646
 8436  9076  9724 10372 11020 11668 12316 12964 13764 14564
15364 16164  8781- 8787  9581- 9587 10381-10387 11181-11187
11981-11987 12781-12787 13581-13587 14381-14387 15181-15187
15981-15987 16788 17396 18004 18612 19220 19828 20444 21244
22044 22844 23644 24444 17061-17067 17861-17867 18661-18667
19461-19467 20261-20267 21061-21067 21861-21867 22661-22667
23461-23467 24261-24267 25068 25676 26284 26892 27500 28108
28724 29524 30324 31124 31924 32724 25341-25347 26141-26147
26581-26587 27741-27747 28541-28547 29341-29347 30141-30147
30941-30947 31741-31747 32541-32547 33348 33956 34564 35172
35780 36388 37004 37804 38604 39404 40204 32821-32827
33621-33627 34421-34427 35221-35227 36021-36027 36821-36827
37621-37627 38421-38427 39221-39227 40021-40027 40821-40827
45284 46084 46884 47684 48484 41101-41107 41901-41907
42701-42707 43501-43507 44301-44307 45101-45107 45901-45907
46701-46707 47501-47507 48301-48307 49101-49107 53564 54364
55164 55964 56764 49381-49382
TOTAL: 503
```

Abbildung 6.6: Auswirkung der aggressiven Vorbestellung von Blöcken: (oben) `debugfs`–Ausgabe für die im Test 4 erzeugte Datei `050` unter ext3-m. (unten) Mit `debugfs` erzeugte Blockliste für dieselbe Datei unter ext3-o.

erzeugten Datei keine einzelnen Blöcke und die Anordnung der Blöcke innerhalb der Datei ist streng monoton steigend.

### 6.2.6 Strukturelle Unterschiede

Im ext2–Dateisystem ist die Lage der Superblöcke, der Deskriptoren für die Blockgruppen, der Bitmapblöcke für die Blöcke und Inodes der Blockgruppe und der Blöcke der Inodetabelle vorhersagbar, wenn Größe und Inodeanzahl des Dateisystems bekannt sind und das Medium keine Lesefehler aufweist, die eine Verlegung der Bitmaps oder der Inodetabelle erfordern würden. Obwohl dies zunächst wie eine sehr scharfe Bedingung klingt, ist sie für die weitaus meisten im Produktionsbetrieb laufenden ext2–Dateisysteme erfüllt: In der Regel füllt ein Dateisystem die gesamte Partition aus, auf der es angelegt ist. Damit kann die Größe des Dateisystems, ohne im Superblock nachzuschlagen, aus der Partitionstabelle der Festplatte ermittelt werden. In der Regel ist das Dateisystem mit der standardmäßigen Anzahl von Inodes eingerichtet worden, d.h. es wurde für jeweils 4096 Bytes Partitionsgröße eine Inode erzeugt. Weiterhin erscheinen moderne Festplatten frei von Defekten, da Zugriffe auf Blöcke mit Lesefehlern von der Platte intern auf Reserveblöcke abgebildet werden, die auf jeder Spur vorhanden sind.

Damit läßt sich ein ext2–Dateisystem auch dann noch vollständig rekonstruieren, wenn alle seine Superblöcke, Blockgruppendeskriptoren und Bitmaps überschrieben worden sind. Die Informationen in den Bitmapblöcken wird vom `fsck` in jedem Fall bei einem Reparaturlauf vollständig regeneriert. Ebenso wird die statistische Information in den Blockgruppendeskriptoren neu erzeugt. Die Superblöcke und Deskriptoren können mit einem speziellen `mke2fs`–Aufruf gewaltsam neu erzeugt werden, so daß zur Rekonstruktion des Dateisystems unter den o.a. Nebenbedingungen alleine die vollständige Inodetabelle notwendig ist. Abbildung 6.7 zeigt, wie in einem ext2–Dateisystem durch einen `mke2fs -S`–Aufruf die Superblöcke und Blockgruppendeskriptoren überschrieben werden. Der anschließende Aufruf von `e2fsck` rekonstruiert alle darin enthaltenen Informationen.

Diese Rekonstruktion ist für das ext2–Dateisystem möglich, weil die Lage der Inodetabellenblöcke vorhersagbar ist. Diese Bedingung ist im ext3–Dateisystem nicht mehr gegeben: Inodetabellenblöcke können an irgendeiner Stelle im Dateisystem angelegt worden sein, und sie sind ohne die Inode der Inodetabellendatei nicht mehr auffindbar. Wird diese Inode durch einen Systemfehler oder defekte Hardware zerstört, ist die gesamte Inodetabelle verloren. Damit sind sämliche Dateien auf dem Medium nicht mehr verfügbar. Das ext3–Dateisystem ist damit in hohem Grade abhängig von der Verfügbarkeit und Integrität der Inodetabellendatei.

Solange diese eine Inode unbeschädigt bleibt, sind die anwendbaren Reparaturmechanismen jedoch dieselben: Das Dateisystem kann durch Durchlesen und Auswerten der Inodetabellendatei vollständig rekonstruiert werden. Das in Abbildung 6.7 gezeigte Experiment gelingt dann auch mit einem ext3–Dateisystem.

```
root@white:~# mke2fs /dev/hda6
...
root@white:~# mount -t ext2 /dev/hda6 /mnt
root@white:~# cp /etc/termcap /mnt
root@white:~# umount /mnt
root@white:~# mke2fs -S /dev/hda6
...
root@white:~# fsck -y /dev/hda6
Parallelizing fsck version 1.04 (16-May-96)
e2fsck 1.04, 16-May-96 for EXT2 FS 0.5b, 95/08/09
/dev/hda6 contains a file system with errors, check forced.
Pass 1: Checking inodes, blocks, and sizes
Pass 2: Checking directory structure
Pass 3: Checking directory connectivity
Pass 4: Checking reference counts
Pass 5: Checking group summary information
Fix summary information<y>? yes

Free blocks count wrong for group 0 (7932, counted=7738).  FIXED
Free blocks count wrong (307851, counted=307657).  FIXED
Free inodes count wrong for group #0 (2040, counted=2028).  FIXED
Directories count wrong for group #0 (0, counted=2).  FIXED
Free inodes count wrong (79560, counted=79548).  FIXED

/dev/hda6: ***** FILE SYSTEM WAS MODIFIED *****
/dev/hda6: 12/79560 files (0.0% non-contiguous), 10335/317992 blocks
root@white:~# mount -t ext2 /dev/hda6 /mnt
root@white:~# diff -u /etc/termcap /mnt/termcap
root@white:~#
```

Abbildung 6.7: Zerstörung und Rekonstruktion eines ext2–Dateisystems. Bei einem ext2–Dateisystem werden die Superblöcke und Blockgruppendeskriptoren durch einen Aufruf von `mke2fs` mit der Option `-S` überschrieben. Ein anschließender Lauf von `e2fsck` regeneriert diese Informationen vollständig.

### 6.2.7 Bewertung

Die Gesamtleistung des ext3–Dateisystems ist der eines ext2–Dateisystems vergleichbar: Während ext3 meist leicht bessere Gesamtzeiten erzielt, verbraucht es mehr CPU–Zeit als ext3. Im Gegensatz zum ext2–Dateisystem kann ein ext3–Dateisystem eingerichtet werden, ohne schon beim Anlegen des Dateisystems Inodes für eventuell später einmal eintretende Spitzenlasten reservieren zu müssen. Dabei erzielt ext3 auch bei sehr großen Inodezahlen vernünftige Systemleistungen.

Der in Abschnitt 5.5.2 gewählte Grenzwert für die aggressive Vorbestellung von Blöcken ist für extreme Konkurrenzsituationen zu groß gewählt, erzeugt aber weitgehend zusammenhängende Dateien. Die Einbußen in der Systemleistung in solchen Situationen sind jedoch nicht hinnehmbar.

Das ext3–Dateisystem ist sehr empfindlich gegenüber einer Zerstörung der Inode, die die Inodetabelle beschreibt. Davon abgesehen ist es genauso robust wie das ext2–Dateisystem.

# Kapitel 7

# Zusammenfassung und Ausblick

## 7.1 Zusammenfassung

Ziel dieser Arbeit war es, die Anzahl der Inodes in einem UNIX–Dateisystem dynamisch zu gestalten. Es sollte ein lauffähiges Dateisystem geschaffen werden, das mit einer kleinen Anzahl von Inodes angelegt wird und zur Laufzeit selbsttätig und ohne Eingriff eines Systemverwalters nach Bedarf weitere Inodes beschafft und gegebenenfalls wieder freigibt.

Zu diesem Zweck wurden das Linux–ext2–Dateisystem und die zu diesem Dateisystem gehörenden Werkzeuge analysiert. Auf der Grundlage des Kernelquelltextes für das ext2–Dateisystem wurde ein formales Modell des Dateisystems beschrieben. Dabei wurde ein besonderer Schwerpunkt auf die Aspekte der Block– und Inodeverwaltung gelegt, während die Verwaltung von Verzeichnisstrukturen bewußt ausgeklammert wurde, da dies für das angestrebte Ziel unerheblich war.

Das Modell des Dateisystems wurde anschließend so verändert, daß es nicht mehr mit einer Inodetabelle statischer Größe und Positionierung arbeitet, sondern statt dessen die Inodetabelle und die Inodebelegungsbitmap in regulären Dateien handhabt. Im Gegensatz zur statischen Inodetabelle des Ausgangsdateisystems lassen sich diese Datenstrukturen dynamisch nach Bedarf erweitern und modifizieren und sind damit zur Darstellung einer veränderlich großen Anzahl Inodes geeignet.

Der Inodeallokator des Modelldateisystems wurde um Funktionen zur Vergrößerung und Verkleinerung dieser Dateien erweitert, d.h. das Modell des Dateisystems wurde so verändert, daß es tatsächlich neue Inodes beschafft, wenn die bisherige Inodetabelle erschöpft ist, und daß mehr benutzte Inodeblöcke wieder freigibt.

Die Änderungen am Modell des Dateisystems wurden am Kernelquellcode und an den Hilfsprogrammen zur Dateisystemwartung nachvollzogen. Es entstand ein neues Dateisystem mit der Bezeichnung ext3 für den Linux–Kernel, Version 2.0. Dieses Dateisystem erfüllt die Anforderung, zur Laufzeit selbsttätig weitere Inodes zu beschaffen und nicht mehr genutzte Inodeblöcke wieder freizugeben. Der Durchsatz des neuen Dateisystems ist mit der Leistung des originalen ext2–Dateisystems vergleichbar, sofern die aggressive Vorbestellung von Blöcken nicht verwendet wird. Mit aggressiver Vorbestellung von Blöcken erzeugt das neue Dateisystem besser zusammenhängende Dateien, aber die Systemleistung sinkt in einigen pathologischen Fällen erheblich.

Im modifizierten Dateisystem ist nicht mehr wie im Ausgangsdateisystem vorhersagbar, an welchen Stellen des Mediums die Inodetabelle gespeichert ist. Die Kenntnis dieser Information ist jedoch im Falle eines beschädigten Dateisystems für Reparaturversuche am Dateisystem essentiell. Das neue Dateisystem ist daher gegenüber Beschädigungen der Inodetabelle empfindlicher als das Ausgangsdateisystem — ohne Kenntnis der Metainformationen aus der Inode dieser Datei ist das modifizierte Dateisystem praktisch nicht mehr zu reparieren. Wünschenswert wären Mechanismen, die das Dateisystem gegenüber solchen Beschädigungen unempfindlicher machen oder generell die Möglichkeiten zur Rekonstruktion von beschädigten Dateisystemen verbessern.

## 7.2 Erweiterungsmöglichkeiten

Diese Arbeit bietet in verschiedenen Bereichen Möglichkeiten für weitergehende und verwandte Projekte:

- Der gesamte Komplex der Benennung von Inodes ist in dieser Arbeit nur wenig behandelt und auch nicht mitmodelliert worden, da er von den angestrebten Erweiterungen und Umbauten am Dateisystem nicht betroffen war. Es wäre denkbar, in einer Nachfolgearbeit sämtliche Aspekte der Verzeichnisverwaltung ähnlich wie in dieser Arbeit zu behandeln. Eine mögliche Erweiterung wäre die Organisation von Verzeichnissen als Hashlisten oder B–Bäume statt als lineare Listen, um einen schnelleren Zugriff auf Dateinamen in sehr großen Verzeichnissen zu erreichen.
- Das ext2–Dateisystem beinhaltet bereits Ansatzpunkte zur Implementierung von *access control lists* nach [POSIX.1e], aber es existiert noch kein veröffentlichter Code zu diesem Thema. Das Posix–Rechtesystem stellt einen erweiterbaren Rahmen für detaillierte Zugriffsrechte auf Dateien und Verzeichnisse. In der Norm werden jedoch keine Anstrengungen unternommen, ein solches Rechtesystem zu formalisieren.
- Um die Abhängigkeit des ext3–Dateisystems von den Metainformationen in der Inodetabelle und der Inodebelegungsbitmap zu verringern und Reparaturprozesse am Dateisystem zu beschleunigen, könnte man es zu einem transaktionsorientierten Dateisystem umformen, in dem alle Operationen auf Metainformationen von Dateien in einem Logbuch nachvollzogen werden.

## 7.3 Verwandte Arbeiten

Es existiert eine ganze Reihe von Dateisystemen, die eine variable Anzahl von Inodes implementieren und dabei die genannten Erweiterungsmöglichkeiten in verschiedenen Ausprägungsgraden berücksichtigen. In keinem Fall wird aber der Versuch unternommen, das Dateisystem oder Teile davon formal zu modellieren.

Die meisten dieser Dateisysteme verfügen über einen Logbuch–Mechanismus, mit dem die gezielte Reparatur eines beschädigten Dateisystems möglich ist. Außerdem werden in ihnen vielfach weitergehende Konzepte verwirklicht, die notwendig sind, um sehr große Dateisysteme effizient betreiben zu können:

- Logging–Mechanismus zum sicheren Wiederanlauf nach einem Systemhalt;
- Zusammenarbeit mit einer definierten Form von Speichermanagementsystem, das Dateisysteme und Dateien ermöglicht, die die Grenzen von Medien überspannen können;
- Datenstrukturen für Verzeichnisse, die mit einer sehr großen Anzahl von Dateien und sehr tief verschachtelten Verzeichnishierarchien effizient umgehen können;
- Datenstrukturen für Inodes, die geeignet sind, auch sehr große Dateien effizient zu verwalten.

Einige dieser Dateisysteme sollen an dieser Stelle detaillierter vorgestellt werden, um die genannten weiterführenden Aspekte zu beleuchten und alternative Implementierungen vorzustellen.

### 7.3.1 LFS — Log–Structured File System in Sprite und BSD

In [Ou88] beschreibt John Ousterhout ein Dateisystem, dessen Struktur vom Ansatz herkömmlicher Dateisysteme vollkommen abweicht. Während herkömmliche Dateisysteme versuchen, die Daten beim Schreiben zu strukturieren und zu organisieren, arbeitet das LFS–Dateisystem wie ein Logbuch: Daten werden im Buffer Cache des Systems gesammelt und bei erzwungenen Synchronisationsoperationen (Schließen einer Datei, `fsync()`–Systemaufruf o.ä.) oder nach einer gewissen Zeit linear in Form eines Logbuches auf das Medium geschrieben. Da im Gegensatz zu herkömmlichen Dateisystemen keine Strukturierung der Daten stattfinden muß, kann die volle Arbeitsgschwindigkeit des Mediums beim Schreiben von Daten ausgenutzt werden. In [Ro91] wird eine Implementierung dieses Dateisystems vorgestellt.

Seltzer et. al. nehmen dieses Ansatz in [BSD93] wieder auf, wo sie ein verbessertes LFS als Bestandteil von 4.4BSD UNIX vorstellen. Das von Seltzer et. al. erweiterte LFS-Dateisystem arbeitet dabei mit einer variablen Anzahl von Inodes. Es ist außerdem transaktionsorientiert, da das Dateisystem selbst ein Logbuch ist. Probleme des Speichermanagements und sehr großer Dateisysteme und Dateien werden nicht gesondert angesprochen.

#### 7.3.1.1 Aufbau von LFS

Wie andere UNIX–Dateisysteme auch arbeitet das LFS mit Inodes, die Datenblöcke auf die übliche Weise über direkte oder ein– und mehrfach indirekte Datenblöcke adressieren. Anders als beim FFS liegen diese Inodes aber nicht an festen Adressen, sondern werden als Abschluß einer jeden Dateioperation, die den Inhalt einer Datei verändert, erneut geschrieben: Wenn Datenblöcke verändert werden und nach einiger Zeit die Buffer Cache–Blöcke dieser Datei zurückgeschrieben werden müssen, sortiert das LFS diese Blöcke in der Datei nach Inodenummer und relativer Blocknummer und schreibt sie sequentiell an der aktuellen Schreibposition des Logbuchs auf das Medium. Die Positionen dieser Datenblöcke müssen in den indirekten Blöcken oder in der Inode vermerkt werden, die sich dadurch ebenfalls verändern und an der aktuellen Schreibposition erneut geschrieben werden. Schreibvorgänge sind im LFS also strikt linear und erzeugen keine langen Kopfbewegungen.

Trotzdem ist über die Inodes, die die Datenblöcke der Datei indizieren, ein effizienter wahlfreier Zugriff auf die Bytes einer Datei möglich — das LFS schreibt Daten zwar sequentiell vergleichbar dem Logbuch einer Datenbank, kann aber mit der Inode als Index effizient Daten an beliebigen Dateipositionen lesen.

Da LFS Datenblöcke ständig neu schreibt und alte Datenblöcke niemals überschreibt, wird durch das Dateisystem fortlaufend Platz verbraucht. Zurück bleiben alte Versionen von Datenblöcken und alte Versionen von Inodes, die auf diese alten Datenblöcke verweisen. Es ist also im LFS möglich, auf vorhergehende Versionen von Dateien zurückzugreifen. Ein *cleaner*–Prozeß, der in BSD als Usermode–Programm realisiert ist, durchläuft das Dateisystem, sobald der freie Platz auf dem Medium unter einen definierten Schwellenwert sinkt. Der Cleaner versucht, alte Versionen von Dateien zu finden und den von ihnen belegten Platz freizugeben. Die so freigegebenen Datenblöcke stehen dem Logbuch schreibenden Teil des Dateisystems dann wieder zur Verfügung.

Um zu verhindern, daß der durch den Cleaner freigegebene Platz bei fortlaufender Benutzung des Dateisystems immer stärker fragmentiert wird, unterteilt das LFS das Medium in sogenannte Segmente, die den Blockgruppen eines ext2–Dateisystems vergleichbar sind. Der Cleaner gibt nicht länger referenzierte Datenböcke nicht einfach frei, sondern er verfügt über ein zu säuberndes Quellsegment und über ein vollständig freies Zielsegment. Das Aufräumen besteht darin, die noch aktiven Datenblöcke aus dem Quellsegment so in das Zielsegment umzukopieren, daß sie dort zusammenhängend den vorderen Teil des Zielsegmentes belegen. Der freie Platz im aufgeräumten Zielsegment steht also als ein zusammenhängender Block am Ende des Segmentes zur Verfügung.

Um entscheiden zu können, welche Blöcke in einem Segment belegt und welche frei sind, schreibt das Dateisystem am Ende jedes Segmentes einen *summary block*, der Auskunft über die Verwendung der einzelnen Blöcke eines Segmentes gibt. Zusätzlich unterhält der Kernel noch eine *segment usage table*, in der für jedes Segment die Anzahl der durch gültige Inodes belegten Datenblöcke und der Zeitstempel der letzten Änderung in diesem Segment mitgeführt werden. Diese Informationen erlauben dem Cleaner die Auswahl des nächsten zu säubernden Segmentes nach strategischen Gesichtspunkten.

#### 7.3.1.2 Variable Anzahl von Inodes

Während das ursprüngliche LFS im Sprite–Betriebssystem den Cleaner als Kernelprozeß implementiert hat, ist er in der BSD–Implementation von LFS ein normales Programm im Usermode, das sich über einige privilegierte Systemaufrufe und eine spezielle Datei Zugriff auf die zur Erfüllung seiner Aufgabe benötigten Informationen verschafft. Diese spezielle Datei, `IFILE` genannt, enthält statistische Informationen über alle Segmente und Zeiger auf alle Inodes von Dateien. Die `IFILE`–Datei erfüllt damit Aufgaben, die im ext2–Dateisystem von den Deskriptoren der Blockgruppen und der Inodetabellendatei übernommen werden. Freie Inodes verwaltet das LFS in einer linearen Liste und nicht mit einer Bitmap, da die Position der Inodes variabel ist und es daher nicht darauf ankommt, eine Inode mit einer bestimmten Inodenummer zu belegen. Wie die Inodetabellendatei im ext3–Dateisystem ist das `IFILE` des LFS eine reguläre Datei, und analog zum ext3–Dateisystem werden neue Inodes in LFS durch Erweiterung des `IFILE` erzeugt.

### 7.3.2 Episode – lokale Dateisystemkomponente von OSF DCE

Chutani et. al. beschrieben in [OSF92] das Epsiode–Dateisystem. Episode soll das lokale Dateisystem werden, auf dem das *distributed computing environment* (DCE) der *Open Software Foundation* (OSF) aufsetzen wird. Episode trennt vollständig die Bereiche Freispeicherverwaltung und Inodemanagement auf der einen Seite und Benennung von Dateisystemobjekten auf der anderen Seite. Dazu wird eine Verallgemeinerung von Inodes, *Anodes*, und eine Verallgemeinerung von Dateisystemen, *filesets*, eingeführt.

Episode arbeitet transaktionsorientiert mit einem Logbuch, kann Dateisysteme verwalten, die Mediengrenzen überspannen und implementiert *access control lists* nach [POSIX.1e].

#### 7.3.2.1 Aufbau von Episode

In Episode wird ein Dateisystem durch ein Aggregat beschreiben. Dabei handelt es sich um eine Ansammlung von zu verwaltenden Blöcken, die aus einer oder mehreren Partitionen eines oder mehrerer physikalischer Laufwerke stammen können. Am Anfang eines Aggregates steht ein Aggregatkopf, der in etwa die Aufgabe eines Superblocks wahrnimmt und die zum Aggregat gehörenden Medien in Form von Gerätenummernpaaren und für das Gerät zugelassenen Blocknummernbereichen aufzählt. Der Aggregatkopf beherbergt auch eine Reihe von Anodes, die man sich als in den Superblock verlegte Verwaltungsinodes vorstellen kann. Das durch eine Anode beschriebene Objekt nennt Episode einen Container; es hat alle Eigenschaften einer Datei mit Ausnahme der Tatsache, daß es nicht unbedingt in einer Inodetabelle stehen muß. Die erste Anode beschreibt den Aggregatkopf und die zu ihm gehörenden Blöcke, so daß der Aggregatkopf selbst eine Struktur sein kann, die sich über mehrere Datenblöcke erstreckt. Weitere Anodes beschreiben eine Blockbelegungsbitmap für das gesamte Aggregat, ein Transaktionslogbuch, in dem alle Operationen auf dem Aggregat verzeichnet werden, und sogenannte *fileset anode tables*, bei denen es sich um Anodetabellen für unterschiedliche Filesets im Aggregat handelt. Ein Fileset entspricht dabei in etwa einer Inodetabelle im ext3–Dateisystem.

#### 7.3.2.2 Variable Anzahl von Inodes

Ein Fileset in einem Aggregat besteht aus einer Reihe von Anodes, von denen eine das Wurzelverzeichnis für diesen Fileset beschreibt. Da ein Fileset selbst durch eine Anode beschrieben wird, die im Aggregatkopf gespeichert ist, ist er erweiterbar, so daß die Anzahl der Anodes im Fileset variabel ist. Die in Episode verwendete Konstruktion ähnelt dabei der des ext3–Dateisystems: Die Anodes (Inodes) eines Dateisystems werden in einem Container (einer Datei) gespeichert, der selbst durch eine Anode (Inode) beschrieben wird. In Episode steht die Anode, die den Container beschreibt, nicht im Container selbst, sondern im Aggregatkopf. Dies entspräche einem ext3–Dateisystem, bei dem die Inode für die Inodetabellendatei vollständig im Superblock untergebracht wird.

Da jedoch in Episode der Aggregatkopf keine feste Größe hat und sich selbst ebenfalls über mehrere Blöcke erstrecken kann, muß er ebenfalls als Container interpretiert und durch eine Anode beschrieben werden. Diese Anode steht am Anfang des Aggregatkopfes und beschreibt den Aggregatkopf selbst. Man kann sich den Aggregatkopf im Episode–Dateisystem in den Begriffen von ext3 also als eine Meta–Inodetabellendatei vorstellen, die Inodes zur Selbstbeschreibung und zur Beschreibung der Blockbelegungsbitmap, eines Transaktionslogbuches und mehrerer Inodetabellendateien enthält. Nicht nur die Größe der einzelnen Inodetabellendateien ist variabel, sondern auch die Anzahl der Inodetabellendateien in einem Aggregat.

Jede Inodetabellendatei kann unabhängig als Dateisystem angemeldet werden. Die Datenblöcke für die durch die Inodes einer Inodetabellendatei beschriebenen Dateien kommen dabei aus demselben Aggregat. Episode erlaubt es, Inodetabellendateien zu *clonen*, d.h. eine Inodetabellendatei zu duplizieren und dabei alle Zeiger auf Datenblöcke oder indirekte Blöcke im Original und in der Kopie als *copy on write* zu markieren. Der Clone einer Inodetabellendatei kann dann unabhängig vom Originaldateisystem an anderer Stelle des Dateibaums angemeldet werden. Auf diese Weise ist es z.B. möglich, den Clone eines Dateisystems zu sichern, während der normale Produktionsbetrieb auf dem Originaldateisystem fortgesetzt wird. Datenblöcke und indirekte Blöcke, die nach dem Clonen im Original modifiziert werden, dupliziert Episode nach Bedarf. Episode unterscheidet bei einem Fileset nach dem Clonen Original und Kopie: Die Kopien können nur lesend verfügbar gemacht werden, um zu verhindern, daß divergierende Kopien eines Filesets entstehen.

### 7.3.3 XFS — auf B+–Bäumen basierendes Dateisystem in Irix 5.3

In [Sw96] beschreibt Adam Sweeney das XFS–Dateisystem, wie es die Firma Silicon Graphics Inc. seit Irix Version 5.3 mit ihren Maschinen ausliefert. XFS ist ein Dateisystem für extrem große Speichersysteme und extrem große Dateien. Es arbeitet mit dem Irix–Speichermanagement zusammen und implementiert ein Logbuch, hauptsächlich um die Zeiten zur Reparatur des Dateisystems nach einem Systemhalt in einem zumutbaren Rahmen zu halten. XFS kommt ohne Bitmaps aus und verwaltet allen Speicher statt dessen in Extents der Form (Startblock, Länge), die in balancierten B+–Bäumen organisiert werden. Auch Verzeichnisse werden von XFS ab einer bestimmten Größe als B+–Bäume realisiert. Zusätzlich ist mit XFS ein Echtzeitbetrieb möglich: Anwendungen können beim Dateisystem Ein– und Ausgabekanäle mit einer bestimmten Bandbreite reservieren, die dann vom Dateisystem garantiert wird.

#### 7.3.3.1 Aufbau von XFS

Um Anwendungen mit sehr großen Datenmengen und sehr hohen Datenraten realisieren zu können, begann Silicon Graphics mit der Entwicklung eines Dateisystems, das speziell auf diese Bedürfnisse zugeschnitten ist. Das Resultat dieser Entwicklung, XFS, wird seit Dezember 1994 ausgeliefert und ist seit Anfang 1996 das Standarddateisystem für alle Silicon Graphics Workstations.

Für die Handhabung großer Datenmengen sind alle globalen Zähler und Datenstrukturen in XFS 64 Bit breit. Um jedoch eine Erweiterung auch der lokalen Datenstrukturen auf 64 Bit zu vermeiden, unterteilt XFS ein Dateisystem in *allocation groups* (AGs) von 0,5 bis 4 GB Größe und verwendet innerhalb einer solchen Einheit kürzere relative Bezeichner von 32 Bit Breite. Trotzdem können Verzeichnisse und auch einzelne Dateien als globale Strukturen in einem Dateisystem größer als eine AG werden. Jede AG besitzt eigene Datenstrukturen zur Verwaltung von Freispeicher und Inodes, die unabhängig von denen anderer AGs verwaltet werden, so daß eine nebenläufige Belegung von Freispeicher oder Inodes in unterschiedlichen AGs möglich ist.

XFS stellt Freispeicher und die zu einer Datei gehörenden Datenblöcke in der Form von Extents dar, weil sich große zusammenhängende Strukturen in dieser Form kürzer als mit Bitmaps notieren lassen. Der freie Speicher wird vom Dateisystem in zwei B+–Bäumen einmal nach der Startadresse und einmal nach der Länge indiziert organisiert, um eine schnelle und effiziente Suche nach passenden Stücken Freispeicher zu ermöglichen. Um die Datenblöcke zu beschreiben, die zu einer Datei gehören, verwendet XFS zunächst direkt in der Inode abgelegte Extents. Falls mehr Extents benötigt werden, als in eine Inode passen, um eine Datei zu beschreiben, legt XFS einen B+–Baum an, dessen Wurzel in der Inode untergebracht wird. Verzeichnisse werden in XFS ebenfalls ab einer gewissen Größe als B+–Bäume organisiert, so daß auch sehr große Verzeichnisse effizient durchsucht werden können.

Um Dateien zusammenhängend abspeichern und damit die Möglichkeiten der extentbasierten Notation in einer Inode optimal ausnutzen zu können, wird Datenblöcken im Buffer Cache von XFS noch keine feste Adresse auf dem Medium zugewiesen, sondern die Datenblöcke werden über ihre Inodenummer und ihre relative Blocknummer in der Datei identifiziert. Auf diese Weise lassen sich zusammenhängend zu speichernde Daten im Buffer Cache leicht erkennen und beim Schreiben auf das Medium zu einem Extent gruppieren. Falls der Buffer Cache ausreichend groß ist, um eine Datei vollständig im Speicher zu halten, wird die gesamte Datei als ein Extent auf das Medium geschrieben. Besonders kurzlebige oder häufig geänderte Dateien gelangen oft niemals auf das Medium, sondern werden schon im Cache durch neuere Versionen ersetzt (siehe auch [Sa81] und [Ou88]).

#### 7.3.3.2 Variable Anzahl von Inodes

XFS verwendet keine bei der Erzeugung des Dateisystems festgelegte Anzahl von Inodes, sondern erzeugt diese bei Bedarf dynamisch. Dazu wird in jeder AG eine Inodetabellendatei angelegt, d.h. XFS erzeugt einen B+–Baum, der die Lage von Inodeblöcken innerhalb der AG beschreibt. Jeder Inodeblock ist in XFS eine Einheit von 64 Inodes, und der Eintrag im

Verwaltungsbaum enthält außer einem Zeiger relativ zum Beginn der lokalen AG auch eine Belegungsbitmap mit 64 Einträgen für die von diesem Eintrag verwalteten Inodes. Inodenummern sind als globale Datenstrukturen eines Dateisystems ebenfalls 64 Bit groß.

XFS verwendet damit eine einzige Verwaltungsstruktur, um sowohl die Lage der Blöcke der Inodetabellendatei als auch ihren Belegungszustand zu beschreiben. Der Raum der Inodenummern ist in XFS dünn besetzt, d.h. jeder AG wird ein bestimmter Bereich des Inodenummernbereichs zugeteilt, aus dem die AG nach Bedarf neue Inodenummern schöpfen kann.

## 7.4 Ausblick

In den meisten wissenschaftlichen und kommerziellen Projekten auf dem Gebiet der Betriebssysteme wird Code entwickelt, ohne das modellierte System mit formalen Methoden zu beschreiben. Gerade im Hinblick auf die zunehmende Abhängigkeit der entstehenden Informationsgesellschaft von zuverlässig funktionierenden Rechnersystemen sind revisionsfähige Hard– und Software [TCSEC] und dokumentierbare Produktionsprozesse mit garantierter Qualität [ISO9000] von zunehmender Bedeutung. Aus diesem Grund kann eine solche Modellbildung, wie sie für den Teilbereich eines Dateisystems in dieser Arbeit vorgenommen wurde, einen Schwerpunkt für weitere Entwicklungen setzen.

Dabei kann die Tatsache, daß diese formalen Methoden zur Weiterentwicklung des populären, frei verfügbaren Betriebssystems Linux eingesetzt wurden, zur weiteren Verbreitung solcher Techniken beitragen. Zugleich wird die praktische Anwendbarkeit formaler Methoden im Rahmen eines aktuellen und weit verbreiteten Produktes demonstriert und das Vorurteil von Kritikern entkräftet, formale Methoden seien mit einem praktischen Einsatz unvereinbar.

# Literaturverzeichnis


[Ba86] Maurice J. Bach, „The Design of the UNIX Operating System“, *Prentice–Hall*, London, 1984

[Bau84] F.L. Bauer, H. Wössner, „Algorithmische Sprache und Programmentwicklung“, *Springer Verlag*, Berlin, 1984

[Be94] Michael Beck et. al., „Linux–Kernel–Programmierung: Algorithmen und Strukturen der Version 1.0“, *Addison-Wesley*, Bonn, 1994

[BSD84] McKusick, Karels, Leffler, „A Fast File System for UNIX“, *ACM Transactions on Computer Systems, Vol. 2, No. 3*, August 1984, Seiten 181–197

[BSD89] Leffler, McKusick, Karels, Quarterman, „The Design and Implementation of the 4.3BSD UNIX Operating System“, *Addison–Wesley*, Bonn, 1989

[BSD93] Seltzer, Bostic, McKusick, Staelin, „An Implementation of a Log–Structured File System for UNIX“, *Proceedings of the USENIX 1993 Summer Conference*, San Diego, 1993

[BSD96] Infomagic, „CD-ROM: BSDisc; NetBSD and FreeBSD UNIX Sources“, Flagstaff, 1996

[Bo95a] Bögeholz, Schneider, Schnurer, „Byte Milliardäre; Moderne Festplatten mit SCSI– und IDE–Interface“, *c't, Magazin für Computertechnik*, Heise, Hannover, Juni 1995, Seite 120 ff.

[Bo95b] Harald Bögeholz, „Platten–Karussell; Aktuelle Festplatten im Vergleich“, *c't, Magazin für Computertechnik*, Heise, Hannover, November 1995, Seite 114 ff.

[Bo96a] Harald Bögeholz, „Platten–Karussell; Festplatten mit EIDE– und SCSI–Schnittstelle im Überblick“, *c't, Magazin für Computertechnik*, Heise, Hannover, April 1996, Seite 268 ff.

[Bo96b] Harald Bögeholz, „Platten–Karussell; Festplatten mit EIDE– und SCSI–Schnittstelle im Überblick“, *c't, Magazin für Computertechnik*, Heise, Hannover, September 1996, Seite 272 ff.

[Ca93] R. Card, E. Commelin, S. Dayras, F. Mével, „The MASIX Multi–Server Operating System“, *OSF Workshop on Microkernel Technology for Distributed Systems*, Juni 1993

[Ca95] Rémy Card, „Design and Implementation of the Second Extended Filesystem“, *Proceedings of the First Dutch International Symposium on Linux*, 1995,
auch als Appendix A in Lars Wirzenius, *Linux System Administrator's Guide*, `ftp://ftp.sunsite.unc.edu/pub/Linux/doc/ldp/sag-*`

[Ga94] Gregory Ganger, Yale Patt, „Metadata Update Performance in File Systems“, *Proceedings of the USENIX 1994 Symposium on Operating Systems Design and Implementation*, Kalifornien, 1994, Seiten 49–60

[Hd i.Vorb.] Achim Held, Dissertation (in Vorbereitung) am Institut für Informatik und Praktische Mathematik der Christian–Albrechts–Universität zu Kiel

[He93] Eric H. Herrin II, Raphael A. Finkel, „The VIVA Filesystem“, *Technical Report 225–93*, University of Kentucky, 1993

[Ho93] Charles Antonelli, Peter Honeyman, „Integrating Mass Storage and File Systems“, *CITI Technical Report 93–2*, Michigan, 1993

[Hp i.Vorb.] Klaus Hoppe, Diplomarbeit (in Vorbereitung) am Institut für Informatik und Praktische Mathematik der Christian–Albrechts–Universität zu Kiel

[ISO9000] DIN EN ISO 9000, Teil 3, Juni 1992,
DIN EN ISO 9001, August 1994,
DIN EN ISO 9004, Teil 1, August 1994, Beuth Verlag, Berlin

[Ja90] Alfons Jammel, „Betriebssysteme I und II“, *Vorlesungsmitschrift*, SS 90 und WS 90/91, Institut für Informatik und Praktische Mathematik der Christian–Albrechts–Universität zu Kiel

[Ja91] Alfons Jammel, „Rechnernetze I und II“, *Vorlesungsmitschrift*, SS 91 und WS 91/92, Institut für Informatik und Praktische Mathematik der Christian–Albrechts–Universität zu Kiel

[Ja92] Alfons Jammel, „UNIX — Struktur und Implementierung“, *Vorlesungsmitschrift*, SS 92, Institut für Informatik und Praktische Mathematik der Christian–Albrechts–Universität zu Kiel

[Ke90] Kernighan, Ritchie, „Programmieren in C; ANSI-C“, *Prentice–Hall*, London, 1990

[Kl85] Steve Kleiman et. al., „Design and Implementation of the Sun Network File System“, *Proceedings of the USENIX 1985 Summer Conference*, 1985, Seiten 119–130

[Kl86] Steve Kleiman, „Vnodes: An Architecture for Multiple File System Types in Sun UNIX“, *Proceedings of the USENIX 1986 Summer Conference*, Atlanta, 1986

[Kn91] Donald Knuth, „The TEXbook“, *Addison–Wesley*, Reading, Mass, 1991

[Ko91] Helmut Kopka, „LaTeX — Eine Einführung“, *Addison–Wesley*, Bonn, 1991

[La96] Kevin Lai, Mary Baker, „A Performance Comparison of UNIX Operating Systems on the Pentium“, *Proceedings of the USENIX 1996 Winter Technical Conference*, 1996

[Li96] Linus Torvalds et. al., „Quelltext: Linux 2.0 Operating System“, `ftp://ftp.funet.fi/pub/Linux/kernel/linux/v2.0/*`, auch S.u.S.E., „Linux; Unix für PCs“, *S.u.S.E. GmbH*, CD–ROM–Set, Fürth, Dezember 1996

[Ma94] Massiglia, „The RAIDbook; A Source Book for Disk Array Technology“, *The RAID Advisory Board*, St. Peter, Maine, 1994

[MSDOS] Microsoft Corp., „Benutzerhandbuch für Microsoft MS-DOS 6.2; Betriebssystem plus zusätzliche Hilfsprogramme“, *Microsoft*, Seattle, 1996

[OSF92] Sailesh Chutani et.al., „The Episode File System“, *Proceedings of the USENIX 1992 Winter Conference*, 1992, Seiten 43–60

[Ou85] Ousterhout, Da Costa, Harrison, Kunze, Kupfer, Thompson, „A Trace–Driven Analysis of the Unix 4.2BSD File System“, *Proceedings of the 10th Symposium on Operating Systems Principles*, ACM, 1985, Seiten 15–24

[Ou88] Ousterhout, Douglis, „Beating the I/O Bottleneck: A Case for Log Structured File Systems“, *UCB/CSD 88/467, Computer Science Division (EECS)*, University of California, Berkeley, October 1988

[Ou90] Ousterhout, „Why Aren‘t Operating Systems Getting Faster As Fast As Hardware?“, *Proceedings of the USENIX 1990 Summer Conference*, Anaheim, 1990

[Pa88] Patterson, Gibson, Katz, „A case for redundant arrays of inexpensive disks (RAID)“, *Proc. SIGMOD*, Chicago, Illinois, Juni 1988, Seiten 109–116

[Pi90] Pike, Presotto, Thompson, Trickey, „Plan 9 from Bell Labs“, *Proceedings of the Summer 1990 UKUUG Conference*, London, Juli 1990, Seiten 1–9

[POSIX.1] „Portable Operating System Interface (POSIX); Part 1: System Application Program Interface (API) [C Language]“, *IEEE Information Technology Standard 1003.1, ISO/IEC 9945–1*

[POSIX.1e] „Portable Operating System Interface (POSIX); Part 1: System Application Interface (API); Amendment: Protection, Audit and Control Interfaces [C Language]“, *IEEE Information Technology UNAPPROVED DRAFT Standard 1003.1e*, Version vom 15. Dezember 1995

[Pw94] Pawlowski, Juszczak et. al., „NFS Version 3: Design and Implementation“, *Proceedings of the USENIX 1994 Summer Conference*, Boston, 1994, Seiten 137–151

[RFC1014] Sun Microsystems, „External Data Representation Standard“, *Requests for Comments, RFC 1014*, Juni 1987

[RFC1057] Sun Microsystems, „RPC: Remote Procedure Call, Protocol Specification, Version 2“, *Requests for Comments, RFC 1057*, Juni 1988

[RFC1094] Sun Microsystems, „NFS: Network File System Protocol Specification, Version 2“, *Requests for Comments: RFC 1094*, März 1989

[RFC1813] Sun Microsystems: Pawlowski, Callaghan, Staubach, „NFS Version 3 Protocol Specification“, *Requests for Comments: RFC 1813*, Juni 1995

[Ri86] Dennis Ritchie, „The UNIX I/O System“, *UNIX User‘s Supplementary Documents*, University of California, Berkeley, April 1986

[Ro91] Rosenblum, Ousterhout, „The Design and Implementation of a Log–Structured File System“, *Proceedings of the Symposium on Operating System Principles*, Monterey, Oktober 1991

[Sa81] Satyanarayanan, „A Study of File Sizes and Functional Lifetimes“, *Proceedings of the Eigth Symposium on Operating System Principles*, Dezember 1981, Seiten 96–108.

[Schn95] Georg Schnurer, „Fire, Fibre, SSA; Die neuen seriellen Massenspeicherschnittstellen“, *c‘t, Magazin für Computertechnik*, Heise, Hannover, Juni 1995, Seite 126 ff.

[Se95a] Margo Seltzer, Keith A. Smith et. al., „File System Logging Versus Clustering: A Performance Comparison“, *Proceedings of the USENIX 1995 Annual Technical Conference*, Kalifornien, 1995

[Se95b] Margo Seltzer, Trevor Blackwell, Jeffrey Harris, „Heuristic Cleaning Algorithms in Log–Structured File Systems“, *Proceedings of the USENIX 1995 Annual Technical Conference*, Kalifornien, 1995

[Se96] Margo Seltzer, Keith A. Smith, „A Comparison of FFS Disk Allocation Policies“, *Proceedings of the USENIX 1996 Annual Technical Conference*, San Diego, 1996

[Sw96] Adam Sweeney et.al., „Scalability in the XFS File System“, *Proceedings of the USENIX 1996 Annual Technical Conference*, San Diego, Januar 1996

[Ta87] Andrew Tanenbaum, „Operating Systems, Design and Implementation“, *Prentice–Hall International Editions*, London, 1987

[TCSEC] „Department of Defense; Trusted Computer System Evaluation Criteria“, *DoD 5200.28-STD, „Orange Book“*, USA, Dezember 1985

[Th78] Ken Thompson, „UNIX Time Sharing System: UNIX Implementation“, *Bell System Technical Journal*, Vol. 57(6), Juli-August 1978, Seiten 1931–1946

[Vo91] Larry McVoy, S.R. Kleiman, „Extent–like Performance from a UNIX File System“, *Proceedings of the USENIX 1991 Winter Conference*, Dallas, 1991, Seiten 33–44

[Wa i.Vorb.] Andreas Wacknitz, Diplomarbeit (in Vorbereitung) am Institut für Informatik und Praktische Mathematik der Christian–Albrechts–Universität zu Kiel

[We94] Brent Welch, „A Comparison of Three Distributed File System Architectures: Vnode, Sprite and Plan 9“, *Computing Systems*, Vol. 7(2), 1994, Seiten 175–199

[Wi94] Ulrich Widmann, „Krach gemessen; Gehörbezogene Geräuschbewertung“, *c‘t, Magazin für Computertechnik*, Heise, Hannover, April 1994, Seite 166 ff.

[Wu i. Vorb.] Karsten Wulf, Studienarbeit (in Vorbereitung) am Institut für Informatik und Praktische Mathematik der Christian–Albrechts–Universität zu Kiel

Hiermit erkläre ich, daß ich die vorliegende Arbeit ohne fremde Hilfe angefertigt habe; außer der angegebenen Literatur habe ich keine weiteren Hilfsmittel benutzt.