# Diplomarbeit

zum Thema

## Entwicklung eines echtzeitfähigen Dateisystems

an der

Technischen Universität Dresden
Fakultät Informatik
Institut für Betriebssysteme, Datenbanken und Rechnernetze
Lehrstuhl für Betriebssysteme

| | |
|---|---|
| Eingereicht von: | Lars Reuther |
| Eingereicht am: | 15. Januar 1998 |

Verantwortlicher Hochschullehrer:

Prof. Dr. H. Härtig

Betreuer:

Dr. Claude-Joachim Hamann
Dipl.-Inf. Sebastian Schönberg

# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

Für den Begriff Echtzeit existieren eine Reihe von Definitionen. Ursprünglich wurden Echtzeitsysteme zur Steuerung von Maschinen oder Industrieanlagen eingesetzt. Diese Systeme mußten Zusagen über Reaktions- und Antwortzeiten auf z.B. Sensorensignale einhalten können um eine korrekte Funktionsweise der entsprechenden Anlage zu gewährleisten. Die Zeitauflösungen gehen dabei bis in den Mikrosekundenbereich und die Systeme müssen garantieren, daß die Zusagen generell eingehalten werden. Eine derartige Charakterisierung wird auch mit *harter Echtzeit* bezeichnet. Diese Systeme sollen hier jedoch nicht betrachtet werden. Der Begriff Echtzeit wird in letzter Zeit auch immer häufiger in Verbindung mit Multimediasystemen gebraucht. Damit sind Systeme gemeint, die verschiedene Medientypen wie Video, Ton oder Text gleichzeitig verwenden. Die Bezeichnung als Echtzeitsystem beruht auf den Eigenschaften einiger dieser Medientypen. Beispielsweise erfordert die Verwendung von Videosequenzen, daß das System ebenfalls Zusagen über die Geschwindigkeit machen kann, mit der diese Daten gelesen werden können, um eine störungsfreie Anzeige dieser Videos zu ermöglichen. Allerdings sind diese Zusagen weit weniger streng zu sehen als bei einer Anlagensteuerung, weshalb diese Form auch als *weiche Echtzeit* bezeichnet wird.

Im Rahmen des Projekts *Dresden Real Time Operating System (DROPS)* am Lehrstuhl für Betriebssysteme der Technischen Universität Dresden wird gegenwärtig untersucht, wie derartige Zusagen durch die Verwendung von *Quality of Service (QoS)* Parametern unterstützt werden können. Ein Teilprojekt beschäftigt sich dabei mit der Entwicklung eines zusagefähigen Speichersystems. Dieses besteht aus einem zusagefähigen SCSI-Treiber zum Zugriff auf die Festplatten sowie einem Dateisystem zur Verwaltung des Speicherplatzes und der Dateien. Ein separater Teil des Dateisystems beschäftigt sich mit der Admission Control, der Entscheidung über das Akzeptieren neuer Anforderungen basierend auf deren QoS-Beschreibungen. Die vorliegende Arbeit beschreibt den Entwurf und die Implementierung des Dateisystems; der SCSI-Treiber und die Admission Control sind Gegenstand anderer Arbeiten.

Warum ist nun aber die Entwicklung neuer Speichersysteme für diese Daten notwendig? Video- und Audiodaten unterscheiden sich von den herkömmlichen Datentypen wie etwa Text im wesentlichen durch zwei Eigenschaften:

- Kontinuierliches Abspielen
  Die Daten bestehen meistens aus einer Menge von Teilobjekten (Bildern oder Samples), die für eine korrekte Wiedergabe mit einer bestimmten Geschwindigkeit abgespielt werden müssen. Aufgrund dieser Eigenschaft werden Video- und Audioströme auch als *Continuous Media Data* bezeichnet.

- Datenvolumen
  Im Vergleich zu Textdokumenten müssen vor allem bei Videoströmen sehr große Datenmengen verwaltet werden. Ein einzelnes Video kann durchaus mehrere GByte an Speicherplatz erfordern.

Diesen Anforderungen sind herkömmliche Dateisysteme nicht gewachsen. An der Entwicklung von Dateisystemen für die Verwaltung von Continuous Media Daten wurde bereits an verschiedenen Stellen gearbeitet, und für einige Spezialanwendungen existieren auch eine Reihe von Lösungen, z.B. für Video-Server. Es besteht allerdings nach wie vor noch der Bedarf nach universellen Systemen, die neben Continuous Media Daten sowohl andere Daten mit Echtzeitanforderungen als auch Nicht-Echtzeitdaten gleichermaßen speichern können.

## Gliederung

Das folgende Kapitel enthält einen Überblick über die wichtigsten Mechanismen zur Dateiverwaltung sowie eine genauere Beschreibung der für den Entwurf des Dateisystems besonders bedeutsamen Verfahren. Anschließend daran werden beispielhaft einige bekannte Dateisysteme kurz erläutert. In Kapitel 3 wird ausgehend von diesen Grundlagen der Entwurf des Echtzeitdateisystems für DROPS beschrieben. Über einige Details der Implementierung wird im darauf folgenden Kapitel eingegangen. Die Arbeit wird mit einer zusammenfassenden Bewertung sowie einem Ausblick auf weitere Arbeiten beschlossen.

# Kapitel 2

# Stand der Technik

Das permanente Speichern von Daten gehört neben der Verwaltung des Hauptspeichers und von Prozessen zu einer der wichtigsten Aufgaben eines Betriebssystems. Mit der fortschreitenden Entwicklung der Betriebssysteme wurden daher auch auf dem Gebiet der Dateisysteme verschiedene Verfahren zur Verwaltung persistenter Daten entwickelt. Im folgenden sollen einige der grundlegenden Mechanismen erläutert sowie verschiedene Dateisysteme beispielhaft beschrieben werden.

## 2.1 Dateisystem-Mechanismen

Jedes Dateisystem muß unabhängig von seinem Einsatzgebiet eine Reihe grundlegender Aufgaben erfüllen: es muß den Speicherplatz des Hintergrundspeichers (z.B. der Festplatte) verwalten, für die Daten muß Speicherplatz reserviert werden und die Dateien müssen in einer geeigneten Form nach außen repräsentiert werden. Für jede dieser Aufgaben existieren eine Reihe verschiedener Lösungen [Tan94]:

- **Freispeicherverwaltung**
  Für die Verwaltung des freien Speichers werden hauptsächlich zwei verschiedene Techniken angewendet: Freispeicherlisten und Bitmaps. Bei der ersten Variante werden die freien Bereiche in Form einer verketteten Liste verwaltet, wobei jedes Element dieser Liste den Verweis auf einen freien Block enthält. Demgegenüber wird bei der Verwendung einer Bitmap jeder Block durch ein Bit in diesem Bitfeld repräsentiert und durch die Belegung dieses Bits der Block als frei bzw. belegt gekennzeichnet.

  Die Verwendung einer Bitmap erfordert in der Regel weniger Speicherplatz als eine Freispeicherliste, für die Suche eines freien Blocks in einer Bitmap wird allerdings mehr Zeit benötigt. Ein Vorteil einer Bitmap ist die konstante Größe, wodurch deren Verwaltung wesentlich einfacher als die einer Freispeicherliste ist.

- **Block-Allokation und Verwaltung**
  In einem ersten Schritt lassen sich die Verfahren zur Block-Allokation in kontinuierliche und nicht-kontinuierliche Allokation unterteilen. Bei einer kontinuierlichen Reservierung werden für eine Datei aufeinanderfolgende Blöcke verwendet, für die Beschreibung dieser Datei ist nur die Kenntnis des ersten Blocks und der Dateigröße notwendig. Dieses Verfahren hat den Vorteil, daß sequentielle Lese- oder Schreiboperationen ohne Neupositionierungen des Festplatten-Kopfes auskommen. Allerdings führt es sehr schnell zu einer starken Fragmentierung der Festplatte, so daß für das Speichern von Dateien kein genügend großer kontinuierlicher Speicherbereich zur Verfügung steht. Durch eine nicht-kontinuierliche Allokation der Blöcke kann dieses Problem gelöst werden, allerdings muß dann für jede Datei eine Liste mit den zu der Datei gehörenden Festplattenblöcken verwaltet werden. Dies kann auf verschiedene Weise erfolgen:

- Jeder Block enthält einen Verweis auf den nächsten Block, zum Zugriff auf die Datei ist nur das Speichern der Adresse des ersten Blocks notwendig. Ein großer Nachteil dieser Variante ist, daß beim zufälligen Zugriff auf die Datei alle Blöcke bis zu dem angeforderten Block gelesen werden müssen, um dessen Adresse zu ermitteln.
- Der Verweis auf den nächsten Block wird nicht in dem Datenblock selbst, sondern in einer dafür vorgesehenen Tabelle gespeichert. Der Index dieser Tabelle ist die Nummer des Plattenblocks, der Inhalt der Verweis auf den nächsten Block der Datei. Bei einem zufälligen Zugriff auf eine Datei muß zwar nach wie vor die gesamte Liste durchsucht werden, da diese jetzt jedoch separat verwaltet wird kann dies deutlich schneller erfolgen.
- Die zu einer Datei gehörenden Blöcke werden in einem Index-Knoten (*Inode*) gespeichert. Die Adressen der ersten Blöcke werden direkt in dieser Inode gespeichert, für größere Dateien werden Datenblöcke zum Speichern der Adressen verwendet, diese werden durch eine Baumstruktur verwaltet. Bei sehr großen Dateien kann diese Struktur bis zu drei Ebenen enthalten, die für die Knoten des Baums verwendeten Datenblöcke enthalten dann wiederum Verweise auf die Datenblöcke, in denen die Blockliste gespeichert ist.
- In [RO92] wird ein Verfahren beschrieben, bei dem zumindestens beim Schreiben die Vorteile der kontinuierlichen Allokation genutzt werden sollen. Dazu werden mehrere Schreiboperationen zu einer einzelnen großen zusammengefaßt, dieser *Log* wird dann kontinuierlich auf die Platte geschrieben, zusätzlich werden für das Lesen Indexstrukturen erzeugt.
- Gelegentlich (z.B. in [Rei97]) werden Baumstrukturen zur Verwaltung der Dateiinformationen benutzt. Durch die Verwendung derartiger Strukturen wird besonders der wahlfreie Zugriff auf Dateien beschleunigt, sie haben allerdings den Nachteil einer sehr aufwendigen Implementierung im Gegensatz zu den anderen erwähnten Verfahren.

• **Repräsentation der Daten**
Bei der Repräsentation der Daten kann man im wesentlichen zwischen einer „flachen“ und einer strukturierten Darstellung unterscheiden. Bei der flachen Darstellung besteht die Datei aus einer Folge von Bytes, die durch das Dateisystem nicht weiter interpretiert werden. Diese Art der Darstellung wird von den meisten Standard-Dateisystemen verwendet. Von Datenbanksystemen wird vorzugsweise eine strukturierte Darstellung verwendet, bei der die Daten z.B. als Felder (*Records*) verwaltet werden.

### 2.1.1 Verwendung mehrerer Festplatten – Striping

Dateisysteme, die zum Lesen oder Schreiben mit hohen Datenraten bzw. zur Verwaltung von großen Datenmengen verwendet werden, benutzen häufig mehrere Festplatten gleichzeitig, um die gestellten Anforderungen zu erfüllen. Da die beiden genannten Anforderungen, hohe Datenraten und Datenmengen, insbesondere auch für Multimediadaten gelten, sollen die Techniken zur Verwaltung mehrerer Festplatten an dieser Stelle etwas genauer beschrieben werden.

#### RAID

Ursprünglich wurden mehrere Festplatten gleichzeitig eingesetzt, um den Leistungsnachteil (was sowohl die Speicherkapazität als auch die Übertragungsraten betrifft) kostengünstiger Festplatten gegenüber teuren Hochleistungsplatten zu kompensieren. In [PGK88] werden eine Reihe von Techniken vorgestellt, die mit RAID (*Redundant Arrays of Inexpensive Disks*) bezeichnet werden. Dabei soll vor allem durch Redundanz die hohe Fehleranfälligkeit, die durch den gleichzeitigen Einsatz mehrerer Festplatten entsteht, reduziert werden. Tabelle 2.1 gibt einen Überblick über die RAID-Hierarchie.

Bei einer bitweisen Verteilung der Daten befinden sich alle Schreib-/Leseköpfe der einzelnen Festplatten immer an der gleichen Position, weshalb dieses Verfahren auch als *„spindelsynchron“* bezeichnet wird. Durch dieses synchrone Verhalten der Festplatten kann das Array auch als eine große Festplatte mit einer Datenrate angesehen werden, die der Summe der Datenraten der einzelnen Festplatten entspricht.

| RAID-Stufe | Beschreibung |
|---|---|
| 1 | Daten werden auf einer zweiten Festplatte gespiegelt |
| 2 | Verwendung eines Hamming-Codes, Daten werden bitweise über die Festplatten verteilt |
| 3 | Benutzung eines Paritätsbits, Daten werden bitweise verteilt |
| 4 | wie 3, Daten werden jedoch blockweise über die Festplatten verteilt |
| 5 | wie 4, Paritätsblöcke werden jedoch zyklisch über alle Festplatten verteilt |

Tabelle 2.1: RAID-Hierarchie

### Striping

Neben den RAID-Systemen existieren eine Reihe von Systemen, die ohne Redundanz arbeiten (gelegentlich werden derartige Systeme in die RAID-Hierarchie auf der Stufe 0 eingeordnet). Diese Verfahren werden meistens mit *Striping*[1] bezeichnet. Für den gleichzeitigen Einsatz mehrerer Festplatten gibt es hauptsächlich drei Gründe:

- die Speicherkapazität einer einzelnen Festplatte ist für das Speichern der Daten nicht ausreichend;
- die geforderten Datenraten sind höher als die der Festplatten;
- eine Verteilung der Aufträge in einem Mehrbenutzer-System.

Die Klassifizierung der Verfahren kann anhand von zwei Kriterien vorgenommen werden:

1. Granularität der Verteilung der Daten
2. Anzahl der pro Zugriff verwendeten Festplatten.

Anhand der Granularität werden Verfahren mit einer feinen Aufteilung der Daten (bit-, byte- oder sektorweise) und mit einer groben Aufteilung (bis zu mehreren 100 KByte) unterschieden. Wie bereits weiter oben erwähnt, führt eine sehr feine Aufteilung der Daten zu einem synchronen Betrieb der Festplatten, so daß pro Zugriff in der Regel alle Festplatten gleichzeitig verwendet werden. Bei einer groben Verteilung der Daten wird in der Regel nur eine Festplatte pro Zugriff verwendet, es sei denn die Bandbreite einer einzelnen Festplatte ist nicht ausreichend, so daß zwei oder mehr Festplatten verwendet werden müssen.

In [RzS96] werden diese beiden Vorgehensweisen (dort mit *fine-grained striping* und *coarse-grained striping* bezeichnet) miteinander verglichen. Als Grundlage für diesen Vergleich dient ein Video-Server, der mehrere Videoströme gleichzeitig liefern soll. Die Videoströme werden in kleine Teilstücke zerlegt. In einer *Runde* wird dann jeweils genau ein Teilstück jedes Videostroms gelesen. Das Ergebnis dieses Vergleichs ist, daß aufgrund der nicht vorhandenen Nebenläufigkeit beim Lesen der Ströme die Latenz bei fine-grained Striping höher ist als bei coarse-grained Striping, die durch einen größeren Puffer ausgeglichen werden muß[2].

Im Bereich der Multimedia- und Video-Server wird überwiegend das grobe Striping-Schema verwendet. Dies liegt vor allem daran, daß in der Regel die Bandbreite einer einzelnen Festplatte für die Bedienung eines Datenstroms ausreichend ist und das Hauptaugenmerk daher mehr auf der gleichzeitigen Lieferung

[1] Der Begriff *Striping* wird teilweise mit verschiedenen Bedeutungen verwendet. Im weiteren werden damit Systeme bezeichnet, die die Daten ohne die Verwendung von Redundanz über mehrere Festplatten verteilen.

[2] Der Video-Server war dabei so konfiguriert, daß bei einer gegebenen Anzahl an Festplatten und einer gegebenen Speichergröße die maximale Anzahl an Videoströmen bedient werden konnte

mehrerer Datenströme liegt. Bei einem derartigen System kommt es vor allem auf eine gleichmäßige Verteilung der Daten über alle verfügbaren Festplatten an, welches durch ein geeignetes Layout der Daten erreicht werden kann. Bei der Entwicklung eines solchen Systems sind vor allem zwei Fragen zu beantworten: wie ist die Blockgröße zu wählen, mit der die Allokation des Festplattenplatzes erfolgt und mit welchem Layout werden die Daten über die Festplatten verteilt.

Zu der ersten Fragestellung wurden in [SV97] umfangreiche theoretische Untersuchungen durchgeführt. Ziel dieser Untersuchungen war die Bestimmung der Blockgröße, bei der die Bedienungszeit (*service time*) eines Auftrags am geringsten ist. Als wichtige Faktoren, die durch die Wahl einer bestimmtem Blockgröße beeinflußt werden, werden für diese Arbeit die Lastverteilung (*load balance*) und die Verzögerung durch die Positionierung des Lesekopfs (*seek and rotational latency*) verwendet. Durch die Wahl einer kleinen Blockgröße wird eine gute Lastverteilung erreicht, was sich in einer geringen Anfangsverzögerung widerspiegelt, es ist jedoch ein erhöhter Positionierungsaufwand notwendig. Demgegenüber ist bei der Verwendung großer Blöcke der Positionierungsaufwand gering, während durch eine schlechtere Lastverteilung mit einer höheren Anfangsverzögerung zu rechnen ist. In Abbildung 2.1 ist dieser Sachverhalt graphisch dargestellt.

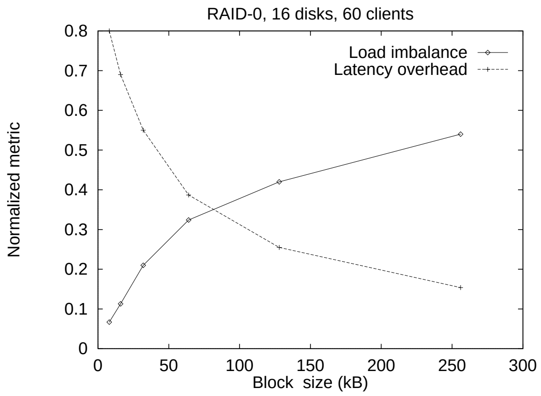



Abbildung 2.1: Lastverteilung und Latenz in Abhängigkeit der Blockgröße [SV97].

Der Wert für die optimale Blockgröße liegt am Schnittpunkt der beiden Kurven. Dieser ist von einer Reihe Systemparameter abhängig, dazu gehören u.a. die Anzahl der Festplatten und Klienten. So wird z.B. bei einer hohen Anzahl Klienten eine so große Datenmenge gelesen, daß auch mit größeren Blöcken eine ausreichende Lastverteilung erreicht wird (siehe Abb. 2.2). Die bei diesen Untersuchungen ermittelten Blockgrößen liegen für die verschiedenen Systeme im Bereich von 50 KByte bis 200 KByte.

Bei der Verteilung der Daten auf die Festplatten werden in der Regel Techniken angewendet, bei denen die Daten nacheinander über die zur Verfügung stehenden Festplatten verteilt werden (Block 1 auf Platte 1, Block 2 auf Platte 2, ..., Block n auf Platte n, Block n + 1 auf Platte 1, ... bei n Festplatten). Durch die dadurch entstehende zyklische Anordnung der Daten[3] wird eine gleichmäßige Verteilung der Daten erreicht, jede Festplatte speichert annähernd ein n-tel des Datenstroms. In [BGMJ94] ist ein derartiges Verfahren (*Staggered Striping*) beschrieben. Es ermöglicht sowohl das Lesen von Datenströmen mit hohen Bandbreitenanforderungen (durch das gleichzeitige Verwenden von mehreren Festplatten) als auch von Datenströmen mit geringen Anforderungen, diese werden durch das Auslassen von Lesezyklen und Pufferung realisiert. Abbildung 2.3 stellt das Datenlayout bei Verwendung von Staggered Striping dar. Das System speichert drei Datenströme $X, Y$ und $Z$, wobei $X$ die dreifache Bandbreite einer Festplatte, $Y$ die vierfache und $Z$ die doppelte Bandbreite beansprucht.

[3] im folgenden wird diese Art des Stripings mit *Round-Robin Striping* bezeichnet.

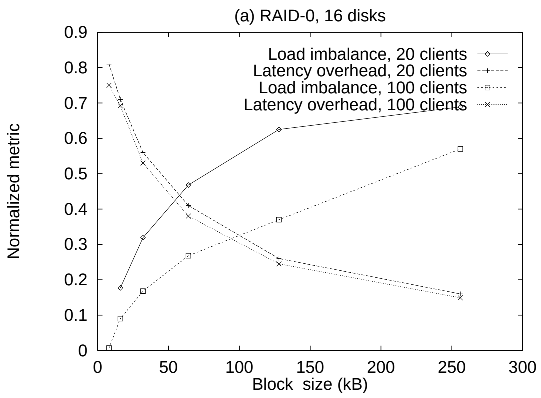



Abbildung 2.2: Optimale Blockgröße bei verschiedener Anzahl Klienten [SV97].

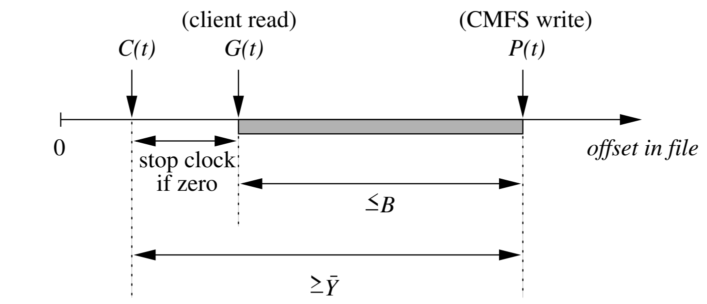
| | Disk 0 | | | | | | | | | | | Disk 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Subobject 0 | Y0.0 | Y0.1 | Y0.2 | Y0.3 | X0.0 | X0.1 | X0.2 | Z0.0 | Z0.1 | | | |
| 1 | | Y1.0 | Y1.1 | Y1.2 | Y1.3 | X1.0 | X1.1 | X1.2 | Z1.0 | Z1.1 | | |
| 2 | | | Y2.0 | Y2.1 | Y2.2 | Y2.3 | X2.0 | X2.1 | X2.2 | Z2.0 | Z2.1 | |
| 3 | | | | Y3.0 | Y3.1 | Y3.2 | Y3.3 | X3.0 | X3.1 | X3.2 | Z3.0 | Z3.1 |
| 4 | Z4.1 | | | | Y4.0 | Y4.1 | Y4.2 | Y4.3 | X4.0 | X4.1 | X4.2 | Z4.0 |
| 5 | Z5.0 | Z5.1 | | | | Y5.0 | Y5.1 | Y5.2 | Y5.3 | X5.0 | X5.1 | X5.2 |
| 6 | X6.2 | Z6.0 | Z6.1 | | | | Y6.0 | Y6.1 | Y6.2 | Y6.3 | X6.0 | X6.1 |
| 7 | X7.1 | X7.2 | Z7.0 | Z7.1 | | | | Y7.0 | Y7.1 | Y7.2 | Y7.3 | X7.0 |
| 8 | X8.0 | X8.1 | X8.2 | Z8.0 | Z8.1 | | | | Y8.0 | Y8.1 | Y8.2 | Y8.3 |
| 9 | Y9.3 | X9.0 | X9.1 | X9.2 | Z9.0 | Z9.1 | | | | Y9.0 | Y9.1 | Y9.2 |
| 10 | Y10.2 | Y10.3 | X10.0 | X10.1 | X10.2 | Z10.0 | Z10.1 | | | | Y10.0 | Y10.1 |
| 11 | Y11.1 | Y11.2 | Y11.3 | X11.0 | X11.1 | X11.2 | Z11.0 | Z11.1 | | | | Y11.0 |
| 12 | Y12.0 | Y12.1 | Y12.2 | Y12.3 | X12.0 | X12.1 | X12.2 | Z12.0 | Z12.1 | | | |
| ⋮ | | | | | | | | | | | | |

Abbildung 2.3: Datenlayout bei *Staggered Striping* [BGMJ94].

### 2.1.2 Auftragsplanung

Im Bereich der Continuous Media Dateisysteme ist es besonders wichtig, daß die Daten zu einem genau festgelegten Zeitpunkt gelesen oder geschrieben werden. Werden Daten zu spät gelesen, kommt es beispielsweise bei einem Video zu einer ruckhaften Darstellung, werden Daten zu früh gelesen, kann es zu Pufferüberläufen kommen. Diese Anforderung macht eine Auftragsplanung in dem Dateisystem notwendig; dafür existieren eine Reihe von Ansätzen.

- In [AOG91] wird eine logische Uhr zur Planung der Aufträge verwendet. Diese wird normalerweise anhand der von einer Anwendung geforderten Datenrate vorgestellt, kann aber auch angehalten werden, falls die Anwendung mit dem Auslesen der Daten aus dem Puffer nicht nachkommt. Das Dateisystem ist mit dem Lesen immer mindestens um einen *Workahead*-Parameter voraus, um evtl. Schwankungen innerhalb des Datenstroms ausgleichen zu können (siehe Abb. 2.4).
- Bei Video-Servern werden die Leseoperationen häufig Runden bzw. Zyklen organisiert. In einer Runde wird dabei genau ein Teilstück (häufig ein Block auf der Festplatte) jedes Videostroms gelesen.




$\bar{Y}$: maximum work ahead
*P(t):* the index of the next byte to be put into the FIFO by CMFS at time t
*G(t):* the index of the next byte to be removed from the FIFO by the client at time t
*C(t):* the value of the logical clock at time t
*B:* the size of the FIFO buffer

Abbildung 2.4: Die logische Uhr des Berkeley CMFS [AOG91].

Die Dauer einer Runde ist konstant, sie ist abhängig von der Zeit, die zum Lesen aller Blöcke einer Runde benötigt wird bzw. der Zeit, die zum Anzeigen der Videosequenzen benötigt wird. Die Planung erfolgt dann anhand des durch die Rundendauer vorgegebenen Zeitrasters.

- In [BFD97] wird ein Verfahren beschrieben, bei dem die Planung anhand eines zentralen Plans (*schedule*) erfolgt. Ein Eintrag in diesem Plan (*Slot*) entspricht einem Leseauftrag für einen Block. Die Daten sind nach dem Round-Robin Verfahren über die Festplatten verteilt, so daß sich ein periodisches Zugriffsverhalten auf die Festplatten ergibt. Die Länge dieser Periode ist bestimmt durch die der Anzahl der Festplatten sowie der Zeit, die zum Anzeigen eines Blocks benötigt wird (*block play time*). Während der Anzeige eines Blocks können bereits weitere Blöcke gelesen werden (in der Regel ist die Zeit zum Lesen eines Blocks (*block service time*) geringer als die block play time), so daß die Anzahl der Slots in dem Plan dem Produkt aus Anzahl Slots pro block play time und Anzahl der Festplatten entspricht. Für die Auftragsplanung wird pro Festplatte ein Zeiger auf diesen zentralen Plan verwaltet, der in Echtzeit bewegt wird. Beim Erreichen eines Slots der betreffenden Festplatte wird dieser Block gelesen.

## 2.2 Beispiele für Dateisysteme

Da, wie bereits erwähnt, das permanente Speichern der Daten zu den wichtigen Aufgaben eines Betriebssystems gehört, existieren eine Vielzahl von Dateisystemen. Im folgenden sollen einige davon exemplarisch beschrieben werden, geordnet nach dem jeweiligen Einsatzgebiet.

### 2.2.1 Standard-Dateisysteme

Die Dateisysteme der bekannten Betriebssysteme (UNIX, Microsoft Windows) sind in der Regel für viele, kleine Dateien konzipiert. Daraus resultiert u.a. die Verwendung kleiner Blöcke (in der Regel 512 Byte bis 4 KByte).

**Extended Secondary File System**

Das *Extended Secondary File System (ext2fs)* ist das Standard-Dateisystem von Linux [BBD 95], die Struktur des ext2fs ist stark an das *BSD Fast File System* angelehnt [MJLF84]. Eine Festplatte (bzw. eine Partition) ist in mehrere *Blockgruppen* aufgeteilt, die je nach verwendeter Blockgröße 8 MByte bis 32 MByte groß sind (siehe Abb. 2.5).

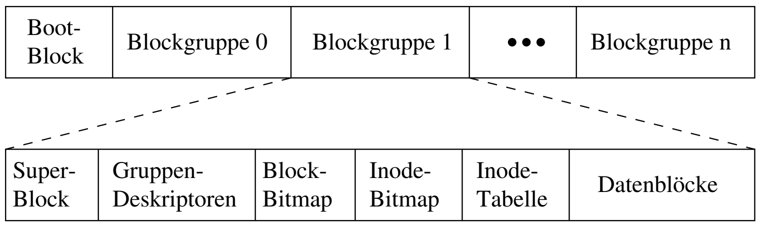



Abbildung 2.5: Aufteilung der Festplatten in Blockgruppen im *ext2fs* [BBD 95].

Jede dieser Blockgruppen enthält am Anfang neben einer Sicherungskopie des Superblocks die Verwaltungsinformationen (Block-Bitmap, Inode-Tabelle) für diese Gruppe. Ziel dieser Aufteilung ist es, die Verwaltungsinformationen möglichst nahe zu den Dateien zu speichern. Aus diesem Grund wird versucht, bei der Blockallokation zuerst Blöcke zu verwenden, die in der Blockgruppe der Inode der Datei liegen und die Dateien werden möglichst in der Nähe ihrer Verzeichnisse gespeichert. Für die Verwaltung der Dateien werden im ext2fs Inodes verwendet.

**NT File System**

Neben dem von MS-DOS bekannten FAT-Dateisystem unterstützt Microsoft Windows NT das *NT File System (NTFS*) [Löw97, Cus94]. NTFS besitzt eine Reihe interessanter Eigenschaften. So werden die Metadaten (z.B. die Block-Bitmap) als normale Datei behandelt und nicht, wie z.B. bei ext2fs, statisch beim Anlegen des Dateisystems erzeugt. Dadurch ist es beispielsweise möglich, ein bestehendes Dateisystem durch das Hinzufügen einer weiteren Festplatte zu vergrößern, indem die Bitmapdatei entsprechend erweitert wird (dadurch entstehen sog. *Volume Sets*). Eine Datei wird im NTFS durch eine Menge von Attributen beschrieben, dazu gehören u.a. der Name und eine Zugriffssteuerliste, und auch die Daten werden als Attribut verwaltet. Alle Attribute einer Datei werden zu einem *File Record* zusammengefaßt, welcher in einer speziellen Datei, der *Master File Table*, gespeichert wird. Weitere Eigenschaften des NTFS sind die Unterstützung von Software-RAID sowie das transaktionsorientierte Speichern der Metadaten (dadurch wird eine bessere Wiederherstellung der Konsistenz nach einem Systemausfall gewährleistet).

### 2.2.2 Multimedia-Dateisysteme

Die Speicherung von Continuous Media Daten stellt Anforderungen, die von den Standard-Dateisystemen nicht erfüllt werden. An verschiedenen Stellen wurden daher spezielle Dateisysteme entwickelt. Für eine Unterscheidung dieser Systeme sind mehrere Gesichtspunkte von Bedeutung:

- Datenlayout
  Wie werden die Daten auf den Festplatten gespeichert?

- Akzeptanz-Test (Admission Control)
  Welches Verfahren wird zur Entscheidung über das Zulassen neuer Datenströme angewendet?

- Auftragsplanung
  Wie werden die Aufträge zum Lesen oder Schreiben erzeugt?

### Berkeley Continuous Media File System

Einer der ersten Vertreter der Multimedia-Dateisysteme ist das *Berkeley Continuous Media File System (CMFS)* [AOG91]. Das Dateisystem verwendet die bereits erwähnte logische Uhr (vgl. Abb. 2.4). Für die Admission Control bzw. die Planung der Aufträge beim Lesen eines Datenstroms müssen immer folgende „Axiome“ erfüllt sein:

$$P(t) - G(t) \leq B \tag{2.1}$$
$$P(t) - C(t) \geq \bar{Y} \tag{2.2}$$
$$G(t) \leq P(t) \tag{2.3}$$

Durch diese Bedingungen wird ausgedrückt, daß das Dateisystem nie mehr Daten im voraus liest als der Puffer aufnehmen kann (erste Formel), das Dateisystem immer der logischen Uhr um mindestens den *Workahead*-Parameter $\bar{Y}$ voraus ist und daß die Anwendung nie das Dateisystem einholt. Für die Admission Control wird versucht, ein *operation set* $\phi$ aufzustellen, so daß die Axiome noch erfüllt werden. Die Menge $\phi$ enthält dabei für jeden Datenstrom die Anzahl Datenblöcke, die für diesen Strom während eines Zeitabschnitts gelesen werden sollen.

Für das Datenlayout werden durch das Dateisystem keine Vorgaben gemacht. Statt dessen verwenden die Algorithmen zur Admission Control und zur Auftragsplanung als Eingangsgrößen Funktionen, anhand derer die Worst-Case Lesezeiten für eine bestimmte Anzahl Datenblöcke von der Festplatte bestimmt werden. Dies ermöglicht die Verwendung verschiedener Datenlayouts.

### Mitra

Der in [GZS 96] vorgestellte Continuous Media Server Mitra verwendet eine Hierarchie von verschiedenen Speichersystemen zur Verwaltung der Datenströme. Diese Hierarchie besteht aus einem großen, langsamen Hintergrundspeicher (im Beispiel eine Jukebox von MO-Wechselplatten) und einem Festplatten-Array, welches zum Zwischenspeichern der aktiven Datenströme verwendet wird. Die Verwendung einer derartigen Hierarchie ist sinnvoll, da MO-Wechselplatten oder auch Bandlaufwerke im Vergleich zu Festplatten für die Speicherung großer Datenmengen deutlich kostengünstiger sind, aber zum direkten Lesen der Daten nicht geeignet sind.

Für die Verwaltung des Festplatten-Arrays wird ein Dateisystem Namens *EVEREST* verwendet. Zum Abspeichern der Datenströme werden diese in kleinere Teilstücke aufgeteilt. Diese Aufteilung wird so vorgenommen, daß die Dauer zum Anzeigen der Teilstücke für die verschiedenen Datenströme gleich ist. Durch EVEREST wird versucht, diese Teilstücke möglichst kontinuierlich auf den Festplatten zu speichern. Um dabei Fragmentierungsprobleme zu vermeiden, werden die Daten nicht vollständig kontinuierlich gespeichert, sondern in möglichst großen Stücken. Dazu wird der zur Verfügung stehende Festplattenplatz anhand des aus der Hauptspeicherverwaltung bekannten Buddy-Algorithmus [WJNB95] verwaltet. Der zur Speicherung der Teilstücke benötigte Festplattenplatz wird dann aus möglichst großen Blöcken zusammengesetzt (so wird z.B. bei der Verwendung eines binären Buddy-Systems ein 169 KByte großes Teilstück in einem 128 KByte, einem 32 KByte, einem 8 KByte und einem 1 KByte Block gespeichert). Um die Fragmentierung innerhalb des Buddy-Systems zu minimieren, wird nach jedem Löschen eines Datenstroms ggf. ein Umspeichern und Zusammenfassen der Blöcke vorgenommen, so daß der freie Speicherplatz immer in größtmöglichen Blöcken vorliegt[4]. Werden durch EVEREST mehrere Festplatten verwaltet, werden die Datenblöcke mittels *Staggered Striping* [BGMJ94] über die Festplatten verteilt.

[4] es wird argumentiert, daß der dadurch entstehende Mehraufwand vernachlässigt werden kann, da das Löschen eines Datenstroms nicht häufig vorkommt

Die Planung bzw. das Scheduling der Aufträge erfolgt nach dem *Grouped Sweeping Scheme (GSS)*. Eine Zeitperiode, deren Länge durch die Abspieldauer eines Teilstücks des Datenstroms bestimmt ist, wird in mehrere gleichlange Gruppen aufgeteilt. Jedem Datenstrom wird eine Gruppe zugewiesen, in der dessen Datenblöcke gelesen werden. Innerhalb einer Gruppe werden die Datenblöcke nach dem SCAN-Algorithmus gelesen, die einzelnen Gruppen werden nach Round-Robin abgearbeitet. Für die Admission Control wird für den neuen Strom zuerst die Zeit zum Lesen eines Teilstücks (*disk service time*) bestimmt und dann eine Gruppe gesucht, die noch genügend Leerlaufzeit hat, um diesen Strom zu bedienen.

Eine weitere Eigenschaft des EVEREST-Dateisystems ist, daß Anwendungen Einfluß auf die Auftragsplanung nehmen können. Ist das System nicht vollständig ausgelastet, können durch das Dateisystem in den entstehenden Leerlauf-Pausen bereits Blöcke im voraus gelesen werden. Um einen Pufferüberlauf in den Anwendungen zu verhindern, wird der „Füllstand“ des Puffers überwacht, beim Erreichen einer gewissen Marke sendet die Anwendung eine Stop-Nachricht an den Scheduler des Dateisystems, der daraufhin das Lesen neuer Böcke für einen angegebenen Zeitraum unterbricht.

### Video Server auf RT-Mach

Die Besonderheit des in [NOM 97] beschriebenen Video-Servers ist, daß die Datenströme nicht durch eine Blockliste (z.B. in einer Inode) beschrieben werden, sondern die Position der Datenblöcke berechnet werden kann. Dies wird durch ein fest vorgegebenes Datenlayout erreicht, so daß zur Beschreibung einer Datei nur die Kenntnis der Position des ersten Blocks sowie die Parameter des Layouts erforderlich sind. Der große Nachteil dieser Systeme ist allerdings, daß durch das fest vorgegebene Layout sehr schnell Fragmentierungsprobleme entstehen, besonders wenn häufig Datenströme gelöscht und andere neu geschrieben werden. Aus diesem Grund eignen sich diese Systeme nur für relativ statische Umgebungen, z.B. Video-Server bei denen nur selten neue Datenströme geschrieben werden.

### Symphony

Ziel von Symphony war es, ein Dateisystem zu entwickeln, das sowohl Continuous Media Daten als auch „normale“ Daten (z.B. Text) speichern kann [SGRV97]. Dabei werden die Daten auf ein und demselben Dateisystem gespeichert und nicht auf getrennten Dateisystemen, die zu einem logischen System zusammengefaßt werden. Durch diesen Ansatz kann eine bessere Leistung erzielt werden, da für jeden Datentyp das vollständige System zur Verfügung steht; der Ansatz erfordert jedoch eine aufwendige, sehr flexible Implementierung.

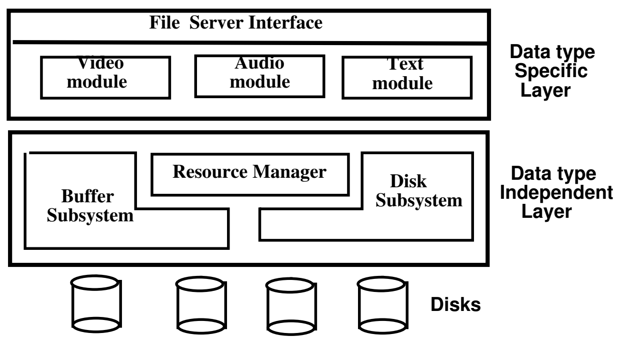



Abbildung 2.6: Systemstruktur des Symphony Multimedia File System [SGRV97].

Symphony verwendet einen zweischichtigen Aufbau (siehe Abbildung 2.6). Die untere Schicht ist unabhängig von dem verwendeten Datentyp und stellt grundlegende Dateisystemfunktionen zur Verfügung:

- Pufferverwaltung
  Die Pufferverwaltung ist in der Lage, verschiedene Cache-Strategien gleichzeitig zu verwenden. Dazu werden für jede Strategie getrennte Pufferbereiche verwaltet.

- Ressourcenverwaltung
  Durch die Ressourcenverwaltung wird im wesentlichen die Admission Control durchgeführt.

- Festplatten-Verwaltung
  In diesem Teil der unteren Schicht werden die Auftragsplanung und Blockverwaltung realisiert. Für die Auftragsplanung werden die Datenströme in 3 Kategorien eingeteilt:

  1. Periodische Echtzeitaufträge
  2. Nicht-periodische Echtzeitaufträge
  3. Nicht-Echtzeitaufträge

  Die Aufträge für jede Kategorie werden in getrennten Warteschlangen verwaltet. Für die Generierung der Festplattenaufträge wird eine gemeinsame Warteschlange benutzt, dabei werden die Aufträge für die Echtzeit-Datenströme (periodisch und nicht-periodisch) nach SCAN-EDF in diese Warteschlange eingefügt, die Nicht-Echtzeitaufträge nach Best-Effort. Um zu gewährleisten, daß die Aufträge gleichberechtigt ausgeführt werden, wird jeder Kategorie ein Zeitanteil garantiert. Aufträge werden nur dann in die globale Warteschlange aufgenommen, falls für die entsprechende Auftragskategorie noch genügend Zeit zur Verfügung steht bzw. keine Aufträge anderer Kategorien vorhanden sind. Durch die Vorgehensweise soll erreicht werden, daß Echtzeitaufträge entsprechend ihrer Zeitschranken ausgeführt werden und auf der anderen Seite die Antwortzeiten für Nicht-Echtzeitaufträge so gering wie möglich gehalten wird.

  Die Blockverwaltung soll die Möglichkeit bieten, verschiedene Layouts gleichzeitig zu verwenden. Dazu kann einer Allokationsanforderung ein „Hinweis“ gegeben werden, auf welcher Festplatte und an welcher Stelle der Festplatte ein Block reserviert werden soll. Außerdem können Blöcke unterschiedlicher Größe angelegt werden. Das Dateisystem versucht dann, den Block an der angegebenen Position oder in deren Nähe zu allokieren.

  Die Metadaten werden ähnlich zu den UNIX-Inodes verwaltet, es wird allerdings zusätzlich ein Index gespeichert, der die logische Struktur eines Datenstromes (z.B. die einzelnen Bilder eines Videos) auf Bytepositionen in der Datei abbildet.

In der oberen, vom jeweiligen Datentyp abhängigen Schicht werden die eigentlichen Strategien zur Speicherung der Datenströme implementiert. Für jeden Datentyp existiert dafür ein separates Modul. Das Modul zur Speicherung von Videoströmen verwendet z.B. große Blöcke und verteilt diese nach einem geeigneten Striping-Schema über die Festplatten, während das Text-Modul mit kleinen Blöcken arbeitet und diese strikt nach Round-Robin verteilt.

### 2.2.3 Echtzeit-Datenbanken

Im wesentlichen haben Datenbanksysteme zur Verwaltung von Echtzeit- oder Multimedia-Daten die gleichen Probleme zu lösen wie die Dateisysteme: Admission Control, Auftragsplanung und Datenlayout [RzS97, KGM93]. Hinzu kommt das Problem einer geeigneten Indexstruktur. Die bis jetzt beschriebenen Verfahren (Blocklisten oder spezielle Layoutvarianten) sind vor allem dann ungeeignet, wenn das System Operationen wie das Einfügen oder Löschen von Teilen des Datenstroms (z.B. einiger Bilder eines Videos) unterstützen soll. In [Bil92] wird für dieses Problem eine Lösung beschrieben. Dort werden B-Bäume für die Verwaltung der Dateien verwendet (siehe Abb. 2.7). Diese ermöglichen das Einfügen oder Löschen von Teilen innerhalb der Datei, erfordern allerdings einen erhöhten Aufwand für die Implementierung.

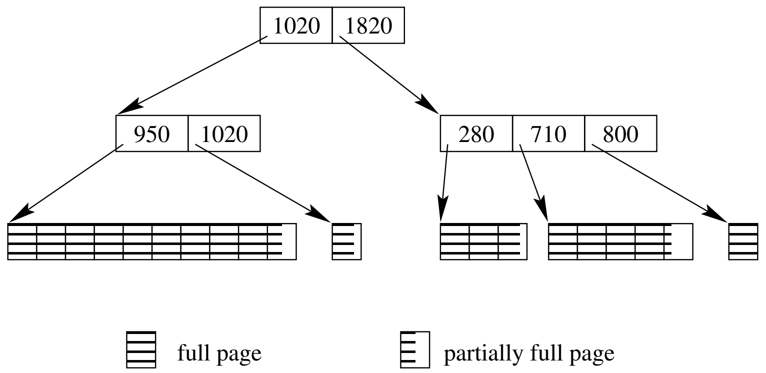



Abbildung 2.7: B-Bäume zur Blockverwaltung in EOS [Bil92].

# Kapitel 3

# Entwurf

## 3.1 Grundlagen

Bevor die verschiedenen Entwurfsmöglichkeiten beschrieben werden, sollen an dieser Stelle zunächst die wichtigsten Grundlagen und Rahmenbedingungen erläutert werden.

### 3.1.1 SCSI-Auftragsplanung

Das Dateisystem soll zum Zugriff auf die Festplatten den im Rahmen des DROPS-Projekts von Frank Mehnert entwickelten SCSI-Treiber verwenden [Meh98]. Dieser Treiber ist in der Lage, Zusagen über die Abarbeitungsdauer der Aufträge zu geben. Dazu werden die Aufträge in *Slots* abgearbeitet (siehe Abb. 3.1). Diese Slots haben eine feste Länge, die durch die Worst-Case Bedienungszeit[1] bestimmt ist.

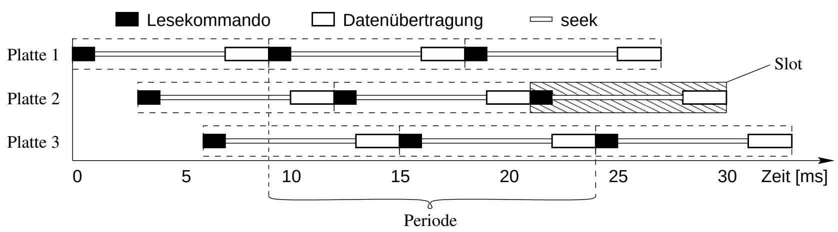



Abbildung 3.1: Auftragsbearbeitung des SCSI-Treibers [Meh98].

Während der Kopfpositionierung einer Festplatte, bei der der SCSI-Bus durch diese nicht belegt ist, können Aufträge an andere Festplatten übertragen werden. Die maximale Anzahl der Aufträge, die in einer solchen *Periode* bearbeitet werden können, kann aus dem Verhältnis der Übertragungsdauer des Kommandos und der Daten sowie der Positionierungszeit berechnet werden. Die maximale Größe eines Datenblocks, der innerhalb eines Slots gelesen werden kann, wird durch das SCSI-System anhand der Eigenschaften der verwendeten Hardware bestimmt, übliche Werte sind 64 KByte oder 128 KByte. Das Dateisystem sollte möglichst auch mit dieser Größe arbeiten, da bei kleineren Blöcken nicht die volle Leistungsfähigkeit erreicht werden kann.

Für die Übertragung der Aufträge an den SCSI-Treiber existieren zwei verschiedene Varianten:

[1] Die Bedienungszeit setzt sich aus der Zeit die für die Übertragung des Lesekommandos und der Daten sowie der Positionierungszeit des Festplattenkopfes zusammen.

```
struct request {
    unsigned int period;
      /* diskrete Zeitangabe, in welcher Periode der Auftrag auszuführen ist */
    unsigned int slot;
      /* welcher Slot soll belegt werden */
    scsi_block_t blk;
      /* Angaben über Partitionsnummer, Blocknummer und -länge */
    byte_t *map_address;
      /* Adresse des Schreib-/Lesepuffers */
    word_t status;
      /* Status Bits */
    word_t reserved;
};
```

Abbildung 3.2: Auftragsstruktur für Echtzeit-Aufträge [Meh98]

- Die Aufträge für Echtzeitdatenströme werden in Form von Auftragsfeldern an den SCSI-Treiber gesendet. Jeder einzelne Auftrag wird durch eine Request-Struktur (siehe Abb. 3.2) beschrieben. Der SCSI-Treiber sortiert die Aufträge in einen Plan ein, der durch die Verwendung der Perioden und Slots entsteht. Zu beachten ist, daß die angegebene Adresse des Puffers die physische Speicheradresse ist, da der SCSI-Treiber die Daten direkt durch DMA (*Direct Memory Access*) in den Speicher überträgt.

- Nicht-Echtzeitaufträge werden einzeln an den Treiber gesendet. Dieser versucht sie so schnell wie möglich zu bearbeiten.

### 3.1.2 DROPS / L⁴Linux

Am Lehrstuhl für Betriebssysteme der Technischen Universität Dresden wird derzeit an der Entwicklung eines Echtzeitbetriebssystems, dem *Dresden Realtime Operating System*, gearbeitet. Dieses System soll durch die Verwendung spezieller Algorithmen zur Verwaltung der Systemressourcen (CPU-Scheduling, Speicherverwaltung, ...) die Abarbeitung von Echtzeitanwendungen parallel zu UNIX-Anwendungen unterstützen (siehe Abb. 3.3).

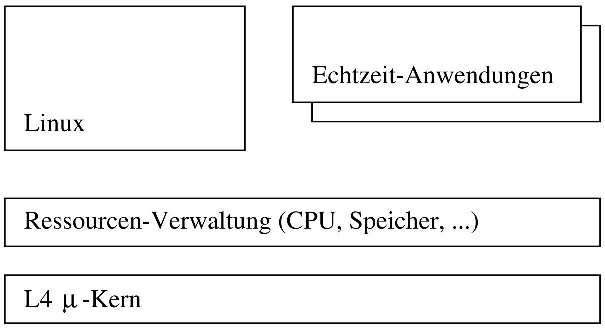



Abbildung 3.3: Struktur des *Dresden Realtime Operating System (DROPS)*

Als Grundlage für dieses System wird der Mikrokern L4 verwendet [Lie96], für die Unterstützung der Nicht-Echtzeit Anwendungen wurde der Linux-Kern auf L4 portiert [Hoh96]. Als Hauptanwendungsgebiete werden Multimedia-Systeme betrachtet, z.B. die Verarbeitung von Audio- oder Videoströmen.

### 3.1.3 QoS-Parameter

Die Anforderungen eines Datenstroms müssen dem Dateisystem auf eine geeignete Weise übergeben werden. Für konstante Datenströme, z.B. unkomprimierte Videoströme, ist das recht einfach, die Angabe der geforderten Datenrate ist für das Dateisystem ausreichend, um die Einplanung vorzunehmen. Andere Daten, z.B. komprimierte Videos, weisen jedoch keine konstante Datenrate auf, die Datenmengen, die pro Zeiteinheit gelesen werden müssen, schwanken. In [MHN95] wird eine Möglichkeit beschrieben, derartige Datenströme zu charakterisieren. Dabei wird der Datenstrom in sehr kleine Abschnitte zerlegt (die Dauer eines Abschnitts kann z.B. 0,5 s betragen) und für jeden dieser Abschnitte die Datenmenge bestimmt. Die so entstehende Zahlenfolge wird als Grundlage für die Einplanung verwendet. Diese Vorgehensweise ermöglicht eine sehr genaue Planung, erfordert jedoch aufgrund der großen Menge an Beschreibungsdaten einen sehr hohen Aufwand.

Um diese großen Datenmengen zu vermeiden, werden für die Beschreibung der Datenströme nicht die pro Zeiteinheit gelesenen Datenmengen betrachtet sondern ein Datenstrom wird durch die Angabe einer mittleren Datenrate sowie eines Parameters beschrieben, der die Schwankung der tatsächlichen um diese mittlere Datenrate widerspiegelt. In [Ham97] ist das dieser Beschreibung zugrunde liegende Modell erläutert. Da durch die Verwendung einer mittleren Datenrate die genaue Position der einzelnen Schwankungen nicht mehr bekannt ist, kann die Planung nicht mehr so genau vorgenommen werden wie bei der Verwendung der zuerst erläuterten Beschreibung, die entstehende Abweichung kann jedoch begrenzt werden, indem ein Strom in wenige große Abschnitte aufgeteilt wird, für die ein separater Parametersatz verwendet wird.

### 3.1.4 Admission Control

Aufgabe der Admission Control ist die Entscheidung über die Zulassung neuer Aufträge. Dazu werden die Anforderungen eines Auftrags anhand der übergebenen Beschreibung berechnet und mit den zur Verfügung stehende Ressourcen sowie der aktuellen Systemlast verglichen [Rud97a]. Für das Dateisystem sind im Moment zwei Ressourcen von Bedeutung: der benötigte Pufferplatz und die Festplattenbandbreite. Der benötigte Pufferplatz läßt sich aus der geforderten Datenrate und der Dauer berechnen, die der Puffer von der Anwendung benötigt wird.

Die Planung der verfügbaren Festplattenbandbreite ist sehr stark durch die beschriebene Auftragsplanung des SCSI-Treibers beeinflußt. Die maximale Bandbreite wird erreicht, indem alle verfügbaren Slots einer Periode belegt werden[2]. Kleinere Bandbreiten können zunächst erreicht werden, indem nicht alle Slots einer Periode belegt werden. Die kleinste Bandbreite entspricht dann der Bandbreite, die durch das Verwenden eines einzelnen Slots erzielt wird. Für reale Werte (Periodendauer 40 ms, 128 KByte Daten pro Slot) ergibt sich dadurch eine Bandbreite von 3,125 MByte/s. Die Bandbreitenanforderungen realer Datenströme liegen jedoch deutlich darunter (ein MPEG-2 Strom hat z.B. eine maximale Datenrate von 500 KByte/s). Diese können unterstützt werden, indem nicht alle aufeinanderfolgenden Perioden für den Datenstrom verwendet werden. Je kleiner die geforderten Bandbreiten sind, um so mehr Perioden müssen ausgelassen werden. Eine derartige Abfolge von verwendeten Perioden wird im weiteren ein *Zyklus* genannt, die Länge des Zyklus wird durch die minimale unterstützte Bandbreite bestimmt.

### 3.1.5 Begriffsbestimmung

Im folgenden werden die Zugriffe auf die Dateien in drei Kategorien unterteilt:

[2] Dabei ist zu berücksichtigen, daß für eine Festplatte in einer Periode nur maximal ein Auftrag bearbeitet werden kann. Um die maximale Bandbreite des SCSI-Buses ausnutzen zu können, müssen die Daten geeignet über alle Festplatten verteilt werden.

**Kontinuierlicher Echtzeitzugriff** Diese Zugriffsart entspricht dem Vorgehen bei der Bedienung von Continuous Media Daten. Die Datei wird linear bearbeitet, für ein Video werden z.B. die einzelnen Bilder der Reihe nach gelesen. Die Teilobjekte der Datei müssen dabei zu einem durch die verwendete Datenrate bestimmten Zeitpunkt bearbeitet werden.

**Nicht-kontinuierlicher Echtzeitzugriff** Bei dieser Kategorie wird auf die Datei wahlfrei zugegriffen, die Position der einzelnen Zugriffe ist im voraus nicht bekannt. Das Dateisystem muß hier gewährleisten, daß die Aufträge innerhalb eines angeforderten Zeitbereichs abgearbeitet werden.

**Nicht-Echtzeitzugriff** Bei dieser Zugriffsart müssen durch das Dateisystem keine Zusagen hinsichtlich der Bearbeitungsdauer eines Auftrags eingehalten werden.

## 3.2 Entwurfsziele

Ziel dieser Arbeit ist die Entwicklung eines echtzeitfähigen Dateisystems unter Verwendung des in Abschnitt 3.1.1 beschriebenen SCSI-Treibers. Das Dateisystem stellt dabei eine Echtzeitkomponente des DROPS dar. Der Entwurf soll folgende Bedingungen berücksichtigen:

- Das Dateisystem soll verschiedene Datentypen gleichzeitig unterstützen können. Dazu gehören die klassischen Continuous Media Daten wie zum Beispiel Video- oder Audioströme, aber auch nicht-kontinuierliche Echtzeitdaten und Nicht-Echtzeitdaten wie etwa Texte oder Bilder.

- Der Zugriff auf die Daten soll sowohl von Echtzeitanwendungen als auch von $L^4$Linux-Anwendungen möglich sein.

- Für die Übertragung der Daten an die Echtzeitanwendungen soll ein geeignetes Puffersystem entwickelt werden.

Als mögliche Anwendungsgebiete für das Dateisystem sind kleinere bis mittlere Informationsysteme, z.B. auf Messeständen oder Flughäfen oder auch kleinere Video-Server denkbar.

## 3.3 Systemstruktur

Abbildung 3.4 gibt einen groben Überblick über die Struktur des Gesamtsystems. Der SCSI-Treiber ist eine eigenständige DROPS-Komponente, das Dateisystem enthält als eine Teilkomponente die Admission Control. Im folgenden sollen das Zusammenwirken dieser einzelnen Komponenten sowie die sich daraus ergebenden Schnittstellen zwischen den einzelnen Komponenten beschrieben werden.

### 3.3.1 Zusammenwirken Admission Control – Dateisystem – SCSI Treiber

Durch die Admission Control wird überprüft, ob anhand der augenblicklichen Systemlast ein neuer Datenstrom akzeptiert werden kann oder nicht. Stehen noch genügend Ressourcen zur Bearbeitung der Anforderung zur Verfügung, wird eine entsprechende Zusage an die Anwendung gegeben. Die Aufgabe des Dateisystems und des SCSI-Treibers ist, die Einhaltung dieser Zusage zu garantieren. Für einige Ressourcen (z.B. den benötigten Pufferspeicher) kann das durch eine statische Reservierung erfolgen. Für die Garantierung der zugesagten Bandbreite sind mehrere Modelle denkbar:

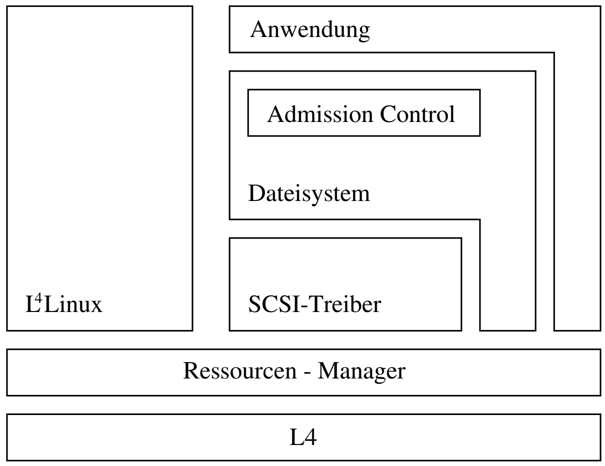



Abbildung 3.4: Struktur des Gesamtsystems

#### Getrennte Admission Control und Auftragsplanung

Der Test in der Admission Control erfolgt auf Basis der verfügbaren Gesamtbandbreite, die sich durch die Anzahl der Slots pro Periode sowie die Länge und gelesene Datenmenge eines Slots ergibt. Es wird überprüft, ob die geforderte Bandbreite (aufgerundet auf ein Vielfaches der Minimalbandbreite) noch verfügbar ist. Das Dateisystem erzeugt dann in gewissen Abständen eine Liste von Aufträgen und übergibt diese dem SCSI-Treiber.

Damit die durch die Admission Control gemachte Zusage garantiert werden kann, sind durch das Dateisystem einige Bedingungen einzuhalten. Die wichtigste Bedingung ist, daß die Aufträge eines Zyklus gesammelt werden und bereits eine Zyklusdauer im voraus bekannt sein müssen. Dies ist erforderlich, da durch die Admission Control lediglich gewährleistet wird, daß innerhalb eines Zyklus ausreichend Slots zur Verfügung stehen, deren Position jedoch nicht festgelegt wird. Diese wird erst im SCSI-Treiber bestimmt. Aus dieser Bedingung folgt, daß immer mindestens ein vollständiger Zyklus gepuffert werden muß, um die Zusagen einzuhalten. In Abhängigkeit von der Zyklusdauer entsteht dadurch ein sehr großer Pufferbedarf (siehe Tabelle 3.1 auf Seite 26), wodurch diese Vorgehensweise besonders für Ströme mit kleinen Bandbreitenanforderungen (und damit großen Zykluslängen) nicht praktikabel ist.

#### Gekoppelte Admission Control und Auftragsplanung

Der benötigte Pufferbedarf kann verringert werden, indem bereits durch die Admission Control die genauen Positionen bestimmt werden, an denen die Aufträge eines einzelnen Datenstroms gelesen werden [Här97b, Rud97a] (siehe Abb. 3.5).

Die Kenntnis der genauen Positionen kann durch das Dateisystem für eine genauere Planung verwendet werden. Die Aufträge eines Datenstroms werden immer nur in den dafür vorgesehenen Slots abgearbeitet. Dies kann erreicht werden, indem entweder die Slot-Nummern direkt an den SCSI-Treiber übergeben werden oder indem die einzelnen Aufträge durch das Dateisystem eine ihrer Slot-Position entsprechende Deadline zugewiesen bekommen.

### 3.3.2 Schnittstellen

Das Dateisystem soll zwei Zugriffsvarianten unterstützen:

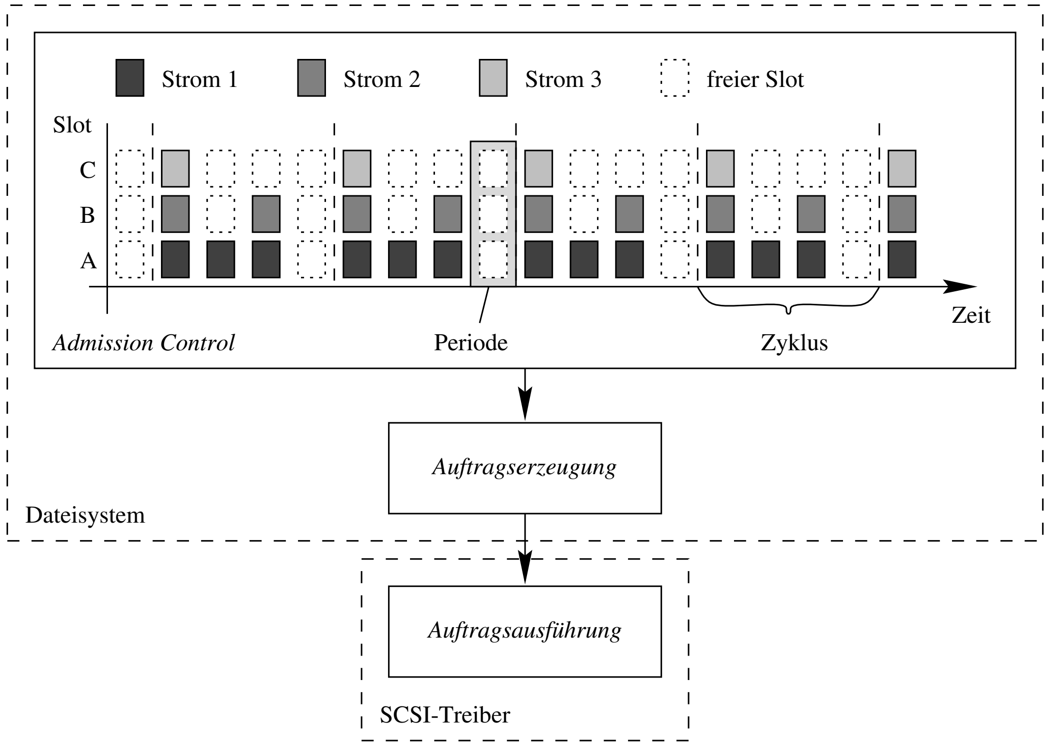



Abbildung 3.5: Zuordnung der SCSI-Slots durch die Admission Control

1. Echtzeitanwendungen sollen auf die Daten in Echtzeit und in Nicht-Echtzeit zugreifen können
2. L$^4$Linux-Anwendungen sollen auf die Daten in Nicht-Echtzeit zugreifen können.

Die Schnittstellen für den Zugriff einer Echtzeitanwendung sind für die drei Zugriffsarten unterschiedlich realisiert:

- Kontinuierliche Echtzeitdaten
  Diese Daten sind durch ein vorhersagbares Zugriffsverhalten gekennzeichnet, da die Dateien linear und mit einer festgelegten Datenrate bearbeitet werden. Die Kenntnis dieser Eigenschaft kann sich das Dateisystem zu Nutze machen, indem es selbstständig die Aufträge generiert und die Daten an die Anwendung weiterleitet (so können die Daten durch das Dateisystem in einen mit der Anwendung gemeinsam genutzten Speicherbereich geschrieben werden). Durch dieses aktive Vorgehen des Dateisystems kann der Kommunikationsaufwand mit der Anwendung auf wenige Nachrichten zum Starten der Übertragung reduziert werden, und dies ermöglicht eine bessere Kontrolle über die Auftragsplanung. Sollen allerdings solche Operationen wie das schnelle Vor- oder Rückspulen bei Videos unterstützt werden, sind zusätzliche Nachrichten zu definieren.

  Auch beim Schreiben der Echtzeitdaten soll das Dateisystem aktiv die Aufträge zum Schreiben der Daten erzeugen. Dies ist erforderlich, um eine Einhaltung des vorgegebenen Datenlayouts zu erreichen (siehe dazu Abschnitt 3.4.2).

- Nicht-kontinuierliche Echtzeitdaten
  Bei nicht-kontinuierlichen Datenströmen sind die Zugriffe auf die Daten nicht vorhersagbar, es fehlt dem Dateisystem die Grundlage für ein aktives Senden der Daten. Es muß also auf konkrete Aufträge warten, die von der Anwendung erzeugt werden. Dieses Verfahren entspricht der Zugriffsart bei

klassischen Dateisystemen, es wird jedoch zusätzlich zu jedem Auftrag ein Zeitpunkt angegeben, bis wann dieser zu bearbeiten ist.

- Nicht-Echtzeitdaten
  Auf die Nicht-Echtzeitdaten wird analog zu den klassischen Dateisystemen zugegriffen. Die Anwendung erzeugt für jeden Zugriff einen Auftrag, den das Dateisystem bearbeitet.

Um eine möglichst einheitliche Schnittstelle für $L^4$Linux-Anwendungen zu bieten, soll das Dateisystem in das *Virtual File System (VFS)* [BBD 95] integriert werden. Dadurch kann das Dateisystem in einen bestehenden Verzeichnisbaum eingebunden werden, und der Zugriff auf die Dateien erfolgt über $L^4$Linux-Systemrufe.

## 3.4 Block-Verwaltung

Durch das Block-Subsystem werden die physischen Blöcke auf den Festplatten verwaltet und es stellt Mechanismen zu Allokation und Freigabe dieser Blöcke zur Verfügung.

### 3.4.1 Freispeicherverwaltung

Die Fragestellungen beim Entwurf der Freispeicherverwaltung sind wieviele Blockgrößen unterstützt werden sollen (eine feste oder mehrere) und wie groß diese sind.

Standard-Dateisysteme verwenden in der Regel eine feste Blockgröße, die beim Erzeugen des Dateisystems festgelegt wird. Für das hier betrachtete Dateisystem könnte die Größe entsprechend der Datenmenge gewählt werden, die pro Slot gelesen werden kann (z.B. 128 KByte). Die Verwendung so großer Blöcke führt allerdings bei Nicht-Echtzeitdaten zu einem sehr großen Verschnitt, da Dateien nur in Vielfachen von dieser Blockgröße angelegt werden können. Eine Lösung besteht in der Verwendung von verschiedenen Blockgrößen für Echtzeit- und Nicht-Echtzeitdaten wie z.B. in [Kli97]. Dort werden zwei verschiedene, feste Blockgrößen unterstützt.

Bei der Bestimmung der verwendeten Blockgröße ist auch zu berücksichtigen, daß die in einem Slot gelesene Datenmenge $d_{Slot}$ Einfluß auf die Länge des in Abschnitt 3.1.4 beschriebenen Zyklus hat. Die Länge $t_{Zyklus}$ und damit auch die Anzahl $n_{Zyklus}$ der Perioden in einem Zyklus ergibt sich bei gegebener Periodendauer $t_{Periode}$ und geforderter Minimaldatenrate $b_{min}$ durch:

$$t_{Zyklus} = \frac{d_{Slot}}{b_{min}} \tag{3.1}$$

$$n_{Zyklus} = \frac{d_{Slot}}{b_{min} * t_{Periode}} \tag{3.2}$$

Bei sehr kleinen Minimaldatenraten kann die Zykluslänge sehr groß werden, für reale Werte kann sie mehrere Sekunden betragen (siehe Tabelle 3.1).

Durch große Zykluslängen können zwei Probleme entstehen:

- Soll das Dateisystem viele Datenströme mit der minimalen Datenrate bearbeiten, entsteht ein sehr hoher Pufferbedarf. Dieser resultiert daraus, daß für jeden Datenstrom mindestens ein Puffer mit der Größe von $d_{Slot}$ über die komplette Dauer des Zyklus benötigt wird. Die Daten werden in einem Slot des Zyklus in diesen Puffer gelesen, die Anwendung liest diese über die Dauer des Zyklus aus diesem Puffer. In Tabelle 3.1 ist für den Extremfall, daß das Dateisystem die maximale Anzahl an Datenströmen mit der minimalen Datenrate liefert, der benötigte Puffer aufgeführt. In diesem Fall wird dann durch jeden Slot des Zyklus ein anderer Datenstrom bearbeitet.

  Für die Begrenzung des benötigten Puffers gibt es zwei Möglichkeiten, zum einen kann die Anzahl der Datenströme in Abhängigkeit des zur Verfügung stehenden Pufferspeichers eingeschränkt werden, zum anderen kann die Zykluslänge verkürzt werden, indem pro Slot weniger Daten gelesen

| min. Bandbreite $b_{min}$ [KByte/s] | Zykluslänge $t_{Zyklus}$ [s] | Anzahl Perioden pro Zyklus $n_{Zyklus}$ | Pufferbedarf [MByte] |
|---|---|---|---|
| 3200 | 0,04 | 1 | 0,625 |
| 1067 | 0,12 | 3 | 1,875 |
| 128 | 1,00 | 25 | 15,625 |
| 64 | 2,00 | 50 | 31,250 |
| 8 | 16,00 | 400 | 250,000 |
| 1 | 128,00 | 3200 | 2000,000 |

Tabelle 3.1: Zykluslänge und Pufferbedarf (bei voller Auslastung) in Abhängigkeit der minimalen Bandbreite ($t_{Periode} = 40ms, d_{Slot} = 128KByte$, 5 Slots pro Periode).

werden. Die erste Variante hat den Vorteil, daß in den freien Slots, die durch die Begrenzung der Anzahl der Datenströme entstehen, Nicht-Echtzeitdaten gelesen werden können.

- Das zweite Problem ist die hohe Latenz beim Starten bzw. beim Wiederaufnehmen einer Datenübertragung. Eine Anwendung muß dabei solange warten, bis innerhalb des Zyklus der Slot erreicht ist, der der Anwendung durch die Admission Control zugewiesen wurde. Eine Verringerung dieser Zeit kann nur erreicht werden, indem die Zykluslänge durch einen kleineren Wert für $d_{Slot}$ begrenzt wird.

Die Frage ist nun, ob die verschiedenen Werte für $d_{Slot}$ auch bei der Blockallokation berücksichtigt oder ob die Dateien generell mit der maximalen Größe von $d_{Slot}$ angelegt werden. Durch die Verwendung kleinerer Blockgrößen können mehr Anwendungen gleichzeitig auf die Datei zugreifen, bei der Allokation mit der festen Blockgröße ist die maximale Datentrate höher, mit der eine Datei theoretisch gelesen werden könnte. Für das Dateisystem ist eher die gleichzeitige Bedienung mehrerer Datenströme von Bedeutung, so daß auch variable Blockgrößen für die Allokation verwendet werden[3].

Für die Behandlung variabler Blockgrößen existieren verschiedene Ansätze. Ein einfacher Weg ist die Reservierung separater Bereiche auf der Festplatte für jede Blockgröße. Diese Aufteilung ist jedoch nicht flexibel, was besonders bei im voraus nicht bekannten Datenströmen zu Problemen führt, da Bereiche oft benötigter Blockgrößen bereits aufgebraucht sind, während andere noch ausreichend Speicherplatz enthalten, der jedoch nicht benutzt werden kann.

Flexibler kann der Speicherplatz durch die Verwendung des aus der Hauptspeicherverwaltung bekannten Buddy-Algorithmus verwaltet werden. Ausgehend von der kleinsten Blockgröße $B_{Basis}$, die z.B. durch die Hardware oder den SCSI-Treiber vorgegeben wird, werden benachbarte Blöcke zu einem größerem Block zusammengefaßt (siehe Abb. 3.6)[4].

Die Anzahl $S$ der verfügbaren Blockgrößen $B_i$ ist abhängig von der Kapazität der Festplatte und $B_{Basis}$:

$$S = \left\lceil \mathrm{ld}\left(\left\lfloor \frac{C_{Festplatte}}{B_{Basis}} \right\rfloor\right)\right\rceil \qquad (3.3)$$

$$B_i = B_{Basis} * 2^i \qquad (0 \leq i < S) \qquad (3.4)$$

Die Initialisierung wird so vorgenommen, daß der Speicherplatz in die größtmöglichen Blöcke aufgeteilt ist. Bei der Allokation wird zunächst versucht, einen Block auf der entsprechenden Ebene zu finden. Ist dort kein Block verfügbar, wird ein Block einer höheren Ebene geteilt. Beim Freigeben wird überprüft, ob der Block mit dem benachbarten zu einem größeren Block zusammengefaßt werden kann.

[3] Diese Entscheidung sollte nach der Implementierung noch einmal überprüft werden, da die Verwendung kleiner Blockgrößen bei der Allokation einen großen Einfluß auf die Gesamtleistung des Systems hat.

[4] Hier wird der *binäre Buddy-Algorithmus* verwendet, d.h. es werden immer zwei benachbarte Blöcke zusammengefaßt. Es sind auch Systeme mit einer größeren Anzahl an zusammenzufassenden Blöcken denkbar, bei diesen ist der Abstand zwischen den verfügbaren Blockgrößen jedoch größer.

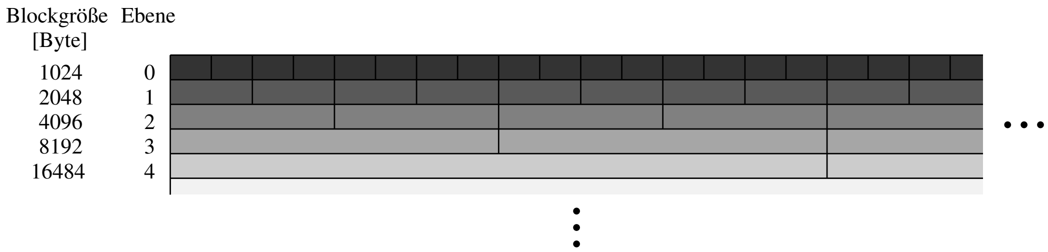



Abbildung 3.6: Blockverwaltung mittels Buddy-Algorithmus

### 3.4.2 Datenlayout

Bei der Einteilung der Perioden durch den SCSI-Treiber werden die Worst-Case-Zeiten für die Positionierung des Festplattenkopfes verwendet. Es ist daher nicht sinnvoll, im Datenlayout Rücksicht auf die Lage der einzelnen Blöcke auf einer Festplatte zu nehmen. Vielmehr muß durch das Datenlayout gewährleistet werden, daß in einer Periode nicht mehrere Aufträge für eine Festplatte vorliegen. Weiterhin soll durch das Layout eine möglichst hohe Datenrate unterstützt werden. Daraus resultiert zunächst, daß die Blockgröße, die zum Speichern der Dateien verwendet werden, in der Regel anhand der Datenmengen, die pro Slot gelesen werden, gewählt wird. Kleinere Blockgrößen werden nur dann verwendet, wenn dies aufgrund einer zu großen Zykluslänge erforderlich wird (siehe letzter Abschnitt).

Um zu verhindern, daß in einer Periode mehrere Aufträge für eine Festplatte vorliegen, muß durch das Layout gewährleistet werden, daß aufeinanderfolgende Datenblöcke einer Datei nicht auf der gleichen Festplatte liegen. Für wieviele aufeinanderfolgende Datenblöcke das gilt, hängt von der Anzahl der Slots ab, die ein Datenstrom pro Periode benötigt. Belegt ein Datenstrom maximal einen Slot, so kann dieser theoretisch vollständig auf einer Festplatte gespeichert werden. Benötigt der Datenstrom jedoch mehrere Slots, so müssen entsprechend viele Datenblöcke auf unterschiedlichen Festplatten verteilt sein (bei zwei verwendeten Slots pro Periode müssen jeweils benachbarte Blöcke auf unterschiedlichen Festplatten liegen, bei drei Slots jeweils drei aufeinanderfolgende Blöcke usw.). Auch wenn eine Datei nur einen Slot pro Periode benötigt, ist es jedoch auch dort nicht sinnvoll, diese auf einer einzelnen Festplatte zu speichern. Dies hat vor allem zwei Gründe:

- Soll der Datenstrom durch verschiedene Anwendungen gleichzeitig an unterschiedliche Positionen gelesen werden können, so müssen die Daten über möglichst viele Festplatten verteilt sein. Wird die Datei nur auf einer Festplatte gespeichert und wird pro Periode ein Slot zum Lesen benötigt, kann nur eine Anwendung diesen Datenstrom lesen. Wird dieselbe Datei jedoch z.B. über zwei Festplatten zyklisch verteilt, können auch zwei Anwendungen die Datei lesen, indem die Aufträge ineinander geschachtelt werden (in der ersten Periode liest die erste Anwendung von der ersten Festplatte, die zweite von der zweiten, in der darauf folgenden liest die erste Anwendung von der zweiten und die zweite Anwendung von der ersten Festplatte).
- Wird das Dateisystem zum Speichern und Abspielen von Videos verwendet und soll ein schneller Vorlauf ermöglicht werden, ist ebenfalls eine Verteilung über viele Festplatten notwendig. Durch den schnellen Vorlauf wird der Abarbeitungsplan zusammengestaucht, d.h. bei einem Vorlauf mit vierfacher Geschwindigkeit benötigt ein Datenstrom auch die vierfache Anzahl an Slots pro Periode. Ein Datenstrom, der normalerweise genau einen Slot pro Periode benötigt, benötigt dann vier Slots innerhalb einer Periode. Dies erfordert, daß vier aufeinanderfolgende Blöcke der Datei auf jeweils unterschiedlichen Festplatten gespeichert sind.

Diese Anforderungen werden durch eine Round-Robin-Verteilung der Daten über die Festplatten ähnlich dem *Staggered Striping* [BGMJ94] erreicht. Die Blöcke einer Datei werden der Reihe nach über die zur

Verfügung stehenden Festplatten verteilt, beim Erreichen der letzten Festplatte wird wieder mit der ersten Festplatte begonnen.

Durch eine geeignete Abwandlung dieses Datenlayouts kann noch ein weiteres Problem gelöst werden. Durch die Admission Control werden die geforderten Bandbreiten der Datenströme auf Vielfache der minimalen Bandbreite aufgerundet und die Festlegung der Slots anhand dieser gerundeten Bandbreite vorgenommen. Dadurch werden die Daten mit einer größeren Bandbreite gelesen als die Anwendung angefordert hat, das Dateisystem liest also mehr Daten als die Anwendung verarbeitet. Um ein zu starkes Auseinanderlaufen der Anwendung und des Dateisystems zu verhindern, werden, sobald der Vorlauf groß genug ist, Datenblöcke bei der Allokation ausgelassen [Rud97b].

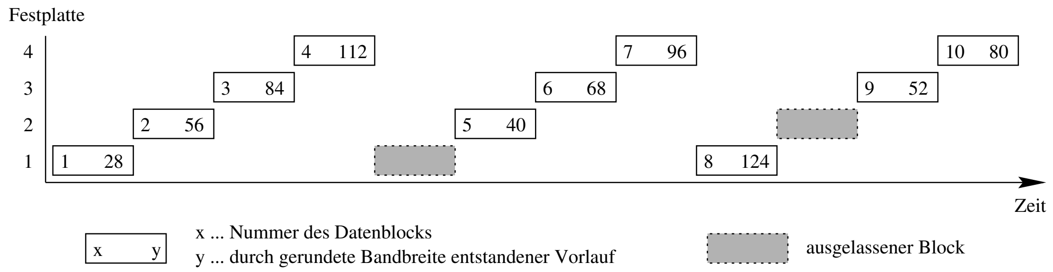



Abbildung 3.7: Anpassung der Lese- und Verarbeitungsbandbreiten durch Auslassen von Datenblöcken

Abbildung 3.7 verdeutlicht dies an einem Beispiel. Das Dateisystem liefert pro Periode einen 128 KByte-Datenblock, während die Anwendung im selben Zeitraum nur 100 KByte verarbeitet. Nach dem vierten Block ist der dadurch entstehende Vorlauf so groß, daß der nächste Block übersprungen werden kann und statt dessen die Anwendung die Daten aus dem Puffer bearbeitet.

Für die ausgelassenen Datenblöcke werden durch das Dateisystem keine Aufträge für den SCSI-Treiber erzeugt, d.h. dieser kann die eigentlich dafür vorgesehenen Slots für die Bedienung anderer Aufträge verwenden.

### Anlegen der Dateien

Für das Anlegen der Dateien und damit die Erzeugung des entsprechenden Datenlayouts sind zwei Verfahren zu unterscheiden:

- **Nicht-Echtzeitschreiben einer Datei**
  Beim Schreiben einer Echtzeit-Datei in Nicht-Echtzeit, z.B. von $L^4$Linux aus, kann diese bevor sie geschrieben wird analysiert und anhand der bestimmten realen und gerundeten Bandbreite das Layout berechnet werden.

- **Echtzeitschreiben einer Datei**
  Soll eine Echtzeit-Datei auch in Echtzeit geschrieben werden, so besteht das Problem, daß die tatsächlichen Bandbreitenanforderungen der Datei im voraus nicht bekannt sind, es können meistens nur obere Grenzwerte für die zu erwartenden Anforderungen angegeben werden. Somit fehlen für eine genaue Bestimmung des Datenlayouts die erforderlichen Eingangsdaten. Ein erster Ansatz, die Bestimmung des Layouts anhand der oberen Grenzwerte vorzunehmen und die Daten auch kontinuierlich entsprechend diesen Layouts zu speichern, ist nicht anwendbar, da der reale Schreibablauf und der durch die Admission Control vorbestimmte Verlauf in der Regel voneinander abweichen und somit die Gefahr besteht, daß Schreiboperationen aufgrund anderer Anforderungen im SCSI-Treiber nicht ausgeführt werden können.

Um ein Auseinanderlaufen des realen mit dem eingeplanten Schreibablauf zu verhindern, muß das Dateisystem den geplanten Ablauf durchsetzen, auch wenn dadurch Lücken innerhalb der Datei entstehen. Um das Echtzeit-Schreiben garantieren zu können, wird die Einplanung durch die Admission Control und auch das Datenlayout anhand der oberen Grenzwerte für den Datenstrom durchgeführt und auch ausgeführt. Nachdem die Datei so geschrieben wurde, muß eine Reorganisation erfolgen, die den tatsächlichen Bandbreitenbedarf der Datei ermittelt und die Daten dann entsprechend dieser Anforderung neu auf die Festplatten schreibt.

## 3.5 Dateien

Die zwei wesentlichen Möglichkeiten zur Verwaltung der Dateien bestehen in einer blocklisten-orientierten Verwaltung wie bei den bekannten Standard-Dateisystemen oder einer Verwaltung, die die Datei anhand eines speziellen Layouts beschreibt (wie z.B. in [NOM 97]). Aufgrund der bisher vorgestellten Vorgehensweise beim Datenlayout ist die zweite Variante nicht anwendbar, da durch das Layout nur die Festplatte, aber nicht die Position auf dieser bestimmt wird.

Für die Verwaltung der Blockliste wird eine Struktur verwendet, die den UNIX-Inodes ähnlich ist. Inodes sind im allgemeinen als gut geeignet für die Verwaltung kleiner Dateien bekannt. Bei großen Dateien besteht jedoch das Problem, daß durch die indirekte Speicherung der Blocknummern mehrere Plattenzugriffe notwendig sind, um auf die Blockliste der Datei zuzugreifen. Um dieses Problem zu umgehen, ist der Bereich für die Speicherung der direkten Blocknummern innerhalb der Inode gegenüber UNIX-Inodes deutlich größer. Der Bereich wird so groß gewählt, daß die gesamte Blockliste der Datei mit einem Zugriff gelesen werden kann, d.h. daß diese Liste maximal mit einer einfachen Indirektion gespeichert ist. Die dadurch entstehenden Inode-Größen liegen im Bereich der verwendeten Datenblock-Größen (die Blockliste einer 2 GByte großen Datei ist bei der Verwendung von 128 KByte-Blöcken und einer 4 Byte großen Blocknummer 64 KByte groß), aus diesem Grund werden die Inodes auch nicht in einer separaten Inode-Tabelle gespeichert, sondern als normale Datenblöcke.

Das Echtzeitdateisystem unterstützt auch Verzeichnisse. Diese werden analog zu den Verzeichnissen im ext2fs durch eine einfach verkettete Liste verwaltet. Ein Eintrag in dieser Liste ordnet dem Dateinamen eine Inode zu, die Inode wird durch die Nummer des Blocks, in dem sie gespeichert ist, bezeichnet.

## 3.6 Puffersystem

Der Entwurf des Puffersystems ist sehr stark mit dem Verarbeitungsmodell in DROPS verbunden, so daß dieses hier kurz erläutert werden soll. Bei diesem Verarbeitungsmodell wird davon ausgegangen, daß ein Datenstrom durch eine Kette von nacheinanderfolgenden Anwendungen bearbeitet wird. Eine solche Kette könnte beispielsweise aus dem Dateisystem, welches ein MPEG-Video liefert, einem Dekoder, der dieses Video dekodiert und einem Grafikkartentreiber, der dieses Video dann anzeigt, bestehen. Die Daten werden zwischen diesen Anwendungen über gemeinsam genutzten Speicher ausgetauscht, um möglichst selten Daten zu kopieren.

Für die Verwaltung des gemeinsam genutzten Speichers gibt es zwei verschiedene Varianten. Der Speicher kann permanent in allen Anwendungen verfügbar sein (dies entspricht dem klassischen *shared memory*). Um eine Konsistenz des Puffers zu gewährleisten, müssen Zugriffe auf den Puffer synchronisiert werden. Diese Synchronisation wird besonders dann aufwendig, wenn innerhalb einer Kette mehr als zwei Anwendungen auf diesen Speicher zugreifen.

Die zweite Variante zur Verwaltung des Puffers besteht darin, den verfügbaren Pufferbereich in mehrere einzelne gleichgroße Teile aufzuteilen. Jeder dieser kleineren Puffer gehört zu jedem Zeitpunkt maximal einer Anwendung, und nur diese kann auf den Puffer zugreifen. Ist eine Anwendung mit der Bearbeitung der Daten eines Puffers fertig, wird der Puffer an die nächste Anwendung der Kette weitergegeben. Durch diese Vorgehensweise ist keine Synchronisation der Zugriffe auf die Puffer notwendig, die Anwendungen arbeiten unabhängig voneinander, wodurch ein Blockieren einer Anwendung durch eine andere verhindert werden kann.

Für die Umsetzung der zweiten Variante muß die Pufferverwaltung also eine Menge getrennter Puffer verwalten. Die Anzahl und Größe dieser Puffer wird durch die Admission Control in Abhängigkeit der geforderten Bandbreite und Verweildauer des Puffers bei der Anwendung bestimmt. Durch die Angabe der Verweildauer können auch Anwendungsketten beschrieben werden, die die Puffer über mehr als zwei Elemente weitergeben; aus Sicht des Dateisystems ist dann die Dauer, die die Puffer in der zum Dateisystem nächsten Anwendung benötigt werden, entsprechend größer.

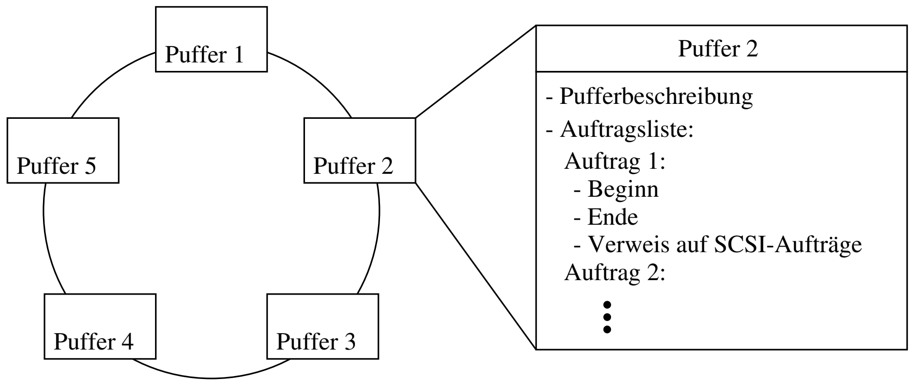



Abbildung 3.8: Ringliste zur Verwaltung der Übertragungspuffer

Die Puffer werden in einer Ringliste organisiert (siehe Abb. 3.8), jedes Element dieser Liste enthält neben einer Beschreibung des Puffers eine Liste mit den Aufträgen, die für diesen Puffer noch ausstehen. Ein Auftrag ist dabei durch einen Zeitraum $(Beginn, Ende)$ beschrieben, indem dieser Puffer der Anwendung zur Verfügung stehen muß. Anhand des Verweises auf die SCSI-Aufträge kann überprüft werden, ob die Aufträge zum Lesen/Schreiben der Daten des Puffers auch korrekt ausgeführt wurden.

Die Verwendung einer derartigen Auftrags-Warteschlange auch für die Puffer ermöglicht eine flexible Auftragsplanung, da die SCSI-Aufträge unabhängig davon erzeugt werden können, ob der Puffer noch in Benutzung ist.

## 3.7 Auftragsplanung

Aufgrund der Aufteilung der SCSI-Slots durch die Admission Control ist die grundlegende Vorgehensweise bei der Auftragsplanung bereits vorgegeben. Das Dateisystem verwaltet einen Plan, in dem jedem Slot des Zyklus ein Datenstrom zugeordnet ist (siehe Abbildung 3.9). Für die Erzeugung der konkreten Aufträge werden in festgelegten Zeitabständen die Aufträge für einen Ausschnitt aus diesem Plan erzeugt, indem für die jeweiligen Datenströme die nächsten Blöcke geliefert werden. Die Länge dieses Ausschnitts und der Abstand zwischen der Generierung dieser Auftragsgruppen ist von der Anzahl an Aufträgen abhängig, die der SCSI-Treiber auf einmal verwalten kann.

Des weiteren muß auch die Planung der Pufferübertragung erfolgen. Dazu müssen die entsprechenden Einträge in den Auftragswarteschlangen der Puffer erzeugt werden. Da die Puffer in der Regel größer als ein einzelner Datenblock sind, werden mehrere SCSI-Aufträge zum Lesen eines Puffers benötigt. Der frühestmögliche Zeitpunkt, an dem der Puffer an die Anwendung geliefert werden kann liegt demzufolge nach dem Abschluß des letzten Leseauftrags. Der Zeitpunkt, an dem der Puffer für andere Aufträge wieder verfügbar ist, wird durch die Verweildauer des Puffers bei der Anwendung bestimmt. Beim Schreiben eines Datenstroms wird anders verfahren. Der Zeitraum, in dem der Puffer der Anwendung zur Verfügung

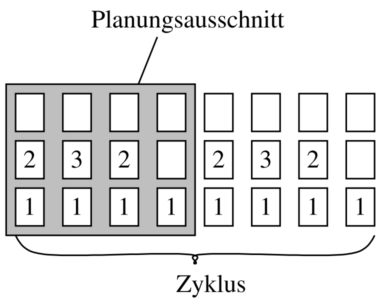



Abbildung 3.9: Plan zur Auftragsgenerierung

steht, wird durch das festgelegte Datenlayout bestimmt. Die Einhaltung dieses Zeitraums wird durch das Dateisystem durchgesetzt, d.h. die Puffer sind meistens nicht vollständig gefüllt. Für das Schreiben der Daten aus dem Puffer werden dann die Slots verwendet, die für diesen Datenstrom nach dem Zeitpunkt des Zurückholens des Puffers reserviert sind.

## 3.8 Schwankungsbeschränkte Datenströme

Durch die beschriebene Vorgehensweise beim Lesen der Datenströme werden diese mit einer konstanten Bandbreite geliefert. Reale Datenströme weisen jedoch häufig Schwankungen der Datenrate auf, z.B. durch Kompressionsverfahren wie MPEG. Diese Schwankungen können bis zu einem gewissen Punkt durch eine Pufferung ausgeglichen werden. Datenströme, bei denen ein derartiger Ausgleich möglich ist, können auch als *schwankungsbeschränkte Datenströme* bezeichnet werden.

Für den Ausgleich der Schwankungen ist die Verwendung von zusätzlichem Pufferspeicher notwendig. Durch die Admission Control wird eine Datenmenge bestimmt, die bereits vor dem Start der eigentlichen Bearbeitung des Datenstroms gelesen werden muß. Die Bedienung des Datenstroms erfolgt durch das Dateisystem anhand der mittleren Bandbreite des Datenstroms. Durch die Schwankungen beim Lesen des Datenstroms seitens der Anwendung wird der Vorlauf, der durch das Voraus-Lesen der Daten entstanden ist, nach und nach aufgebraucht.

Da das Lesen der Daten von der Festplatte auch bei schwankungsbeschränkten Strömen mit einer konstanten Bandbreite erfolgt, können die bisher vorgestellten Verfahren für die Planung der SCSI-Aufträge weiterhin verwendet werden. Für die Übertragung der Puffer an die Anwendung müssen die Schwankungen der Datenrate jedoch berücksichtigt werden. Abbildung 3.10 stellt die entsprechende Situation dar. Aufgrund der Abweichung von der mittleren Bandbreite beim Lesen der Daten ergibt sich ein Bereich, in dem sich der Lesezeiger der Anwendung befinden kann. Durch die verwendete Strombeschreibung ist jedoch nicht bekannt, an welcher Stelle sich die Anwendung genau befindet, so daß die obere und untere Grenze dieses Bereichs über den kompletten Zeitraum berücksichtigt werden muß. Das Intervall, für das ein Puffer der Anwendung zur Verfügung stehen muß, ergibt sich dann aus der Betrachtung der möglichen Grenzfälle. Dies ist für den Beginn des Intervalls wenn die Anwendung mit dem Lesen um den maximal erlaubten Wert voraus ist, für das Ende des Intervalls falls die Anwendung um den erlaubten Wert hinterher ist. Das Intervall ist dann durch diese beiden Werte sowie die Verweildauer des Puffers bei der Anwendung bestimmt.

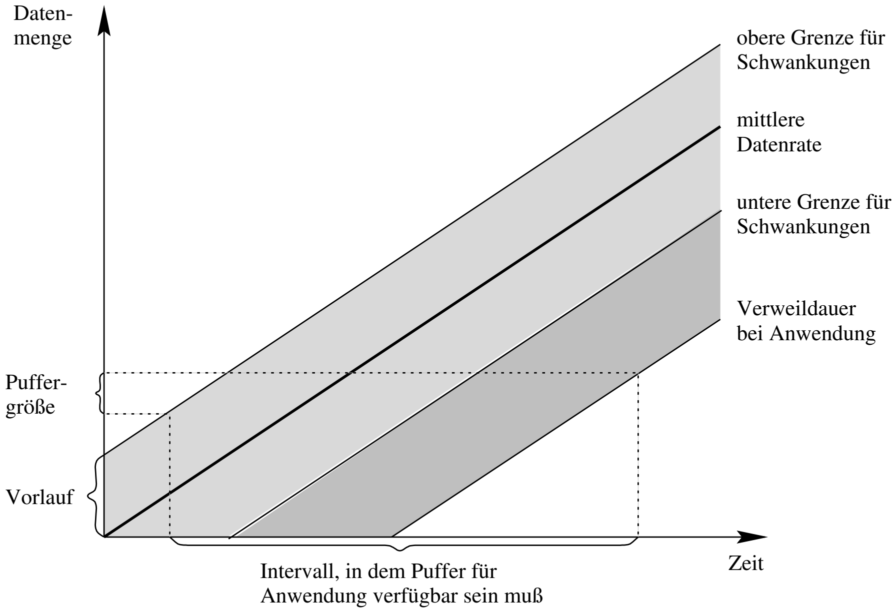



Abbildung 3.10: Planung bei schwankungsbeschränkten Datenströmen

## 3.9 Zusammenfassung

Abbildung 3.11 stellt noch einmal einen Überblick über die wichtigsten Komponenten des Echtzeit-Dateisystems sowie die verwendeten Mechanismen dar. Damit das Dateisystem Zusagen über Datenraten geben kann, arbeiten die Admission Control, die Auftragsplanung und der SCSI-Treiber eng zusammen. Durch die Admission Control wird bereits festgelegt, zu welchen Zeitpunkten die einzelnen Aufträge der Datenströme bearbeitet werden.

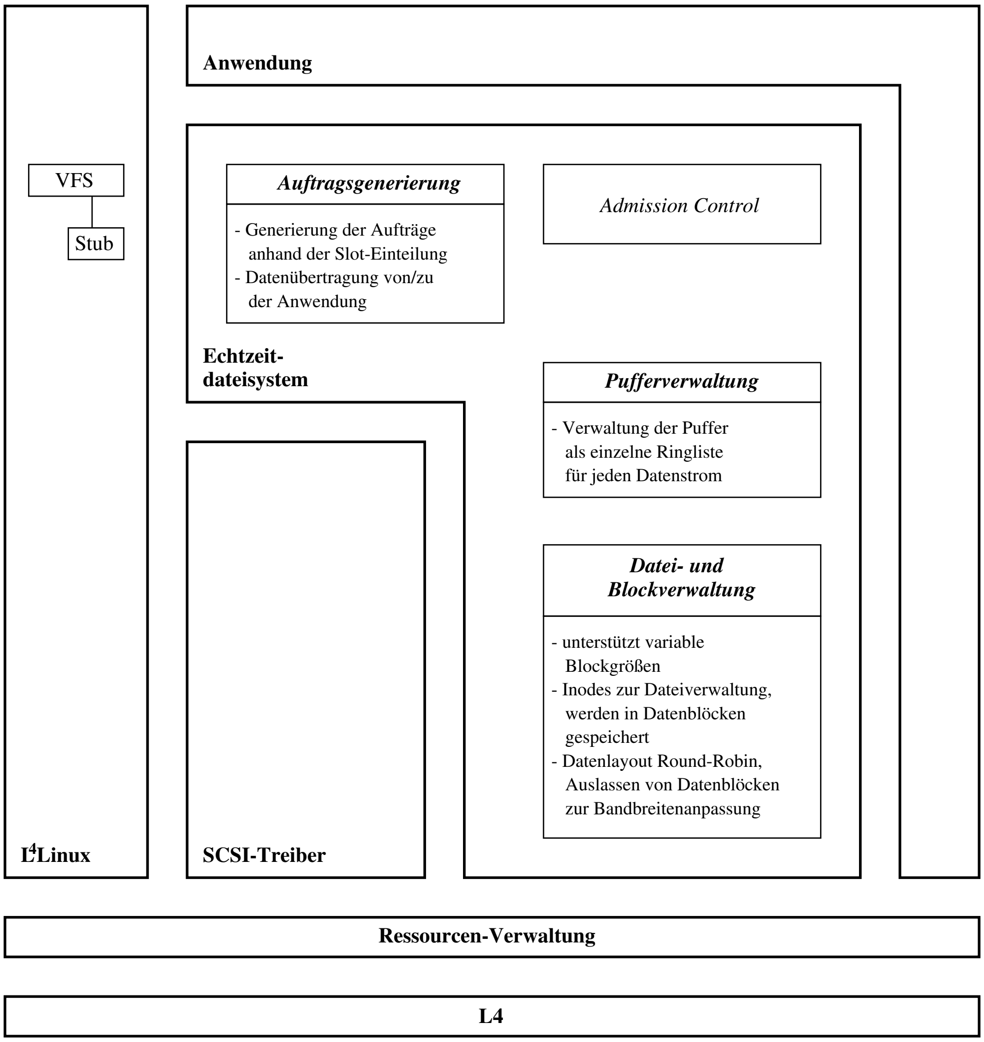



Abbildung 3.11: Gesamtstruktur des Echtzeitdateisystems

# Kapitel 4

# Implementierung

Im vorangegangenen Kapitel wurden die Konzepte vorgestellt, die das Dateisystem zur Verwaltung der Daten verwendet. Daran anschließend soll jetzt auf einige Aspekte der Implementierung eingegangen werden.

Als Grundlage für die Implementierung dient der Mikrokern L4 [Lie96] und der von Frank Mehnert portierte Treiber für die NCR53c8xx SCSI-Hostadapter Familie [Meh98].

## 4.1 Threadstruktur

Durch die Verwendung von Threads kann der Eigenschaft vieler Anwendungen Rechnung getragen werden, daß diese mehrere nebenläufige Abarbeitungspfade besitzen. Diese Abarbeitungspfade können dann auch auf Programmebene entkoppelt werden, was zu einer geringeren Beeinflussung dieser Pfade untereinander führt.

Da man auch die Bearbeitung eines einzelnen Datenstroms durch das Dateisystem als separaten Abarbeitungspfad ansehen kann, wäre als Struktur für das Dateisystem die Verwendung eines Threads für jeden Datenstrom denkbar, um eine Beeinflussung der Datenströme untereinander zu vermeiden. Gegen eine derartige Struktur spricht allerdings der Umstand, daß L4 nur 128 Threads innerhalb eines Adreßraums unterstützt. Für einige Anwendungen, etwa einen Video-Server, ist das bei der verwendeten Hardware ausreichend, für andere Anwendungen stellt es jedoch eine Limitation dar, da die Hardware dann theoretisch mehr Datenströme liefern könnte, als das Dateisystem verarbeiten kann, z.B. bei einer Musik-Datenbank.

Die in Abbildung 4.1 dargestellte Threadstruktur nutzt die Eigenschaft des Dateisystementwurfs aus, daß für kontinuierliche Datenströme das Dateisystem von sich aus die Auftragsgenerierung und Datenübertragung steuert. Diese beiden Aufgaben werden in getrennten Threads abgearbeitet. Der Thread zur Auftragsplanung erzeugt anhand des in Abschnitt 3.7 vorgestellten Plans die SCSI-Aufträge, der Thread der Pufferplanung ist für die Übertragung der Puffer an die Anwendung entsprechend der festgelegten Zeitintervalle verantwortlich.

Da die Aufträge nicht-kontinuierlicher Datenströme für das Dateisystem nicht vorhersagbar sind, werden diese durch getrennte Threads bearbeitet, um ein gegenseitiges Blockieren der Datenströme zu vermeiden.

## 4.2 Auftragsbearbeitung

Ausgehend von der beschriebenen Threadstruktur erfolgt die Bearbeitung der Aufträge unterschiedlich für die einzelnen Zugriffsarten, dies ist in Abbildung 4.2 schematisch dargestellt.

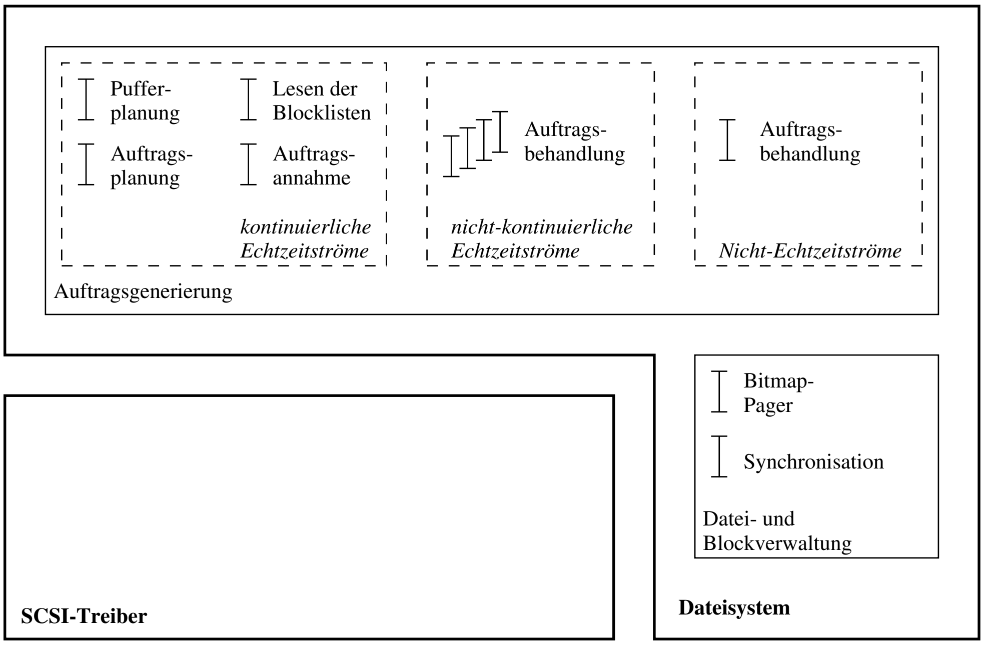



Abbildung 4.1: Threadstruktur des Dateisystems

- **Kontinuierliche Echtzeitdaten**
  Für die Bearbeitung eines kontinuierlichen Datenstroms wendet sich eine Anwendung zunächst an die Admission Control mit einer Anforderung zum Öffnen der entsprechenden Datei. Als Argument werden dabei die benötigte Datenrate und Verweildauer der Puffer übergeben. Bei erfolgreicher Admission wird der Auftrag an den Thread zur Auftragsbehandlung weitergegeben. Für die Erzeugung der SCSI-Aufträge werden in dem im Abschnitt 3.7 beschriebenen Plan die entsprechenden Felder belegt und es wird die Ringliste der Übertragungspuffer erzeugt. Die Blocklisten der einzelnen Dateien können nicht permanent im Speicher gehalten werden, durch einen separaten Thread werden die im Moment benötigten Ausschnitte der Blocklisten gelesen. Die Übertragung der Daten erfolgt durch das Einblenden der Puffer in den Adreßraum der Anwendung.

- **Nichtkontinuierliche Echtzeitdaten**
  Die Bearbeitung eines nicht-kontinuierlichen Datenstroms erfolgt durch einen separaten Thread pro Datei. Die Anwendung wendet sich ebenfalls zuerst an die Admission Control. Diese startet bei erfolgreicher Admission einen neuen Thread zur Behandlung der Aufträge für diese Datei. Die Anwendung kommuniziert dann mit diesem Thread für das Lesen oder Schreiben der Daten.

- **Nicht-Echtzeitdaten**
  Alle Aufträge für Nicht-Echtzeitdateien werden durch einen einzigen Thread bearbeitet, der beim Hochfahren des Systems gestartet wird.

## Zeitsteuerung

Die Bearbeitung der kontinuierlichen Datenströme erfordert eine periodische Abarbeitung der Threads zur SCSI- und Pufferplanung sowie des Threads zum Lesen der Blocklistenabschnitte. Ziel des DROPS-Projekts ist es, derartige periodische Threads durch ein geeignetes CPU-Scheduling zu unterstützen [Wol97]. Bedingung dafür ist, daß neben der Periodenlänge auch die maximale Bearbeitungsdauer innerhalb einer

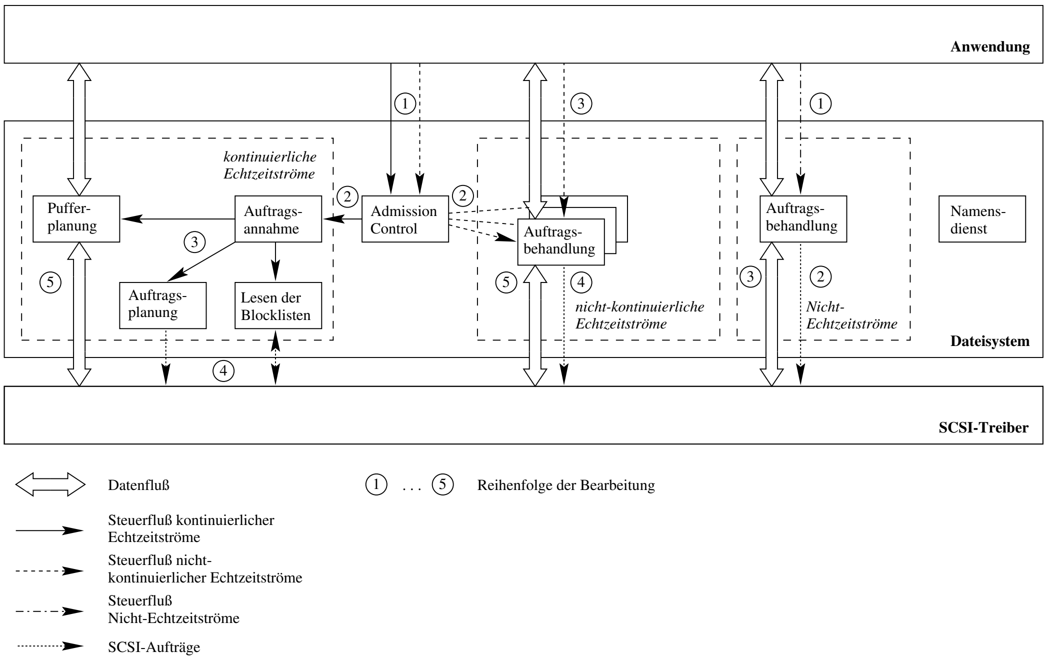



Abbildung 4.2: Bearbeitungsablauf

Periode bekannt ist. Für den Thread zur Erzeugung der SCSI-Aufträge ist die Periodenlänge durch die Größe des Planungsausschnitts (vgl. Abschnitt 3.7), die Bearbeitungsdauer durch die Anzahl der Aufträge pro Planungsausschnitt bestimmt[1]. Die Übertragung der Puffer erfolgt durch das vorgestellte Modell ebenfalls periodisch, die Periodenlänge ist abhängig von der verwendeten Datenrate[2].

## 4.3 Speicherverwaltung

Die Speicherverwaltung läßt sich in zwei Teile untergliedern: Verwaltung der Adreßräume des Dateisystems und des SCSI-Treibers sowie die Bereitstellung der zur Übertragung der Daten benötigten Puffer.

### 4.3.1 Pager

Die Ressourcenverwaltung von DROPS stellt derzeit nur Speicher zur Verfügung, der eins zu eins zu dem physischen Speicher in einen Adreßraum eingeblendet ist. Eine Anwendung muß davon ausgehend für die Verwaltung ihrer virtuellen Adreßräume sorgen. Der für das Dateisystem verwendete Pager basiert auf einer Arbeit von Torsten Paul [Pau97]. Dieser verwendet eine invertierte Seitentabelle, um die Adreßräume mehrerer Anwendungen verwalten zu können. Der Pager wird beim Hochfahren des Systems geladen und startet seinerseits dann das Dateisystem bzw. den SCSI-Treiber. Abbildung 4.3 stellt die so entstehende Adreßraumstruktur dar.

Eine weitere Aufgabe der Speicherverwaltung ist die Bereitstellung dynamisch allokierbaren Speichers für die Anwendungen. Dies ist besonders für das Dateisystem wichtig, um eine effiziente Verwaltung der

[1] Damit die Bearbeitungsdauer für einen Planungsabschnitt möglichst genau bestimmt werden kann, sollte die Aufteilung des Zyklus in einzelne Abschnitte so erfolgen, daß pro Abschnitt möglichst die gleiche Anzahl an Aufträgen erzeugt werden muß.

[2] Bei der Verwendung unterschiedlicher Datenraten kann die Bestimmung der Periodenlänge durch die Berechnung des größten gemeinsamen Teilers der Periodenlängen der einzelnen Datenströme erfolgen.

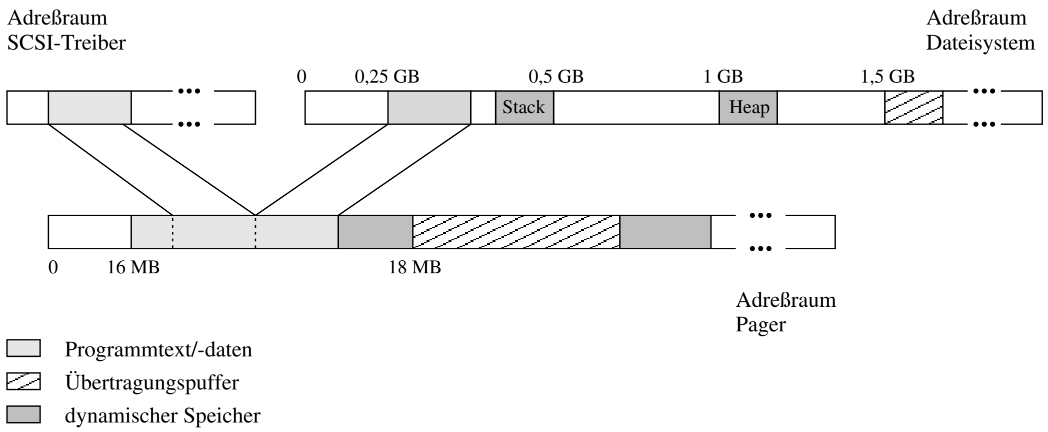



Abbildung 4.3: Adreßraumstruktur

diversen Listen und Warteschlangen zu erleichtern. Um dies zu ermöglichen, wurde der Pager so erweitert, daß er pro Anwendung einen Speicherbereich verwaltet, dessen Größe dynamisch verändert werden kann. Dieser Bereich wird von einem Speicherverwaltungsalgorithmus verwendet, um der Anwendung dynamischen Speicher zur Verfügung zu stellen [Lea96]. Die Verwendung dieses Speichers erfolgt über die üblichen `malloc`-, `free`- und `realloc`-Funktionen.

### 4.3.2 Pufferverwaltung

Die zweite Aufgabe der Speicherverwaltung besteht in der Bereitstellung der für die Übertragung der Daten an die Anwendung erforderlichen Puffer. Die Verwaltungskomponente muß dabei zwei Bedingungen erfüllen:

- Die physische Adresse der Pufferbereiche muß bekannt sein. Dies ist erforderlich, da die Daten durch den SCSI-Treiber mittels DMA direkt in den Speicher geschrieben werden.
- Die Puffer sollten aus möglichst großen zusammenhängenden Speicherbereichen bestehen. Der SCSI-Treiber ist zwar durch *Scattered Gathering* in der Lage, Daten in verstreut liegende Speicherbereiche zu schreiben, die Verwaltung dieser Bereiche ist jedoch nur für eine kleine Anzahl einzelner Speicherabschnitte ausgelegt, so daß für die Puffer möglichst wenige getrennte Speicherbereiche verwendet werden sollten.

Die Verwaltung der Puffer erfolgt in zwei Stufen. In der ersten Stufe wird durch einen separaten Thread im Pageradreßraum ein zusammenhängender Speicherbereich verwendet, der am Anfang in den Adreßraum so eingeblendet wird, daß die physischen und virtuellen Adressen übereinstimmen. Für die Organisation dieses Bereichs wird die in Abbildung 4.4 gezeigte Struktur verwendet.

Für ein schnelles Allokieren der Speicherbereiche werden dynamisch erzeugte Freispeicherlisten verwendet. Zu Beginn existiert nur eine Liste mit einem Element, durch das der gesamte Speicherplatz beschrieben wird. Zur Allokation eines Speicherbereichs werden dann bei Bedarf größere Bereiche geteilt. Um ein schnelles Zusammenfassen der Speicherbereiche beim Freigeben zu gewährleisten und somit eine Fragmentierung des Speicherplatzes zu verhindern, werden die Speicherbereiche noch durch eine doppelt verkettete Liste verbunden, so daß die angrenzenden Speicherbereiche einfach zu ermitteln sind.

Ausgehend von dieser Listenstruktur werden dann Anwendungen (in diesem Fall dem Dateisystem) Puffer zur Verfügung gestellt. Bei der Anforderung eines Puffers wird zunächst versucht, einen einzigen zusammenhängenden Speicherbereich für den Puffer zu verwenden, gelingt dies nicht, werden mehrere Bereiche

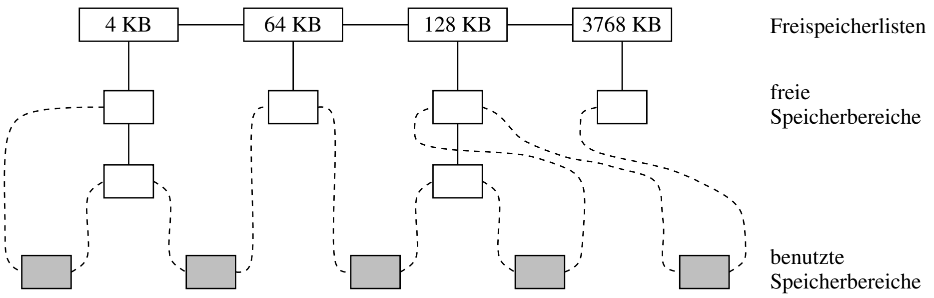



Abbildung 4.4: Pufferverwaltung durch Freispeicher- und Bereichsliste

verwendet. Für die Beschreibung des Puffers wird der Anwendung eine Liste aller verwendeten Speicherbereiche übergeben, ein Speicherbereich ist in dieser Liste durch seine physische Adresse und Länge definiert.

Die zweite Stufe der Verwaltung wird durch das Dateisystem realisiert. Die Puffer werden durch die bereits im Entwurf beschriebene Ringliste verwaltet, dafür werden sie in den Adreßraum des Dateisystems eingeblendet. Bei diesem Einblenden sind einige Eigenschaften des L4 Mikrokerns zu beachten. In L4 werden Speicherbereiche durch die Verwendung von *Flexpages* zwischen Adreßräumen übergeben. Eine Flexpage beschreibt einen Ausschnitt aus dem Adreßraum einer Anwendung durch die Angabe der Startadresse und der Größe. Die Größe muß dabei eine Zweierpotenz sein, die kleinstmögliche Größe ist die einer physischen Speicherseite (bei x86-Systemen sind das 4 KByte). Ein Speicherbereich, dessen Größe keine Zweierpotenz ist, muß durch mehrere Flexpages beschrieben werden. Eine weitere Bedingung ist, daß die Flexpage entsprechend ihrer Größe ausgerichtet sein muß und zwar sowohl im Quell- als auch im Zieladreßraum. Aus diesem Grund übergibt das Dateisystem bei der Anforderung zum Einblenden eines Puffers die Ausrichtung der Zieladresse, die Pufferverwaltung im Pageradreßraum erzeugt dann die Flexpages entsprechend dieser und der Ausrichtung der Speicherbereiche in ihrem Adreßraum. Damit von Seiten des Dateisystems immer eine optimale Ausrichtung der Zieladresse gewährleistet werden kann, wird der für das Einblenden der Puffer vorgesehene Adreßbereich durch einen binären Buddy-Algorithmus verwaltet. Dadurch kann erreicht werden, daß Puffer der Größe einer Zweierpotenz immer an einer nach dieser Größe ausgerichteten Adresse eingeblendet werden, für andere Puffer wird ein Adreßbereich mit der Größe der nächst höheren Zweierpotenz verwendet.

## 4.4 Festplattenverwaltung

Für die Implementierung der Festplattenverwaltung stellt sich zuerst die Frage, ob alle verfügbaren Festplatten als eine logische Platte oder voneinander getrennt behandelt werden. Da durch das Datenlayout zwischen einzelnen Festplatten unterschieden wird, wird eine getrennte Verwaltung verwendet, die Organisation als eine logische Platte wäre bei der Blockallokation eher hinderlich.

Die einzelnen Festplatten werden in der Form von Partitionen verwaltet. Dies ermöglicht es, daß auf einer Festplatte theoretisch mehrere Dateisysteme angelegt werden[3]. Für den Eintrag der Partitionen in die Partitiontabelle der Festplatten wird die bisher noch ungenutzte Dateisystem-Id 0x86 verwendet.

Abbildung 4.5 stellt den Aufbau einer Partition dar. Am Anfang jeder Partition steht ein Superblock, der die Partition beschreibt (siehe Abb. 4.6). Er enthält neben den Parametern des Buddy-Algorithmus eine Partition-Id sowie eine Liste aller zu dem Dateisystem gehörenden Partitionen. Die Partition-Id wird für

[3] Eine denkbare Anwendung dafür wäre, die langsameren inneren Bereiche einer Festplatte für ein Linux-Dateisystem zu verwenden, während die äußeren, schnellen Bereiche für das Echtzeitdateisystem genutzt werden. Voraussetzung dafür ist, daß der SCSI-Treiber auch direkt von Linux aus angesprochen werden kann und er diese Aufträge in freie Slots einordnen kann.

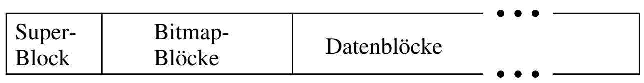



Abbildung 4.5: Aufbau einer Partition

die eindeutige Identifikation der Partition innerhalb des Dateisystems verwendet, sie entspricht im Moment der SCSI-Device-Nummer der Partition. Letztendlich wird die Partition-Id jedoch unabhängig vom SCSI-System sein, die Auflösung der Partition-Id in die SCSI-Device-Nummer soll dann erst durch den SCSI-Treiber erfolgen. Dadurch soll z.B. das Verschieben einer Partition auf eine andere Festplatte auf eine einfache Art ermöglicht werden. Die neue Zuordnung muß nur im SCSI-Treiber berücksichtigt werden, alle weiter oben liegenden Verwaltungsstrukturen bleiben von der Änderung unberührt.

```
struct rtfs_superblock
{
  unsigned partition_id:16;       /* partition id */
  unsigned buddy_om:16;           /* buddy basis */
  unsigned b_basis:16;            /* smallest blocksize */
  unsigned b_bitmap:16;           /* bitmap blocksize */

  dword_t data_blocks;            /* number of data blocks */
  dword_t bitmap_blocks;          /* number of bitmap blocks */

  rtfs_blockid_t root;            /* location of the root directory */

  dword_t free_count[NO_BLOCKSIZES];
                                  /* counter for free blocks per size */

  byte_t reserved[492 - 4 * NO_BLOCKSIZES];

  word_t partitions[256];         /* other rtfs partitions */
};
```

Abbildung 4.6: Superblock

Die nächste Fragestellung lautet, wie die Verwaltungsstrukturen des Buddy-Algorithmus gespeichert werden. Die Standardimplementierung des Buddy-Algorithmus für die Hauptspeicherverwaltung besteht in der Verwendung je einer Freispeicherliste pro verfügbarer Blockgröße. Diese Struktur besitzt für den Einsatz für die Verwaltung des Festplattenplatzes zwei Nachteile:

- Ein eher kleineres Problem ist, daß die Listen keine konstante Länge haben.
- Das Hauptproblem besteht darin, daß aufgrund des Zusammenfassens benachbarter Datenblöcke beim Freigeben auch Lücken innerhalb der Listen entstehen. Für eine dynamisch erzeugte Liste im Hauptspeicher ist das kein Problem, für eine Liste, die statisch durch die Verwendung von Datenblöcken auf der Festplatte realisiert ist, bedeutet das jedoch ein Umkopieren aller Daten hinter dem freigegebenen Listenelement. Die Lösungsvariante, die Freispeicherlisten während der Bearbeitung im Hauptspeicher zu halten und nur gelegentlich auf die Festplatte zu schreiben, ist aufgrund des hohen Speicherbedarfs nicht realisierbar.

Für das Dateisystem wurde daher eine andere Darstellungsart gewählt. Für jede Blockgröße wird eine Bitmap mit je einem Bit pro verfügbaren Block verwaltet. Die Bitmap für die maximale Blockgröße enthält damit ein Bit, die der Basisblockgröße die maximale Anzahl an Bits. Ein gesetztes Bit in der Bitmap einer Blockgröße bedeutet, daß der entsprechende Block dieser Größe verfügbar ist. Da kleinere Blöcke durch die Aufteilung größerer Blöcke entstehen, bezieht sich ein Bit in der Bitmap einer Blockgröße auch auf die Blöcke kleinerer Blockgrößen, die durch eine Teilung dieses Blocks entstehen. Ist ein Block als verfügbar gekennzeichnet, sind die zu diesem Block gehörenden Blöcke kleinerer Größen zunächst nicht verfügbar (die Bits in den entsprechenden Bitmaps sind nicht gesetzt), erst durch eine Teilung des größeren Blocks können die kleineren verfügbar gemacht werden (siehe Abb. 4.7). Durch diesen Umstand werden für die Verwaltung eines Blocks der Basisblockgröße effektiv zwei Bits verwendet.

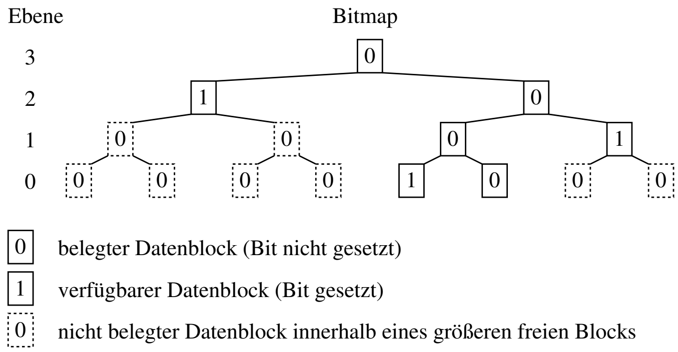



Abbildung 4.7: Implementierung des Buddy-Algorithmus mittels Bitmap

Der für die Speicherung der Bitmaps benötigte Speicherplatz ist konstant, er wird durch die Größe der Partition bestimmt. Die Bitmap einer 2 GByte Partition bei einer minimalen Blockgröße von 4 KByte ist 128 KByte groß. Für größere Dateisysteme können die Bitmaps also durchaus einige MByte groß sein, so daß diese nicht vollständig permanent im Hauptspeicher gehalten werden können, sie müssen bei Bedarf von Festplatte geladen werden. Um dies transparent zu gestalten, verwendet das Dateisystem einen Pager. Die Bitmaps werden in den Adreßraum des Dateisystems eingeblendet. Bei Bedarf lädt der Pager einen Bitmapblock an die entsprechende Adresse, dazu werden eine feste Anzahl an Speicherseiten verwendet. Ist keine freie Speicherseite verfügbar, wird ein anderer Bitmapblock auf die Festplatte zurückgeschrieben und diese Speicherseite verwendet[4]. Um zu vermeiden, daß durch den Zugriff auf eine benachbarte Blockebene beim Aufteilen oder Zusammenfassen eines Blocks auch auf einen anderen Bitmapblock zugegriffen wird, werden die Bitmaps durch das in Abbildung 4.8 dargestellte Schema auf die Festplatten-Blöcke verteilt.

Durch die Verwendung von 4 KByte Festplatten-Blöcken für die Speicherung der Bitmaps können bei einer minimalen Blockgröße von 4 KByte durch einen Bitmapblock 64 MByte Festplattenplatz verwaltet werden, durch die in Abbildung 4.8 dargestellte zweistufige Hierarchie 2 TByte.

Für die Synchronisation der in den Speicher geladenen Bitmapblöcke mit den Blöcken auf den Festplatten wird ein separater Synchronisations-Thread verwendet, der in regelmäßigen Abständen alle Bitmaps auf die Festplatten zurückschreibt.

[4]Für die Auswahl der zu verdrängenden Speicherseite wird im Moment kein spezieller Algorithmus verwendet. Die zur Verfügung stehenden Speicherseiten werden durch eine Ringliste verwaltet und entsprechend der Reihe nach verwendet.

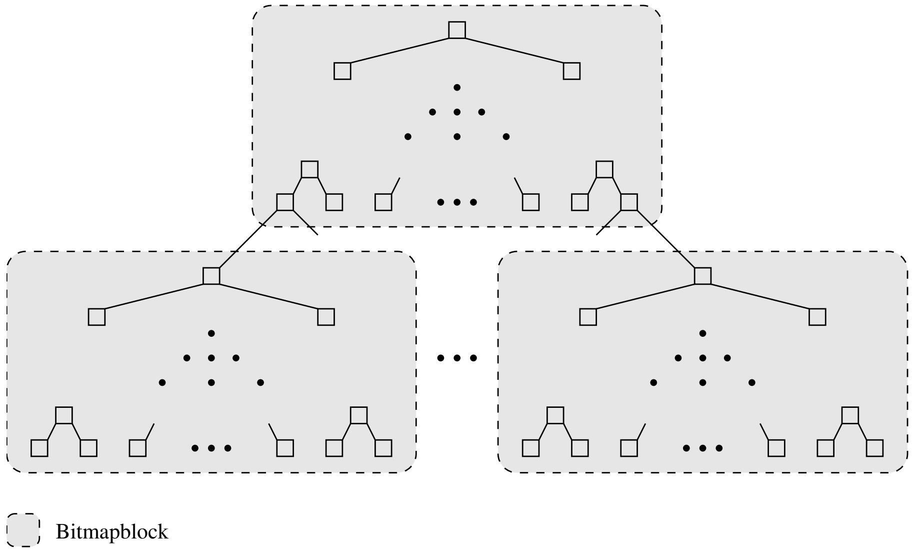



Abbildung 4.8: Zuordnung der Bitmaps zu Bitmapblöcken

## 4.5 Dateien

Eine Datei wird durch die in Abbildung 4.10 dargestellte Inode-Struktur beschrieben. Diese Struktur ist in zwei Teile untergliedert, den Inode-Kopf und den direkt in der Inode gespeicherten Abschnitt der Blockliste.

Neben den üblichen Angaben enthält der Inode-Kopf die für die Datei verwendete Datenblockgröße sowie die Größe des Inode-Blocks. Der restliche Speicherplatz im Inode-Block wird für die Speicherung des ersten Teils der Blockliste der Datei verwendet.

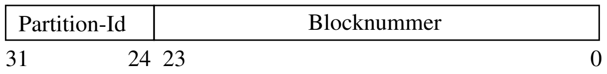



Abbildung 4.9: Block-Id

Ein Block der Datei wird durch die Angabe einer Block-Id (siehe Abb. 4.9) beschrieben. Diese enthält die Id der Partition, auf der der Block gespeichert ist, sowie die Nummer des Blocks innerhalb dieser Partition.

Die Benennung der Dateien erfolgt analog zu ext2fs, die Zuordnung eines Namens zu einer Inode erfolgt durch den Eintrag in dem Elternverzeichnis der Datei. Die Verzeichnisse werden ebenfalls durch eine Inode beschrieben, die Organisation erfolgt durch eine einfach verkettete Liste, Abbildung 4.11 stellt einen Eintrag in dieser Liste dar.

Ausgehend vom Wurzelverzeichnis, dessen Inode-Block-Id im Superblock der Partition gespeichert ist, wird die Verzeichnisstruktur analog zu ext2fs aufgebaut.

```
/* Inode header */
typedef struct rtfs_inode_header
{
  word_t          i_type;           /* file type */
  word_t          i_uid;            /* owner */
  word_t          i_gid;            /* owner */
  word_t          i_unused1;
  rtfs_time_t     i_time_create;    /* creation time */
  rtfs_time_t     i_time_access;    /* last access */
  rtfs_time_t     i_time_modify;    /* last modification */
  word_t          i_blk_size;       /* log2 blocksize, blk_size = 2^n * B_0 */
  word_t          i_striping_unit;  /* striping unit -> size = n * blk_size */
  dword_t         i_inode_size;     /* log2 inode size, size = 2^n */
  dword_t         i_blk_count;      /* number of blocks */
  dword_t         i_unused2;
  qword_t         i_size;           /* file size */
  dword_t         i_unused3[7];
  rtfs_blockid_t  i_indirect1;      /* single indirect blocks */
  rtfs_blockid_t  i_indirect2;      /* double indirect blocks */
  rtfs_blockid_t  i_indirect3;      /* triple indirect blocks */
} rtfs_inode_header_t;

/* Inode */
typedef struct rtfs_inode
{
  rtfs_inode_header_t i_header;      /* Inode header */
  rtfs_blockid_t      i_direct[1];   /* direct blocks */
} rtfs_inode_t;
```

Abbildung 4.10: Aufbau der Inode

### 4.5.1 Anlegen der Dateien

Die Allokation neuer Datenblöcke für eine Datei erfolgt dann, wenn auf eine Position nach dem Ende der Datei geschrieben wird. Es werden dabei immer für den kompletten Bereich zwischen dem aktuellen Ende der Datei und der Schreibposition Blöcke allokiert, so daß keine Lücken innerhalb der Datei entstehen[5]. Die Suche nach einem freien Block erfolgt in zwei Schritten. Zuerst wird die Festplatte bestimmt, auf der der Block gespeichert werden soll, im Moment wird dafür eine strikte Round-Robin-Verteilung verwendet. Im zweiten Schritt wird auf dieser Festplatte ein Datenblock der entsprechenden Größe reserviert.

Wie in Abschnitt 3.4.2 beschrieben, soll beim Speichern einer Datei in Echtzeit das Datenlayout bereits im voraus bestimmt werden. Dies kann beim gegenwärtigen Entwicklungsstand erfolgen, indem die Größe der Datei durch eine `seek`- und `write`-Operation auf das Ende der Datei festgelegt wird; da wie oben beschrieben dadurch alle Blöcke der Datei reserviert werden, stehen beim Schreiben der Daten die Blöcke bereits zur Verfügung.

[5] In UNIX-Dateisystemen wird die Allokation eines Blocks meistens erst dann vorgenommen, falls auf diesen konkreten Block zugegriffen wird. Dadurch können innerhalb einer Datei Bereiche existieren, für die keine Blöcke allokiert sind.

```
/* directory entry */

struct rtfs_dir_entry
{
  rtfs_blockid_t inode;              /* inode block number */
  unsigned short rec_len;            /* entry length */
  unsigned short name_len;           /* name length */
  char name[RTFS_FILENAME_LENGTH];   /* file name */
};
```

Abbildung 4.11: Eintrag in Verzeichnis-Liste

### 4.5.2 Speicherung der Strombeschreibungen

Die Speicherung der für die Beschreibung der Echtzeitdateien verwendeten Daten erfolgt in separaten Dateien; dadurch wird ein einfacher Zugriff auf diese Daten auch durch $L^4$Linux-Anwendungen möglich. Die Zuordnung der Dateien erfolgt durch spezielle Dateinamenserweiterungen, die in Tabelle 4.1 aufgeführt sind.

Neben den in Abschnitt 3.1.3 beschriebenen Parametersätzen werden in einer weiteren Datei datentypspezifische Informationen gespeichert, wie z.B. die Auflösung und das verwendete Kompressionsverfahren bei einem Video.

| Dateiname | Inhalt |
|---|---|
| `name.`*`data`* | Daten |
| `name.`*`para`* | Parametersätze der Strombeschreibung |
| `name.`*`desc`* | Beschreibung des Dateityps |

Tabelle 4.1: Verwendete Dateinamenserweiterungen

## 4.6 Stand der Implementierung

Die derzeitige Implementierung umfaßt die Speicher-, Festplatten- und Dateiverwaltung sowie die Behandlung der Nicht-Echtzeitdateien. Für die Steuerung und das Testen des Dateisystems wurde zusätzlich eine serielle Konsole implementiert, diese besteht aus einem Server, der die Kommunikation mit der seriellen Schnittstelle durchführt sowie einem separaten Thread des Dateisystems, der die Darstellung der Konsole übernimmt.

Der Zugriff auf die Dateien erfolgt über eine UNIX-ähnliche Schnittstelle. Diese umfaßt die `open`-, `create`-, `close`-, `read`-, `write`- und `seek`- sowie spezielle `fcntl`- und `fstat`-Funktionen und ist auf eine IPC-Kommunikation mit dem Behandlungs-Thread im Dateisystem abgebildet. Über diese Schnittstelle wird im Moment auch von $L^4$Linux-Anwendungen aus auf das Dateisystem zugegriffen; eine Bibliothek mit den entsprechenden Funktionen wurde dazu entwickelt.

Die Implementierung umfaßt ca. 12500 Zeilen C-Quelltext für das Dateisystem (inklusive Pager und serielle Konsole) sowie weitere 1500 Zeilen für die $L^4$Linux-Bibliothek und darauf basierende Programme zum Zugriff auf die Dateien von $L^4$Linux aus.

# Kapitel 5

# Leistungsbewertung

Bei dem derzeitigen Stand der Implementierung können noch keine umfangreichen Tests zur Leistungsfähigkeit des Dateisystems vorgenommen werden. Es kann jedoch bereits überprüft werden, welchen Einfluß die Wahl der Blockgröße auf die Datenrate beim Lesen und Schreiben von Dateien hat.

## 5.1 Testumgebung

Die Messungen wurden auf einem PC mit einem 100 MHz Intel Pentium Prozessor, 32 MByte Hauptspeicher, Asus SC200 SCSI-Controller (NCR53c810) und einer 2,1 GByte Quantum VP32210 Festplatte durchgeführt. Für die Ermittlung von Vergleichswerten wurde das Programm *h2bench* verwendet, welches die Eigenschaften der Festplatte (Zoneneinteilung, Datenraten bei verschiedenen Blockgrößen und Zugriffszeiten) durch direkten Zugriff über BIOS-Funktionen bestimmt [Bög96].

Für die Bestimmung der Datenraten wurde eine 64 MByte große Datei auf der schnellsten Zone der Festplatte[1] geschrieben bzw. gelesen. Die Blöcke der Datei lagen linear hintereinander, eine zufällige Verteilung der Daten ist im Moment nicht möglich, da die Blockallokation innerhalb einer Platte nicht beeinflußt werden kann. Für die Vergleichswerte wurden daher die Ergebnisse des Test beim linearen Lesen und Schreiben von h2bench verwendet. Die Datei wurde von L$^4$Linux aus über die IPC-Schnittstelle geschrieben bzw. gelesen. Bei den einzelnen Aufträgen wurde dabei generell auf 128 KByte der Datei zugegriffen. Die Blockliste wurde permanent im Speicher gehalten, da auch bei der Bearbeitung der Echtzeit-Datenströme so vorgegengen werden soll. Für die Messung der Dauer der Auftragsbearbeitung wurde der Time Stamp Counters des Pentium Prozessors verwendet.

## 5.2 Meßergebnisse

In Tabelle 5.1 sind die Ergebnisse der Messungen aufgeführt, in den Abbildungen 5.1 und 5.2 werden diese mit den durch h2bench ermittelten Maximalwerten verglichen[2]. Die in der Tabelle angegebenen Zeiten beziehen sich auf die Bearbeitung eines einzelnen Schreib-/Leseauftrags, die Datenraten wurden aus diesen Werten berechnet.

Bei einer Blockgröße von 64 KByte erreicht das Dateisystem beim Schreiben 38% und beim Lesen 51% der maximalen Datenrate. Der große Unterschied läßt sich zum Teil damit erklären, daß die Daten zwischen dem Dateisystem und der L$^4$Linux-Anwendung durch eine L4 String-Message übertragen werden, bei der die Daten durch den L4 Kern kopiert werden müssen. Um den dadurch entstehenden Aufwand zu bestimmen, wurde zusätzlich die Zeit bestimmt, die durch das Dateisystem für die Bearbeitung eines Auftrags

[1] Die schnellste Zone wurde durch h2bench ermittelt, bei der Quantum VP32210 liegt diese bei Block 0.

[2] Die Messungen von h2bench werden nur bis zu einer Blockgröße von 64 KByte durchgeführt.

| | Schreiben | | Lesen | |
|---|---|---|---|---|
| Blockgröße [KByte] | Bearbeitungsdauer [ms] | Datenrate [KByte/s] | Bearbeitungsdauer [ms] | Datenrate [KByte/s] |
| 4 | 103,67 | 1234 | 62,01 | 2064 |
| 8 | 88,80 | 1441 | 49,32 | 2595 |
| 16 | 69,47 | 1842 | 45,39 | 2820 |
| 32 | 57,86 | 2212 | 44,18 | 2897 |
| 64 | 56,63 | 2260 | 44,08 | 2903 |
| 128 | 53,02 | 2414 | 43,13 | 2967 |

Tabelle 5.1: Meßergebnisse

intern benötigt wird. Die Differenz der Gesamtbearbeitungszeit und dieser Zeit ist der Kommunikationsaufwand zwischen Dateisystem und der Anwendung. Für die bei der Messung verwendete Datengröße von 128 KByte wurde dafür eine Zeit von ca. 20 ms gemessen. Berücksichtigt man diese Zeit bei der Berechnung der Datenrate, wird durch das Dateisystem beim Lesen annähernd die Maximaldatenrate, beim Schreiben ca. 60% der maximalen Datenrate erreicht.

Besonders beim Lesen sind die Datenraten über einen größeren Bereich relativ konstant. Aufgrund der Meßergebnisse von h2bench läßt sich dies jedoch auf die kontinuierliche Allokation der Datei zurückführen, für eine zufällige Anordnung der Blöcke ergeben sich deutliche Unterschiede in den erreichten Datenraten.

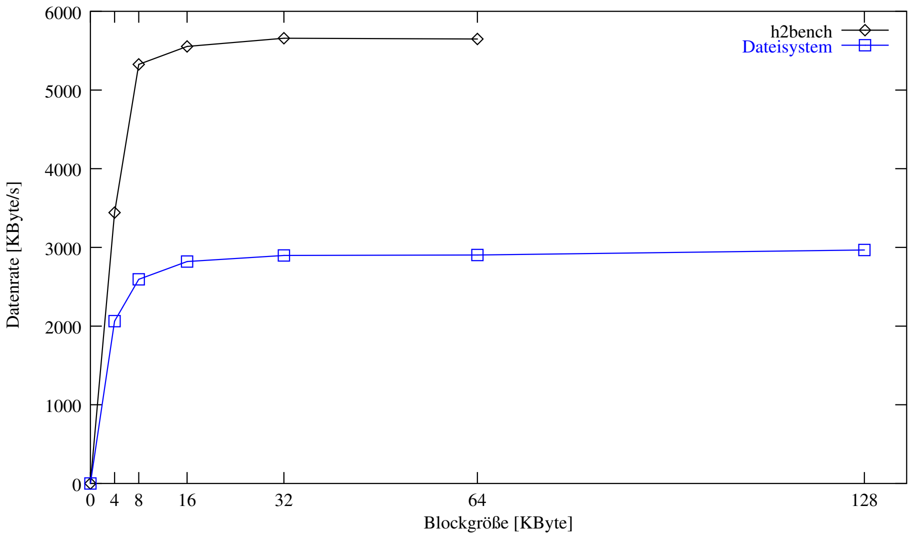



Abbildung 5.1: Datenraten beim Lesen

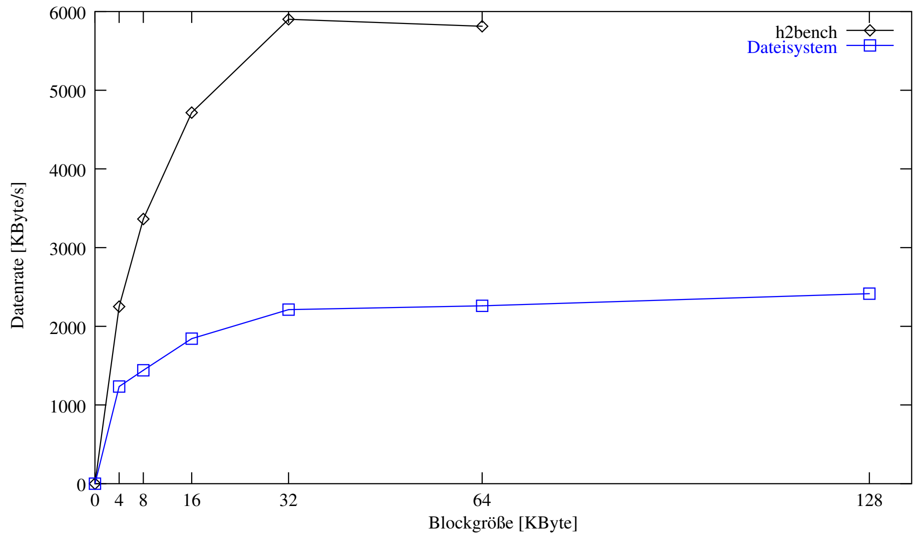



Abbildung 5.2: Datenraten beim Schreiben

# Kapitel 6

# Zusammenfassung und Ausblick

In dieser Arbeit wurden der Entwurf und Teile der Implementierung eines echtzeitfähigen Dateisystems beschrieben. Das Dateisystem verwendet zum Speichern der Daten mehrere Festplatten, um den hohen Bedarf an Speicherplatz und Bandbreite der betrachteten Datenströme erfüllen zu können. Um den speziellen Eigenschaften der unterschiedlichen Datentypen Rechnung zu tragen, können bei der Allokation des Speicherplatzes unterschiedliche Blockgrößen verwendet werden.

Für den Zugriff auf die Daten verwendet das Dateisystem einen zusagefähigen SCSI-Treiber, der in der Diplomarbeit von Frank Mehnert beschrieben ist [Meh98]. Die für den Entwurf verwendeten Ideen zur Admission Control beruhen auf einer Arbeit von Sven Rudolph [Rud97a].

Für weitere Arbeiten an dem Dateisystem in der nächsten Zeit gibt es folgende Problemstellungen:

- Abschluß der Implementation, dies umfaßt die Behandlungsroutinen für kontinuierliche und nichtkontinuierliche Datenströme.
- Ausgehend von dieser Implementierung eine Überprüfung der getroffenen Entwurfsentscheidungen. Insbesondere gilt das für die Verwendung variabler Blockgrößen, es ist zu überprüfen, ob bei den letztendlich zur Admission Control und Planung verwendeten Verfahren dies noch sinnvoll ist.
- Die derzeitige Organisation der Dateien ist stark durch die Vorgehensweise bei der Bearbeitung kontinuierlicher Datenströme beeinflußt. Es ist zu untersuchen, inwieweit für die Behandlung nichtkontinuierlicher Daten geeignetere Organisationsstrukturen verwendet werden müssen.
- Eine Aufgabenstellung für einen späteren Zeitpunkt ist die Entwicklung eines großen Speichersystems auf Basis mehrerer miteinander verbundener Dateisysteme.

Die bei ersten Messungen erreichten Leistungen liegen noch deutlich unter den theoretisch möglichen Werten. Die Gründe dafür sind jedoch zum Teil bekannt, so daß bei einer entsprechenden Implementierung deutlich bessere Ergebnisse erwartet werden können.

# Anhang A

# Glossar

**Block** Einheit, in der Speicherplatz auf der Festplatte reserviert werden kann. Die minimale Blockgröße ist durch die Festplatten-Hardware bestimmt, das Dateisystem kann darauf aufbauend größere logische Blöcke definieren.

**Continuous Media Data** Bezeichnung für Datentypen, auf die in der Regel kontinuierlich zugegriffen wird, d.h. die einzelnen Teilobjekte werden der Reihe nach bearbeitet. Beispiele dafür sind Video- und Audioströme, die einzelnen Teile (Bilder bzw. Samples) werden zum Abspielen der Reihe nach gelesen. Der zeitliche Abstand zwischen den Teilobjekten wird durch die Datenrate bei der Bearbeitung bestimmt.

**DMA** *Direct Memory Access*, direkter Zugriff auf den Hauptspeicher ohne Mitwirkung der CPU.

**Fragmentierung** Aufteilung des Speicherplatzes in benutzte und unbenutzte Teilbereiche, die durch das Löschen von einzelnen Dateien entsteht.

**Hamming-Code** Verfahren zur Prüfsummenberechnung, bei dem auch Mehrfachfehler erkannt und korrigiert werden können.

**Metadaten** Verwaltungsdaten einer Datei, die zusätzlich zu den Nutzdaten gespeichert werden müssen.

**MPEG** *Moving Pictures Experts Group*

**Pager** Komponente zur Verwaltung eines virtuellen Adreßraums.

**Positionierungszeit** Zeit, die durch die Festplatte zur Positionierung des Schreib-/Lesekopfs über den angeforderten Block benötigt wird.

**SCAN** Algorithmus zur Planung von Festplatten-Aufträgen. Die Aufträge werden der Reihe nach bearbeitet, kann während der Positionierung jedoch bereits ein anderer Auftrag bearbeitet werden, wird dieser bereits im voraus bearbeitet.

**SCAN-EDF** Wie *SCAN*, die Reihenfolge der Aufträge wird jedoch durch den Zeitpunkt bestimmt, an dem ein Auftrag ausgeführt sein muß.

**SCSI** *Small Computer Systems Interface*, Standard zum Anschluß vom Peripheriegeräten

**Verschnitt** Differenz zwischen dem reservierten Speicherplatz und der tatsächlichen Größe der Datei. Diese entsteht, da die Reservierung nur in Vielfachen der verwendeten Blockgröße erfolgen kann.

**Worst-Case** ungünstigster anzunehmender Anwendungsfall

**VFS** *Virtual File System*, einheitliche Schnittstelle des Linux-Kerns zum Zugriff auf verschiedene Dateisysteme.

# Literaturverzeichnis


[AOG91] ANDERSON, David P. ; OSAWA, Yoshitomo ; GOVINDAN, Ramesh: Real-Time Disk Storage and Retrieval of Digital Audio/Video Data / CS Division, EECS Department, University of California at Berkeley. 1991 ( CSD-91-646). – Forschungsbericht

[BBD 95] BECK, Michael ; BÖHME, Harald ; DZIADZKA, Mirko ; KUNITZ, Ulrich ; MAGNUS, Robert ; VERWORNER, Dirk: *Linux-Kernel-Programmierung - Algorithmen und Strukturen der Version 1.2*. Addison-Wesley, 1995

[BFD97] BOLOSKY, William J. ; FITZGERALD, Robert P. ; DOUCEUR, John R.: Distributed Schedule Management in the Tiger Video Fileserver. **In:** *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, 1997

[Bög96] BÖGEHOLZ, Harald. H2bench. zu finden unter ftp://ftp.heise.de/pub/ct/pci/h2bench.zip. 1996

[BGMJ94] BERSON, Steven ; GHANDEHARIZADEH, Shahram ; MUNTZ, Richard ; JU, Xiangyu: Staggered Striping in Multimedia Information Systems. **In:** *Proceedings of ACM SIGMOD*, 1994

[Bil92] BILIRIS, Alexandros: An Efficien Database Storage Structure for Large Dynamic Objects. **In:** *Proceeding of the IEEE Data Engineering Conference, Phoenix*, 1992

[Cus94] CUSTER, Helen: *Inside the Windows NT File System*. Microsoft Press, 1994

[GVKR95] GEMMELL, D. J. ; VIN, Harrick M. ; KANDLUR, Dilip D. ; RANGAN, P. V.: Multimedia Storage Servers: A Tutorial and Survey. **In:** *IEEE Computer* (1995)

[GZS 96] GHANDEHARIZADEH, Sharam ; ZIMMERMANN, Roger ; SHI, Weifeng ; REJAIE, Reza ; IERADI, Doug ; LI, Ta-Wei: Mitra: A Scalable Continuous Media Server / University of Southern California. 1996. – Forschungsbericht

[Ham97] HAMANN, Claude-Joachim: On the Quantitative Specification of Jitter Constrained Periodic Streams. **In:** *Proceedings of MASCOTS*, 1997

[Hoh96] HOHMUTH, Michael: *Linux-Emulation auf einem Mikrokern*, TU Dresden, Diplomarbeit, 1996

[Här97a] HÄRTIG, Hermann. Betriebssysteme. Vorlesungsskript TU Dresden. 1997

[Här97b] HÄRTIG, Hermann. Ein Planungsmodell für das Echtzeitdateisystem. mündliche Mitteilung. 1997

[KGM93] KAO, Ben ; GARCIA-MOLINA, Hector: An Overview of Real-Time Database Systems. **In:** *Proceedings of NATO Advanced Study Institute on Real-Time Computing*, 1993

[Kli97] KLIX, Thomas: *Multimedia-Dateisystem auf L4*, Technische Universität Dresden, Diplomarbeit, 1997

[Lea96] LEA, Doug. A Memory Allocator. zu finden unter http://g.oswego.edu/dl/html/malloc.html. 1996

[Lie96] LIEDTKE, Jochen. L4 Reference Manual for 486, Pentium and Pentium Pro. zu finden auf http://os.inf.tu-dresden.de/L4/l4refx86.ps.gz: IBM Watson Technical Report. 1996

[Löw97] LÖWIS, Martin: Im verborgenen: Microsofts NT Filesystem. **In:** *iX* (1997), 4, S. 136 – 139

[Meh98] MEHNERT, Frank: *Ein zusagenfähiges SCSI-Subsystem für DROPS*, Technische Universität Dresden, Diplomarbeit, 1998

[MHN95] MAKAROFF, Dwight J. ; HUTCHINSON, Norman C. ; NEUFELD, Gerald W.: The UBC Distributed Continuous Media File System: Internal Design of Server / Department of Computer Science at University of British Columbia, Canada. 1995. – Forschungsbericht

[MJLF84] MCKUSICK, Marshall K. ; JOY, William N. ; LEFFLER, Samuel J. ; FABRY, Robert S.: A Fast File System for UNIX. **In:** *ACM Transactions on Computer Systems (TOCS)* 2 (1984), Nr. 3

[NOM 97] NISHIKAWA, Junji ; OKABAYASHI, Ichirou ; MORI, Yasuhiro ; SASAKI, Shinji ; MIGITA, Manabu ; OBAYASHI, Yoshimasa ; FURUYA, Shinji ; KANEKO, Katsuyuki: Design and Implementation of Video Server for Mixed-rate Streams. **In:** *Proceedings of the 7th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, 1997

[Pau97] PAUL, Torsten. Videopräsentationen mit Echtzeitsystemen. Großer Beleg, TU Dresden. 1997

[PGK88] PATTERSON, David A. ; GIBSON, Garth ; KATZ, Randy H.: A Case for Redundant Arrays of Inexpensive Disks (RAID). **In:** *Proceedings of the SIGMOD Conference*, 1988

[Rei97] REISER, Hans. Trees Are Fast. zu finden unter http://ideom.com/∼beverly/reierfs.html. 1997

[RO92] ROSENBLUM, Mendel ; OUSTERHOUT, John K.: The Design and Implementation of a Log-Structured File System. **In:** *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, 1992

[Rud97a] RUDOLPH, Sven. Admission Control im RTFS. mündliche Mitteilung. 1997

[Rud97b] RUDOLPH, Sven. Bandbreitenanpassung durch Datenlayout. mündliche Mitteilung. 1997

[RzS96] RASTOGI, Rajeev ; ÖZDEN, Banu ; SILBERSCHATZ, Avi: Disk Striping in Video Server Environments. **In:** *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, 1996

[RzS97] RASTOGI, Rajeev ; ÖZDEN, Banu ; SILBERSCHATZ, Avi: Multimedia Support for Databases. **In:** *Proceedings of the ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, 1997

[SGRV97] SHENOY, Prashant J. ; GOYAL, Pawan ; RAO, Sriram S. ; VIN, Harrick M.: Symphony: An Integrated Multimedia File System / Department of Computer Sciences, Univ. of Texas at Austin. 1997 ( TR-97-09). – Forschungsbericht

[SV97] SHENOY, Prashant J. ; VIN, Harrick M.: Efficient Striping Techniques for Multimedia File Servers. **In:** *Proceedings of the 7th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 97)*, 1997

[Tan94] TANENBAUM, Andrew S.: *Moderne Betriebssysteme*. Carl Hanser Verlag / Prentice Hall, 1994

[WJNB95] WILSON, Paul R. ; JOHNSTONE, Mark S. ; NEELY, Michael ; BOLES, David: Dynamic Storage Allocation: A Survey and Critical Review. **In:** *Proceedings of the International Workshop on Memory Management*, 1995

[Wol97] WOLTER, Jean. Erste Ideen zum Scheduling im DROPS-Projekt. Vortrag Echtzeit-AG, TU Dresden. November 1997