Entwurf und Implementierung eines schnellen Filesystems für UNIX

unter besonderer Berücksichtigung der technischen Parameter

optischer Speichermedien und multimedialer Anwendungen

Jörg Schilling
Matrikel-Nr. 45181

Zusammenfassung

Das Ziel dieser Arbeit ist es, ein Filesystem für UNIX† zu implementieren, das sowohl den besonderen Effizienz-Anforderungen von Multimedia-Anwendungen gerecht wird, als auch die technischen Parameter der heutigen optischen Speichermedien inclusive Worm-Medien (Write once read many) berücksichtigt.

Dazu wird zunächst das in SunOS vorhandene (und in System V release 4 übernommene) Kernel-interne Filesystem-Interface untersucht und eine Dokumentation dieser Schnittstelle erarbeitet. Danach kann dann unter Verwendung dieser Dokumentation ein einfaches, aber schnelles Filesystem auf der Basis zusammenhängender Dateien entworfen und implementiert werden.

Beim Entwurf wird insbesondere untersucht, welche Folgerungen sich aus dem gewünschten Einsatz von *Worm*-Medien ergeben und ob die Komplexität und die Performance des zu entwerfenden Filesystems durch diese Folgerungen beeinflußt werden.

† UNIX ist ein Warenzeichen von AT&T.

1. Einführung

Die heute zunehmend an Bedeutung gewinnenden multimedialen Anwendungen erfordern es unter anderem, digitalisierte Videodaten lesen und schreiben zu können. Bei Daten dieser Art ist es notwendig, eine Vorhersage über die beim Lesen oder Schreiben erreichten Datentransferraten treffen zu können. Auch bei Anwendung modernster Kompressionsverfahren ist zur Speicherung digitaler Videodaten ein konstanter Durchsatz von 1.5 bis 2 Mbits/s erforderlich. Würde man solche Videodaten auf einem UNIX-Filesystem speichern, dann würde die im laufenden Betrieb eines solchen Filesystems entstehende Fragmentierung des Mediums zu nicht mehr vorhersagbaren Datentransferraten führen. Datentransfers mit konstantem Durchsatz und ohne Unterbrechungen lassen sich bei Verwendung dieses Filesystems nicht erreichen. Um einen konstanten Durchsatz der Daten garantieren zu können, müssen die Daten daher kontinuierlich auf dem Medium abgelegt werden, denn nur dann sind keine unberechenbaren Unterbrechungen des Datentransfers durch *Seeks* (Armbewegungen) auf dem Medium zu erwarten.

Da bei multimedialen Anwendungen große Datenmengen anfallen, entsteht ein Wunsch nach der Verwendung von Wechselmedien. Um die anfallenden Datenmengen dabei zu akzeptablen Kosten und mit geringem Platzbedarf archivieren zu können, im Betrieb aber Direktzugriff zu erlauben, kommen zur Zeit nur optische Speichermedien in Frage.

1.1. Optische Speicher Medien

Es sind zur Zeit drei Typen von optischen Speichermedien verfügbar:

Beim Typ *CD-Rom* werden die Daten in der Fabrik dem Medium mechanisch eingeprägt, sie sind beim Anwender nur noch lesbar. Bei diesem Laufwerkstyp wird die von der *Compact-Disk* her bekannte Technologie verwendet.

Das uns zugängliche *CD-ROM*-Laufwerk (ein CDU-8012 von der Firma Sony) hat eine kontinuierliche Lesegeschwindigkeit von ca. 150 kBytes/s. Die mittlere Seekzeit beträgt ca. 700 ms, die maximale Seekzeit liegt deutlich über 2 Sekunden.

Der Typ *Worm* (Write Once Read Many) erlaubt es, jeden Sektor des Mediums einmal zu beschreiben, danach sind die Daten nur noch lesbar. Bei dieser Technologie werden mit Hilfe eines starken Infrarot-Lasers "Brandblasen" auf einer speziellen Kunststoffschicht erzeugt, die sich auf dem Glassubstrat des Mediums befindet. Beim anschließenden Auslesen mit geringerer Energie des Lasers können dadurch Helligkeitsunterschiede erkannt werden.

Bisher konnte von uns keines der neueren *Worm*-Laufwerke getestet werden. Das uns zugängliche Worm-Laufwerk ist ein 5¼-Zoll-Laufwerk (ein RXT-800S der Firma Maxtor). Es hat eine maximale kontinuierliche Lesegeschwindigkeit von ca. 100 kBytes/s und eine kontinuierliche Schreibgeschwindigkeit von ca. 30 kBytes/s, die mittlere Seekzeit liegt bei 400 ms.

Die dritte Gruppe bilden die wiederbeschreibbaren optischen Speichermedien. Auf dem Markt sind dies zur Zeit magneto-optische Laufwerke,[1] bei denen der Faraday-Effekt beim Durchgang von polarisiertem Laserlicht durch eine magnetisch aktive Kunststoffschicht auf einem Glassubstrat ausgenutzt wird. Die magnetischen Domänen in der Kunststoffschicht lassen sich neu ausrichten, wenn die gewünschte Stelle des Mediums mit Hilfe eines Lasers über die Curie-Temperatur der magnetisch aktiven Schicht erhitzt wird. Durch diese Technologie lassen sich diese Medien beliebig häufig wiederbeschreiben, sie sind aber, bedingt durch die hohe Masse der zu bewegenden Optik, in ihren Armbewegungen deutlich langsamer als Magnetplatten. Da zur Ummagnetisierung der magnetischen Domänen des Mediums aber ein starkes Magnetfeld benötigt wird, kann dafür kein örtlich begrenzt wirkender Magnet eingesetzt werden. Der Ort der Ummagnetisierung wird durch die vom Laser erhitzten Stellen bestimmt. Daher ist bei diesem Mediumtyp ein Überschreiben eines Sektors nur in zwei Durchläufen möglich; im ersten Durchlauf wird die Magnetisierung im gesamten Sektor zurückgesetzt, im zweiten Durchlauf werden die Stellen geschrieben, die eine anders gerichtete Magentisierung bekommen sollen.

Das beste uns zur Zeit zugängliche *magneto-optische* Laufwerk ist das Sony SMO-C501. Es erreicht eine kontinuierliche Lesegeschwindigkeit von ca. 400 kBytes/s und eine kontinuierliche Schreibgeschwindigkeit von ca. 100 kBytes/s, die mittlere Seekzeit liegt bei 90 ms, die maximale Seekzeit bei 160 ms.

Die Angaben über die Datentransferraten beziehen sich bei dem CD-Rom Laufwerk und dem magneto-optischen Laufwerk auf eine Sektorgröße von 512 Bytes, bei dem Worm-Laufwerk sind nur Medien mit einer Sektorgröße von 2048 Bytes erhältlich, daher beziehen sich die Angaben dort auf diese Sektorgröße.

1.2. UNIX-Filesystem auf *Worm*-Medien?

Würde man versuchen, ein UNIX-Filesystem auf einem *Worm*-Medium zu installieren, dann würde man sofort scheitern: Beim Entwurf des UNIX-Filesystems wurde davon ausgegangen, daß jeder Sektor des Mediums beliebig häufig beschrieben werden kann. Beim Anlegen einer neuen Datei werden Sektoren an mehreren Bereichen des Mediums modifiziert.[2] Daher könnte man bei Verwendung eines *Worm*-Mediums nicht eine einzige Datei auf diesem Filesystem anlegen.

Das seit UNIX Version 7 verwendete Filesystem arbeitet bei allen Schreib- und Lese-Operationen mit einer Blockgröße von 512 Bytes. Ein *Worm*-Medium mit einer Sektorgröße von 2048 Bytes ist daher mit diesem Filesystem nicht zu verwenden.

---

[1] In der Entwicklung befinden sich auch Laufwerke, die auf der Farbumschlagreaktion einer optisch aktiven Schicht beruhen.

[2] Bei den modifizierten Bereichen handelt es sich um die Directory, in der die neue Datei steht, den *Inode* der Directory, den *Inode* der neuen Datei, und den Superblock. Diese Bereiche werden im nächsten Kapitel noch näher beschrieben.

Will man trotzdem Erfahrungen sammeln, dann kann man ein Berkeley 4.2-Filesystem auf einem Magnetplattenlaufwerk vorbereiten und die Magnetplatte danach physisch auf ein *Worm*-Medium kopieren. Wenn man die so angefertigte Kopie nur zum Lesen in den Filebaum einhängt, dann kann man dieses Filesystem auf dem *Worm*-Medium benutzen. Es verbleibt aber das Problem, *Worm*-Laufwerke auch auf beschreibbare Weise in den UNIX-Filebaum zu integrieren.

Die einfachste Lösung für die Verwaltung von Dateien auf einem *Worm*-Medium ist die Simulation der Funktionen eines Bandlaufwerks. Wenn man mit *tar* (oder einem *tar*-ähnlichen Programm) auf das Medium schreibt, dann lassen sich mehrere Dateien auf dem *Worm*-Medium unterbringen.

Wenn man es bei Anwendung dieser Methode jedoch zulassen will, daß einzelne Dateien modifiziert werden können, wird der Zugriff auf die Dateien sehr langsam. Denn wenn man auf eine Datei zugreifen will, muß jedesmal das gesamte Medium durchsucht werden, um sicherzustellen, daß sich keine neuere Version der Datei als die zuerst gefundene auf dem Medium befindet. Da bei dem Programm *tar* jede Dateibeschreibung den kompletten Dateinamen enthält, wäre zudem eine *rename*-Operation auf eine Directory nicht möglich.

1.3. Performance Aspekte

Die für die Performance eines Filesystems wichtigen Kenngrößen sind die Datentransferrate, die Zeit, die zum Übersetzung eines Dateinamens in die Dateibeschreibung benötigt wird, sowie die Zeit zum Anlegen einer neuen Datei auf dem Filesystem. Ein schnelles Filesystem sollte logische Transferraten haben, die in der Nähe der Transferraten des Mediums liegen. Ein Filesystem, das Transferraten in der Nähe der Transferraten des Mediums aufweist, ist auf allen Medien schnell.

Das Hauptproblem, auf optischen Speichermedien eine gute Performance zu erreichen, ist die geringe *Seek*-Geschwindigkeit dieser Medien. Die *Seek*-Geschwindigkeit ist aber auch bei Magnetplattenlaufwerken ein bestimmender Faktor für die Performance eines Filesystems. Es ist daher beim Entwurf eines schnellen Filesystems nicht erheblich, ob dabei von optischen oder magnetischen Medien ausgegangen wurde. Es muß nur darauf geachtet werden, daß die physische Transferrate des verwendeten Mediums einen ausreichenden Sicherheitsabstand zu der bei der gewünschten Anwendung benötigten Transferrate bietet.

Wenn das Filesystem auch auf *Worm*-Medien lauffähig sein soll, dann muß beim Entwurf allerdings die nicht vorhandene Wiederbeschreibbarkeit dieser Medien berücksichtigt werden.

1.4. Struktur dieser Arbeit

Im zweiten Kapitel werden die Datenstrukturen auf dem Medium, sowie die Überlegungen, die dazu führten, entwickelt und begründet.

Das dritte Kapitel behandelt die (undokumentierte) *Filesystemswitch*-Schnittstelle von SunOS 4.0.

Im vierten Kapitel werden die bei der Implementierung des Filesystems im Kern angewandten Algorithmen beschrieben.

Im fünften, abschließenden Kapitel werden mögliche Erweiterungen zur Erschließung zusätzlicher Anwendungen für das Filesystem aufgezeigt.

2. Datenstrukturen auf dem Medium

Im vorigen Kapitel ist dargestellt worden, daß die wesentlichen beim Entwurf eines effizienten Filesystems zu beachtenden Punkte unabhängig davon sind, ob sich das Filesystem auf einem mehrfach beschreibbaren oder auf einem nur einmal beschreibbaren Medium befindet. Wenn man ein Filesystem entwirft, das jeden Sektor des Mediums nur einmal beschreibt und noch nicht beschriebene Sektoren nicht zu lesen versucht, sieht man keinen Unterschied zwischen den beiden Medien-Typen. Daraus ergibt sich, daß ein Filesystem, das für *Worm*-Medien entworfen wurde, auch auf *magneto-optischen* und *magnetischen* Medien benutzbar ist. Es ist also offenbar ohne Einschränkung der Allgemeinheit möglich, bei dem Entwurf des Filesystems nur von *Worm*-Medien auszugehen. Daher wird das Filesystem im folgenden *Worm*-Filesystem genannt.

Die wichtigsten Forderungen an ein effizientes Filesystem sind:

- Beim Lesen oder Schreiben von Dateiinhalten sollten möglichst wenige *Seek*-Operationen auf dem Medium erforderlich sein.
- Beim Zugriff auf die zu den Dateien gehörenden Beschreibungsdaten sollten ebenfalls möglichst wenige *Seek*-Operationen auf dem Medium erforderlich sein.
- Beim Zugriff auf die zu den Dateien gehörenden Beschreibungsdaten sollten möglichst wenige Mediumzugriffe benötigt werden.

Sind *Seek*-Operationen auf dem Medium nicht zu vermeiden, dann sollte darauf geachtet werden, daß sie selten sind und daß nur kurze *Seeks* vorgenommen werden.

Die Anwendung für unser Filesystem ist aus Sicht eines Filesystems im wesentlichen eine Benutzung durch einen einzelnen Prozess, daher kann man mit kontinuierlich geschriebenen Daten praktisch die Datentransferrate des Mediums erreichen.

2.1. Grundstrukturen eines Filesystems

Ein Filesystem soll es ermöglichen, mehrere Dateien auf einem Medium so zu speichern, daß die einzelnen Dateien unabhängig voneinander erzeugt, beschrieben und gelesen werden können. Dazu werden folgende Informationen benötigt, um die Dateien zu verwalten:

- Eine Beschreibung der Parameter des Filesytems. Das sind Werte, die die Größe, und möglicherweise die Geometrie, des Filesystem beschreiben, sowie konfigurierbare Parameter, mit denen das Filesystem an spezielle Wünsche und Anwendungsfälle des Anwenders angepaßt werden kann.
- Eine Beschreibung der einzelnen Dateien. Diese Beschreibung enthält Daten wie den Eigentümer der Datei, die Zugriffsrechte, die Größe und die Beschreibung, an welchen Stellen des Mediums die Dateiinhalte gespeichert sind.
- Eine Beschreibung der Struktur des Filesystems. Bei einem hierarchisch organisierten Filesystem, wie es unter UNIX verwendet wird, muß in einer geeigneten Weise der Zusammenhang zwischen Directories und den darin enthaltenen Dateien hergestellt werden können.

2.2. Die wesentlichen Strukturen des Berkeley 4.2-Filesystems

Um die Gedanken, die zu der hier zu beschreibenden Struktur des *Worm*-Filesystems geführt haben, zu erläutern, werden wir zur Einleitung zuerst kurz die wesentlichen Strukturen des *BSD*-Filesystems [Berkeley 4.2 SMM Kap. 14] erläutern.[3]

Das *BSD*-Filesystem teilt den Gesamtbereich des Mediums, unabhängig von der sonstigen Struktur in mehrere Zonen ein, die *Zylindergruppen* genannt werden. Diese Einteilung dient dazu, die für den Zugriff auf Dateien notwendigen *Seek*-Operationen möglichst klein zu halten.

Unabhängig von der eben beschriebenen Zylindergruppen-Aufteilung gibt es folgende Bereiche auf dem Medium:

- Der Superblock enthält die Daten, die die Geometrie des Mediums, sowie das Filesystemlayout beschreiben.
- Die *Inode*-Blöcke am Anfang jeder Zylindergruppe, enthalten, bis auf die Beschreibung des logischen Orts im Filesystembaum, alle Eigenschaften und beschreibenden Daten der Dateien. Bei den Beschreibungsdaten handelt es sich z.B. um den Besitzer der Datei, die Zugriffszeiten, die Dateigröße, sowie die Blöcke des Mediums, auf denen sich die Inhalte der Datei befinden.
- Die Inhalte der Dateien belegen die Bereiche einer Zylindergruppe, die sich jeweils hinter den *Inode*-Blöcken einer Zylindergruppe befinden.

Die Daten, die die logische Struktur des Filesystems repräsentieren, befinden sich innerhalb von *Directories.* Das sind spezielle Dateien, deren Inhalt aus Mengen von Paaren von Pfadnamen-Komponenten und *Inode*-Nummern besteht. Eine *Inode*-Nummer verweist auf einen einzelnen Eintrag innerhalb eines *Inode*-Blocks. Dieser Eintrag enthält die restlichen Beschreibungs-Daten der Datei.

Die eben beschriebene Struktur hat jedoch gewisse Nachteile, falls man sie auf ein Filesystem für ein *Worm*-Medium abbilden will. Das Anlegen einer neuen Datei, das Löschen einer Datei, sowie das Ändern eines Dateinamens sind hier Schreiboperationen auf die Directory, in der sich die Datei befindet. Das bedeutet, daß sich der Inhalt der Directory in diesen Fällen ändern würde.

Da bei *Worm*-Medien einzelne Blöcke nicht überschrieben werden können, müßte man in diesen Fällen immer eine neue Directory mit dem aktualisierten Inhalt anlegen. Das bliebe nicht ohne Einfluß auf die Dateibeschreibung der Directory, die dann ebenfalls zu aktualisieren wäre.

Um solche Schreiboperationen auf Directories zu vermeiden, muß man Wege suchen, die es ermöglichen, die Struktur eines Filesystems auf andere Weise darzustellen. Dazu muß es jedoch gelingen, die Baumstruktur des Filesystems zu beschreiben, ohne die im UNIX-Filesystem angewandte Methode zu verwenden.

---

[3] Mit Ausnahme der Tatsache, daß das *BSD*-Filesystem das Medium in Zylindergruppen einteilt, sind die Unterschiede des UNIX-Version 7-Filesystems und des *BSD*-Filesystems, für die hier besprochenen Eigenschaften ohne Belang.

2.3. Die Struktur des Filesystems

Bei dem Entwurf der Struktur des *Worm*-Filesystems müssen folgende Randbedingungen beachtet werden:

- Der UNIX-Gerätetreiber erwartet, daß auf dem ersten Sektor des Mediums ein sogenanntes *Label* steht. Das Label wird vom UNIX-Gerätetreiber für die Partitionierung, sowie für eventuell notwendige Informationen über die Geometrie des Mediums verwendet.
- Die dem Label folgenden Sektoren werden üblicherweise bei UNIX für ein primäres Bootstrap-Programm bereitgehalten.

Hinter diesen Bereich schreibt man sinnvollerweise eine Beschreibung der Parameter des Filesystems. Diese Beschreibung wird im Folgenden *Primärer Superblock* genannt.

Der restliche Bereich des Mediums steht zur freien Disposition für die Daten des Filesystems. Für eine grobe Einteilung des Mediums kann man im wesentlichen von zwei gleichzeitig wachsenden Bereichen ausgehen:[4]

- Die Dateiinhalte
- Die Dateibeschreibungen.

Eine Verschränkung dieser Bereiche kann man vermeiden, wenn man einen Bereich vom Anfang des Mediums zum Ende des Mediums und den anderen Bereich vom Ende des Mediums zum Anfang wachsen läßt. Auf diese Weise kann man vermeiden, daß man von Anfang an einem der Bereiche eine feste Größe zuweisen muß.

Dadurch ist dann, im Gegensatz zum UNIX-Filesystem, nicht schon zum Zeitpunkt der Erzeugung des Filesystems festgelegt, wieviele Dateien auf dem Filesystem erzeugt werden können und wieviel Platz für Dateiinhalte zur Verfügung steht.[5] Dadurch kann man, ohne sich schon bei der Erzeugung des Filesystems festlegen zu müssen, den auf dem Medium verfügbaren Speicherplatz optimal ausnutzen.

Da das Filesystem die Größe von Dateien nicht voraussagen kann, müssen sich die Dateiinhalte, damit sie später kontinuierlich auf dem Medium stehen, auf dem Bereich befinden, der vom Anfang zum Ende des Mediums wächst.

---

[4] In einem der folgenden Abschnitte wird noch darauf eingegangen, daß es wegen der Berücksichtigung der Besonderheiten der *Worm*-Medien noch einen dritten wachsenden Bereich geben muß. Er wird in dem folgenden Bild als *Superblock-Updates* bezeichnet.

[5] Bei einem BSD-Filesystem von 400 MBytes Größe werden bei Standardparametern für das *mkfs(8)*-Programm fast 24 MBytes für *Inodes* reserviert. Dadurch sind ca. 17% der Kapazität des Mediums, unabhängig von der wirklich benötigten Menge, nicht für Dateiinhalte verfügbar.

Sektor #0	Label (1 Sektor)
	Bootstrap (falls benötigt)
	Primärer Superblock (auf fester Adresse, Minimalgröße 8kBytes)
	1 Sprung Superblock Update
	63 Superblock Updates
	Filedaten ↓
	1 Sprung Superblock Update
	63 Superblock Updates
	Filedaten ↓
	... Pufferzone für Filedaten und Generations Knoten ...
	↑ Generations Knoten
Letzter Sektor	Sicherheitskopie des primären Superblocks

Bild 1: Layout des Filesystems für *Worm*-Medien.

Dabei ist der Sektor #0 der erste adressierbare Sektor innerhalb einer Partition. Mit dem letzten Sektor ist entsprechend der letzte adressierbare Sektor der betreffenden Partition gemeint. Auf die sonstigen in dieser Abbildung bezeichneten Bereiche wird in den folgenden Abschnitten näher eingegangen.

2.4. Der *Generations*-Knoten als Beschreibung der Dateien

Im UNIX-Filesystem ist der *Inode* der Informations-Knoten für die Datei. Er enthält, wie schon beschrieben, bis auf die Information über den logischen Ort der Datei im Filesystembaum, sämtliche Informationen für eine Datei. Unter anderem enthält er auch die Zeiten des letzten Lesezugriffs, des letzten Schreibzugriffs und der letzten Statusänderung der Datei. Das bewirkt jedoch, daß er, nach dem Superblock, zu den am häufigsten geschriebenen Bereichen auf dem Medium gehört.

Da bei *Worm*-Medien jeder Sektor nur einmal beschrieben werden kann, ist eine Abbildung der im UNIX-Filesystem angewandten Methode auf das *Worm*-Filesystem nicht ohne weiteres möglich. Man könnte zwar eine Verringerung der notwendigen Schreiboperationen erreichen, wenn man bei der Verwendung von *Worm-Medien* die Zeit des letzten Lesezugriffs nicht aktualisierte. Damit ist das Hauptproblem jedoch nicht gelöst.

Eine Lösung für das Problem ergibt sich jedoch, wenn man die notwendigen Updates der Dateibeschreibung jeweils an eine andere Stelle des Mediums schreibt. Man hat dann für jeden Zustand der Datei eine zu diesem Zustand gehörende Dateibeschreibung auf dem Medium.

Um aus diesen Beschreibungen die aktuelle herausfinden zu können, muß man sie nur in einer festgelegten Richtung auf das Medium schreiben. Wenn man dann von dem logischen Ende des Bereiches, auf dem sich die Beschreibungen auf dem Medium befinden, rückwärts[6] nach dem Eintrag für eine bestimmte Datei sucht, dann findet man die aktuelle Beschreibung dieser Datei zuerst.

Da man durch die soeben beschriebene Methode für jede Generation einer Datei eine zu dieser Generation gehörende Beschreibung bekommt,[7] wird diese Beschreibung der Datei im folgenden *Generations*-Knoten genannt.

Der *Generations*-Knoten ist, entsprechend dem *Inode* des UNIX-Filesystems, der *Informations*-Knoten für Dateien im *Worm*-Filesystem.

2.4.1. Probleme durch den Update von Generations-Knoten an anderer Stelle

Durch diese oben beschriebene Methode des Updates von Generations-Knoten gibt es allerdings Probleme, eine Directory-Struktur, wie sie beim UNIX-Filesystem verwendet wird, anzulegen. Wenn man für den Generations-Knoten einer Datei keinen festen Platz auf dem Medium vorsehen kann, dann ist es auch nicht möglich, eine feste Rechenvorschrift anzugeben, mit der man aus dem Directory-Eintrag für eine Datei, auf den Ort des zu dieser Datei gehörenden Generations-Knotens auf dem Medium kommt.

Dieser Zusammenhang, der beim UNIX-Filesystem gegeben ist, läßt sich offensichtlich beim *Worm*-Filesystem nicht beibehalten, denn eine Veränderung an einer Datei erfordert einen Update des Generations-Knotens dieser Datei. Der Update des Generations-Knotens an anderer Stelle würde aber einen Update der Directory, in der sich die Datei befindet, bedeuten, was sich dann in einer Kette bis zur *Root*-Directory des betreffenden Filesystems fortsetzte.

[6] d.h. entgegen der Wachstumsrichtung.

[7] Wie noch im Abschnitt über die Modifikation von Dateien gezeigt wird, könnte man beim *Worm*-Filesystem, durch eine Erweiterung der Funktionen des Filesystemcodes, auch auf ältere Versionen einer Datei zugreifen.

2.4.2. Der Dateiname im *Generations*-Knoten

Wenn es gelingt, den Dateinamen im Generations-Knoten unterzubringen und die Baumstruktur des Filesystems auf eine andere Art als beim UNIX-Filesystem darzustellen, dann kann man auf einen Directory-Inhalt, wie beim UNIX-Filesystem, ganz verzichten. Um die Möglichkeit der Unterbringung des Dateinamens im Generations-Knoten zu überprüfen, werden zunächst einige Überlegungen über die Länge von Dateinamen im Betriebssystem UNIX durchgeführt:

Die maximale Länge einer Pfadnamen-Komponente beträgt bei UNIX inclusive *NULL*-Byte 256 Bytes *(MAXNAMLEN)*. Da für die restlichen Daten des Generations-Knotens ca. 96 Bytes benötigt werden, ist es problemlos möglich, den Generations-Knoten inclusive der Pfadnamen-Komponente in einem Sektor von 512 Bytes Größe unterzubringen.

Um einen besseren Überblick über die in der Praxis vorkommenden Längen von Pfadnamen-Komponenten zu erhalten, wurden Messungen an einem Rechner, der mit SunOS 4.0.1 betrieben wird, vorgenommen.

Auf dem `/usr`-Filesystem dieses Rechners wurden dabei folgende Werte ermittelt:

- Es befanden sich insgesamt 5984 Dateien inclusive Directories, Hard-Links und Symbolischer Links auf dem Filesystem.
- Es befanden sich 267 Directories auf dem Filesystem.
- Es befanden sich 800 Hardlinks,[8] ohne Berücksichtigung der Einträge "." und ".." in den Directories, auf dem Filesystem.
- Es befanden sich 202 Symbolische Links auf dem Filesystem.

An diesen Dateien wurden Untersuchungen über die Länge der Pfadnamen-Komponenten durchgeführt. (Dabei wurde das abschließende *NULL*-Byte nicht berücksichtigt; Wenn man die folgenden Werte für die Dimensionierung von zu reservierendem Speicherplatz bei C-Programmen verwenden will, muß man sie folglich um eins erhöhen.) Es wurden folgende Werte ermittelt:

Mittlere Länge einer Pfadnamen-Komponente	7.6 Bytes
Maximale Länge einer Pfadnamen-Komponente	28 Bytes
Mittlere Linknamenlänge	15.9 Bytes
Maximale Linknamenlänge	48 Bytes

Aus diesen Messungen kann man schließen, daß sich alle in der Praxis vorkommenden Pfadnamen-Komponenten in 32 Bytes unterbringen lassen. Zusammen mit den ca. 96 Bytes, die für die restlichen Daten des Generations-Knotens benötigt werden, kommt man damit auf eine Größe von 128 Bytes. Daher ist 128 Bytes als die Mimimalgröße eines Generations-Knotens gewählt worden.

[8] Dieser hohe Anteil von Hard-Links ist untypisch für UNIX, er wird hervorgerufen durch 733 Hard-Links auf Einträge in `/usr/share/lib/terminfo` der Terminalbeschreibung von System V.

2.5. Die Realisierung der Filesystemstruktur

Damit die Filesystemstruktur eindeutig zu beschreiben ist, muß man zusätzlich zum Dateinamen folgende Informationen im Generations-Knoten vorsehen:

- Jede Datei bekommt eine eindeutige *Inodenummer* zur Identifizierung.
- Jeder Generations-Knoten enthält die *Inodenummer* der Directory in der er steht. Dieser Eintrag wird im folgenden *Parent*-Zeiger genannt.

Da man mit Hilfe des *Parent*-Zeigers die Directory finden kann, in der die Datei steht, kann man beim *Worm*-Filesystem auf die Einträge ".." und ".", die im UNIX-Filesystem benötigt werden, verzichten. Die Semantik dieser Einträge kann stattdessen durch besondere Maßnahmen im Dateinamenparser realisiert werden. Eine Directory muß folglich durch die Verwendung des *Parent*-Zeigers keine systembedingten Einträge mehr enthalten. Im *Worm*-Filesystem ist eine Directory daher nur eine spezielle leere Datei, die nicht modifiziert werden muß, um Einträge in ihr zu erzeugen.

Auf diese Weise ist zwar die Struktur des Filesystembaumes eindeutig beschrieben, ein Durchlaufen des Baumes in der im Betriebssystem UNIX üblichen Weise ist jedoch nur schwer möglich. Wollte man z.B. die Dateien auflisten, die sich innerhalb einer Directory befinden, so müßte man den gesamten Bereich des Mediums, auf dem sich die Generations-Knoten befinden, durchsuchen.

Dies kann man jedoch vermeiden, wenn man einen Cache vorsieht, der im Systemaufruf *mount* gefüllt wird und die Daten aller aktiven Generations-Knoten enthält. Beim Füllen dieses Caches kann man gleichzeitig die Struktur umkehren. Danach kann man auf das Filesystem in der gewohnten Weise zugreifen.

2.5.1. Dateiinhalte und Modifikation von Dateien im *Worm*-Filesystem

Wenn eine neue Datei auf dem *Worm*-Filesystem angelegt werden soll, dann werden die Inhalte dieser Datei kontinuierlich, beginnend an der ersten freien Stelle des Mediums, in aufsteigender Richtung auf das Medium geschrieben. Die Position dieser freien Stelle ergibt sich aus dem Eintrag für das aktuelle Ende des Datenbereichs im Superblock.

Damit man es erreichen kann, daß alle Dateiinhalte kontinuierlich auf dem Medium liegen, muß man die Anzahl der gleichzeitig zum Schreiben geöffneten Dateien auf einem Filesystem, auf eins beschränken. Man könnte zwar prinzipiell das simultane Beschreiben von mehreren Dateien ermöglichen, wenn man die Dateiinhalte auf dem Swap-Bereich des Betriebssystems zwischenpufferte, dann könnte es aber zu Situationen kommen, bei denen kein temporärer Platz mehr verfügbar ist. Da diese Situationen aber dem Benutzerprogramm willkürlich erscheinen würden und es bei der beabsichtigten Anwendung für das *Worm*-Filesystem nicht nötig ist, mehr als eine Datei gleichzeitig zum Schreiben offen zu haben, wurde diese einfache und klare Beschränkung gewählt.

Da bei *Worm*-Medien einzelne Sektoren nicht überschrieben werden können, ist es nicht möglich, bei der Modifikation einer Datei den von dieser Datei belegten Datenbereich auf dem Medium wiederzuverwenden. Man muß vielmehr, ähnlich wie bei den Generations-

Knoten, die komplette neue Version der Datei auf einen bisher noch nicht benutzten Bereich des Mediums schreiben.

Zu jeder Generation einer Datei verbleiben, bei Anwendung dieser Methode, ein Generations-Knoten und der dazugehörige Datenbereich unverändert auf dem Medium. Daher kann man, wenn man eine ältere Version eines Generations-Knotens für eine Datei findet, jederzeit die zu diesem Generations-Knoten gehörende Dateiversion lesen.

Das Betriebssystem UNIX bietet aber im Gegensatz von z.B. *VMS*[9] kein Benutzer-Interface um auf diese Dateien zugreifen zu können. Um unter UNIX trotzdem Zugang zu alten Versionen von Dateien zu bekommen gibt es zwei Möglichkeiten der Implementierung:

- Mit Hilfe des Systemaufrufs *fcntl(2)* kann, nachdem die aktuelle Version der Datei geöffnet wurde, die Version dieser Datei erfragt, sowie der Zugriff auf andere Versionen ermöglicht werden.
- Durch eine geeignete Erweiterung der Einträge einer Directory[10] kann auf die älteren Versionen von Dateien mit Hilfe von modifizierten Dateinamen zugegriffen werden.

Bei der ersten beschriebenen Möglichkeit können nur Programme, die dieses für UNIX untypische Interface kennen, auf eine andere als die aktuelle Version einer Datei zugreifen. Es wäre also im speziellen nicht möglich, ein Directory-Listing der verfügbaren Versionen zu erhalten. Daher erscheint die zweite Methode als die geeignetere für eine Implementierung. In der aktuellen Version des Filesystems wurde aber wegen Zeitmangel auf eine Implementierung dieses Features verzichtet.

2.5.2. Das „Löschen“ von Dateien im *Worm*-Filesystem

Da auf *Worm*-Medien Datenbereiche nicht überschrieben werden können, ist es ohne besondere Maßnahmen auch nicht möglich, Dateien zu löschen.

Da es aber mit den oben beschriebenen Methoden möglich ist, einen Update für einen Generations-Knoten zu erzeugen, kann man in diesem Update des Generations-Knotens ein *Flag*-Bit setzen, das die betreffende Datei als gelöscht kennzeichnet. Beim Einlesen der Generations-Knoten wird dann dieses *Flag*-Bit erkannt und die Datei ignoriert. Auf der Anwenderprogrammebene ist die betreffende Datei dann nicht mehr sichtbar.

Da die Datei dabei auf dem Medium verbleibt, wäre es auch hier möglich, trotzdem einen Zugriff auf gelöschte Dateien zu ermöglichen. Ein Zugriff auf die gelöschten Dateien könnte, wie im vorigen Abschnitt beschrieben, wie ein Zugriff auf ältere Versionen dieser

[9] *VMS* ist ein auf Rechnern der Firma *Digital* übliches Betriebssystem.

[10] Man könnte z.B. in jeder Directory eine Subdirectory mit dem Namen "..." anlegen, die die älteren Versionen der Dateien aus der ersten Directory trägt. Zur Unterscheidung der einzelnen Versionen der dort gefundenen Dateien wird am Ende jedes Dateinamens ein String in der Form *";Versionsnummer"*, oder *"#Versionsnummer"* angehängt, so daß ein aktuell dort vorgefundener Dateiname z.B. `testdatei;17` oder `testdatei#17` lauten könnte.

Dateien zu implementiert werden.

2.5.3. Methoden zur Implementierung von *Symbolischen* Links

Symbolische Links lassen sich im *Worm*-Filesystem genauso implementieren, wie dies im UNIX-Filesystem geschieht. Ein symbolischer Link ist dort eine besondere Datei, deren Datenbereich den Namen einer anderen Datei enthält – den *Linknamen*.

Die Länge eines Dateinamens ist in UNIX auf *MAXPATHLEN* (1024 Bytes) beschränkt. Deshalb können die Daten dieser speziellen Datei sowohl innerhalb des Bereiches des Mediums angeordnet werden, wo auch die Daten anderer Dateien liegen, als auch innerhalb des Bereichs, der von den Generations-Knoten belegt wird. Letzteres erreicht man, wenn man den Datenraum des Generations-Knotens hinter dem Dateinamen um die Größe des Linknamens erweitert.

Die zweite Methode ist der ersten aus zwei Gründen vorzuziehen:

- Das *Worm*-Filesystem läßt sich effizient nur in Verbindung mit einem Cache für die Generations-Knoten implementieren. Dann ist es aber wünschenswert, daß auch die Linknamen mit in den Cache aufgenommen werden, denn der Linkname wird z.B. bei einem `ls -l` benötigt. Um beim Einlesen der Generations-Knoten in den Cache keine Zeit durch *Seeks* zu den Linkdaten zu verlieren, sollten die Linkdaten innerhalb des Bereichs der Generations-Knoten auf dem Medium liegen.
- Selbst ohne Cache für die Generations-Knoten ist davon auszugehen, daß die Linkdaten häufig gleichzeitig mit den Daten aus den Generations-Knoten benötigt werden. Auch in diesem Fall ist es sinnvoll, beim Zugriff auf die Daten *Seeks* zu vermeiden bzw. kurz zu halten und die Linkdaten im Bereich der Generations-Knoten zu halten.

Wenn man den Linknamen im Bereich der Generations-Knoten unterbringt, muß man allerdings beachten, daß es möglich ist, daß die Gesamtgröße des Generations-Knotens inclusive Pfadnamen-Komponente und Linknamen die Größe eines Sektors auf dem Medium überschreiten kann. Da für den Generations-Knoten ohne die Pfadnamen-Komponente ca. 96 Bytes benötigt werden, bleiben bei der kleinsten denkbaren Sektorgröße von 512 Bytes noch 416 Bytes für Pfadnamen-Komponente und Linkname übrig. Wenn die Pfadnamen-Komponente die bei UNIX maximal erlaubte Länge von 255 Bytes *(MAXNAMLEN)* erreicht, dann bleiben für den Linknamen noch 160 Bytes inclusive *NULL*-Byte übrig.

Bei den oben beschriebenen aktuell erreichten Dateinamen- und Linknamenlängen bleibt man jedoch erheblich unter der 512 Byte-Grenze; bei den mittleren Werten passen die Daten des Generations-Knotens sogar noch in die Minimalgröße von 128 Bytes.

Zu den möglichen Folgen bei einem Systemzusammenbruch während des Schreibens eines sich über mehr als einen Sektor erstreckenden Generations-Knotens sei auf den Abschnitt *Sicherheit bei Systemzusammenbrüchen* verwiesen.

2.5.4. Methoden zur Implementierung von *Hard* Links

Im UNIX-Filesystem besteht kein direkter Verweis vom *Inode* zum Dateinamen. Für eine normale Datei gibt es in einer Directory einen Eintrag, der auf den *Inode* der Datei verweist. Wenn ein *Hard*-Link zu einer Datei angelegt werden soll, dann wird ein weiterer Verweis zu dem *Inode* dieser Datei erzeugt. Nach dem Anlegen des *Hard*-Links ist es nicht mehr möglich, den ersten Eintrag für die Datei von dem angelegten *Hard*-Link zu unterscheiden.

Das *Worm*-Filesystem weicht in zwei für die Implementierung von *Hard*-Links wichtigen Punkten vom UNIX-Filesystem ab:

- Der Generations-Knoten des *Worm*-Filesystems enthält den Namen der Datei.
- Der Generations-Knoten befindet sich im Gegensatz zum *Inode* im UNIX-Filesystem nicht immer an der selben Stelle des Mediums. Durch die für die Berücksichtigung der *Worm*-Medien notwendige Methode zum Update der Generations-Knoten gibt es sogar eventuell mehrere Generations-Knoten für eine Datei.

Da es beim *Worm*-Filesystem keine Directories im Sinne des UNIX-Filesystems gibt, ist es nicht möglich, einen *Hard*-Link als zusätzliche Directory-Referenz auf den *Inode* zu implementieren. Man müßte für die Speicherung des zusätzlichen Dateinamens einen weiteren Generations-Knoten verwenden. Würde man für diesen Generations-Knoten, wie beim UNIX-Filesystem, die gleiche *Inode*-Nummer wie für den Generations-Knoten der Originaldatei verwenden, so würde es nicht möglich sein diesen von einem Update des Generations-Knotens für die Originaldatei zu unterscheiden.

Eine Implementierung der *Hard*-Links in einer vollständig zu UNIX vergleichbaren Weise ist also nicht möglich. Man kann aber offenbar *Hard*-Links in einer Weise implementieren, die ähnlich der eben beschriebenen Methode für *Symbolische* Links ist.

Für einen *Hard*-Link wird dazu, wie bei einem symbolischen Link, ein eigener Generations-Knoten mit einer eigenen *Inode*-Nummer angelegt. Dieser Generations-Knoten wird aber nicht, wie beim symbolischen Link, als namensbezogener Link angelegt, sondern als *inodebezogener* Link ausgeführt. Dazu wird im Generations-Knoten ein Flag-Bit gesetzt, um ihn als *Hard*-Link zu kennzeichnen. Die Relation zur Zieldatei wird dadurch hergestellt, daß in seinem Generations-Knoten die *Inode*-Nummer der Zieldatei eingetragen wird. Das kann zur Platzersparnis an der Stelle des Generations-Knotens geschehen, wo sonst die Nummer des ersten Datenblocks einer Datei steht, denn ein *Hard*-Link benötigt keinen eigenen Datenraum.

Bei dieser Methode muß allerdings beachtet werden, daß der Generations-Knoten der Originaldatei nicht als gelöscht gekennzeichet werden darf, wenn noch *Hard*-Links auf ihn verweisen, denn nur er enthält die für den Zugriff auf die Daten wichtigen Informationen. Will man die Originaldatei löschen, dann hat man zwei Möglichkeiten der Implementierung:

- Der Generations-Knoten der Originaldatei wird mit einem Flag-Bit als nicht mehr zugreifbar, aber noch nicht gelöscht gekennzeichnet. Erst wenn der letzte auf diese Datei verweisende *Hard*-Link gelöscht wird, kann auch der Generations-Knoten der Originaldatei als gelöscht gekennzeichnet werden. Bei dieser Methode muß beim Löschen des letzten auf die Datei verweisenden *Hard*-Links sowohl für diesen *Hard*-Link, als auch für die Originaldatei ein Generations-Knoten mit Löschvermerk geschrieben werden.

- Es wird ein Generations-Knoten eines auf die Originaldatei weisenden *Hard*-Links so modifiziert, daß er die *Inode*-Nummer und, bis auf den Namen und den *Parent*-Zeiger, sämtliche Daten des Generations-Knotens der Originaldatei enthält. Da bei dieser Methode die *Inode*-Nummer des modifizierten *Hard*-Links, in die *Inode*-Nummer der Originaldatei gewandelt wird, ist es nötig, zusätzlich einen Generations-Knoten mit Löschvermerk für den zu modifizierenden *Hard*-Link zu schreiben.

Bei der zweiten Methode bleibt die Struktur des gesamten Filesystems klarer. Daher wurde die zweite Methode für die Implementierung gewählt.

Die *Inode*-Nummer des für den *Hard*-Link angelegten Generations-Knotens wird nach außen nicht sichtbar, sie wird nur für die innere Verwaltung benötigt.

Der *Parent*-Zeiger und die Pfadnamen-Komponente des Generations-Knoten des *Hard*-Links können sich vom *Parent*-Zeiger der Originaldatei unterscheiden, er gibt die Position des *Hard-Links* innerhalb des Filesystembaums an.

2.6. Der Superblock auf dem *Worm*-Filesystem

Der primäre Superblock auf dem *Worm*-Filesystem enthält die Layoutparameter eines aktuellen Filesystems, sowie statistische Daten. Die wichtigsten Layoutparameter sind:

- Die Sektorgröße des verwendeten Mediums.
- Die Anzahl der Sektoren auf dem Medium, die zum Filesystem gehören.
- Die Version des *Worm*-Filesystem, die zum Erzeugen des Filesystems auf dem Medium verwendet wurde.

Wegen der notwendigen Kompatibilität des für das *Worm*-Filesystem verwendeten Mediums mit dem UNIX-Gerätetreiber für das Laufwerk, muß, wie bereits erwähnt, auf dem ersten Sektor des Mediums ein sogenanntes *Label* stehen.

Für den primären Superblock gibt es folgende Randbedingungen, die beachtet werden müssen:

- Der Superblock muß sich hinter dem Label und den für den primären Bootstrap bereitgehaltenen Sektoren befinden.
- Der Offset innerhalb des Mediums, auf dem der Superblock steht, muß durch die Sektorgröße teilbar sein, damit es möglich ist, das primäre Bootstrap-Programm und den Superblock getrennt voneinander zu schreiben.[11]

---

[11] Würde man diese Bedingung nicht beachten, dann müßte der Gerätetreiber vor dem Schreiben des ersten Sektors des Superblocks zunächst diesen Sektor vom Medium einlesen. Ist dieser Sektor noch nicht beschrieben, dann führt das bei einem *Worm*-Medium zu einem Fehler, ist er bereits

- Die Größe des Superblocks muß durch die Sektorgröße teilbar sein.

Da die Sektorgröße vor dem Lesen des primären Superblocks nicht bekannt ist, ist auch nicht bekannt, an welcher Stelle des Mediums sich der Superblock befindet. Es muß also eine Methode gefunden werden, die es ermöglicht, ohne Kenntnis der Sektorgröße und der aktuellen Größe des primären Superblocks, diesen zuverlässig vom Medium einzulesen.

Wenn man dafür sorgt, daß die Größe des primären Superblocks der größten zu erwartenden Sektorgröße entspricht, und ihn auf eine durch diesen Wert teilbaren Adresse innerhalb des Mediums schreibt, dann kann er immer aufgefunden werden. Da die größte mir bekannte Sektorgröße 2 kBytes[12] ist, wurde eine maximale Sektorgröße von 8 kBytes gewählt. Die Tendenz bei der Entwicklung von optischen Speichermedien scheint eher in Richtung zu der Standardgröße von 512 Bytes zu gehen, und der Faktor 4 scheint als Reserve ausreichend.

Da das Filesystem auch auf *Worm*-Medien lauffähig sein soll, ist es notwendig, genau wie bei den Generations-Knoten die Updates für den Superblock an eine andere Stelle des Mediums zu schreiben.

2.6.1. Wege zum schnellen Auffinden des aktuellen Superblocks

Im laufenden Betrieb muß auf dem Medium spätestens nach jedem *unmount(2)*, das einer Modifikation des Filesystems folgte, ein Superblock-Update geschrieben werden, denn bei jeder Modifikation des Filesystems verschiebt sich das aktuelle Ende des von den Generations-Knoten belegten Bereichs auf dem Medium und gegebenenfalls auch das Ende des von Dateiinhalten belegten Bereichs. Diese Werte müssen aber, um die Konsistenz des Filesystems überprüfen zu können, im Superblocks geführt werden.

Ohne Verschränkung ist es nicht möglich, drei unterschiedliche wachsende Datenbereiche auf einem Medium unterzubringen. Daher müssen die Superblock-Updates entweder im Bereich der Dateiinhalte oder im Bereich der Generations-Knoten liegen. Damit das Einlesen der Generations-Knoten nicht durch eingestreute Superblock-Updates verlangsamt wird, wurde beschlossen die Superblock-Updates im Bereich der Dateiinhalte unterzubringen.

Durch die Methode des Updates ist es allerdings nicht möglich, bei einem erneuten *mount(2)* sofort den aktuellen Superblock-Update zu finden. Er muß vielmehr, nach dem die Layoutdaten durch Lesen des primären Superblocks bekannt sind, gesucht werden.

Da bei *Worm*-Medien jeder Sektor nur einmal beschrieben werden kann, muß sich die Position des nächsten Superblock-Updates implizit aus den bisher bekannten Daten ergeben. Die einfachste Methode, dies zu erreichen, besteht darin, für jeweils eine

beschrieben, dann ist es nicht möglich ihn noch einmal zu beschreiben.

[12] Die Sektorgröße des Mediums der Maxtor RXT-800S (Worm) beträgt 2 kBytes, für das Sony SMO-C501 (Magnetooptisch) sind Kassetten mit Sektorgrößen von 512 Bytes und 1024 Bytes erhältlich.

bekannte Anzahl von Superblock-Updates, Platz auf dem Medium bereitzuhalten. Der nächste Superblock-Update kommt dann auf die dem zuletzt geschriebenen Superblock-Update folgende Adresse. Wird der letzte Superblock-Update in den bereitgehaltenen Platz geschrieben, dann wird in ihm vermerkt, wo der nächste für Superblock-Updates bereitgehaltene Bereich beginnt.

Bei der Verwendung von wiederbeschreibbaren Medien für das *Worm*-Filesystem, muß man dann, wenn man einen neuen Bereich für Superblock-Updates anlegt, diesen Bereich nullen, damit beim Einlesen eine Verwechslung mit dort eventuell stehenden Superblock-Updates eines überschriebenen *Worm*-Filesystems vermieden werden kann.

Es ist damit zu rechnen, daß während der aktiven Gebrauchsdauer eines Mediums einige hundert Superblock-Updates geschrieben werden. Daher muß man eine Methode finden, durch die es vermieden werden kann, daß alle diese Superblock-Updates gelesen werden müssen, um den aktuellen zu finden.

Wenn man in einem Bereich zuerst versucht, den an der zuletzt zu beschreibenden Position stehenden Superblock-Update zu lesen, dann ist es im Falle eines Erfolges nicht mehr erforderlich, die restlichen Superblock-Updates aus diesem Bereich einzulesen. Daher wird die erste Position in einem Bereich von Superblock-Updates zu diesem Zweck verwendet, da dadurch das Medium nur in einer Richtung gelesen werden muß, um den aktuellen Superblock-Update zu finden.

Seien:

n Die Anzahl der Zugriffe ohne Anwendung der oben beschriebenen Methode.

n′ Die Anzahl der Zugriffe bei Anwendung der oben beschriebenen Methode.

m Die Anzahl der zu einem Bereich zusammengefaßten Superblock-Updates.

Dann kann man die sich, durch Anwendung der oben beschriebenen Methode, ergebende Verringerung der notwendigen Zugriffe zum Auffinden des aktuellen Superblock-Updates wie folgt ausdrücken:

$$n' = (n / m) + (n \textit{ modulo } m)$$

Wird *m* auf den Wert 64 festgesetzt, d.h. jeweils 64 Superblock-Updates werden zu einem dieser Bereiche zusammenfaßt, dann sinkt, falls sich 1000 Superblock-Updates auf dem Medium befinden, die Anzahl der zum Auffinden des aktuellen Superblock-Updates erforderlichen Zugriffe auf das Medium, auf 35.

Wenn es erforderlich sein sollte, ein Filesystem wesentlich häufiger mit einem Superblock-Update zu versehen, kann man die Anzahl der Superblock-Updates, die zu einem oben beschriebenen Bereich zusammengefaßt werden, konfigurierbar machen. Wenn man (statt 64) 256 Superblock-Updates zu einem Bereich zusammenfaßt, dann senkt sich die Anzahl der bei 10 000 vorhandenen Superblock-Updates nötigen Zugriffe von 172 auf 55.

Als grobe Richtlinie kann man annehmen, daß der Blockungsfaktor für die Superblock-Updates so gewählt werden sollte, daß er größer ist als die Quadratwurzel aus der erwarteten Gesamtanzahl der Superblock-Updates.

2.7. Sicherheit bei Systemzusammenbrüchen

Bei jedem Filesystem können sich bei Systemzusammenbrüchen Inkonsistenzen der Datenstrukturen auf dem Medium ergeben. Dabei kann man grundsätzlich zwei Ursachen unterscheiden:

- Systemausfälle z.B. infolge von Spannungszusammenbrüchen. Hier ist es sehr wahrscheinlich, daß ein Sektor, wenn der Spannungszusammenbruch während einer Schreiboperation auftritt, unvollständig geschrieben wird und daher von der Hardware nicht mehr lesbar ist.
- Betriebssystemzusammenbrüche infolge von Softwarefehlern. Bei diesen Ausfällen ist bei der Wiederinbetriebnahme des Systems damit zu rechnen, daß die Daten auf dem Medium unvollständig oder logisch inkonsistent sind.

Um die Auswirkungen von Systemzusammenbrüchen diskutieren zu können, muß zunächst betrachtet werden, welche Typen von Daten sich auf dem Medium befinden und in welchem Zusammenhang sie geschrieben werden. Dann kann man diskutieren, welche Auswirkungen auf die Konsistenz des Filesystems sich bei einem Systemzusammenbruch oder Hardwarefehler innerhalb einer Schreiboperation ergeben.

Da UNIX einen Buffer-Cache für die Filesysteme bereitstellt, ist der Zustand des Filesystems auf dem Medium nicht zu jedem Zeitpunkt identisch mit dem Zustand, der von den Anwenderprogrammen gesehen wird. Es muß also zusätzlich diskutiert werden, in welcher Reihenfolge und Priorität die einzelnen Datentypen des Filesystems zu schreiben sind, damit die Auswirkungen eines Systemzusammenbruchs so gering wie möglich sind und das Filesystem an Hand des vorgefundenen Zustandes wieder konsistent gemacht werden kann.

Auf dem *Worm*-Filesystem befinden sich folgende Datentypen:

- Der primäre Superblock
- Die Superblock-Updates
- Die Dateiinhalte
- Die Generations-Knoten

2.7.1. Fehler am primären Superblock

Der primäre Superblock wird beim Anlegen des Filesystems geschrieben. Bei einem *Worm*-Medium ist es nicht möglich, diesen Superblock nachträglich zu überschreiben. Eine Zerstörung ist nur durch Materialfehler des Mediums möglich. Wird der primäre Superblock durch solche Einflüsse unlesbar, dann können die notwendigen Daten entweder aus der Sicherheitskopie des primären Superblocks, oder durch Suchen eines Superblock-Updates (der mit Hilfe einer *Magic Number* gefunden werden könnte) rekonstruiert werden.

Bei mehrfach beschreibbaren Medien ist es prinzipiell denkbar, daß der primäre Superblock versehentlich überschrieben wird. Da in diesem Fall der Superblock physisch lesbar bleibt, benötigt man hier heuristsche Methoden oder eine Checksumme, um zu erkennen, daß die Daten im Superblock fehlerhaft sind.

2.7.2. Fehler in einem Superblock-Update

Die Daten im Superblock-Update werden immer dann ungültig, wenn das Filesystem modifiziert wird, denn alle Änderungen am Filesystem bewirken, daß sich entweder das Ende des benutzten Datenbereichs oder die Position des letzten benutzten Generations-Knotens ändern. Daher müßte man den Superblock-Update bei jeder Modifizierung des Filesystems neu schreiben.

Aus dem Zusammenhang der im Superblock vorhandenen Daten ist weiterhin ersichtlich, daß es sinvoll ist, daß der Superblock-Update erst nach dem Schreiben der Dateiinhalte und der Generations-Knoten auf das Medium geschrieben wird, denn seine Daten lassen sich aus den anderen Daten ableiten.

Um den Verbrauch von verfügbaren Sektoren so gering wir möglich zu halten, ist es bei Verwendung eines *Worm*-Mediums für das Filesystem nicht zu empfehlen, den Superblock-Update häufiger als bei jedem *unmount(8)* vorzunehmen, das einer Modifizierung des Filesystems folgte. Es ist dann aber sehr wahrscheinlich, daß es häufig vorkommt, das der aktuelle Superblock-Update fehlt. Dies ist insbesondere deshalb anzunehmen, weil SunOS 4.0 bei einem *reboot(2)* kein *unmount(2)* auf die montierten Filesysteme vornimmt. Um diesen Fall korrekt behandeln zu können, muß das Filesystem so angelegt sein, daß es möglich ist, die so verlorengegangenen Werte zu rekonstruieren.

2.7.3. Fehler an Dateiinhalten

Wenn beim Schreiben von Dateiinhalten ein Fehler auftritt, in dessen Folge ein Sektor physisch nicht mehr lesbar ist, so ist davon nur die Datei betroffen, zu deren Daten dieser Sektor gehört. Eine Inkonsistenz in der Filesystemstruktur entsteht dadurch nicht.

Werden durch einen Systemzusammenbruch einzelne Blöcke nicht geschrieben, die zu Inhalten von Dateien gehören, so sind ebenfalls nur die Dateien betroffen, zu denen diese Blöcke gehören.

Wie groß die entstehenden Folgen sind, hängt im wesentlichen davon ab, ob mehrere Dateien gleichzeitig beschrieben werden und in welcher Reihenfolge Dateiinhalte und Generations-Knoten geschrieben werden. Da bei dem Entwurf des *Worm*-Filesystems davon ausgegangen wurde, daß zu einem Zeitpunkt jeweils nur eine Datei zum Schreiben geöffnet sein kann, kann auch nur eine Datei von den beschriebenen Fehlern betroffen sein.

Wenn die Generations-Knoten nach den Dateiinhalten auf das Medium geschrieben werden, dann bleibt in diesem Fall bei einem Systemzusammenbruch die letzte Version der Datei wirksam. Die Folgen können dadurch geringer gehalten werden als bei einem vergleichbaren Fehler auf dem UNIX-Filesystem.

2.7.4. Fehler an Generations-Knoten

Wird durch einen Hardwarefehler ein Sektor zerstört, in dem sich ein Generations-Knoten befindet, dann ist die aktuelle Version der durch diesen Generations-Knoten beschriebenen Datei verloren. Der gleiche Effekt entsteht, wenn durch einen Systemzusammenbruch ein Generations-Knoten nicht mehr auf das Medium geschrieben werden konnte.

Auch hier führt die Annahme, daß zu einem Zeitpunkt jeweils nur eine Datei beschrieben werden darf, dazu, daß nur die aktuelle Version einer Datei von diesem Fehler betroffen ist und die vorherige Version der betreffenden Datei wirksam bleibt. Dazu muß nur sichergestellt werden, daß bei jedem *close(2)* auf eine gerade beschriebene Datei, spätestens jedoch beim Eröffnen der nächsten Datei oder beim *unmount(2)* auf das Filesystem, ein Update für den Generations-Knoten auf das Medium geschrieben wird.

Wenn sich der Generations-Knoten eines *Symbolischen* Links über mehrere Sektoren erstreckt, und durch einen Systemzusammenbruch während des Schreibens nur ein Teil dieses Generations-Knotens geschrieben werden konnte, dann ist ein Teil des Linknamens inkonsistent. Es gehen durch diesen Fehler keine Dateiinhalte verloren, denn der von diesem Fehler betroffene *symbolische* Link stellt nur eine Referenz auf eine bereits bestehende Datei dar.

Für die Wiederherstellung der Konsistenz des Filesystems stellt dieser Fehler dann kein Problem dar, wenn der Sektor des Generations-Knotens, der die Dateibeschreibung enthält zuerst geschrieben wird und sich Sektoren, die intakte Generations-Knoten enthalten, von anderen Sektoren unterscheiden lassen.

2.8. Erkennen von Inkonsistenzen und Methoden der Rekonstruktion

Um das *Worm*-Filesystem auf Inkonsistenzen zu untersuchen, muß man, wie in den folgenden Abschnitten noch näher erläutert wird, folgende Punkte überprüfen:

- Fehlt der aktuelle Superblock-Update?
- Sind die Daten im Superblock-Update korrekt?
- Fehlen Generations-Knoten, die bereits auf dem Medium befindliche Daten beschreiben?

Hat man eine dieser Inkonsistenzen erkannt, dann muß man versuchen, die dadurch verlorengegangenen Informationen so zu rekonstruieren, daß dadurch das Filesystem wieder in einen konsistenten Zustand überführt wird.

Damit die Wiederherstellung der Konsistenz des Filesystems möglich wird, sind folgende Dinge zu beachten, auf die ebenfalls im folgenden näher eingegangen wird:

- Bei den einzelnen Bereichen (Superblock, Dateiinhalte und Generations-Knoten) ist eine bestimmte Reihenfolge bei den Schreiboperationen auf das Medium zu beachten.
- Generations-Knoten müssen immer einzeln und in der Reihenfolge des Aufbaus der Liste geschrieben werden, damit eine klare Abgrenzung des Endes der fehlerhaften Information möglich ist.

2.8.1. Die Rekonstruktion des Superblock-Updates

Die häufigste Inkonsistenz auf dem *Worm*-Filesystem dürfte das Fehlen eines aktuellen Superblock-Updates sein. Daher ist es wichtig, daß dieser Fehler immer sicher erkannt wird, und der aktuelle Superblock rekonstruiert werden kann. Wenn ein anderer Fehler auftritt, dann sind in jedem Fall nach dessen Rekonstruktion auch Werte im Superblock-Update zu korrigieren. Daher sollte die Rekonstruktion des Superblock-Updates immer zuletzt erfolgen.

Die bei einem fehlenden Superblock-Update verlorenen und daher zu rekonstruierenden Daten sind der Ort des zuletzt geschriebenen Generations-Knotens und das obere Ende der Daten, sowie davon abgeleitete statistische Informationen.

Wird das *Worm*-Filesystem auf einem *Worm*-Medium verwendet, ist die Erkennung eines fehlenden Superblock-Updates relativ einfach:

> Bei einem *Worm*-Laufwerk lassen sich die Blöcke, die noch nicht beschrieben wurden, nicht lesen d.h. es tritt ein sogenannter *BLANK_CHECK* auf. Ein UNIX-kompatibler Geräte-Treiber liefert in diesem Fall *EIO*.
>
> Wenn der Superblock-Update korrekt erfolgt ist, dann muß sowohl der Block auf dem Medium, der den zuletzt geschriebenen Daten folgt, als auch der Block, der vor dem zuletzt geschriebenen Generations-Knoten steht, bei einem Leseversuch einen *IO*-Error liefern. Außerdem müssen der im Superblock bezeichnete letzte benutzte Datenblock und der letzte im Superblock bezeichnete Generations-Knoten lesbar sein.

Will man die Daten aus dem fehlenden Superblock-Update rekonstruieren, dann muß man nur ausgehend vom Wert des letzten korrekten Superblock-Updates den Datenbereich in aufsteigender Richtung solange durchsuchen, bis ein *IO*-Error auftritt. Gleiches gilt in umgekehrter Richtung für den Generations-Knotenbereich. Da sich das obere Ende des Datenbereichs auch indirekt über den letzten Generations-Knoten rekonstruieren läßt, kann man diesen Wert zusätzlich verifizieren.

Wird das *Worm*-Filesystem auf einem mehrfach beschreibbaren Medium verwendet, dann versagt die eben beschriebene Methode. Daher ist es sinnvoll, daß sowohl ein Superblock als auch ein Generations-Knoten durch eine *Magic Number* und eine *Checksumme* eindeutig zu identifizieren sind.

2.8.2. Die Rekonstruktion von Dateiinhalten

Werden durch einen Systemzusammenbruch einzelne Sektoren von Dateien nicht geschrieben, dann fehlt in jedem Fall auch der diese Daten beschreibende Generations-Knoten, denn er wird zeitlich nach den Dateiinhalten geschrieben. Eine Rekonstruktion in dem Sinne, daß die vor dem Systemzusammenbruch geschriebenen Daten einer Datei sinnvoll zuzuordnen sind, ist nicht möglich. Diese schon geschriebenen Daten sind verloren. Dies ergibt sich aus der Organisation des *Worm*-Filesystems, das, wegen der geforderten Verträglichkeit mit den *Worm*-Medien, jede Modifikation einer Datei an einer anderen Stelle des Mediums ablegt und die Updates für den Generations-Knoten nach den Dateiinhalten schreibt.

Befindet sich ein Filesystem, das nur einen der oben beschriebenen Fehler aufweist, auf einem *Worm*-Medium, dann stimmt nur die im letzten Superblock-Update eingetragene obere Grenze des Datenbereichs nicht. In diesem Fall ist nur nach dem neuen Ende des Datenbereichs zu suchen und der so gefundene neue Wert in den zur Rekonstruktion zu schreibenden Superblock-Update einzutragen. Die Daten, die sich in dem Bereich befinden, der zwischen dem alten und den neuen Wert des oberen Datenbereichs im Superblock liegt, sind verloren und werden durch Referenz aus dem Filesystem nicht mehr erreichbar.

Befindet sich das Filesystem hingegen auf einem mehrfach beschreibbaren Medium, dann muß keine Reparatur am Filesystem erfolgen. Die durch den letzten Superblock-Update und die Generations-Knoten beschriebenen Daten sind in diesem Fall konsistent. Denn bei einer weiteren Benutzung des Filesystems, können die kurz vor dem Systemzusammenbruch beschriebenen Sektoren bei der nächsten Filesystem-Modifikation wieder verwendet werden.

2.8.3. Die Rekonstruktion eines Generations-Knoten

Wenn ein Generations-Knoten infolge eines Systemzusammenbruchs nicht mehr korrekt auf das Medium geschrieben werden konnte, dann fehlt in jedem Fall auch der aktuelle Superblock-Update, denn er wird aus schon erwähnten Gründen immer nach den Generations-Knoten geschrieben. Es ist deshalb in jedem Fall zuerst zu testen, ob der letzte Superblock-Update korrekt ist, bevor man zur eventuellen Rekonstruktion eines Generations-Knotens schreitet.

Wenn ein Eintrag für einen Generations-Knoten nicht oder nicht korrekt auf dem Medium ist, dann ist er auch nicht mehr zu rekonstruieren. Es ist weiterhin nicht mehr zu rekonstruieren, welcher Datei die bei der Rekonstruktion aufgefundenen Daten zuzuordnen sind.

Wenn der Generations-Knoten eine *Magic Number* und eine *Checksumme* enthält, dann kann man in jedem Fall, unabhängig davon ob sich das Filesystem auf einem *Worm*-Medium oder auf einem wiederbeschreibbaren Medium befindet, einen noch nicht im letzten Superblock-Update verzeichneten Bereich von Generations-Knoten erkennen.

Wenn man auf einem wiederbeschreibbaren Medium, auf dem sich schon ein *Worm*-Filesystem befand, ein neues *Worm*-Filesystem anlegt, dann findet man allerdings bei dem Versuch einer Rekonstruktion ferhlerhafterweise die Generations-Knoten des alten Filesystems. Um das zu verhindern, müßte man den gesamten Bereich, auf dem sich möglicherweise Generations-Knoten befinden, bei der Erzeugung eines neuen *Worm*-Filesystems mit *NULL* überschreiben.

Da beim *Worm*-Filesystem das gesamte Medium Generations-Knoten tragen kann, müßte man als Folgerung aus dem letztem Absatz das gesamte Medium nullen. Dafür würde man aber eine nicht unerhebliche Zeit benötigen. Diese bei der Erzeugung eines neuen Filesystems benötigte Zeit kann man sich ersparen, wenn man im *Worm*-Filesystem sicherstellt, daß immer eine gewisse Anzahl von Sektoren hinter[13] dem Ende des gültigen Bereichs der Generations-Knoten genullt sind. Dazu muß man den Algorithmus im Filesystemcode, der für das Schreiben der Generations-Knoten verantwortlich ist, entsprechend erweitern und einen Konfigurationsparameter für die Größe des zu nullenden Bereichs im Superblock vorsehen.

Um die sich im Bereich der Generations-Knoten auf dem Medium befindlichen Sektoren, die als Erweiterung von Linknamen benötigt werden, erkennen zu können, muß man diese Sektoren ebenfalls mit einer *Magic Number,* die sich von der *Magic Number* der Generations-Knoten unterscheidet und einer *Checksumme* vesehen. Für die *Magic Number* und die *Checksumme* werden jeweils 2 Bytes am Ende eines Erweiterungs-Sektors vorgesehen.

Bei einer Sektorgröße von 512 Bytes kann es allerdings passieren, daß man bis zu 2 Sektoren für eine Linknamen-Erweiterung benötigt. Eine Beschränkung auf maximal einen Sektor als Erweiterung eines Linknamens stellt keine befriedigende Lösung dar, da in diesem Falle der Benutzer mit einer für ihn wenig verständlichen Einschränkung der Länge des Linknamens konfrontiert würde. Daher muß beim Einlesen von Linknamen, die mehr als einen Erweiterungs-Sektor benötigen, die durch die *Magic Number* und die *Checksumme* bedingte Diskontinuität der Daten beseitigt werden.

Aus Zeitgründen wurde jedoch auf die Implementierung von Linknamen, die im Generations-Knoten keinen Platz finden, verzichtet.

---

[13] d.h. unter Berücksichtigung der Wachstumsrichtung der Generations-Knoten auf kleineren Sektoradressen.

2.8.4. Generelle Vorgehensweise bei der Rekonstruktion

Um zu überprüfen, ob ein *Worm*-Filesystem konsistent ist, muß man also zuerst den letzten Superblock-Update finden. Befinden sich vor der Stelle des Mediums, die in diesem Superblock-Update als das Ende des Bereichs der Generations-Knoten vermerkt ist, keine weiteren korrekten Generations-Knoten und stimmt das sich aus diesem Generations-Knoten zu berechnende Ende des Datenbereichs mit dem Wert im Superblock-Update überein, dann ist das Filesystem, falls es auf einem wiederbeschreibbaren Medium liegt, konsistent. Bei einem Filesystem auf einem *Worm*-Medium muß zusätzlich überprüft werden, ob die hinter den im Superblock-Update bezeichneten Grenzen für den Daten und Generations-Knoten Bereich noch nicht beschrieben wurden.

Werden bei den eben beschriebenen Untersuchungen zusätzliche Generations-Knoten gefunden, dann muß der Superblock auf die sich neu ergebenden Enden für den Daten und Generations-Knoten Bereich korrigiert werden. Bei Verwendung von *Worm*-Medien müssen die so ermittelten Werte eventuell noch um bereits beschriebene und als nicht rekonstruierbar erkannte Bereiche korrigiert werden.

Da die für die Rekonstruktion eines *Worm*-Filesystems notwendigen Aktionen nur einen geringen Aufwand erforden, wurde beschlossen, den dafür notwendigen Algorithmus in den Systemaufruf *mount* des *Worm*-Filesystems zu integrieren.

Da hierbei der Systemaufruf *mount* die Funktionen eines Filesystemcheck-Programms übernimmt, muß nach dem eine Inkonsistenz im Filesystem festgestellt wurde, im Gegensatz zum bisher geäußerten, ein korrigierter Superblock-Update geschrieben werden. Damit wird vermieden, daß sich die Inkonsistenzen das Filesystems ständig vergrößern.

3. Das virtuelle Filesystem von *SunOS*

Anwenderprogramme sehen das UNIX-Filesystem als eine Abstraktionsebene über dem Magnetplattenlaufwerk, auf dem sich die Filesystemspezifischen Daten befinden. Sie erwarten, daß ihnen das Filesystem gewisse standardisierte Zugriffsmöglichkeiten bietet.

Dazu gehört in erster Linie die Möglichkeit auf einem Filesystem mehrere Dateien zu speichern und diese Dateien unabhängig voneinander und gleichzeitig lesen und beschreiben zu können. Zusätzlich erwarten sie, daß man Dateien erzeugen und löschen kann, sowie ein Inhaltsverzeichenis der vorhandenen Dateien bekommen kann.

Da UNIX ein Mehrbenutzer Betriebssystem ist, müssen die Dateien gegen ungewollten Zugriff durch andere Benutzer geschützt werden können.

Eine hierarchisch orientierte Struktur, die es ermöglicht Unterverzeichnisse anzulegen, in denen Dateien und weitere Unterverzeichnisse liegen können, erleichtert die Ordnung und den Zugriff.

3.1. Die Architektur des UNIX-Filesystems

3.1.1. Was für Geräte unterstützt UNIX ?

UNIX teilt traditionell die angeschlossenen Geräte in zwei Klassen ein:

- Der *Blockdevice*-Typ. Dabei handelt es sich generell um Magnetplattenlaufwerke. UNIX stellt für diesen Typ von Geräten einen Cache – den *Bufferpool* – zur Verfügung. Dieser *Bufferpool* wird vom UNIX-Filesystem zur Effektivitätssteigerung verwendet, in dem er dafür sorgt, daß jeder vom Filesystem benötigte Block möglichst selten gelesen bzw. geschrieben werden muß. Dieser Cache ist aber nicht speziell auf bestimmte Eigenschaften des UNIX-Filesystems abgestimmt.
- Der *Characterdevice*-Typ. Dieser Treibertyp ermöglicht den Anschluß beliebiger Geräte. Dabei handelt es sich hauptsächlich um Terminals, oder um Magnetplattenlaufwerke, die ohne Cache betrieben werden. Es lassen sich jedoch unter anderem auch Magnetbandgeräte, Laserbelichter oder Bildabtaster über diesen Treibertyp betreiben.

3.1.2. Der normale Zugriff auf Geräte unter UNIX

Bei den Geräten unterscheidet man zwischen unstrukturierten Geräten wie Terminals, und strukturierten Geräten wie Magnetplatten. Bei den Magnetplatten gibt es durch das Funktionsprinzip eine feste Blockstruktur und die Möglichkeit diese Blöcke einzeln zu adressieren. Wie schon erwähnt, verfügen Magnetplatten sowohl über einen *Blockdevice*-Treibereintrag, als auch über einen *Characterdevice*-Treibereintrag.

Unstrukturierte Geräte haben keinen *Blockdevice*-Treibereintrag. Bei ihnen erfolgt der Zugriff nur über das *Characterdevice.*

Der direkte Zugriff auf Magnetplatten mit Hilfe des *Blockdevice*-Treibers kommt in der Praxis nicht vor, denn dieser Zugriff ist im allgemeinen langsamer als der Zugriff über das *Characterdevice.* Auch der Zugriff auf Magnetplatten mit Hilfe ihres *Characterdevice*-Treibers stellt eher die Ausnahme dar, er wird eigentlich nur beim Filesystemcheck verwendet.

Üblicherweise erfolgt der Zugriff auf ein strukturiertes Gerät über das *UNIX-Filesystem.* Mit Hilfe des Filesystems werden den Anwenderprogrammen standardisierte Zugriffsmöglichkeiten auf Dateien gegeben. Dies sind z.B. die Systemaufrufe *creat(), open(), close(), read()* und *write().* Mit diesen *Systemaufrufen* ist es den Anwenderprogrammen möglich, ohne Wissen über die Eigenschaften und die Größe des Gerätes, auf dem das Filesystem liegt, zu arbeiten.

3.1.3. Architektur für den Zugriff auf ein strukturiertes Gerät

Der mögliche Datenfluß beim Zugriff auf ein strukturiertes Gerät sieht dann folgendermaßen aus:

<table>
<tr><td>Rawdevice
Zugang</td><td>Blockdevice
Zugang</td><td>Filesystem
Zugang</td></tr>
<tr><td rowspan="2"></td><td></td><td>Filesystem</td></tr>
<tr><td colspan="2">Buffer Cache</td></tr>
<tr><td colspan="3">Gerätetreiber</td></tr>
</table>

Bild 2: Der Zugriff auf strukturierte Geräte in UNIX

3.1.4. Die Schnittstelle zwischen den Anwenderprogrammen und UNIX

Für die Kommunikation mit dem Filesystem stellt UNIX den Anwenderprogrammen eine Reihe von Systemaufrufen zur Verfügung. Diese stellen den *Filesystemzugang* aus obigem Bild dar. Die Wichtigsten davon sind:

open(), creat(), close()	Für das Erzeugen, Öffnen und Schließen von Dateien.
read(), write()	Für das Schreiben und Lesen von Inhalten der Dateien.
mkdir(), rmdir()	Für das Manipulieren von Directories.
getdents(), stat()	Für das Durchsuchen des Filebaums und das Auflisten der Dateiattribute.
chown(), chmod()	Für das Manipulieren der Attribute, die für die Zugriffsrechte wichtig sind.
utimes(), truncate()	Für das Manipulieren anderer Attribute.

3.2. Möglichkeiten der Einbindung von Geräten in eine UNIX-Umgebung

3.2.1. Grenzfälle

Einen Grenzfall für die oben beschriebene grobe Geräteeinteilung stellen die Magnetbandlaufwerke dar. Bei ihnen sind außer den für die schon beschriebenen Geräte üblichen Schreib- und Leseoperationen zusätzliche Funktionen wie z.B. Umspulen, Löschen und das Schreiben von Bandmarken zu bedienen.

Für diese Zusatzfunktionen stellt das traditionelle UNIX keine standardisierten Bedienmöglichkeiten zur Verfügung. Es gibt zwar einen Systemaufruf namens *ioctl,* der ermöglicht aber nur das unspezifische Auslösen von Sonderfunktionen in Gerätetreibern des Typs *Characterdevice,* in dem er das Durchreichen von Daten vom Anwenderprogramm zum Treiber und zurück ermöglicht.

Damit allein ist aber keine treiberunabhängie Bedienung von Bandlaufwerken möglich, denn jeder Treiberprogrammierer könnte sich seine eigene Methode für die Bedienung z.B. der Umspulfunktion ausdenken, in dem er sich eigene Parameter für die Bedienung definiert.

Eine Lösung für dieses spezielle Problem fand die Universität von Berkeley in der Definition eines treiberunabhängigen *ioctl* für Magnetbandlaufwerke durch Festlegung der dafür notwendigen Parameter sowie der Einführung eines dazugehörigen Bedienprogramms.

3.2.2. Grenzen des Systems

Es gibt jedoch Geräte, die sich einer so einfachen Lösung, wie der bei den Magnetbandgeräten, widersetzen. Ein Gerät, das man nicht in die Gruppe der unstrukturierten Geräte einordnen kann, aber auch nicht vollständig über die Eigenschaften von Magnetplatten verfügt, ist das *Worm*-Plattenlaufwerk.[14]

Versucht man ein solches *Worm*-Plattenlaufwerk an einen UNIX Rechner anzuschließen, ergibt sich mit den bisher beschriebenen Mitteln keine befriedigende Lösung. Das herkömmliche UNIX-Betriebssystem kennt nur einen Typ von Filesystem, das UNIX-Filesystem. Es ist fest zwischen dem *Bufferpool* und den Filesystemspezifischen *Systemaufrufen* installiert. Dieses Filesystem ist aber nicht für ein *Worm*-Plattenlaufwerk geeignet, denn es schreibt teilweise wesentlich häufiger als einmal auf einen bestimmten Plattenblock.

[14] Ein *Worm*-Plattenlaufwerk ist ein optisches Plattenlaufwerk, das es ermöglicht jeden Block einmal zu schreiben und dann beliebig häufig zu lesen.

Damit bliebe als einzige Zugriffsmöglichkeit für *Worm*-Plattenlaufwerke das direkte Ansprechen des Gerätes über die Basisfunktionen des *Block-* oder *Characterdevice*-Treibers. Bei den Basisfunktionen, wie Lesen und Schreiben, gibt es keine Probleme. Sie lassen sich ohne Schwierigkeiten mit einem *Block-* oder *Characterdevice*-Treiber bedienen.

Wenn es jedoch darauf ankommt, standardisierte Zugriffsmethoden für *Worm*-Laufwerke anzubieten, versagt der Rahmen, den das herkömmliche UNIX-Betriebssystem bietet. Denn das UNIX-Filesystem ist nicht in der Lage, mit Geräten zu arbeiten, bei denen ein Block nicht beliebig häufig beschreibbar ist. Unter ausschließlicher Verwendung von Schreib/Leseoperationen, die direkt auf das Gerät führen, ist aber keine standardisierte, und damit sinnvolle Nutzung möglich.

Um eine sinnvolle Verwendung von *Worm*-Laufwerken zur allgemeinen Datenspeicherung zu ermöglichen, benötigt man ein Filesystem, das auf die Eigenheiten dieser Laufwerke abgestimmt ist und voll in das Betriebssystem integriert ist. Dann können die Anwenderprogramme über die ihnen bekannte *Systemaufruf*-Schnittstelle von UNIX für das Filestem arbeiten. Diese Schnittstelle stellt einen Standard für Anwenderprogramme dar.

3.2.3. Der Ausweg: das virtuelle Filesystem

Der UNIX-Bufferpool verfügt über eine definierte Schnittstelle zum Filesystemcode. Wenn es gelänge, die feste Verbindung zwischen dem UNIX-Filesystem und der *Systemaufruf*-Schnittstelle aufzubrechen, würde man damit den Anschluß weiterer Filesysteme ermöglichen. Dann hätte man die Möglichkeit das Filesystem nicht mehr als eine fest vorgeschriebene Schicht zwischen dem *Bufferpool* und der *Systemaufruf*-Schnittstelle aufzufassen, sondern wäre in der Lage, ein Filesystem als ein mögliches Modul zwischen *Bufferpool* und *Systemaufruf*-Schnittstelle zu begreifen.

In *SunOS* hat man zwischen den Filesystemcode und die *Systemaufruf*-Schnittstelle eine zusätzliche Schicht eingezogen. Diese Schicht wird *Virtuelles Filesystem* genannt, und stellt gewissermaßen einen Wahlschalter dar, mit dem man den passenden Filesystemcode unter die *Systemaufruf*-Schnittstelle setzen kann. Dadurch ist man nicht mehr an die speziellen Eigenschaften des UNIX-Filesystems gebunden.

Durch das *Virtuelle Filesystem* ist es möglich, die speziellen Eigenschaften von Geräten mit Hilfe von angepaßten Filesystemen besser zu nutzen. Es ist sogar möglich, völlig neuartige Filesysteme zu integrieren, wie z.B. das *Proc*-Filesystem, bei dem die Inhalte von Dateien durch den Datenraum von Prozessen gebildet werden, oder das *Network Filesystem (NFS),* bei dem die Filesystemspezifischen Daten mit Hilfe eines Netzwerkes von bzw. zu weiteren Rechnern transferiert werden.

3.2.4. Die neue Architektur für den Zugriff auf ein strukturiertes Gerät

Die möglichen Zugriffsweisen auf strukturierte Geräte (Plattenlaufwerke), wie sie sich nach Einführung des virtuellen Filesystems zeigen, kann man graphisch dann wie folgt darstellen:

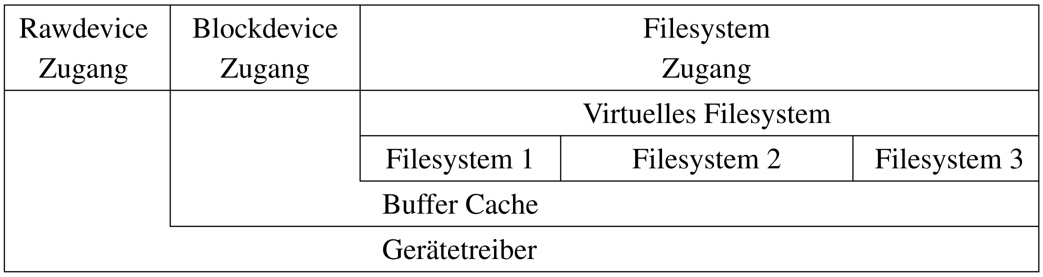

Bild 3: Die Einbindung der Filesysteme in SunOS

3.3. Überblick über die Schnittstelle des virtuellen Filesystems

3.3.1. Die *VFS*-Schnittstelle von *SunOS* 4.0.x

Da über die Schnittstelle bei früheren Versionen von *SunOS* (z.B. SunOS 3.5) keine Informationen verfügbar waren und nicht bekannt ist, ob bei der aktuellen Version (SunOS 4.1) außer den Erweiterungen inkompatible Änderungen vorgenommen wurden, können hier nur Aussagen über SunOS 4.0.*x* gemacht werden.

In *SunOS* 4.0 werden das *Berkeley 4.2*-Filesystem, im folgenden UNIX-Filesystem genannt, und das *Network Filesystem,* im Folgenden *NFS* genannt, unterstützt. Zusätzlich gibt es, bedingt durch das *Virtuelle Filesystem,* die Möglichkeit, weitere Filesysteme einzubinden. Es gibt keine Dokumentation für die Schnittstelle des *Virtuellen Filesystems;* für die Einbindung von zusätzlichen Filesystemen selbst wird aber kein Quellcode von *SunOS* benötigt, denn die einzige Verbindung von *SunOS* zum virtuellen Filesystem ist ähnlich wie bei Gerätetreibern eine Tabelle. Diese Tabelle heißt *Filesystem Switch Table.* Sie befindet sich in der Datei `/sys/sys/vfs_conf.c,` die bei jeder Binärversion des Betriebssystems mit ausgeliefert wird.

Da für das Verständnis der Einbindung des Filesystems für *Worm*-Platten die Kenntnis der Schnittstelle benötigt wird, wird im folgenden eine Beschreibung des Wissens, das sich aus Quellenstudium ergeben hat, angefügt.

Gegenüber älteren Versionen von UNIX hat *Sun,* wie schon beschrieben, zwischen den eigentlichen Filesystemcode und die Systemaufrufe des Betriebssystems eine zusätzliche Schicht eingezogen, die sich *Virtuelles Filesystem* nennt. Um dies zu ermöglichen, wurden die *inodes* in den nicht zum UNIX-Filesystem gehörigen Teilen des Betriebssystems durch *vnodes* ersetzt.

Ein *vnode* ist eine generalisierte Struktur, die im Gegensatz zum *inode* keine am physischen Filesystemaufbau orientierten Daten enthält. Sie enthält einen Zeiger auf einen *struct vnodeops,* der auf die filesystemspezifischen Zugriffsfunktionen verweist sowie einen Zeiger auf einen assoziierten *struct vfs* zur Beschreibung des gemounteten Filesystems, der unter anderem einen Zeiger auf einen *struct vfsops* enthält. Damit enthält jeder *vnode* Verweise auf alle für ihn wichtigen Zugriffsroutinen. Die Schicht des *Virtuellen Filesystems* kann durch diese Verweise über Makros die Zugriffsfunktionen aus dem *struct vnodeops* des speziellen Filesystems aufrufen.

rootvfs

vnode this vfs vfs-ops vnode-ops

next

vfs mounted here

vnode_covered

next

Bild 4 zeigt, wie man von einem *vnode* die *vfs_ops* oder die *vnode_ops* des betreffenden Filesystems erhält und wie man, falls der *vnode* zu einer Directory gehört, die einen Mountpunkt darstellt, auf das gemountete Filesystem gelangt.

3.3.2. Der Systemaufruf *mount* stellt Verbindungen her

Um nun einen bestimmten Filesystemtyp benutzen zu können, muß eine Verbindung zwischen der virtuellen Filesystemschicht und dem speziellen Filesystemcode hergestellt werden. Dies geschieht mit dem Systemaufruf *mount.*

Der Systemaufruf *mount* stellt zwei Verbindungen her:

- Die erste entsteht zwischen dem im UNIX integrierten Virtuellen Filesystem und dem speziellen Filesystemcode.
- Die zweite zwischen dem Filesystemcode und dem das spezielle Filesystem tragende Medium. Da die zweite Verbindung ausschließlich im speziellen Filesystemcode behandelt wird, sind ihr kaum Beschränkungen auferlegt. Damit wären auch Filesysteme denkbar, die mehrere Resourcen benützen, z.B. ein zusätzliches Gerät, auf dem ein Cache liegt, oder ein Filesystem, das nicht physisch auf dem selben Rechner liegt *(NFS).*

3.3.3. Der Systemaufruf *mount*

```
#include <sys/mount.h>

int mount(type, dir, M_NEWTYPE|flags, data)
     char *type;
     char *dir;
     int flags;
     caddr_t data;
```

Parameter:

type
: Der erste Parameter wird für die Identifizierung des Filesystemtyps verwendet. Er dient als Selektor für die *Filesystem Switch Table.* Mögliche Werte für den aktuellen Parameter sind hier z.B. `"4.2"` oder `"nfs"`.

dir
: Der zweite Parameter ist der Pfadname der Directory, auf die das neu zu integrierende Filesystem montiert werden soll.

flags
: Der dritte Parameter enthält bitorientierte Optionen (wie *M_RDONLY, M_NOSUID* oder ähnliche).

data
: Nur der vierte Parameter ist vom Filesystemtyp abhängig und ist ein Zeiger auf eine Struktur, die alle für das spezielle Filesystem benötigten zusätzlichen Daten enthält. Beim UNIX-Filesystem enthält diese Struktur nur den Namen des Gerätes, das das Filesystem trägt.

  In diesem Fall sieht der vierte Parameter folgendermaßen aus:

```
struct ufs_args {
        char    *fspec;
};
```

  Bei anderen Filesystemen kann diese Struktur auch wesentlich mehr Informationen enthalten.

3.3.3.1. Die Funktion des Systemaufrufs *mount*

Im Systemaufruf *mount* wird zunächst in der *Filesystem Switch Table* nach dem gewünschten Filesystemtyp gesucht. Dies geschieht, in dem der Parameter *type* des Systemaufrufs *mount* mit dem Feld *vfs_name* in der *Filesystem Switch Table* verglichen wird. Die *Filesystem Switch Table* hat folgenden Aufbau:

Aus `<sys/vfs.h>`:

```
/*
 * Filesystem type switch table
 */
struct vfssw {
        char            *vsw_name;      /* type name string */
        struct vfsops   *vsw_ops;       /* filesystem operations vector */
};

extern struct vfssw vfssw[];            /* table of filesystem types */
extern struct vfssw *vfsNVFS;           /* vfs switch table end marker */
```

In der Konfigurationsdatei `/sys/os/vfs_conf.c` sieht man ein Beispiel für eine mögliche *Filesystem Switch Table.*

```
struct vfssw vfssw[] = {
        "spec", &spec_vfsops,            /* SPEC */
        "4.2",  &ufs_vfsops,             /* UFS */
        "nfs",  &nfs_vfsops,             /* NFS */
        "pc",   &pcfs_vfsops,            /* PC */
        "lo",   &lo_vfsops,              /* LOopback */
        "rfs",  &rfs_vfsops,             /* RFS */
};

#define NVFS    (sizeof vfssw / sizeof vfssw[0])

struct vfssw *vfsNVFS = &vfssw[NVFS];
```

Wenn in der *Filesystem Switch Table* der passende Eintrag für den gewünschten Filesystemtyp gefunden wurde, dann erhält man über diese Tabelle einen Zeiger auf einen *struct vfsops* für das gewünschte Filesystem. Dieser Struct enthält die Basisoperationen, die man für den Filesystemspezifischen Teil des *mount* bzw. des *unmount* Systemaufrufes benötigt sowie die Funktionen für alle Zugriffe auf das Filesystem, die sich nicht auf eine bestimmte Datei beziehen.

Der *struct vfsops* sieht folgendermaßen aus:

```
/*
 * Operations supported on virtual file system.
 */
struct vfsops {
        int     (*vfs_mount)();         /* mount file system */
        int     (*vfs_unmount)();       /* unmount file system */
        int     (*vfs_root)();          /* get root vnode */
        int     (*vfs_statfs)();        /* get fs statistics */
        int     (*vfs_sync)();          /* flush fs buffers */
        int     (*vfs_vget)();          /* get vnode from fid */
        int     (*vfs_mountroot)();     /* mount the root filesystem */
        int     (*vfs_swapvp)();        /* return vnode for swap */
};
```

Die Funktionen aus dem *struct vfsops* werden im nachfolgenden Kapitel näher beschrieben.

Für alle gemounteten Filesysteme gibt es eine verkettete Liste von Strukturen des Typs *struct vfs*. Diese Liste beginnt bei einer Variablen namens *rootvfs,* die einen Zeiger auf den *struct vfs* für das Rootfilesystem darstellt.

Für jedes neu zu mountende Filesystem wird hier bei *vfs_next* vom Systemaufruf *mount* ein neuer Struct eingehängt. Dieser Struct wird von der Filesystemspezifischen *Mount*-routine mit den Filesystemspezifischen Daten gefüllt. Das wichtigste Feld ist dabei *vfs_op,* ein Zeiger auf den *struct vfsops* für das betreffende Filesystem. Von dem Systemaufruf *mount* ist vorher in das Feld *v_vfsmountedhere* des vnodes für die Directory, auf die das neue Filesystem montiert werden soll ein Zeiger auf den *struct vfs* für dieses Filesystem eingetragen worden.

rootvfs

vfs

vfs-
ops

vfs_root()

root-
vnode

next

vfs

next

Bild 5 zeigt, wie man den *Rootvnode* eines gemounteten Filesystems erhält.

Damit hat man nun alle notwendigen Verbindungen, um auf Dateien, die sich auf dem neuen Filesystem befinden, zuzugreifen. Der Zugriff auf bestimmte Dateien erfolgt immer über Pfadnamen. Dabei erfolgt über die Funktion *lookupname(),* die die Funktionalität der Routine *namei()* aus älteren UNIX Versionen im weiteren Sinne übernommen hat, die Zuordnung eines *vnodes* zum Dateinamen.

Ohne Einschränkung der Allgemeinheit kann man dabei davon ausgehen, daß es sich um einen sogenannten vollständigen Pfadnamen handelt, der bei der *Root* des Rechners beginnt.

Die Funktion *lookupname()* besorgt sich in diesem Fall zunächst einen *vnode* für die *Rootdirectory* des *Rootfilesystems.* Dies geschieht mit Hilfe des Funktionszeigers *vfs_root* aus dem *struct vfsops*, der sich im *struct vfs* des *rootvfs* Zeigers befindet.

In der Funktion *vfs_root* wird unter anderem auch ein Zeiger auf den *struct vnodeops* des betreffenden Filesystems in den zurückgegebenen *vnode* eingetragen. Damit ist es dann möglich die Funktion *vn_lookup()* des Filesystems mit dem ersten Pfadnamensegment aufzurufen. Der Aufruf der *vn_lookup()* Funktion wird nun solange wiederholt, bis entweder der gesamte Pfadname abgearbeitet ist, oder bis dabei ein *vnode* zurückgeliefert wird, in dessen Feld *v_vfsmountedhere* ein Wert ungleich *NULL* steht. Das ist ein Zeichen dafür, daß dort ein weiteres Filesystem gemountet ist. Nachdem dann von der Routine *lookupname()* daraus einen Zeiger auf den *struct vfsops* für das neue Filesystem gesucht worden ist, kann nun wie oben beschrieben bis zur vollständigen Abarbeitung des Pfadnamens fortgefahren werden.

Der *struct vfs* hat folgendes Aussehen:

Aus `<sys/vfs.h>`:

```
/*
 * Structure per mounted file system.
 * Each mounted file system has an array of
 * operations and an instance record.
 * The file systems are put on a singly linked list.
 * If vfs_stats is non-NULL statistics are gathered, see vfs_stat.h
 */
struct vfs {
        struct vfs      *vfs_next;           /* next vfs in vfs list */
        struct vfsops   *vfs_op;             /* operations on vfs */
        struct vnode    *vfs_vnodecovered;   /* vnode we mounted on */
        int             vfs_flag;            /* flags */
        int             vfs_bsize;           /* native block size */
        fsid_t          vfs_fsid;            /* file system id */
        caddr_t         vfs_stats;           /* filesystem statistics */
        caddr_t         vfs_data;            /* private data */
};
```

Der *struct vnode* hat folgendes Aussehen:

Aus `<sys/vnode.h>`:

```
struct vnode {
        u_short         v_flag;             /* vnode flags (see below) */
        u_short         v_count;            /* reference count */
        u_short         v_shlockc;          /* count of shared locks */
        u_short         v_exlockc;          /* count of exclusive locks */
        struct vfs      *v_vfsmountedhere;  /* ptr to vfs mounted here */
        struct vnodeops *v_op;              /* vnode operations */
        union {
                struct socket   *v_Socket;  /* unix ipc */
                struct stdata   *v_Stream;  /* stream */
                struct page     *v_Pages;   /* vnode pages list */
        } v_s;
        struct vfs      *v_vfsp;            /* ptr to vfs we are in */
        enum vtype      v_type;             /* vnode type */
        dev_t           v_rdev;             /* device (VCHR, VBLK) */
        caddr_t         v_data;             /* private data for fs */
};
```

```
/*
 * Operations on vnodes.
 */
struct vnodeops {
        int     (*vn_open)();
        int     (*vn_close)();
        int     (*vn_rdwr)();
        int     (*vn_ioctl)();
        int     (*vn_select)();
        int     (*vn_getattr)();
        int     (*vn_setattr)();
        int     (*vn_access)();
        int     (*vn_lookup)();
        int     (*vn_create)();
        int     (*vn_remove)();
        int     (*vn_link)();
        int     (*vn_rename)();
        int     (*vn_mkdir)();
        int     (*vn_rmdir)();
        int     (*vn_readdir)();
        int     (*vn_symlink)();
        int     (*vn_readlink)();
        int     (*vn_fsync)();
        int     (*vn_inactive)();
        int     (*vn_lockctl)();
        int     (*vn_fid)();
        int     (*vn_getpage)();
        int     (*vn_putpage)();
        int     (*vn_map)();
        int     (*vn_dump)();
        int     (*vn_cmp)();
        int     (*vn_realvp)();
};
```

Im *Struct vfs* sind Zeiger auf alle Basiszugriffsroutinen für *Vnodes* eines virtuellen Filesystems enthalten.

3.4. Im *SunOS* Kern benutzte Nomenklatur für Modulprefixe

anon	Anonymous pages
as	Address space
hat	Hardware Address Translation
mp	Multiprocessor/ing support
page	Physical page Management
pvn	Paged vnode
rm	Resource Management
seg	Segment Management
segdev	Segment of a mapped device
segmap	Generic vnode mapping segment
segvn	Shared or copy-on-write from a vnode/anonymous memory
segswap	Virtual swap device

3.5. Beschreibung der Filesystemoperationen

Die nun beschriebenen Funktionen beziehen sich auf spezielle Filesysteme. Jeder Funktion ist ein „*xxx*“ vorangestellt um zu verdeutlichen, daß hier ein Name für das betreffende Filesystem einzusetzen ist. So heißt z.B. die *mount* Funktion für das UNIX-Filesystem *ufs_mount,* die für das *NFS*-Filesystem *nfs_mount.*

3.5.1. xxx_mount

```
xxx_mount(vfsp, path, data)
        struct vfs *vfsp;
        char *path;
        caddr_t data;
```

Parameter:

vfsp Ein Zeiger auf eine vom Systemaufruf *mount* allozierte Struktur. Diese Struktur befindet sich in einer verketteten Liste aller gemounteten Filesysteme.

path Pfadname der Directory, auf die das neue Filesystem gemountet werden soll. Der Name befindet sich bereits im Adreßraum des Kerns.

data Ist der vierte Parameter des Systemaufrufs *mount,* und ist ein Zeiger auf eine Struktur *xxx_args.*

Returncode:

error 0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

Das neue Filesystem (d.h *vfsp*) ist während der Laufzeit von *xxx_mount()* mit *vfs_lock()* gesichert.

Diese Funktion wird vom Systemaufruf *mount* gerufen. Die Existenz von *path* sowie die notwendige Bedingung, daß *path* der Pfadname einer Directory ist, sind schon im Systemaufruf überprüft worden. Es ist sichergestellt, daß der rufende Prozess ein Superuser - Prozess ist. Der Zeiger *vfsp* zeigt auf eine Struktur, die im Systemaufruf *mount* schon mit den Elementen *vfs_next* und *vfs_op* gefüllt worden ist. Außerdem sind indirekt über *vfs_add()* die Felder *vfs_vnodecovered* und *vfs_flag* gesetzt worden. Die Struktur *xxx_args* muß mit `copyin(data ,&args, sizeof(xxx_args))` in den Kerneladreßraum geholt werden.

In die Struktur, auf die *vfsp* zeigt, müssen folgende Werte eingetragen werden: *vfs_bsize, vfs_fsid* und *vfs_data.* Dabei ist *vfs_bsize* die natürliche Blockgröße für das betreffende Filesystem. *Vfs_fsid* ist ein Filesystemidentifizierer, der wenigstens innerhalb eines Rechners eindeutig sein sollte, er besteht aus zwei longs (normalerweise major- und minor-

Devicenummer) sowie aus dem Index des Filesystems in der *Filesystem Switch Table. Vfs_data* ist normalerweise ein Zeiger auf Filesysemspezifische Daten z.B. ein *struct mount* beim UNIX-Filesystem. Alle weiteren Aktionen innerhalb von *xxx_mount()* sind abhängig von dem betreffenden Filesystemtyp (z.B. Erzeugen eines vnodes für das Blockdevice beim UNIX-Filesystem).

3.5.2. xxx_unmount

```
xxx_unmount(vfsp)
        struct vfs *vfsp;
```

Parameter:

vfsp Ein Zeiger auf eine aus dem *Root-vnode* des Filesystems abgeleitete Struktur, die beim Systemaufruf *mount* initialisiert wird.

Returncode:

error 0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

Das Filesystem (d.h *vfsp*) ist während der Laufzeit von *xxx_unmount()* mit *vfs_lock()* gesichert.

Diese Funktion wird vom Systemaufruf *unmount* gerufen. Der Systemaufruf *unmount* hat schon seinen Parameter (einen Pfadnamen) als gültigen Rootpfadnamen eines gemounteten Filesystems verifiziert. Danach wurde der Pfadname über den Root - Vnodezeiger in einen Zeiger auf einen *struct vfs* umgewandelt.

Es ist sichergestellt, daß der rufende Prozess ein Superuser - Prozess ist. Der Systemaufruf *unmount* hat vor dem Aufruf von *xxx_unmount()* ein *VFS_SYNC()* ausgeführt. Da auch ein *dnlc_purge()* ausgeführt wurde, darf in *xxx_unmount()* keine Pfadnamensuche mehr durchgeführt werden. Alle weiteren Aktionen innerhalb von *xxx_unmount()* sind abhängig von dem betreffenden Filesystemtyp (z.B. Schließen des vnodes für das Blockdevice beim UNIX-Filesystem).

3.5.3. xxx_root

```
xxx_root(vfsp, vpp)
        struct vfs *vfsp;
        struct vnode **vpp;
```

Parameter:

vfsp Ein Zeiger auf die *vfs*-Struktur für das betreffende Filesystem.

vpp Die Adresse eines Vnodezeigers, der bei Erfolg gefüllt wird.

Returncode:

error 0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

Keine.

Xxx_root() wird von *vfs_mountroot()* und *lookuppn()* aufgerufen. Weitere Aufrufe – jedoch weniger wichtig – sind in *lo_lookup()* und *vafidtovfs()*. Die Aufgabe dieser Funktion ist, mit Hilfe der privaten, unter *vfs_data* verborgenen Daten, einen Zeiger auf einen *vnode* der Rootdirectory für das betreffende Filesystem zu finden.

3.5.4. xxx_statfs

```
xxx_statfs(vfsp, sbp)
        struct vfs *vfsp;
        struct statfs *sbp;
```

Parameter:

vfsp Ein Zeiger auf eine *vfs*-Struktur für das betreffende Filesystem.

sbp Ein Zeiger auf eine *statfs*-Struktur, die im Systemaufruf *statfs* genullt wird.

Returncode:

error 0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

Keine.

Die Funktion *xxx_statfs()* liefert die Filesystemstatistiken für das betreffende Filesystem. Falls nötig werden die internen Daten so konvertiert, daß größtmögliche Ähnlichkeit mit den in der UNIX-Welt erwarteten Daten hergestellt ist. Werte, die nicht verfügbar sind, wie z.B. *f_files* und *f_free* werden auf -1 gesetzt. (Das ist z.B. beim Network-Filesystem der Fall.)

3.5.5. xxx_sync

```
xxx_sync(vfsp)
        struct vfs *vfsp;
```

Parameter:

vfsp Ein Zeiger auf eine *vfs*-Struktur für das betreffende Filesystem.

Returncode:

0 Wird nicht ausgewertet.

Randbedingungen:

Keine

Diese Funktion wird vom *unmount* und *sync*-Systemaufruf gerufen. Wenn der Parameter *vfsp* ein NULL-Pointer ist, dann wird der gesamte Pufferinhalt für alle Filesysteme des betreffenden Typs auf das Medium zurückgeschrieben, anderenfalls nur der Pufferinhalt für das ausgewählte Filesystem.

3.5.6. xxx_vget

```
xxx_vget(vfsp, vpp, fidp)
        struct vfs *vfsp;
        struct vnode **vpp;
        struct fid *fidp;
```

Parameter:

vfsp Ein Zeiger auf eine *vfs*-Struktur für das betreffende Filesystem.

vpp Die Adresse eines Vnodezeigers, der bei Erfolg gefüllt wird.

fidp Zeiger auf einen *Fileidentifier*, für den der dazugehörige *vnode* gefunden werden soll.

Returncode:

error 0, wenn diese Funktion im betreffenden Filesystem unterstützt wird; sonst *EINVAL*.

Randbedingungen:

Keine

Diese Funktion wird nur vom *nfs*-Server aufgerufen und wird benutzt, um einen *Filehandle* in einen Vnodezeiger umzuwandeln. Filesysteme die nicht exportiert werden sollen, sollten *EINVAL* zurückgeben. Wenn ein passender Vnodezeiger nicht gefunden werden kann, oder die Generationsnummer nicht paßt, wird der Adresse des Vnodezeigers ein *NULL*-Pointer zugewiesen. (Die Generationsnummer dient im UNIX-Filesystem zusammen mit der *inode*-Nummer als File -Identifizierung. Da die *inode*-Nummer inzwischen anderweitig vergeben worden sein kann dient die Generationsnummer als Sicherheit dafür, daß der *Filehandle* noch gültig ist.)

Zur Zeit gibt es diese Funktion nur im UNIX-Filesystem, alle anderen Filesysteme geben *EINVAL* zurück.

3.5.7. xxx_mountroot

```
xxx_mountroot(vfsp, vpp, name)
        struct vfs *vfsp;
        struct vnode **vpp;
        char *name;
```

Parameter:

vfsp	Ein Zeiger auf eine *vfs*-Struktur für das betreffende Filesystem.
vpp	Die Adresse eines Vnodezeigers, der bei Erfolg mit einem Vnode auf das Objekt gefüllt wird.
name	Zeiger auf einen leeren String, der *MAXOBJNAME* (127 Buchstaben) Platz bietet. In diesen String wird der Name des Objektes gefüllt, das das Rootfilesystem trägt.

Returncode:

error	0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

Keine

Diese Funktion wird von *vfs_mountroot()* aufgerufen; *Vfs_mountroot()* wird in *init_main()* aufgerufen. In *vfs_mountroot()* wird zunächst *getfstype()* aufgerufen. *Getfstype()* fragt nach dem Rootfilesystemtyp falls mit der -a Option gebootet wurde und liefert dann einen Zeiger auf einen passenden Eintrag in der *Filesystem Switch Table.* Wenn *getfstype()* einen *NULL*-Zeiger liefert, dann geht *vfs_mountroot* durch die komplette *Filesystem Switch Table* und versucht jeweils die *xxx_mountroot()* Funktion aufzurufen, bis kein Fehler auftritt; anderenfalls wird nur ein Mountversuch mit dem gewünschten Filesystem unternommen. Der in *vfs_mountroot()* allozierte *struct vfs* ist mit *VFS_INIT()* initialisiert, bevor *xxx_mountroot()* aufgerufen wird. Bei erfolgreichem Mountroot wird mit *xxx_root()* der Rootvnode des Filesystems geholt.

Xxx_mountroot() muß zunächst nach dem Rootmedium fragen , falls mit der Option `-a` gebootet wurde (`(boothowto & RB_ASKNAME) != 0`). Das kann mit Hilfe einer der Standardfunktionen aus der Datei `sun/swapgeneric.c` erfolgen die bei jeder Binärversion mit ausgeliefert wird. Wenn auf dem Filesystemtyp ein Filesystemcheck nötig sein könnte, sollte die Variable *boothowto* überprüft werden. Falls das Bit *RB_WRITABLE* nicht gesetzt ist, sollte dann das Rootfilesystem readonly gemountet werden. Die sonstigen Aktionen sind identisch mit denen von *xxx_mount()* außer, daß nach erfolgreichem Mount *vfs_add()* explizit aufgerufen werden muß.

In *xxx_montroot()* muß auch ein Aufruf von *inittodr()* erfolgen, um die Systemzeit zu initialisieren. Dabei wird als Parameter, falls das Filesystem eine Zeit trägt, eine *time_t* Variable aus dem Filesystem übergeben oder eine -1 falls nicht.

3.5.8. xxx_swapvp

```
xxx_swapvp(vfsp, vpp, name)
        struct vfs *vfsp;
        struct vnode **vpp;
        char *name;
```

Parameter:

vfsp	Ein Zeiger auf eine *vfs*-Struktur für das betreffende Filesystem.
vpp	Die Adresse eines Vnodezeigers, der bei Erfolg mit einem Vnode auf das Objekt gefüllt wird.
name	Zeiger auf einen leeren String, der *MAXOBJNAME* (127 Buchstaben) Platz bietet. In diesen String wird der Name des Objektes gefüllt, auf das geswapt wird.

Returncode:

error	0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

Keine

Diese Funktion wird von *swapconf()* aufgerufen; *Swapconf()* wird in *init_main()* direkt nach *vfs_mountroot()* aufgerufen. *Xxx_swapvp()* wird von *swapconf* nach den gleichen

Regeln aufgerufen, wie *xxx_mountroot()* von *vfs_mountroot()*. Die Regeln nach denen der Swapbereich gefunden wird stehen der Funktion *xxx_swapvp()* frei. Wenn `((boothowto & RB_ASKNAME) != 0)`, dann kann mit Hilfe einer der Standardfunktionen aus der Datei `sun/swapgeneric.c` nach einem Namen für das Swapobjekt gefragt werden.

3.6. Beschreibung der Vnodeoperationen

3.6.1. xxx_open

```
xxx_open(vpp, flag, cred)
        struct vnode **vpp;
        int flag;
        struct ucred *cred;
```

Parameter:

vpp Die Adresse des Vnodezeigers für die betreffende Datei. Er kann, falls gewünscht, ausgetauscht werden.

flag Die um eins inkrementierten Flags aus dem Systemaufruf *open*. Durch das Inkrementieren wird erreicht, daß für das *read*-Flag und das *write*-Flag getrennte Bits benutzt werden.

cred Ein Zeiger auf die *ucred*-Struktur für den betreffenden Benutzer.

Returncode:

error 0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

Keine

Typische Aufrufreihenfolge:

open() → copen() → vn_open() → xxx_open()

Normalerweise ist diese Funktion ein No-Op, denn der wichtigste Teil der Open-Prozedur des UNIX-Systemaufrufes *open()* spielt sich in *xxx_lookup()* ab. Dort wird zu einem Dateinamen der passende *Vnode* gesucht. Es sind aber Filesysteme denkbar, bei denen beim Öffnen ein neuer *Vnode* generiert wird, das ist z.B. beim *Spec*-Filesystem der Fall, wenn ein *Clone*-Device geöffnet wird. Beim *Spec*-Filesystem ist ein Nebeneffekt, daß die *Open*-Routine des betreffenden Gerätetreibers aufgerufen wird. Beim *NFS*-Filesystem wird die Gültigkeit des Caches überprüft.

3.6.2. xxx_close

```
xxx_close(vp, flag, count, cred)
        struct vnode *vp;
        int flag;
        int count;
        struct ucred *cred;
```

Parameter:

vp	Der Vnodezeiger für die betreffende Datei.
flag	Die File-Flags vom Systemaufruf *open* oder vom Systemaufruf *fcntl*. Siehe auch *xxx_open*.
count	Die Anzahl der noch bestehenden Referenzen auf die betreffende Datei.
cred	Ein Zeiger auf die *ucred*-Struktur für den betreffenden Benutzer.

Returncode:

error	0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

Keine

Typische Aufrufreihenfolge:

close() → closef() → vno_close() → vn_close() → xxx_close()

Auch diese Funktion ist normalerweise ein No-Op. Beim *Spec*-Filesystem ist der Nebeneffekt, daß beim letzten *Close()* die *Close*-Routine des betreffenden Gerätetreibers aufgerufen wird. Beim *NFS*-Filesystem wird überprüft, ob Die Datei noch einen Namen hat.[15]

3.6.3. xxx_rdwr

```
xxx_rdwr(vp, uiop, rw, ioflag, cred)
        struct vnode *vp;
        struct uio *uiop;
        enum uio_rw rw;
        int ioflag;
        struct ucred *cred;
```

[15] Das statuslose *NFS* benennt offene Dateien beim Löschen in *.nfsXXXXX* um, wobei *XXXXX* eine generierte Zahl ist. Solche Dateien müssen beim Schließen der Datei gelöscht werden.

Parameter:

vp Der Vnodezeiger für die betreffende Datei.

uiop Die *uio*-Struktur enthält Angaben über Adressen und Größen der *I/O*-Operation, so daß der Systemaufruf *read* und der Systemaufruf *readv* gleichermaßen behandelt werden können.

rw Ein Aufzählungstyp, der *UIO_READ* oder *UIO_WRITE* enthalten kann.

ioflag *IO_UNIT, IO_APPEND* und *IO_SYNC* (siehe `/usr/sys/sys/vnode.h`)

cred Ein Zeiger auf die *ucred*-Struktur für den betreffenden Benutzer.

Returncode:

error 0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

Es ist durch *vno_rw()* sichergestellt, daß sich die Datei nicht auf einem *read-only* gemounteten Filesystem befindet und geschrieben werden soll.

Typische Aufrufreihenfolge:

read/readv/write/writev() → rwuio() → vno_rw() → xxx_rdwr()

In dieser Funktion befindet sich der gesamte Code für das Lesen und Beschreiben von Dateien. Falls es sich um eine Schreiboperation handelt, die eine gewöhnliche Datei betrifft, muß zunächst überprüft werden ob *RLIMIT_FSIZE*[16] überschritten wird, und gegebenenfalls ein *SIGXFSZ*[17] an den schreibenden Prozess geschickt werden.

In dieser Funktion werden beim Schreiben Plattenblöcke alloziert. Beim Lesen erfolgt hier auch das *Buffered read ahead.*[18]

Dabei werden allerdings an dieser Stelle die vom traditionellen UNIX verwendeten Funktionen *bread()* und *breada()* sowie *bwrite()* und *bdwrite()* nicht benutzt. *SunOS* hat ein neues virtuelles Speicherkonzept, in das auch das Filesystem integriert ist. Das bedeutet, daß auch Dateien über *Page Faults* eingelesen werden. Damit das funktioniert, muß in der Funktion *xxx_rdwr()* ein Mapping eingerichtet werden. Das geschieht mit den *Segmap*-

---

[16] Ein Limit für die Größe einer einzelnen Datei, das mit dem Systemaufruf *setrlimit()* eingestellt werden kann.

[17] Ein Signal, daß anzeigt, daß das mit *setrlimit()* eingestellte Limit für die maximale Dateigröße überschritten ist.

[18] UNIX versucht seinen *Bufferpool* schon im voraus mit dem nächsten Block aus der Datei zu füllen, weil angenommen wird, daß dieser Block als nächstes gelesen wird.

Funktionen:

segmap_getmap()	→ Einrichten eines Mappings.
segmap_release()	→ Freigeben eines Mappings.
segmap_pagecreate()	→ Erzeugen von leeren Seiten für ein Mapping.

```
addr_t
segmap_getmap(seg, vp, off)
        struct seg *seg;
        struct vnode *vp;
        u_int off;
```

Parameter:

seg — Das mapping Segment – normalerweise ist das das generische Kernel mapping Segment *segkmap*.

vp — Der *Vnode*-Zeiger für die Datei.

off — Der Offset innerhalb der Datei, bei dem das Mapping starten soll.

Returncode:

Die virtuelle Adresse innerhalb des Kerneladreßraumes, auf die die gewünschten Dateiteile gemappt wurden.

```
int
segmap_release(seg, addr, flags)
        struct seg *seg;
        addr_t addr;
        u_int flags;
```

Parameter:

seg — Das mapping Segment – normalerweise ist das das generische Kernel mapping Segment *segkmap*.

addr — Ein früherer Rückgabewert von *segmap_getmap()*.

flags — Flags aus `/usr/include/vm/seg_map.h`, mögliche Werte sind: *SM_WRITE, SM_ASYNC, SM_FREE, SM_INVAL* und *SM_DONTNEEDED*. Sie steuern das Verhalten der Routine.

Returncode:

error — 0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

```
void
segmap_pagecreate(seg, addr, len, softlock)
        struct seg *seg;
        addr_t addr;
        u_int len;
        int softlock;
```

Parameter:

seg — Das mapping Segment – normalerweise ist das das generische Kernel mapping Segment *segkmap*.

addr — Ein früherer Rückgabewert von *segmap_getmap()*.

len — Die Anzahl der Bytes, die als leere Seiten erzeugt werden sollen.

softlock — Wenn hier ein Wert ungleich *NULL* steht, dann wird der *keepcount* der erzeugten Seiten nicht dekrementiert. Die Funktion *xxx_rdwr()* übergibt hier eine *NULL*.

Returncode:

Entfällt.

Das Lesen und Schreiben wird dann im Einzelnen folgendermaßen durchgeführt:

Lesen:

1) Einrichten eines Mappings mit *segmap_getmap()*.

2) Kopieren der Daten mit *uiomove()* aus dem Kerneldatenraum in den Benutzerdatenraum. Dabei wird dann gegebenenfalls ein *Page Fault* ausgelöst, der das eigentliche Lesen der Daten von dem das Filesystem tragende Medium auslöst.

3) Freigeben des Mappings mit *segmap_release()*.

Schreiben:

1) Einrichten eines Mappings mit *segmap_getmap()*. Wenn ganze Seiten geschrieben werden sollen, dann werden sie mit *segmap_pagecreate()* als leere Seiten erzeugt. Wenn Seiten nur teilweise beschrieben werden sollen, müssen sie über *Page Faults* vorher eingelesen werden.

2) Kopieren der Daten vom Benutzerdatenraum in den Kerneldatenraum. Dabei wird dann für alle noch nicht mit *segmap_pagecreate()* erzeugten Seiten ein *Page Fault* ausgelöst, der das Einlesen der alten Dateiinhalte bewirkt.

3) Freigeben des Mappings mit *segmap_release()*. Dabei wird im Allgemeinen der Inhalt der Filesystempuffer, die bei der Schreiboperation gefüllt wurden, auf das das Filesystem tragende Medium geschrieben, falls das nicht vorher schon durch ein *sync()* ausgelöst wurde.

Nach erfolgreichem Lesen oder Schreiben werden die betreffenden Zeiteinträge für die Datei korrigiert.

3.6.4. xxx_ioctl

```
xxx_ioctl(vp, com, data, flag, cred)
        struct vnode *vp;
        int com;
        caddr_t data;
        int flag;
        struct ucred *cred;
```

Parameter:

vp — Der Vnodezeiger für die betreffende Datei.

com — Das Kommando (der zweite Parameter des Systemaufrufes *ioctl).*

data — Die Adresse der eventuell zu dem *ioctl* gehörenden Daten (der dritte Parameter des Systemaufrufes *ioctl).*

flag — Die File-Flags vom Systemaufruf *open* oder vom Systemaufruf *fcntl*. Siehe auch *xxx_open.*

cred — Ein Zeiger auf die *ucred*-Struktur für den betreffenden Benutzer.

Returncode:

error — 0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

Keine

Typische Aufrufreihenfolge:

ioctl() → vno_ioctl() → xxx_ioctl()
fcntl() → fioctl() → vno_ioctl() → xxx_ioctl()

Diese Funktion wird von *vno_ioctl()* aufgerufen, wenn der Typ des *Vnodes VCHR* ist. Er sollte eigentlich nur im *Spec*-Filesystem benötigt werden, denn bei dem Auffinden von Geräten sollte die Routine *xxx_lookup()* des betreffenden Filesystems *specvp()* aufrufen und damit den gefundenen *Vnode* durch einen *Spec vnode* ersetzen. Ein Gebrauch des Systemaufrufs *ioctl()* führt dann auf die Routine *spec_ioctl()*. Alle normalen Filesysteme sollten *EINVAL* zurückgeben.

3.6.5. xxx_select

```
xxx_select(vp, which, cred)
        struct vnode *vp;
        int which;
        struct ucred *cred;
```

Parameter:

vp Der Vnodezeiger für die betreffende Datei.

which Enthält die Werte *FREAD* für Lesen , *FWRITE* für Schreiben oder 0 für Exception.

cred Ein Zeiger auf die *ucred*-Struktur für den betreffenden Benutzer.

Returncode:

error 0, wenn die Bedingung, die in *which* beschrieben ist, nicht erfüllt ist; sonst 1.

Randbedingungen:

Keine

Typische Aufrufreihenfolge:

select() → selscan() → vno_select() → xxx_select()

Diese Funktion liefert einen Rückgabewert, der angibt ob eine in *which* angegebene Bedingung erfüllt ist, d.h. ob Daten bei einem folgenden *read*-Systemaufruf zum Lesen bereitstehen, Daten bei einem folgenden *write*-Systemaufruf ohne Verzögerung zu schreiben sind, oder ob eine I/O bedingte Ausnahme aufgetreten ist. Sie wird von *vno_select()* aufgerufen, wenn der Typ des *Vnodes VCHR* oder *VFIFO* ist. Er sollte eigentlich nur im *Spec*-Filesystem benötigt werden (siehe *xxx_ioctl()*).

3.6.6. xxx_getattr

```
xxx_getattr(vp, vap, cred)
        struct vnode *vp;
        struct vattr *vap;
        struct ucred *cred;
```

Parameter:

vp Der Vnodezeiger für die betreffende Datei.

vap Ein Zeiger auf eine *vattr*-Struktur, die mit den Attribut Werten für die Datei gefüllt wird.

cred Ein Zeiger auf die *ucred*-Struktur für den betreffenden Benutzer.

Returncode:

error 0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

Keine

Typische Aufrufreihenfolge:

stat/lstat() → stat1() → vno_stat() → xxx_getattr()

Diese Funktion beschafft die Dateiattribute und wandelt sie in in eine für die Struktur *Vattr* brauchbare Form um. Dabei sind eventuelle Besonderheiten des speziellen Filesystems zu beachten, denn die Struktur *Vattr* ist auf das Betriebssystem UNIX ausgelegt.

3.6.7. xxx_setattr

```
xxx_setattr(vp, vap, cred)
        struct vnode *vp;
        struct vattr *vap;
        struct ucred *cred;
```

Parameter:

vp Der Vnodezeiger für die betreffende Datei.

vap Ein Zeiger auf eine *vattr*-Struktur, die mit den Attribut Werten für die Datei gefüllt ist. Werte ungleich -1 sind dabei signifikant.

cred Ein Zeiger auf die *ucred*-Struktur für den betreffenden Benutzer.

Returncode:

error 0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

Keine

Typische Aufrufreihenfolge:

chmod/chown/utimes/truncate() → namesetattr() → xxx_setattr()
fchmod/fchown/ftruncate() → fdsetattr() → xxx_setattr()
ftruncate() → xxx_setattr()

open() → copen() → vn_open() → xxx_setattr()

Diese Funktion nimmt die Dateiattribute aus dem *Struct vattr* und setzt danach die Attribute der Datei im Filesystem neu. Die Funktion muß selbst überprüfen, ob versucht wird Attribute zu verändern, deren Modifizierung unzulässig ist. Siehe auch *xxx_getattr()*. Die gesamte Überprüfung der Zugriffsrechte für jedes zu verändernde Attribut ist auch Sache der Funktion *xxx_setattr()*.

3.6.8. xxx_access

```
xxx_access(vp, mode, cred)
        struct vnode *vp;
        int mode;
        struct ucred *cred;
```

Parameter:

vp	Der Vnodezeiger für die betreffende Datei.
mode	Der gewünschte Zugriffsmodus.
cred	Ein Zeiger auf die *ucred*-Struktur für den betreffenden Benutzer.

Returncode:

error	0, wenn der Zugriff erlaubt ist; sonst UNIX-Fehlercode (*EROFS EACCES*).

Randbedingungen:

Keine

Typische Aufrufreihenfolge:

access() → → xxx_access()
execve() → xxx_access()
chdir/fchdir/chroot/fchroot() → chdirec() → xxx_access()
open() → copen() → vn_open() → xxx_access()
swapon() → xxx_access()

Diese Funktion muß die gesamte Zugriffsrechtsüberprüfung enthalten. Beim UNIX-Filesystem muß hier also die gewohnte Überprüfung stattfinden. Bei anderen Filesystemen muß die Überprüfung derart gestaltet werden, daß größtmögliche Übereinstimmung mit den UNIX-Regeln für Dateizugriffe sowie der Semantik des fremden Filesystems erreicht wird.

3.6.9. xxx_lookup

```
xxx_lookup(dvp, nm, vpp, cred, pnp, flags)
        struct vnode *dvp;
        char *nm;
        struct vnode **vpp;
        struct ucred *cred;
        struct pathname *pnp;
        int flags;
```

Parameter:

dvp — Der Vnodezeiger für die Directory in der nach *nm* gesucht werden soll.

nm — Ein Zeiger auf den aktuellen Komponentennamen (im Adreßraum des Kerns), nach dem in der aktuellen Directory gesucht werden soll.

vpp — Die Adresse eines Vnodezeigers, der bei Erfolg mit einem *vnode*-Zeiger für die gefundene Komponente gefüllt wird.

cred — Ein Zeiger auf die *ucred*-Struktur für den betreffenden Benutzer.

pnp — Zeiger auf einen *struct pathname,* der den Rest des noch zu Verarbeitenden Pfadnamens enthält.

flags — Die *lookup flags,* sie enthalten z.Zt. nur 0 oder *LOOKUP_DIR*. Sie sind aber für die *xxx_lookup* Funktion unwichtig.

Returncode:

error — 0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

Keine

Typische Aufrufreihenfolge:

*

lookupname() → au_lookupname() → au_lookuppn() → xxx_lookup()
lookuppn() → au_lookuppn() → xxx_lookup()

Diese Funktion hat eine zentrale Bedeutung für das gesamte virtuelle Filesystem. Alle Vnodezeiger eines Filesystems, mit Ausnahme des *Rootvnodezeigers*, werden durch Aufruf dieser Funktion gewonnen. Die Funktion sucht in der Directory, deren Vnodezeiger *dvp* ist, nach einer Datei, deren Name in *nm* steht. Bei Erfolg wird *vpp* mit einem Zeiger auf den *Vnode* für die betreffende Datei gefüllt. Der Datenraum, den der so übergebene *Struct Vnode* einnimmt, muß innerhalb von *xxx_lookup()* bereitgestellt werden. Bei der Suche nach der Datei sollten die Funktionen des *Directory name lookup*

Cache zur Beschleunigung verwendet werden.

Zuerst wird mit *dnlc_lookup()* nach einem Eintrag für *name* gesucht.

```
struct vnode *
dnlc_lookup(dp, name, cred)
        struct vnode *dp;
        register char *name;
        struct ucred *cred;
```

Parameter:

dp	Der Vnodezeiger für die Directory in der nach *name* gesucht werden soll.
name	Der Name der Datei.
cred	Ein Zeiger auf die *ucred*-Struktur für den betreffenden Benutzer.

Returncode:

Der gefundene *Vnodezeiger,* oder 0.

Wenn mit *dnlc_lookup()* kein Eintrag gefunden wurde, dann sollte die Directory im Filesystem durchsucht werden und bei Erfolg ein Eintrag im *Directory name lookup Cache* angelegt werden. Dies geschieht mit *dnlc_enter()*.

```
dnlc_enter(dp, name, vp, cred)
        register struct vnode *dp;
        register char *name;
        struct vnode *vp;
        struct ucred *cred;
```

Parameter:

dp	Der Vnodezeiger für die Directory in der sich *name* befindet.
name	Der Name der Datei.
vp	Der Vnodezeiger für die betreffende Datei.
cred	Ein Zeiger auf die *ucred*-Struktur für den betreffenden Benutzer.

Returncode:

Entfällt.

Wenn es sich bei der gefundenen Datei um ein *Blockdevice, Characterdevice* oder um ein *Fifo* handelt, dann muß der *Vnode* mit *specvp()* in einen *Vnode* des *Spec*-Filesystems umgewandelt werden.

```
     struct vnode *
     specvp(vp, dev, type)
             struct vnode *vp;
             dev_t dev;
             enum vtype type;
```

Parameter:

vp Der alte Vnodezeiger.

dev *Major*- und *Minor*-Devicenummer des gefundenen Gerätes.

type Der Typ des alten *Vnodes.*

Returncode:

Der neue *Vnodezeiger.*Er *Vnodezeiger*, stammt allerdings aus dem *Spec*-Filesystem.

Nach erfolgreicher Arbeit der Funktion *xxx_lookup()* werden die Zeiteinträge für die Datei korrigiert.

3.6.10. xxx_create

```
xxx_create(dvp, nm, vap, exclusive, mode, vpp, cred)
        struct vnode *dvp;
        char *nm;
        struct vattr *vap;
        enum vexcl exclusive;
        int mode;
        struct vnode **vpp;
        struct ucred *cred;
```

Parameter:

dvp Der Vnodezeiger für die Directory in der *nm* erzeugt werden soll.

nm Ein Zeiger auf den Komponentennamen (im Adreßraum des Kerns), der in der aktuellen Directory erzeugt werden soll.

vap Ein Zeiger auf eine *vattr*-Struktur, die mit den gewünschten Attribut Werten für die Datei gefüllt ist. Werte ungleich -1 sind dabei signifikant. Hier wird z.B. auch mitgeteilt, welchen Typ die neue Datei bekommen soll, oder daß die Datei auf die Länge 0 gekürzt werden soll.

exclusive Ein Enumerationsparameter, der angibt, ob die Datei exclusiv erzeugt werden soll d.h. ob das Erzeugen der Datei scheitern soll, wenn die Datei schon vorhanden ist.

mode — Ein Bitfeld, das aus dem gewünschten Zugriffsmodus abgeleitet wurde. Der Zugriffsmodus ergibt sich aus dem zweiten Parameter des *Open*-Systemaufrufs bzw. beträgt *VWRITE* bei Verwendung des *Creat*-Systemaufrufs. Es enthält Werte wie *VREAD, VWRITE*...

vpp — Die Adresse eines Vnodezeigers, der bei Erfolg mit einem *vnode*-Zeiger für die erzeugte Datei gefüllt wird.

cred — Ein Zeiger auf die *ucred*-Struktur für den betreffenden Benutzer.

Returncode:

error — 0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

Es ist sichergestellt, daß das Filesystem mit Schreiberlaubnis gemountet ist und daß es sich bei der Datei nicht um eine schon existierende Datei vom Typ *Socket* handelt.

Typische Aufrufreihenfolge:

creat()/open() → copen() → vn_open() → vn_vreate() → xxx_create()
mknod() → vn_vreate() → xxx_create()

Diese Funktion muß zunächst überprüfen, ob versucht wurde, eine Directory zu erzeugen und gegebenenfalls einen entsprechenden Fehlercode zurückgeben.

Wenn die gewünschte Datei schon existiert, dann sind gegebenenfalls weitere Aktionen durchzuführen:

- Wenn es sich um ein *exklusives open* handelt, dann wird der Fehlercode *EEXIST* zurückgegeben.
- Wenn Schreibzugriff gewünscht wurde und die schon existierende Datei eine Directory ist, dann wird der Fehlercode *EISDIR* zurückgegeben.
- Wenn für den gewünschten Zugriffsmodus keine Zugriffsrechte bei der schon existierenden Datei bestehen, dann wird der Zugriff verweigert.
- Wenn *mode* gleich 0 ist, (d.h. es werden keine Zugriffsrechte gewünscht) dann wird nicht überprüft, ob der Zugriff erlaubt ist. (dies ist möglich, wenn man z.B. `open("filename", O_CREAT + FOPEN, 0)` auf der Systemaufrufebene verwendet. Man erhält damit einen Filedescriptor, der keine Lese- oder Schreibberechtigung hat, aber ein *fstat()* ermöglicht.)
- Reguläre Dateien werden falls gewünscht, auf die Länge 0 gebracht (siehe Beschreibung des *vap*-Parameters).

Wenn es sich bei der gefundenen Datei um ein *Blockdevice, Characterdevice* oder um ein *Fifo* handelt, dann muß der *Vnode* mit *specvp()* in einen *Vnode* des *Spec*-Filesystems umgewandelt werden.

Nach erfolgreicher Arbeit werden die Zeiteinträge für die Datei korrigiert, und durch Aufruf von *VOP_GETATTR()* die fehlenden Einträge im *Struct vattr* gefüllt.

3.6.11. xxx_remove

```
xxx_remove(vp, nm, cred)
        struct vnode *vp;
        char *nm;
        struct ucred *cred;
```

Parameter:

vp	Der Vnodezeiger für die Directory in der die betreffende Datei steht.
nm	Ein Zeiger auf den Komponentennamen (im Adreßraum des Kerns), der in der aktuellen Directory gelöscht werden soll.
cred	Ein Zeiger auf die *ucred*-Struktur für den betreffenden Benutzer.

Returncode:

error	0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

In *SunOS* ist sichergestellt, daß es sich bei dem zu löschenden Directoryeintrag nicht um die Einträge "." oder ".." handelt, daß eine Datei mit dem angegebenen Namen existiert und daß sich die Datei nicht auf einem *read-only* gemounteten Filesystem befindet. Der Versuch die Rootdirectory eines Filesystems zu löschen wird auch vorher abgefangen.
Im speziellen Filesystemcode befinden sich jedoch ähnliche Abfragen. Daher muß davon ausgegangen werden, daß man sich darauf nicht verlassen darf.

Typische Aufrufreihenfolge:

unlink() → vn_remove() → xxx_remove()

Die Funktion *xxx_remove()* entfernt einen Dateieintrag aus einer Directory. Bei diesem Eintrag darf es sich nicht um eine Directory handeln, denn für Directories gibt es die Funktion *xxx_rmdir()*. Wenn versucht wird die Einträge "." oder ".." zu löschen, dann wird der Fehlercode *EINVAL* oder *ENOTEMPTY* zurückgeben. Wenn *dvp* keine Directory ist, wird der Fehlercode *ENOTDIR* zurückgegeben. Das Löschen des Eintrages wird untersagt, wenn für *dvp* nicht Schreib- und Executeberechtigung bestehen, oder wenn in *dvp* das *sticky*-Bit gesetzt ist und der Benutzer nicht der Eigentümer der Datei oder der Superuser ist. Wenn der Eintrag vorhanden ist, erfolgt die Entfernung des Namens für die betreffende Datei aus der Directory *dvp*. Damit ist aber nicht sichergestellt, daß der Datenraum für die betreffende Datei freigegeben werden darf, denn die Datei kann noch über weitere Namen verfügen oder von einem Prozess offengehalten werden. Das Freigeben des Datenraums für die Datei erfolgt über *xxx_inactive()*.[19]

Nach erfolgreicher Arbeit werden die Zeiteinträge für die Datei korrigiert.

3.6.12. xxx_link

```
xxx_link(vp, tdvp, tnm, cred)
        struct vnode *vp;
        struct vnode *tdvp;
        char *tnm;
        struct ucred *cred;
```

Parameter:

vp	Der Vnodezeiger für die Datei zu der ein Link angelegt werden soll.
tdvp	Der Vnodezeiger für die Ziel -Directory in der *tnm* erzeugt werden soll.
tnm	Ein Zeiger auf den Komponentennamen (im Adreßraum des Kerns), der in der Ziel -Directory erzeugt werden soll.
cred	Ein Zeiger auf die *ucred*-Struktur für den betreffenden Benutzer.

Returncode:

error	0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

Durch die Übergabe von *vp* und *tdvp* ist sichergestellt, daß die Datei zu der der Link eingerichtet werden soll sowie die Zieldirectory existieren. In *vn_link()* ist überprüft worden, daß *vp* und *tdvp* sich im selben Filesystem befinden und daß das Filesystem schreibbar gemountet ist.

Typische Aufrufreihenfolge:

link() → vn_link() → xxx_link()

---

[19] *Xxx_inactive* wird über *VN_RELE()* aufgerufen, wenn der Referenzzähler für den der Datei Assoziierten *Vnode* auf 0 gegangen ist.

Filesysteme, die es nicht ermöglichen *Hardlinks*[20] anzulegen, sollten *EINVAL* zurückgeben.

Zuerst holt sich die Funktion *xxx_link()* mit *VOP_REALVP()* den echten *Vnodezeiger* für die Datei, zu der der Link hergestellt werden soll. Für den Fall, daß es sich bei *vp* um eine Directory handelt, muß überprüft werden, ob der Benutzer Superuserprivilegien besitzt. Dann wird durch Erzeugen eines Eintrages in der Zieldirectory der Link hergestellt.

Nach erfolgreicher Arbeit werden die Zeiteinträge für die Datei korrigiert.

3.6.13. xxx_rename

```
xxx_rename(sdvp, snm, tdvp, tnm, cred)
        struct vnode *sdvp;
        char *snm;
        struct vnode *tdvp;
        char *tnm;
        struct ucred *cred;
```

Parameter:

sdvp	Der Vnodezeiger für die Directory in der der alte Dateiname steht.
snm	Ein Zeiger auf den alten Komponentennamen (im Adreßraum des Kerns).
tdvp	Der Vnodezeiger für die Ziel -Directory in der *tnm* erzeugt werden soll.
tnm	Ein Zeiger auf den Komponentennamen (im Adreßraum des Kerns), der in der Ziel -Directory erzeugt werden soll.
cred	Ein Zeiger auf die *ucred*-Struktur für den betreffenden Benutzer.

Returncode:

error	0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

Es ist sichergestellt, daß Datei selbst, die Directory in der sie sich befindet sowie die Zieldirectory existieren. In *vn_rename()* ist überprüft worden, daß *sdvp* und *tdvp* sich im selben Filesystem befinden und daß das Filesystem schreibbar gemountet ist.

[20] Das Anlegen eines *Hardlinks* auf eine Datei bedeutet dieser Datei im Filesystem einen weiteren Namen zuzuordnen. Danach kann man nicht mehr unterscheiden, welcher Name der Datei der originale Name ist.

Typische Aufrufreihenfolge:

rename() → vn_rename() → xxx_rename()

Zuerst muß überprüft werden, der Benutzer die Berechtigung hat in der Quelldirectory Dateien zu löschen, denn die *rename* Operation wird mit Ausnahme der Unteilbarkeit in *unlink(target)* ... *link(source, target)* ... *unlink(source)* umgesetzt.[21] Falls es sich bei *snm* um "." oder ".." handelt, wird *EINVAL* zurückgeben. Danach wird die eigentliche Arbeit des Umbenennens ausgeführt.

Nach erfolgreicher Arbeit werden die Zeiteinträge für die Datei korrigiert.

3.6.14. xxx_mkdir

```
xxx_mkdir(dvp, nm, vap, vpp, cred)
        struct vnode *dvp;
        char *nm;
        struct vattr *vap;
        struct vnode **vpp;
        struct ucred *cred;
```

Parameter:

dvp	Der Vnodezeiger für die Directory in der *nm* erzeugt werden soll.
nm	Ein Zeiger auf den Komponentennamen (im Adreßraum des Kerns), der in der aktuellen Directory erzeugt werden soll.
vap	Ein Zeiger auf eine *vattr*-Struktur, die mit den Attribut Werten für die Datei gefüllt ist. Werte ungleich -1 sind dabei signifikant. Hier wird z.B. auch mitgeteilt, welchen Typ die neue Datei bekommen soll, oder daß die Datei auf die Länge 0 gekürzt werden soll.
vpp	Die Adresse eines Vnodezeigers, der bei Erfolg mit einem *vnode*-Zeiger für die erzeugte Directory gefüllt wird.
cred	Ein Zeiger auf die *ucred*-Struktur für den betreffenden Benutzer.

Returncode:

error	0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

---

[21] In Filesystemen, die keine *Hardlinks* anlegen können, muß der Algorithmus entsprechend angepaßt werden.

Es ist sichergestellt, daß das Filesystem mit Schreiberlaubnis gemountet ist.

Typische Aufrufreihenfolge:

mkdir() → vn_create() → xxx_mkdir()

Diese Funktion erzeugt eine komplette Directory (mit den Einträgen "." und ".."). Die neue Directory erbt das *set-gid*-Bit[22] von der darüberliegenden Directory – unabhängig vom *Mode,* der beim Systemaufruf *mkdir()* angegeben wurde.

Nach erfolgreicher Arbeit werden die Zeiteinträge für die Datei korrigiert.

3.6.15. xxx_rmdir

```
xxx_rmdir(vp, nm, cred)
        struct vnode *vp;
        char *nm;
        struct ucred *cred;
```

Parameter:

vp — Der Vnodezeiger für die Directory in der die betreffende Directory steht.

nm — Ein Zeiger auf den Komponentennamen (im Adreßraum des Kerns), der in der aktuellen Directory gelöscht werden soll.

cred — Ein Zeiger auf die *ucred*-Struktur für den betreffenden Benutzer.

Returncode:

error — 0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

Siehe *xxx_remove().*

Typische Aufrufreihenfolge:

rmdir() → vn_remove() → xxx_rmdir()

Bemerkungen siehe *xxx_remove().* Es ist anzuraten für die Eigentliche Arbeit der Funktionen *xxx_remove()* und *xxx_rmdir()* eine gemeinsame Subroutine zu schreiben.

[22] In *SunOS* erbt eine neu erzeugte Datei die Gruppe von der Directory in der sie entsteht nach *BSD 4.2* Semantik, wenn das *set-gid*-Bit der Directory gesetzt ist. Ist das *set-gid*-Bit der Directory nicht gesetzt, dann wird die Gruppe einer neu erzeugten Datei nach *System V* Semantik durch die Gruppe des sie erzeugenden Prozesses bestimmt.

Um zu verhindern, daß ein gleichzeitiges *mount* und *rmdir* Probleme bringen, muß zusätzlich überprüft werden, daß der zu löschende Eintrag kein Mountpunkt ist.

Nach erfolgreicher Arbeit werden die Zeiteinträge für die Datei korrigiert.

3.6.16. xxx_readdir

```
xxx_readdir(vp, uiop, cred)
        struct vnode *vp;
        struct uio *uiop;
        struct ucred *cred;
```

Parameter:

vp	Der Vnodezeiger für die betreffende Directory.
uiop	Die *uio*-Struktur enthält Angaben über Adresse und Größe der *I/O*-Operation.
cred	Ein Zeiger auf die *ucred*-Struktur für den betreffenden Benutzer.

Returncode:

error	0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

In *getdents()* ist überprüft worden, daß für den Filedescriptor Leseberechtigung existiert sowie daß der rufende Prozess mindestens Platz für einen *Struct Dirent* bereitgestellt hat.

Typische Aufrufreihenfolge:

getdents() → xxx_readdir()

Wenn der Offset in der *Uio*-Struktur einen Wert enthält, der das Ende der vorhandenen Directory Einträge anzeigt, kehrt die Funktion sofort zurück. Wenn der *resid*-Zähler in der *Uio*-Struktur nicht angetastet wurde, wird dem rufenden Programm damit *EOF* signalisiert. Der Offset in der *Uio*-Struktur enthält einen Wert, der nicht zwangsläufig der wirkliche Offset in der Directory ist. Der Wert muß allerdings vom Filesystemcode so interpretierbar sein, daß eine eindeutige Funktion von *seekdir()* gewährleistet ist.

Xxx_readdir() beschafft sich einen Puffer in dem die Filesystemspezifischen Directorydaten in das Filesystemunabhängige Format umgewandelt werden können, packt dort soviele Einträge hinein, wie in den vom rufenden Prozess bereitgestellten Puffer passen und kopiert dann den Pufferinhalt mit *uiomove()* in den Benutzerdatenraum.

Zum Schluß muß noch in `uiop->uio_offset` ein Wert eingetragen werden, der so geartet ist, daß der nächste *xxx_readdir()* Aufruf bei dem Directoryeintrag fortfährt, der dem letzten in den Benutzerpuffer transferierten Eintrag folgt.

Nach erfolgreicher Arbeit werden die Zeiteinträge für die Datei korrigiert.

3.6.17. xxx_symlink

```
xxx_symlink(dvp, lnm, vap, tnm, cred)
        struct vnode *dvp;
        char *lnm;
        struct vattr *vap;
        char *tnm;
        struct ucred *cred;
```

Parameter:

dvp	Der Vnodezeiger für die Directory in der *lnm* erzeugt werden soll.
lnm	Ein Zeiger auf den Komponentennamen (im Adreßraum des Kerns), der in der aktuellen Directory erzeugt werden soll.
vap	Ein Zeiger auf eine *vattr*-Struktur, die mit den Attribut Werten für die Datei gefüllt ist. Werte ungleich -1 sind dabei signifikant. Der Systemaufruf *symlink()* hat hier nur eingetragen, daß der entstehende symbolische Link alle Zugriffsrechte für alle Benutzer bekommen soll.
tnm	Ein Zeiger auf den alten Komponentennamen (im Adreßraum des Kerns), auf den der neue symbolische Link zeigen soll.
cred	Ein Zeiger auf die *ucred*-Struktur für den betreffenden Benutzer.

Returncode:

error	0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

Es ist sichergestellt, daß die Directory, in der der Link erzeugt werden soll existiert und daß sie sich auf einem Filesystem befindet, daß schreibbar gemountet ist.

Typische Aufrufreihenfolge:

symlink() → xxx_symlink()

Filesysteme, die es nicht ermöglichen *Symbolische Links*[23] anzulegen, sollten *EINVAL*

[23] Ein *Symbolischer Link* ist ein besonderer Dateityp, der einen namentlichen Verweis auf eine andere Datei enthält. Mit Hilfe von *Symbolischen Links* ist es möglich Filesystemgrenzen zu

zurückgeben.

Xxx_symlink() füllt zunächst den Typ *VLNK* in den Struct *Vattr* und erzeugt dann den Symbolischen Link. Das geschieht in einer für das betreffende Filesystem eigenen Art und Weise.

Nach erfolgreicher Arbeit werden die Zeiteinträge für die Datei korrigiert.

3.6.18. xxx_readlink

```
xxx_readlink(vp, uiop, cred)
        struct vnode *vp;
        struct uio *uiop;
        struct ucred *cred;
```

Parameter:

vp Der Vnodezeiger für die betreffende Datei.

uiop Die *uio*-Struktur enthält Angaben über Adresse und Größe der *I/O*-Operation.

cred Ein Zeiger auf die *ucred*-Struktur für den betreffenden Benutzer.

Returncode:

error 0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

Es ist sichergestellt, daß es sich bei *vp* um einen *Symbolischen Link* handelt.

Typische Aufrufreihenfolge:

readlink() → xxx_readlink()

Filesysteme, die es nicht ermöglichen *Symbolische Links* anzulegen, sollten *EINVAL* zurückgeben.

Obwohl der Systemaufruf *readlink()* schon sichergestellt hat, daß *vp* einen Symbolischen Link verkörpert, wird in allen vorhanden Implementierungen unter *SunOS* innerhalb der Funktion nochmals überprüft, ob die Datei wirklich ein Symbolischen Link verkörpert.

Nach erfolgreichem Lesen des Links werden Zeiteinträge für die Datei korrigiert.

überspannen.

3.6.19. xxx_fsync

```
xxx_fsync(vp, cred)
        struct vnode *vp;
        struct ucred *cred;
```

Parameter:

vp Der Vnodezeiger für die betreffende Datei.

cred Ein Zeiger auf die *ucred*-Struktur für den betreffenden Benutzer.

Returncode:

error 0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

Keine

Typische Aufrufreihenfolge:

fsync() → xxx_fsync()

Diese Funktion bewirkt das Überführen eventuell vorhandener gepufferter Daten für die Datei, deren Vnode Zeiger *vp* ist, auf das das Filesystem tragende Medium.

3.6.20. xxx_inactive

```
xxx_inactive(vp, cred)
        struct vnode *vp;
        struct ucred *cred;
```

Parameter:

vp Der Vnodezeiger für die betreffende Datei.

cred Ein Zeiger auf die *ucred*-Struktur für den betreffenden Benutzer.

Returncode:

error 0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

Keine

Typische Aufrufreihenfolge:

vn_rele() → xxx_inactive()

Xxx_inactive wird von *vn_rele()* aufgerufen, wenn der Zähler für die Benutzung des *Vnodes* auf null geht.

Wenn sich der *vnode* nicht auf einem schreibbar gemounteten Filesystem befindet, werden in dieser Routine nur eventuell zusätzlich allozierte interne Datenstrukturen freigegeben werden, die diesem *vnode* assoziiert sind. Anderenfalls werden eventuell vorhandene gepufferte Daten für die Datei, deren Vnode Zeiger *vp* ist, auf das das Filesystem tragende Medium überführt. Wenn der Namenszähler für die assoziierte Datei auf *NULL* geht, dann kann der Datenraum für die Datei freigegeben werden. (Siehe auch *xxx_remove()*).

3.6.21. xxx_lockctl

```
xxx_lockctl(vp, ld, cmd, cred, clid)
        struct vnode *vp;
        struct flock *ld;
        int cmd;
        struct ucred *cred;
        int clid;
```

Parameter:

vp	Der Vnodezeiger für die betreffende Datei.
ld	Ein Zeiger auf einen *struct flock,* der die Angaben für den Aufruf enthält. Dabei sind die Felder *l_start* und *l_len* schon schon im Systemaufruf *fcntl* normiert und die Zulässigkeit des Wertebereiches überprüft.
cmd	Der zweite Parameter des Systemaufrufes *fcntl,* d.h. das Kommando aus fcntl().
cred	Ein Zeiger auf die *ucred*-Struktur für den betreffenden Benutzer.
clid	die Prozessid des rufenden Prozess.

Returncode:

error	0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

Keine

Typische Aufrufreihenfolge:

fcntl() → xxx_lockctl()

Hier befindet sich der Filesystemspezifische Teil für das *System V* kompatible *Record Locking*. Diese Funktion füllt eine Struktur vom Typ *lockhandle_t* aus `/usr/include/krpc/lockmgr.h` mit dem *Vnodezeiger* für die Datei, dem Namen des Rechners der die Datei hält (das ist der *Hostname* der Maschine) und einem Filehandle, der die Datei eindeutig beschreibt. Dieser Filehandle läßt z.B. sich durch Aufruf von *VOP_FID()* gewinnen. Danach wird der gemeinsame Code für das *Record Locking* aufgerufen. Das geschieht durch Aufruf der Funktion *klm_lockctl*.

```
klm_lockctl(lh, ld, cmd, cred, clid)
        lockhandle_t *lh;
        struct flock *ld;
        int cmd;
        struct ucred *cred;
        int clid;
```

Parameter:

lh	Ein Zeiger auf den in *xxx_lockctl* aufgebauten *Struct lockhandle_t*.
ld	Ein Zeiger auf einen *Struct flock*.
cmd	Der zweite Parameter des Systemaufrufes *fcntl,* d.h. das Kommando aus fcntl().
cred	Ein Zeiger auf die *ucred*-Struktur für den betreffenden Benutzer.
clid	die Prozessid des rufenden Prozess.

Returncode:

error	0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

3.6.22. xxx_fid

```
xxx_fid(vp, fidpp)
        struct vnode *vp;
        struct fid **fidpp;
```

Parameter:

vp	Der Vnodezeiger für die betreffende Datei.
fidpp	Die Adresse eines Zeigers auf einen *struct fid,* der bei Erfolg mit einem *Filehandle* für die Datei gefüllt wird.

Returncode:

error   0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

Keine

Typische Aufrufreihenfolge:

exportfs() → xxx_fid()
nfs_getfh() → findexivp() → xxx_fid()
makefh() → xxx_fid()

Diese Funktion wird nur in Filesystemen benötigt, die über *NFS* exportiert werden sollen.

Die Struktur *fid* wird innerhalb von *xxx_fid* alloziert. Beim UNIX-Filesystem wird der *Fileidentifier* mit der *Inode Nummer* und der *Generations Nummer* der Datei gefüllt.

3.6.23. xxx_getpage

```
xxx_getpage(vp, off, len, protp, pl, plsz, seg, addr, rw, cred)
        struct vnode *vp;
        u_int off;
        u_int len;
        u_int *protp;
        struct page *pl[];
        u_int plsz;
        struct seg *seg;
        addr_t addr;
        enum seg_rw rw;
        struct ucred *cred;
```

Parameter:

vp — Der Vnodezeiger für die betreffende Datei.

off — Der Offset in der Datei ab der gelesen werden soll. Wenn der Parameter *addr* nicht durch *Pagesize* teilbar ist, dann ist es *off* auch nicht.

len — Die Anzahl von Bytes, die ab *Offset* aus der Datei gelesen werden sollen.

protp — Ein Zeiger auf einen Integer mit Flags zur Beschreibung der Zugriffsrechte für die Seiten. Mögliche Werte sind: *PROT_READ, PROT_WRITE, PROT_EXECUTE.* Dieser Zeiger zeigt auf eine nicht initialisierte Variable. Die Flags werden bei Bedarf von *xxx_getapage()* eingetragen.

pl — Wenn *pl* ein *NULL*-Pointer ist, dann wird asynchroner I/O gefordert. Wenn *pl* kein *NULL*-Pointer ist, dann muß *xxx_getapage* die gewünschten Seiten eventuell erst erzeugen und dann in *pl* für jede allozierte Seite einen *struct Page,* der die Seite beschreibt, einfüllen. Das Ende der *pl*-Liste wird durch einen *NULL*-Pointer abgeschlossen.

plsz — Die Größe des Arrays *pl* in Bytes. Der Wert ergibt sich aus der *Anzahl der Seiten * Pagesize*. (Das Array *pl* hat für jede mögliche Seite einen Platz und einen zusätzlichen Eintrag für einen Abschließenden NULL-Pointer.)

seg — Das mapping Segment – normalerweise ist das das generische Kernel mapping Segment *segkmap*.

addr — Eine virtuelle Adresse innerhalb der ersten zu lesenden Seite.

rw — Ein Enumerationsparameter, der die gewünschte Zugriffsrechte auf die Seiten enthält.

cred — Ein Zeiger auf die *ucred*-Struktur für den betreffenden Benutzer. Kann ein NULL Pointer sein.

Returncode:

error 0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

Keine

Typische Aufrufreihenfolge:

pagefault() → as_fault() → segvn_fault()/segmap_fault() → xxx_getpage()
madvise() → as_faulta() → segvn_faulta()/segmap_faulta → xxx_getpage()
... segvn_fault() → segvn_faultpage() → anon_getpage() → xxx_getpage()
... segvn_faulta() → anon_getpage() → xxx_getpage()

Wenn *off* + *len*, um *Pagesize* oder mehr, das Ende der Datei überschreiten und *seg* nicht das generische Kernel Mappingsegment *segkmap* ist, d.h. das es sich um einen Aufruf der Funktion *xxx_getpage()* handelt, der durch den Systemaufruf *mmap()* ausgelöst wurde und dabei das Dateiende überschritten wurde, dann wird *EFAULT* zurückgegeben, denn Dateien können durch *mmap()* nicht in ihrer Größe verändert werden.

Wenn *protp* kein *NULL*-Pointer ist, dann wird zunächst *PROT_ALL* eingetragen. Später werden die Zugriffsrechte durch *xxx_getapage()* eventuell noch beschränkt.

Die Datei wird gegen gleichzeitige andere Modifikationen verriegelt. Wenn die gewünschte Anzahl von Bytes (*len*) kleiner oder gleich der *Pagesize* ist, dann wird die Routine *xxx_getapage()* direkt aufgerufen, sonst wird sie indirekt über *pvn_getpages()* gerufen:

```
if (len <= PAGESIZE)
        err = xxx_getapage(vp, off, protp, pl, plsz, seg, addr,
                rw, cred);
else
        err = pvn_getpages(xxx_getapage, vp, off, len, protp,
                pl, plsz, seg, addr, rw, cred);
```

Nach erfolgreicher Arbeit werden die Zeiteinträge für die Datei korrigiert. Die Verriegelung gegen gleichzeitige andere Modifikationen wird aufgehoben.

Der Rückgabewert der Funktion ist der Rückgabewert von *xxx_getapage()* oder der Rückgabewert von *pvn_getpages()*.

Die eigentliche Arbeit wird in der Funktion *xxx_getapage()* vorgenommen.

3.6.23.1. xxx_getapage

```
xxx_getapage(vp, off, protp, pl, plsz, seg, addr, rw, cred)
        struct vnode *vp;
        u_int off;
        u_int *protp;
        struct page *pl[];
        u_int plsz;
        struct seg *seg;
        addr_t addr;
        enum seg_rw rw;
        struct ucred *cred;
```

Parameter:

Wie bei *xxx_getpage()*, nur daß bei *xxx_getapage()* der Parameter *len* nicht vorkommt.

Returncode:

error 0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

Während der Laufzeit von *xxx_getapage()* ist die Datei gegen gleichzeitige andere Modifikationen verriegelt.

Die Routine *xxx_getapage()* wird nicht nur als Folge von Leseoperationen aufgerufen. Wenn Seiten teilweise beschrieben werden sollen, dann kann es nötig werden vorher die betreffende Seite aus der Datei zu lesen; das geschieht dann durch die Funktion *xxx_getapage()*.

Der Algorithmus dieser Funktion ist relativ kompliziert. Es wird daher versucht, ihn grob am Beispiel von *ufs_getapage()* zu beschreiben:

Der Basisalgorithmus ist:

```
Anlegen einer Seitenliste mit   pvn_kluster();
Vorbereiten des I/O mit         pageio_setup();
Aufruf des Gerätetreibers mit   (*bdevsw[major(dev)].d_strategy)(bp);
Warten auf den I/O mit          biowait();
Aufräumen mit                   pageio_done();
```

Im Einzelnen:

Zuerst wird die Position der Daten der ersten Seite auf dem Speichermedium bestimmt. Wenn es sich bei dem betreffenden Block um einen sogenannten *fake*-Block[24] handelt, dann wird, falls *protp* kein *NULL*-Pointer ist, die Schreibberechtigung für die betreffende Seite weggenommen.

Wenn die *MMU*-Seitengröße größer als die Blockgröße des Filesystems ist, muß die Seite mit zwei Leseoperationen gefüllt werden. Wenn die *MMU*-Seitengröße gleich der Blockgröße des Filesystems ist wird, falls es möglich ist, eine zweite Leseoperation als *Read ahead*[25] durchgeführt.

Mit der Funktion *page_find* wird festgestellt, ob sich die betreffende Seite schon im Hauptspeicher befindet. *Page_find* hat folgende Parameter:

```
struct page *
page_find(vp, off)
        struct vnode *vp;
        u_int off;
```

Parameter:

vp	Der Vnodezeiger für die betreffende Datei.

off	Der Offset in der Datei an dem die Seite beginnt.

Returncode:

Der Zeiger auf eine Struktur *page* für die gewünschte Seite oder *NULL*, falls die Seite sich nicht im Hauptspeicher befindet.

Randbedingungen:

Keine

Wenn die betreffende Seite nicht existiert, dann muß sie erzeugt werden. Dazu wird zunächst die wahre Blockgröße bestimmt. Das ist nötig da beim UNIX-Filesystem *Fragment*-Blöcke existieren können, die eine geringere Größe als reguläre Blöcke haben.

Wenn für die betreffende Seite noch kein entsprechender Block auf dem Medium angelegt wurde, muß sie als leere Seite erzeugt werden und in das *pl*-Array eingetragen werden. Das geschieht mit der Sequenz:

---

[24] Ein *fake*-Block ist ein Block innerhalb einer Datei, der nicht als Ergebnis einer *Schreib*-Operation, sondern als Ergebnis einer *Seek*-Operation die von einer *Schreib*-Operation gefolgt wurde, entstanden ist. Das Ergebnis ist ein „Loch“ innerhalb der Datei, das beim Lesen dem Benutzerprogramm wie ein genullter Block erscheint.

[25] Siehe auch *xxx_rdwr()*.

```
        pp = rm_allocpage(seg, addr, PAGESIZE, 1);
        if (page_enter(pp, vp, off))
                panic("ufs_getapage page_enter");
        pagezero(pp, 0, PAGESIZE);
        page_unlock(pp);
        pl[0] = pp;
        pl[1] = NULL;
```

Die Parameter der betreffenden Funktionen sind:

Zum Allozieren einer Seite:

```
        struct page *
        rm_allocpage(seg, addr, len, canwait)
                struct seg *seg;
                addr_t addr;
                u_int len;
                int canwait;
```

Parameter:

seg — Das mapping Segment – normalerweise ist das das generische Kernel mapping Segment *segkmap*.

addr — Eine virtuelle Adresse innerhalb der ersten Seite.

len — Die Anzahl von Bytes, die alloziert werden sollen.

canwait — Ein Integer, der entweder 1 oder 0 ist, jenachdem ob auf die Seiten gewartet werden kann oder nicht.

Returncode:

Der Zeiger auf eine Struktur *page*, oder *NULL*, falls sich der Wunsch nicht befriedigen läßt.

Zum Eintragen einer Seite in die Hash-Liste für *page_find()*:

```
        int
        page_enter(pp, vp, offset)
                struct page *pp;
                struct vnode *vp;
                u_int offset;
```

Parameter:

pp — Ein Zeiger auf eine Seite, die in die Liste Eingetragen werden soll, damit sie später mit *page_find* gefunden werden kann.

vp — Der Vnodezeiger für die betreffende Datei.

offset Der Offset in der Datei an dem die Seite beginnt.

Returncode:

-1, wenn die Seite sich schon in der Liste befindet, sonst 0.

Zum Beschreiben eines Teiles einer Seite mit Nullen:

```
pagezero(pp, off, len)
        struct page *pp;
        u_int off, len;
```

Parameter:

pp Ein Zeiger auf die Seite.

off Der Offset innerhalb der Seite, ab der Genullt werden soll.

len Die Anzahl der Bytes, die Genullt werden sollen.

Returncode:

Entfällt

Zum Entriegeln der in *page_enter()* verriegelten Seite:

```
void
page_unlock(pp)
        struct page *pp;
```

Parameter:

pp Ein Zeiger auf die Seite.

Returncode:

Entfällt

Wenn die betreffende Seite einen entsprechenden Block auf dem Medium hat, dann muß die Seite oder die Seiten vom Medium eingelesen werden. Dazu wird zunächst mit *pvn_kluster()* der größte zusammenhängende Block, der *addr* enthält und dem Dateioffset *off* entspricht, bestimmt.

```
      struct page *
      pvn_kluster(vp, off, seg, addr, offp, lenp, vp_off,vp_len,isra)
              struct vnode *vp;
              u_int off;
              struct seg *seg;
              addr_t addr;
              u_int *offp, *lenp;
              u_int vp_off, vp_len;
              int isra;
```

Parameter:

vp — Der Vnodezeiger für die betreffende Datei.

off — Der Offset in der Datei ab der die Seiten der Datei gelesen werden sollen.

seg — Das mapping Segment – normalerweise ist das das generische Kernel mapping Segment *segkmap*.

addr — Eine virtuelle Adresse innerhalb der ersten zu lesenden Seite.

offp — Ein Zeiger auf den Offset in der Datei, an der die von *pvn_kluster*() generierte Seitenliste anfängt. Wird von *pvn_kluster()* gefüllt.

lenp — Ein Zeiger auf die Anzahl der Bytes die entsprechend der von *pvn_kluster()* erzeugten Seitenliste gelesen werden sollen. Wird von *pvn_kluster()* gefüllt.

vp_off — Der auf die logische Filesystemblockgröße abgerundete Dateioffset.

vp_len — Die auf die logische Filesystemblockgröße aufgerundete Datenmenge.

isra — Wenn *isra* ungleich *NULL* ist, dann handelt es sich um *Read ahead*-Blöcke.

Returncode:

Ein Zeiger auf eine Liste von Seiten, oder *NULL*.

Falls die von *pvn_kluster()* generierte Seitenliste größer als *plsz* ist, dann muß ein sinnvoller Ausschnitt gewählt werden, der *off* enthält und *plsz* nicht überschreitet.

Dann wird mit *PAGE_HOLD()* der *keepcount* der Seiten inkrementiert und die Seiten in die Seitenliste *pl* übertragen.

Mit *pageio_setup()* werden dann die Seiten für den I/O bereitgestellt.

```
struct buf *
pageio_setup(pp, len, vp, flags)
        struct page *pp;
        u_int len;
        struct vnode *vp;
        int flags;
```

Parameter:

pp Ein Zeiger auf eine Seitenliste.

len Die Anzahl der Bytes in der Seitenliste.

vp Ein Vnodezeiger, der dem *Struct buf* assoziiert wird (`bp->b_vp`). Beim *UX*-Filesystem ist das ein *Vnode*-Zeiger für das *Block*-Device auf dem sich das Filesystem befindet, beim *Network File*-System ist das der *Vnode*-Zeiger der betreffenden Datei. Mit Hilfe von *vp* kann dann ein bestimmter *Struct buf* aus der Hash-Liste herausgefunden werden.

flags Flags aus `/usr/include/sys/buf.h`, mögliche Werte sind: *B_ASYNC* und *B_READ*.

Returncode:

Ein Zeiger auf einen *Struct buf*, oder *NULL*, wenn kein Speicher vorhanden ist.

In den so gewonnenen Zeiger auf den *Struct buf* muß dann noch in `bp->b_dev` die Major- und Minor- Devicenummer und in `bp->b_blkno` die Blocknummer auf dem das Filesystem tragende Gerät, sowie eventuell der Seitenoffset in `bp->b_un.b_addr` eingetragen werden.

Dann kann mit:

```
(*bdevsw[major(dev)].d_strategy)(bp);
```

die *Strategy*-Routine des Gerätetreibers aufgerufen werden.

Danach muß nur noch eventuell der *Read ahead* vorgenommen werden. Da dabei im Wesentlichen die gleichen Dinge zu beachten sind, wie beim Lesen der wirklich gewünschten Seite, wird auf die Beschreibung des *Read aheads* verzichtet.

Zum Abschluß muß noch mit `biowait(bp)` auf die Beendigung des I/O gewartet werden und mit `pageio_done(bp)` der Zeiger auf den *Struct buf* wieder freigegeben werden.

Wenn es einen Fehler gegeben hat und sich in *pl* eine Liste von Seiten befindet, dann werden die Seiten der Liste mit *PAGE_RELE()* wieder freigegeben.

3.6.24. xxx_putpage

```
xxx_putpage(vp, off, len, flags, cred)
        struct vnode *vp;
        u_int off;
        u_int len;
        int flags;
        struct ucred *cred;
```

Parameter:

vp Der Vnodezeiger für die betreffende Datei.

off Der Offset in der Datei ab der die Seiten der Datei geschrieben werden sollen.

len Die Anzahl der Bytes, die ab *off* in die Datei geschrieben werden sollen. Wenn *len* gleich *NULL* ist, dann sollen alle Seiten ab *off*, die dem *Vnode* assoziiert sind, geschrieben werden.

flags Flags aus `/usr/include/sys/buf.h`, mögliche Werte sind: *B_ASYNC, B_INVAL, B_FREE, B_DONTNEEDED, B_FORCE.*

cred Ein Zeiger auf die *ucred*-Struktur für den betreffenden Benutzer.

Returncode:

error 0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

Keine

Typische Aufrufreihenfolge:

checkpage() → xxx_putpage()
spec_sync() → xxx_putpage()
segu_swapout() → xxx_putpage()
syncip() → xxx_putpage()
segmap_release() → xxx_putpage()
segvn_swapout/sgevn_sync() → xxx_putpage()

Der Basisalgorithmus ist:

```
Erzeugen einer Liste von zu schreibenden Seiten mit
                                                pvn_vplist_dirty();
oder
                                                pvn_range_dirty();
Solange Seiten in der Liste sind:
        Umgruppieren der Seitenliste in zusammenhängende Blöcke mit
                                                        page_sub();
        und                                             page_sortadd();
        Schreiben eines zusammenhängenden Teils mit     xxx_writelbn();

Wenn ein Fehler aufgetreten ist:
        Weiterleiten des Fehlers mit            pvn_fail();
```

Im Einzelnen:

Wenn die gesamte Seitenliste ab *off* geschrieben werden soll, dann wird mit *pvn_vplist_dirty()* die Liste der zu Schreibenden Seiten gesucht. Die Parameter für *pvn_vplist_dirty()* sind:

```
     struct page *
     pvn_vplist_dirty(vp, off, flags)
             struct vnode *vp;
             u_int off;
             int flags;
```

Parameter:

vp — Der *Vnode*-Zeiger für die betreffende Datei.

off — Der Offset in der Datei, ab dem nach veränderten Seiten gesucht werden soll.

flags — Flags aus `/usr/include/sys/buf.h`, mögliche Werte sind: *B_ASYNC, B_INVAL, B_FREE, B_DONTNEEDED.*

Returncode:

Ein Zeiger auf einen *Struct page*, oder *NULL*, wenn keine veränderten Seiten vorhanden sind.

Randbedingungen:

Die Datei muß während der Laufzeit von *pvn_vplist_dirty()* gegen gleichzeitige andere Veränderungen geschützt sein.

Wenn ein Bereich zwischen *off* und *off* + *len* geschrieben werden soll, dann wird mit *pvn_range_dirty()* die Liste der zu Schreibenden Seiten gesucht. Die Parameter für *pvn_range_dirty()* sind:

```
struct page *
pvn_range_dirty(vp, off, eoff, offlo, offhi, flags)
        struct vnode *vp;
        u_int off, eoff;
        u_int offlo, offhi;
        int flags;
```

Parameter:

vp — Der *Vnode*-Zeiger für die betreffende Datei.

off — Der Offset in der Datei, ab dem nach veränderten Seiten gesucht werden soll.

eoff — Das Ende des zu durchsuchenden Bereiches.

offlo — Der auf die Klusterblockgröße abgerundete Anfagswert des zu durchsuchenden Bereiches.

offhi — Der auf die Klusterblockgröße aufgerundete Endwert des zu durchsuchenden Bereiches.
Der Bereich, der durch *offlo* und *offhi* abgedeckt wird, beschreibt den Bereich für den *klustering*[26] durchgeführt wird. Es wird aber angenommen, daß *offlo* <= *off* und *offhi* >= *eoff* ist.

flags — Flags aus `/usr/include/sys/buf.h`, mögliche Werte sind: *B_ASYNC, B_INVAL, B_FREE, B_DONTNEEDED.*

Returncode:

Ein Zeiger auf einen *Struct page*, oder *NULL*, wenn keine veränderten Seiten vorhanden sind.

Randbedingungen:

Die Datei muß während der Laufzeit von *pvn_range_dirty()* gegen gleichzeitige andere Veränderungen geschützt sein.

Aus Sicherheitsgründen wird nochmals überprüft, ob sich die Datei nicht auf einem *read-only* gemounteten Filesystem befindet.

Wenn zu schreibende Seiten für die Datei existieren und die Zeiteinträge für die Datei noch nicht korrigiert sind, dann wird es hier getan.

In der nun folgenden Schleife werden die die Seiten, die mit *pvn_vplist_dirty()* oder *pvn_range_dirty()* gefunden wurden, in aufeinanderfolgende, aneinandergrenzende

[26] Das Sammeln von Seiten, die sich innerhalb einer gewissen Umgebung der Endpunkte des Suchbereiches befinden.

Blöcke gruppiert und diese Blöcke dann mit *xxx_writelbn()* geschrieben. Das Gruppieren der Seiten geschieht, in dem die erste Seite aus der Liste der zu Schreibenden Seiten entfernt wird und dann solange weitere Seiten der Liste entnommen werden und an die erste Seite angehängt, bis eine Seite gefunden wird, die nicht an die vorgehenden paßt, oder die maximale Klustergröße überschritten ist.

Die maximale Klustergröße ist beim UNIX-Filesystem gleich der logischen Blockgröße des Filesystems. Hier ließe sich sicherlich eine Performancesteigerung erreichen, wenn man es zuließe, daß größere zusammenhängende Bereiche auf einmal geschrieben werden können.

Zum Entfernen einer Seite aus einer Liste verwendet man *page_sub():*

```
void
page_sub(ppp, pp)
        struct page **ppp;
        struct page *pp;
```

Parameter:

ppp Ein Zeiger auf einen Zeiger auf eine Liste von Seiten.

pp Ein Zeiger auf eine Seite, die aus der Liste entfernt werden soll.

Returncode:

Entfällt

Randbedingungen:

Keine

Zum nach offset sortierten Einfügen einer Seite in eine Liste verwendet man *page_sortadd():*

```
void
page_sortadd(ppp, pp)
        struct page **ppp;
        struct page *pp;
```

Parameter:

ppp Ein Zeiger auf einen Zeiger auf eine Liste von Seiten.

pp Ein Zeiger auf eine Seite, die in die Liste eingefügt werden soll.

Returncode:

Entfällt

Randbedingungen:

Keine

Wenn ein Fehler aufgetreten ist und die Liste der zu schreibenden Seiten nicht leer ist, dann werden die betreffenden Seiten mit *pvn_fail()* markiert, eventuell freigegeben und die auf den I/O wartenden Prozesse aufgeweckt.

```
void
pvn_fail(plist, flags)
        struct page *plist;
        int flags;
```

Parameter:

plist Ein Zeiger auf eine Liste von Seiten.

flags Flags aus `/usr/include/sys/buf.h`, mögliche Werte sind: *B_ASYNC, B_INVAL, B_FREE, B_DONTNEEDED.*

Returncode:

Entfällt

Randbedingungen:

Keine

Für das Schreiben der Blöcke verwendet man zeckmäßigerweise eine Routine *xxx_writelnbn()*, die die nötigen Vor- und Nachbereitungen, sowie den Transfer der Blöcke zu dem das Filesystem tragenden Medium besorgt.

3.6.24.1. xxx_writelbn

Bei den Einzelnen Filesystemen gibt es hier größere Unterschiede in den Parametern. Daher wird als Beispiel die betreffende Routine des UNIX-Filesystems beschrieben.

```
ufs_writelbn(ip, bn, pp, len, pgoff, flags)
        struct inode *ip;
        daddr_t bn;
        struct page *pp;
        u_int len;
        u_int pgoff;
        int flags;
```

Parameter:

ip Der *Inode*-Zeiger für die betreffende Datei. Aus ihm wird der *Device-vnode*-Pointer für *pageio_setup()* und die Major/Minor-Devicenummer für `bp->b_dev` abgeleitet.

bn Die Blocknummer auf dem das Filesystem tragenden Medium in Einheiten von *DEV_BSIZE*.

pp Ein Zeiger auf eine Seitenliste.

len Die Anzahl der Bytes in der Seitenliste.

pgoff Der Offset in der ersten Seite, die geschrieben werden soll. Dieser Parameter wird benötigt, wenn die zweite Hälfte einer Seite bei einem Filesystem mit einer logischen Blockgröße, die halb so groß wie die MMU-Seitengröße ist, geschrieben werden soll.

flags Flags aus `/usr/include/sys/buf.h`, mögliche Werte sind: *B_ASYNC, B_INVAL, B_FREE, B_DONTNEEDED.*

Returncode:

error 0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

Keine

Zuerst wird mit *pageio_setup()* die Seiten für den I/O bereitgestellt und ihnen ein *Struct buf* zugeordnet. Dann wird in den *Struct buf* noch die Major/Minor-Devicenummer, die Blocknummer und der Offset innerhalb der ersten Seite eingetragen.

Danach wird die *Strategy*-Routine des Gerätetreibers aufgerufen und wenn kein asynchroner I/O verlangt wurde mit *biowait()* und *pageio_done()* auf das Ende des Transfers gewartet und der *Struct buf* wieder freigegeben.

3.6.25. xxx_map

```
xxx_map(vp, off, as, addr, len, prot, maxprot, flags, cred)
        struct vnode *vp;
        u_int off;
        struct as *as;
        addr_t addr;
        u_int len;
        u_int prot;
        u_int maxprot;
        u_int flags;
        struct ucred *cred;
```

Parameter:

vp — Der Vnodezeiger für die betreffende Datei. Abgeleitet aus dem fünften Parameter des *mmap*-Systemaufrufs.

off — Der Offset innerhalb der Datei von dem aus das Mapping starten soll. – Der sechste Parameter aus dem *mmap*-Systemaufruf.

as — Die Struktur *as*, die die Adreßraumbeschreibung des aktuellen Prozesses enthält.

addr — Die gewünschte Zieladresse für das Mapping. – Der erste Parameter aus dem *mmap*-Systemaufruf.

len — Die Anzahl der Bytes für die das Mapping gewünscht wird. – Der zweite Parameter aus dem *mmap*-Systemaufruf.

prot — Die gewünschten Zugriffsrechte für das Mapping. (Lesen, Schreiben Ausführen) – Der dritte Parameter aus dem *mmap*-Systemaufruf.

maxprot — Die maximal zulässigen Zugriffsrechte für das Mapping, abgeleitet aus den Zugriffsrechten für den Filedescriptor.

flags — Eine Beschreibung der Behandlung der gemappten Seiten. *(MAP_SHARED, MAP_PRIVATE ...)* – Der vierte Parameter aus dem *mmap*-Systemaufruf.

cred — Ein Zeiger auf die *ucred*-Struktur für den betreffenden Benutzer.

Returncode:

error — 0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

Es ist sichergestellt, daß *off* und *addr* sich auf einer durch *Pagesize* teilbaren Adresse befinden.

Typische Aufrufreihenfolge:

getxfile() → xxx_map()
smmap() → xxx_map()

Die Funktion muß zuerst prüfen, ob *off* und *len* gültige Werte haben und der Filetype ein mapbares Objekt repräsentiert. Wenn das Filesystem eine komplexe Semantik hat, kann es nötig sein, hier eventuell dafür spezielle Vorkehrungen zu treffen. Bei normalen Filesystemen wird dann eine Struktur *segvn_crargs* aus `/usr/include/vm/seg_vn.h` mit den Parametern gefüllt. Das geschieht normalerweise Folgendermaßen:

```
     struct segvn_crargs vn_a;

     vn_a.vp = vp;
     vn_a.offset = off;
     vn_a.type = flags & MAP_TYPE;
     vn_a.prot = prot;
     vn_a.maxprot = maxprot;
     vn_a.cred = cred;
     vn_a.amp = NULL;
```

Dann wird ein eventuell bestehendes altes Mapping freigegeben und das neue Mapping etabliert. Das geschieht mit der Sequenz:

```
     (void) as_unmap(as, addr, len);
     return (as_map(as, addr, len, segvn_create, (caddr_t)&vn_a));
```

3.6.26. xxx_dump

```
xxx_dump(vp, addr, bn, count)
        struct vnode *vp;
        addr_t addr;
        int bn;
        int count;
```

Parameter:

vp	Der Vnodezeiger für die betreffende Datei.
addr	Die Adresse im Hauptspeicher, an der der Dump beginnen soll.
bn	Die Blocknummer (512 Bytes Blöcke) innerhalb des Gerätes bei der der Dump beginnen soll.
count	Die Anzahl der Blöcke (512 Bytes), die aus dem Ram auf das Gerät “Gedumpt” werden sollen.

Returncode:

error	0, wenn kein Fehler aufgetreten ist; sonst UNIX-Fehlercode.

Randbedingungen:

Keine

Typische Aufrufreihenfolge:

dumpsys() → xxx_dump()

Ein Filesystem auf das geswapt werden soll, sollte eine solche Funktion haben. Sie wird vom Kern nach einem *panic* aufgerufen um den realen Hauptspeicher zu sichern. Im UNIX-Filesystem gibt es eine solche Funktion nicht. Im *Spec*-Filesystem wird in dieser Funktion die *Dump*-Funktion aus der *bdevsw* aufgerufen.

3.6.27. xxx_cmp

```
xxx_cmp(vp1, vp2)
        struct vnode *vp1;
        struct vnode *vp2;
```

Parameter:

vp1 Der Vnodezeiger für die erste Datei.

vp2 Der Vnodezeiger für die zweite Datei.

Returncode:

error 0, wenn die beiden Parameter auf den gleichen *vnode* verweisen; sonst != 0.

Randbedingungen:

Filesysteme, die *Vnode*-Zeiger überdecken, müssen vorher ihre *realvp()*-Funktion auf beide *Vnode*-Zeiger aufrufen und dann mit *VOP_CMP()* die Vergleichsfunktion des realen Filesystems aufrufen.

Typische Aufrufreihenfolge:

segvn_kluster() → xxx_cmp()

Diese Funktion vergleicht zwei *Vnode*-Zeiger darauf, ob sie auf den gleichen *Vnode* verweisen. Da im virtuellen Filesystemcode nur Zeiger auf *Vnodes* verwendet werden ist es im Allgemeinen ausreichend, zu überprüfen ob die Zeiger identisch sind.

3.6.28. xxx_realvp

```
xxx_realvp(vp, vpp)
        struct vnode *vp;
        struct vnode **vpp;
```

Parameter:

vp Der Vnodezeiger für die betreffende Datei.

vpp Die Adresse eines Vnodezeigers, der bei Erfolg gefüllt wird.

Returncode:

error 0, wenn diese Funktion im aktuellen Filesystem verfügbar ist; sonst UNIX-Fehlercode *(EINVAL)*.

Randbedingungen:

Keine

Typische Aufrufreihenfolge:

lo_realvp/spec_realvp() → xxx_realvp()
ufs_link() → xxx_realvp()

Diese Funktion wird nur innerhalb von Filesystemen benötigt, die dem realen *Vnode* einen Schatten-*vnode* überlagern. Die Funktion versucht zunächst den vom betreffenden Filesystem überdeckten *vnode* zu finden. Danach wird *VOP_REALVP()* des überdeckten Filesystems aufgerufen, um gegebenenfalls aufgetretene Mehrfachüberdeckungen aufzulösen.

4. Implementierung im *SunOS*-Kern

Wie schon in der Einleitung erwähnt, ist es für eine Implementierung im SunOS 4.0 Kern notwendig, die *Filesystemswitch*-Schnittstelle und die *bdev*-Schnittstelle des Gerätetreibers einzuhalten. Die *Filesystemswitch*-Schnittstelle ist im vorigen Kapitel eingehend beschrieben worden. Der in SunOS vorhandene Gerätetreiber für *SCSI*-Plattenlaufwerke ist jedoch ohne Modifikationen nicht in der Lage, Medien mit einer anderen Sektorgröße als 512 Bytes sowie die besonderen Fehlersituationen von *Worm*-Medien verarbeiten zu können.

4.1. Notwendige Änderungen am Gerätetreiber

In SunOS 4.0 wird die aus älteren UNIX-Versionen bekannte *bdev*-Schnittstelle zum Gerätetreiber verwendet, die Kommunikation erfolgt über einen *Struct buf* mit der *Strategy*-Routine des Gerätetreibers. Auch bei SunOS 4.0 wird zur Beschreibung der gewünschten Sektoradresse auf dem Medium, auf der der Transfer beginnen soll, eine Blocknummer im *Struct buf* vermerkt, deren Wert auf einer Sektorgröße von 512 Bytes basiert.

Um Speichermedien mit einer sich von 512 Bytes unterscheidenden Sektorgröße benutzen zu können, wurde der *SCSI*-Plattentreiber so modifiziert, daß er mit Hilfe des *SCSI*-Kommandos *READ_CAPACITY* die Sektorgröße der angeschlossenen Laufwerke erfragt. Ist die aktuelle Sektorgröße eines Mediums größer als 512 Bytes, dann wird dies in einer dem Laufwerk assoziierten Datenstruktur vermerkt. Bei späteren Zugriffen auf das Laufwerk werden dann die Blockadressen aus dem *Struct buf* entsprechend umgerechnet.

4.2. Interne Repräsentation der Filesystemstruktur

Auf dem Medium ist das *Worm*-Filesystem, wie im Kapitel *Datenstrukturen auf dem Medium* beschrieben, in einer Weise abgelegt, die für die normale Verarbeitung in UNIX sehr ungünstig ist. Die Relationen, die die Baumstruktur des Filesystems beschreiben, sind dort in der für den normalen Gebrauch entgegengesetzten Weise vermerkt.[27] Daher wurde beschlossen, die Informationen, die die Baumstruktur eines aktuellen Filesystems tragen, in einem *Cache* unterzubringen.

Dieser *Cache* trägt alle wesentlichen Daten aus den Generations-Knoten und wird als Baum angelegt, der so verzeigert ist, daß ein Durchlaufen des Baums in der in UNIX gewohnten Weise möglich ist. Um diesen *Cache* für die Generations-Knoten anlegen zu können, muß innerhalb des Algorithmus, der im Filesystem beim Systemaufruf *mount(2)* durchlaufen wird, der gesamte Bereich der aktiven Generations-Knoten eingelesen werden. Dabei müssen die Daten aus den Generations-Knoten des Mediums in die interne

[27] Die Datenstrukturen auf dem Medium enthalten zur Beschreibung der Baumstruktur des Filesystems nur die Information, in welcher Directory sich eine Datei befindet. Beim normalen Gebrauch eines Filesystems ist es aber notwendig, schnell herausfinden zu können, welche Dateien sich in einer Directory befinden.

Repräsentation umgewandelt werden und die Verzeigerung entsprechend den Relationen auf dem Medium angebracht werden.

Um die Lesbarkeit der nächsten Abschnitte zu erhöhen, wird der Generations-Knoten im folgenden *Gnode* genannt.

4.3. Methoden zum Einlesen der *Gnodes*

Beim Einlesen der *Gnodes* muß beachtet werden, daß ältere Versionen eines *Gnodes* nicht mit in den *Gnode*-Cache übernommen werden.[28] Außerdem sollte, möglichst gleichzeitig mit dem Einlesen, der *Gnode*-Cache in einer effektiven Weise so verzeigert werden, daß die im Betriebssystem UNIX übliche Baumstruktur entsteht.

Es ist sehr einfach, zu verhindern, daß alte Versionen von *Gnodes* in den *Gnode*-Cache übernommen werden: Jeder *Gnode* enthält zu seiner Identifizierung eine innerhalb eines Filesystems eindeutige Nummer, die *Inodenummer*.[29] Wenn man die *Gnodes* entgegen der Reihenfolge ihres Entstehens, d.h. in Richtung steigender Sektornummern liest, dann muß nach dem Einlesen eines *Gnodes* nur überprüft werden, ob sich schon ein *Gnode* mit der Inodenummer des soeben eingelesenen im Cache befindet. Ist das der Fall, wird der soeben eingelesene *Gnode* ignoriert.

Ein größeres Problem stellt der Wunsch dar, während des Einlesens der *Gnodes* in den Cache, in diesem Cache eine Verzeigerung anzubringen, die der in UNIX üblichen Struktur eines Filesystembaums nahekommt. Wie schon erwähnt, wird der Bereich der *Gnodes* auf dem Medium, von seinem logischen Ende ausgehend, in Richtung steigender Sektornummern eingelesen, damit die aktuellen *Gnodes* der Dateien zuerst gefunden werden. Bei dieser Methode findet man allerdings in den seltensten Fällen zuerst die *root*-Directory eines Filesystems und dann die in ihr enthaltenen Dateien und Directories.

Vielmehr ist damit zu rechnen, daß man zuerst den *Gnode* einer beliebigen Datei findet. In diesem *Gnode* wird mit dem *Parent*-Zeiger auf eine Directory verwiesen, die sich im allgemeinen Fall zu diesem Zeitpunkt noch nicht im *Gnode*-Cache befindet. Man könnte in diesem Fall also keinen Zeiger von dem *Gnode* dieser Directory auf den soeben eingelesenen *Gnode* verweisen lassen, denn es gibt diesen *Gnode* der Directory noch nicht.

Man kann aber über den *Parent*-Zeiger der zuerst gefundenen Datei zu diesem Zeitpunkt schon folgende Aussagen machen:

- Der *Parent*-Zeiger verweist auf eine Directory.
- Die *Inode*-Nummer des *Parent*-Zeigers ist die *Inode*-Nummer dieser Directory.

---

[28] Das gilt zumindest solange, wie aus der Benutzerprogrammebene kein Zugriff auf ältere Versionen von Dateien implementiert ist.

[29] Der Begriff *Inodenummer* wurde aus dem UNIX-Filesystem übernommen. Da er bereits bekannt ist, wird so das Verständnis des *Worm*-Filesystems erleichtert.

Mit diesen Angaben kann man einen temporären *Gnode* für die gesuchte Directory anlegen, der einen provisorischen Namen bekommt[30] und mit einem *Flag*-Bit als temporär gekennzeichnet wird.

Damit man auf diese Weise eine Baumstruktur aufbauen kann, in der alle bisher eingelesenen oder temporär erzeugten *Gnodes* gesucht werden können, legt man zweckmäßigerweise zuerst einen weiteren temporären *Gnode* an, der im folgenden temporärer *Wurzel*-Gnode genannt wird. In diesen wird der vorher beschriebene temporäre *Gnode* der Directory so eingehängt, daß er als Directory-Inhalt des temporären *Wurzel*-Gnodes erscheint. Auf diese Weise bekommt man eine Struktur, in der sich alle temporär erzeugten Directories nebeneinander im temporären *Wurzel*-Gnode befinden.

Im dem Moment, wo man beim Einlesen der *Gnodes* vom Medium, eine Directory findet, für die man bereits einen temporären *Gnode* generiert hat, muß man den temporären *Gnode* dieser Directory durch ihren echten *Gnode* ersetzen. Hat man bis zu diesem Zeitpunkt schon einen Eintrag für die *Parent*-Directory dieser Directory gefunden, dann hängt man den *Gnode* der Directory in die Verkettung der in der *Parent*-Directory befindlichen Dateien ein, hat man sie noch nicht gefunden, muß auch hier wieder eine temporäre Directory erzeugt werden. Diesen Algorithmus verfolgt man nun solange, bis sämtliche *Gnodes* vom Medium eingelesen wurden.

Während der Zeit des Aufbaus der Baumstruktur gibt es zwei *Gnode*-Zeiger, die man sich merken muß.

- Der Zeiger auf den temporären *Wurzel*-Gnode, dessen Inhalt die temporären Directories bilden.
- Den Zeiger auf den eventuell schon vorhandenen *Gnode* der *root*-Directory des Filesystems.

Über diese beiden Zeiger lassen sich dann während des Aufbaus der Baumstruktur sämtliche bisher gefundenen oder temporär erzeugten *Gnodes* erreichen.

Es hat sich als zweckmäßig erwiesen, einen temporären *Gnode* für die *root*-Directory und die *lost+found*-Directory anzulegen, bevor man mit dem Einlesen der *Gnodes* vom Medium beginnt. Den temporären *Gnode* der *lost+found*-Directory hängt man so in den temporären *root*-Gnode, daß er als Inhalt der *root*-Directory erscheint. Der temporäre *Gnode* für die *lost+found*-Directory wird dann während der Einleseoperation als temporärer *Wurzel*-Gnode verwendet.

Im UNIX-Kern wird ein Teil der *User*-Struktur eines Prozesses als *Supervisor*-Stack benutzt. Der als *Supervisor*-Stack benutzbare Bereich der *User*-Struktur ist begrenzt und relativ klein. Daher kann man in den Algorithmen, die im UNIX-Kern verwendet werden, keine Rekursion verwenden, ohne damit rechnen zu müssen, daß der zulässige Bereich des *Supervisor*-Stacks verlassen wird.

---

[30] In der aktuellen Implementierung lautet dieser provisorische Name allgemein: *"TMP.#"*, wobei # die *Inode*-Nummer der Directory ist.

Es muß daher eine Möglichkeit geschaffen werden, die im *Gnode*-Cache aufgebaute Baumstruktur ohne Verwendung einer Rekursion durchlaufen zu können. Um dies zu erreichen, wird ein Zeiger vom Ende der Kette der *Gnodes*, die den Inhalt einer Directory darstellen, zu der diesen Inhalt tragenden Directory zurückgeführt.

Um dafür nicht in jedem *Incore-Gnode* einen zusätzlichen Zeiger vorsehen zu müssen, wird dafür der *next*-Zeiger der linearen Verkettung des Directory-Inhaltes verwendet. Dann kann allerdings das Ende dieser linearen Verkettung nicht mehr durch einen *NULL*-Zeiger markiert werden; es wird deshalb mit Hilfe eines *Flag*-Bits im *Gnode* angezeigt.

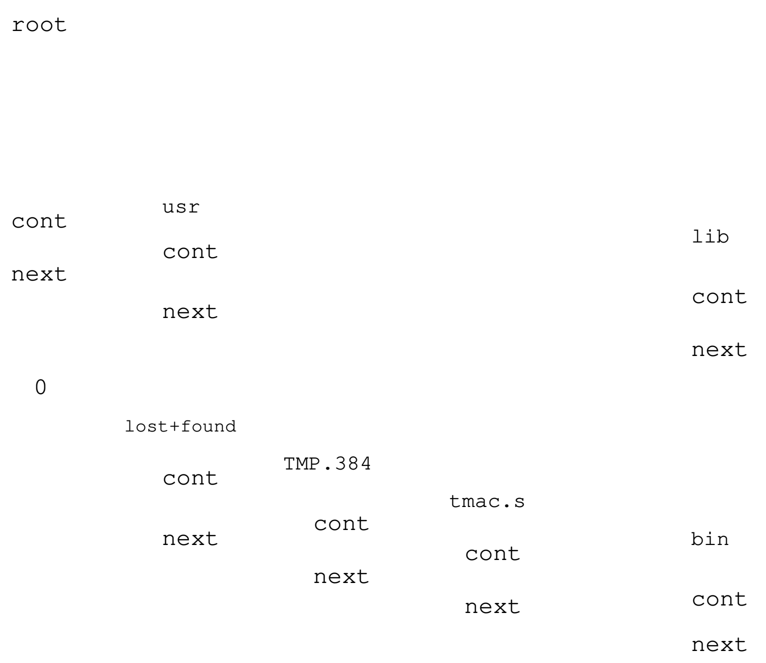

Bild 6: Die Struktur des *Gnode*-Caches während des Aufbaus der Baumstruktur.

4.3.1. Probleme durch Hardlinks

Um die bei den Datenstrukturen auf dem Medium benutzten Relationen zwischen den *Gnodes*, die einen Hardlink repräsentieren und den dazugehörigen Originaldateien auf den *Gnode*-Cache abbilden zu können, müssen zusätzliche Zeiger in der Baumstruktur des *Gnode*-Caches eingefügt werden.

Ähnlich, wie vorher bei den Directories beschrieben, können nun beim Einlesen der *Gnodes* in den Cache die *Gnodes* der Hardlinks vor den *Gnodes* der Originaldatei gefunden werden. In diesem Fall muß ein provisorischer Eintrag für die Originaldatei in der *lost+found*-Directory angelegt werden. Daher können die provisorischen Einträge in der *lost+found*-Directory nun beliebige Dateien sein.

Die *Gnodes* der Hardlinks werden wie Inhalte von Directories in der Baumstruktur an die *Gnodes* der Originaldatei angehängt. Dabei muß im *Gnode* der Originaldatei der *Linkcount* entsprechend den UNIX-Konventionen mitgezählt werden. Um vom *Gnode* eines Hardlinks zum *Gnode* der Originaldatei zu kommen, muß man die Rückwärtsverkettung der Hardlinks solange verfolgen, bis der *Gnode* der Originaldatei erreicht ist.

4.3.2. Optimierungen am Einlesealgorithmus

Da es für eine Datei, falls sie seit ihrer Erzeugung modifiziert wurde, mehrere *Gnodes* auf dem Medium geben kann, ist es während der Zeit des Einlesens notwendig zu wissen, ob ein *Gnode* für eine bestimmte Datei schon gefunden wurde. Da beim Einlesen der *Gnodes* vom Medium in Richtung steigender Sektornummern der aktuelle *Gnode*-Eintrag für eine Datei zuerst gefunden wird, sind alle weiteren *Gnodes* mit der gleichen Inodenummer zu ignorieren.

Um dafür nicht immer die gesamte Baumstruktur der schon eingelesenen *Gnodes* durchsuchen zu müssen, hat es sich als zweckmäßig erwiesen, eine Bitmap, in der sämtliche bereits gefundenen Inodenummern vermerkt sind, anzulegen. Eine weitere Verkürzung der Suchzeiten erreicht man, wenn man eine zweite Bitmap einführt, in der die Inodenummern der Temporär erzeugten *Gnodes* vermerkt sind. Dadurch sieht man, welchen Teil der Baumstruktur man gegebenenfalls zu durchsuchen hat.

Immer, wenn während des Einlesens der *Gnodes* ein neuer *Gnode* gefunden wird, muß in der bisher aufgebauten Baumstruktur nach der *Parent*-Directory und gegebenenfalls nach der Originaldatei eines Hardlinks gesucht werden, um den neuen *Gnode* an der richtigen Position in der Baumstruktur einhängen zu können. Wenn die Dateien sehr zufällig auf dem Filesystem angelegt wurden, dann wird sehr viel Rechenzeit beim Suchen in der Baumstruktur benötigt. Sind die Dateien und Directories völlig zufällig im Bereich der *Gnodes* auf dem Medium verteilt und befinden sich n aktive *Gnodes* auf dem Filesystem, dann sind

$$v = n * (n + 1) / 4$$

Vergleiche notwendig, um alle während des Einlesevorgangs gesuchten *Gnodes* in der Baumstruktur zu finden.

Die Anzahl der zum Aufbau der Baumstruktur im *Gnode*-Cache notwendigen Vergleiche läßt sich jedoch durch Einführung eines weiteren Caches, der eine Relation zwischen *Inodenummern* und *Gnode*-Zeigern herstellt, verringern. Dieser Cache wird im folgenden, zur besseren Unterscheidung vom *Gnode*-Cache, *Inode*-Cache genannt. Ein Element dieses Caches enthält jeweils die *Inodenummer* und die Speicher-Adresse des dazugehörigen *Gnodes* im Kern,[31] es werden also 8 Bytes pro Eintrag benötigt.

Die Größe dieses Caches richtet sich nach der Anzahl der Directories und der Hardlinks auf dem Medium, denn nur nach Directories und den Originaldateien von Hardlinks wird beim Aufbau des *Gnode*-Caches gesucht. Wenn man den *Inode*-Cache so groß macht, daß alle Directories und Originaldateien von Hardlinks darin Platz finden, dann muß man nicht mehr in der Baumstruktur der *Gnodes* suchen. Da kein Hashalgorithmus eine optimale Abbildung des *Inodenummern* der Directories und Originaldateien von Hardlinks auf einen kleinen Wertebereich erreichen kann, müßte dazu der *Inode*-Cache soviele Einträge haben, wie Inodes auf dem Medium vorhanden sind. Da man im allgemeinen den dafür notwendigen Speicherplatz nicht zur Verfügung stellen kann, muß man einen akzeptablen Kompromiß zwischen Leistung und Größe des *Inode*-Caches wählen.

Gegen einen zu großen *Inode*-Cache sprechen folgende Punkte:

- Ein großer *Inode*-Caches hat einen hohen Speicherbedarf.
- Ein sehr großer *Inode*-Cache wird nur zu einem geringen Anteil mit den benötigten Einträgen gefüllt, daher bleibt ein hoher Anteil des möglichen Raums im *Inode*-Cache ohne Nutzen.
- Je mehr "Hits" der Cache liefert, desto größer ist die mittlere Anzahl der Vergleiche beim Suchen in der Baumstruktur in Folge eines Cache "Miss", daher bringt eine beliebige Vergrößerung des *Inode*-Caches nicht die erhoffte Effektivitätssteigerung.

Daher wurden 509 Einträge für den *Inode*-Cache vorgesehen. Mit diesem Wert ergab sich ein guter Kompromiß zwischen Speicherbedarf und Wirksamkeit. Als Abbildungsfunktion für den *Inode*-Cache wurde der *modulo*-Oparator gewählt, da er, bei Verwendung einer Primzahl für die Größe des *Inode*-Caches, eine gute Verteilung der Einträge im *Inode*-Cache bewirkt.[32] Mit diesen Parametern stieg die Zeit, die für den Aufbau der Baumstruktur des *Gnode*-Caches, bei zufälliger Verteilung der *Gnodes* auf dem Medium, benötigt wurde, um weniger als 50% über die Zeit, die bei optimaler Verteilung der *Gnodes* benötigt wurde.

[31] Wie im Abschnitt *Pagebarer Speicher für den Gnode-Cache* noch gezeigt wird, müssen die Incore-Gnodes in zwei Bereiche geteilt werden. Die *Inodenummer* befindet sich in dem Teil des Incore-Gnodes, der nur zeitaufwendig zu erreichen ist. Daher ist es zur Erhaltung der Effizienz des *Inode*-Caches notwendig, den Zeiger auf den *Gnode* und die *Inodenummer* im *Inode*-Cache unterzubringen.

[32] Es ist eventuell zu prüfen, ob der bei einem bestimmten Rechner anfallende Rechenaufwand für die Anwendung des *modulo*-Operators nicht den Vorteil der besseren Verteilung der Einträge im *Inode*-Cache aufhebt. Ist das der Fall, dann sollte auf diesem Rechner anstelle der Primzahl eine Zweierpotenz und als Abbildungsfunktion eine Maskierung der oberen Bits der Inodenummer gewählt werden.

4.3.3. *lost+found*-Dateien

Unter normalen Bedingungen ist es im *Worm*-Filesystem nicht wie beim UNIX-Filesystem möglich, daß Dateien entstehen, die nicht über den Filesystembaum zu erreichen sind, aber noch nicht gelöscht sind. Es ist jedoch denkbar, daß der *Gnode* einer Directory zerstört wird, d.h. nicht mehr korrekt lesbar ist. Wenn dann kein weiterer *Gnode* für diese Directory auf dem Medium vorhanden ist, kann man auf alle Dateien und Directories, die sich im Filesystembaum unterhalb der zerstörten Directory befanden, nicht mehr zugreifen.

Durch die oben beschriebene Methode zum Einlesen der *Gnodes* und zum Aufbau des vorwärts verketteten Filesystembaums wird für jede Directory, deren *Gnode* noch nicht gefunden wurde, ein temporärer *Gnode* angelegt und in die *lost+found*-Directory eingehängt. Daher sind solche „verlorenen“ Dateien, bei Anwendung der oben beschriebenen Methode zum Einlesen der *Gnodes,* nach dem das Filesystem montiert wurde in der *lost+found*-Directory unterhalb einer temporären Directory zu finden. Der Name dieser temporären Directory ist, wie bereits beschrieben, aus der Inodenummer der zerstörten *Parent*-Directory abgeleitet.

Wenn man den urpsrünglichen Zustand des Filesystems wiederherstellen will, dann muß man nur eine *rename*-Operation auf die temporäre Directory, die sich direkt unterhalb der *lost+found*-Directory befindet, ausführen. Da temporäre Directories immer dem Superuser gehören und voreingestellte Zugriffsrechte besitzen, ist auch hier gegebenenfalls eine Anpassung vorzunehmen.

Wenn Hardlinks im *Worm*-Filesystem implementiert sind, dann kann es, bedingt durch die Art der Implementierung, *Gnodes* von Hardlinks auf dem Medium geben, die auf einen nicht mehr existenten *Gnode* einer Originaldatei zeigen. Für diese Originaldatei wird vom oben beschriebenen Einlesealgorithmus in der *lost+found*-Directory ein temporärer Eintrag erzeugt. Da der Einlesealgorithmus keine Informationen über den Dateityp, die Dateigröße, die Zugriffsrechte und den Besitzer dieser Datei besitzt, werden hier Standardwerte verwendet.

Eine Rekonstruktion ist bei diesen Dateien nicht möglich. Man kann jedoch mit Hilfe der *Inodenummer* der temporären Datei sämtliche auf diese Datei weisenden Hardlinks finden und löschen, um einen konsistenten Zustand herzustellen. Dazu kann man das Programm `find` verwenden.

4.4. Filesystem Operationen

Die Algorithmen, die für die Filesystem Operationen und die *Vnode*-Operationen verwendet werden, sind im wesentlichen schon im Kapitel über die Schnittstelle des virtuellen Filesystems von SunOS 4.0 beschrieben worden. Sie können nach den Vorgaben aus dieser Beschreibung implementiert werden. Durch den *Gnode*-Cache sind alle wichtigen, das Filesystem beschreibenden Daten verfügbar, nachdem das Filesystem in den Filesystembaum des Rechners integriert wurde.

Um die Basisoperationen (Erzeugen einer neuen Datei, Beschreiben oder Lesen einer Datei, sowie das Löschen einer Datei) implementieren zu können, müssen zunächst Funktionen implementiert werden, mit denen man neue *Gnodes* im *Gnode*-Cache anlegen und löschen kann. Die anderen Funktionen werden, wie schon erwähnt nach den Vorgaben aus der Beschreibung des virtuellen Filesystems implementiert. Es sind noch in geeigneter Weise die Zeitpunkte zu bestimmen, zu denen die Dateiinhalte, die *Gnodes* und der Superblock-Update auf das Medium zu schreiben sind; dabei sind nur die im Kapitel über die Datenstrukturen auf dem Medium erwähnten Maßnahmen zur Erhaltung der Konsistenz der Daten auf dem Medium zu beachten.

4.5. Pagebarer Speicher für den *Gnode*-Cache

Für den *Gnode*-Cache werden pro *Gnode* ca. 100 Bytes benötigt.[33] Ein typisches `/usr`-Filesystem unter *SunOS* 4.0 hat zwischen 5 000 und 10 000 Dateien. Der *Gnode*-Cache für dieses Filesystem würde also zwischen 0.5 und 1 MByte belegen. Bei größeren Filesystemen würde entsprechend mehr Speicher für diesen Cache benötigt. Bei einem 400 MBytes großem Filesystem und einer mittleren Dateigröße von 10 kBytes ist mit 40 000 *Gnodes* zu rechnen, für die dann ca. 4 MBytes für den Cache benötigt würden. Wenn die mittlere Dateigröße unter 10 kBytes liegt, oder auf dem Medium mehr als 400 MBytes verfügbar sind, wird entsprechend mehr Speicher für den *Gnode*-Cache benötigt. Daher ist es nicht mehr realistisch, diesen Cache mit realem Speicher aus dem Adreßraum des UNIX-Kerns aufzubauen.

Es ist also wünschenswert, den *Gnode*-Cache dem *Paging* zu unterziehen, denn dann ist die Größe dieses Caches nicht mehr so erheblich.

Um das zu erreichen, muß untersucht werden, welche Zugriffsmethoden in SunOS für Speicher, der dem *Paging* unterliegt, verwendet werden. Danach kann die für das *Worm*-Filesystem optimale Benutzung ermittelt und implementiert werden.

4.5.1. Anonymer Speicher in *SunOS* 4.0

SunOS bietet sogenannten anomymen Speicher, das sind Teile des *Swap*-Bereichs die in den virtuellen Adreßraum des Kerns gemappt werden können.

In *SunOS* 4.0 wird anonymer Speicher an folgenden Stellen verwendet:

- Für die *BSS*- und Stack- Bereiche der Anwenderprozesse.
- Für das temporäre Umkopieren der Argumente eines neu zu erzeugenden Prozesses im Systemaufruf *execve()*.

---

[33] Bei durchschnittlicher Länge des Dateinamens (ca. 8 Buchstaben zuzüglich des Nullbytes) beträgt der Anteil der *Gnode*-Daten ca. 80 Bytes. Weitere 20 Bytes werden für die Verzeigerung der *Gnodes* benötigt.

- Für *Shared Memory* mit der in *System V* verwendeten Schnittstelle.
- Für die *u*-Page von Prozessen.

Bei den ersten drei Punkten – dem Mappen des BSS- und Stack- Bereiches, sowie dem Mappen des Temporären Bereichs für die Argumente des Systemaufrufs *execve()* und dem *Shared Memory* – wird der anonyme Speicher mit Hilfe von *as_map()* gemappt. Dabei muß vorher innerhalb des virtuellen Adreßraumes ein noch nicht benutztes Segment mit Hilfe von *as_hole()* ermittelt werden. Bei dieser Methode bekommt man dann ein zusammenhängendes Stück virtuellen Speichers, das an der mit *as_hole()* ermittelten Stelle beginnt.

Bei dem Mappen der *u*-Page wird jeweils nur eine Seite auf eine feste Adresse gemappt. Daher erscheint die dort angewandte Methode für den *Gnode*-Cache uninteressant.

Um die Brauchbarkeit der in *SunOS*4.0 realisierbaren Methoden des Zugriffs auf anonymen Speicher für den *Gnode*-Cache beurteilen zu können, wurde zunächst die Speicherbelegung im *SunOS* 4.0-Kern untersucht.

4.5.2. Die physische Speicherbelegung

SunOS 4.0 ist ein virtuelles Betriebssystem. Daher ist die physische Speicherbelegung für den untersuchten Zweck nur von sekundärem Interesse, wird aber der Vollständigkeit halber hier trotzdem aufgeführt.

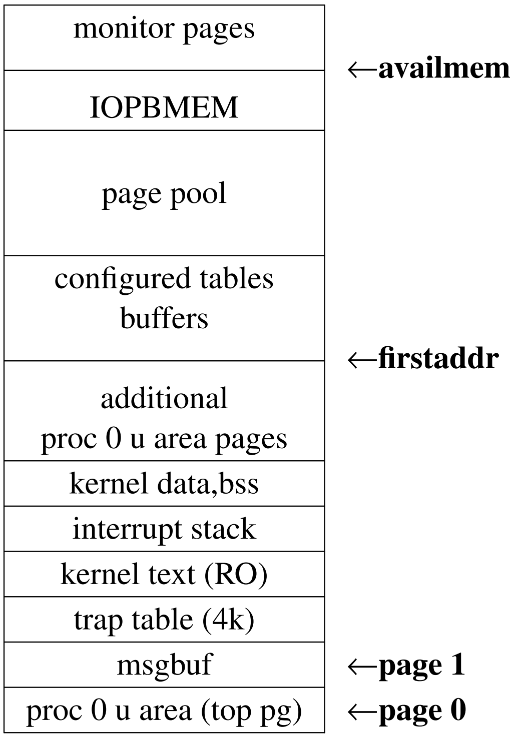

Bild 7: Physische Speicherbelegung unter SunOS 4.0

4.5.3. Die virtuelle Speicherbelegung

Adresse	Belegung	Symbol
	current u page(s)	
0xFFFF8000 →		←**UADDR**
	redzone	
	memory ecc reg	
0x0FFE8000 →		
	interrupt reg	
0x0FFE6000 →		
	memory error reg	
0x0FFE4000 →		
	clock	
0x0FFE2000 →		
	eeprom	
0x0FFE0000 →		
	DVMA IOPB memory	
0x0FF00000 →		←**DVMA**
	monitor	
0x0FE00000 →		←**MONSTART**
	debugger	
0x0FD00000 →		←**DEBUGSTART**
	SEGTEMP2	
0x0FCE0000 →		
	SEGTEMP	
0x0FCC0000 →		
	generic mapping seg	
0x0F908000 →		
	NCARGS + MINMAPSIZE	
0x0F808000 →		←**Syslimit**
	unused	
	Sysmap and other kernel virt addresses	
0x0F700000 →		←**forkutl**
	unused	
	configured tables buffers	
		←**end**
	kernel code	
	trap table (4k)	
0x0F004000 →		←**start**
	msgbuf	
0x0F002000 →		←**msgbuf**
	user copy red zone (invalid)	
0x0F000000 →		←**KERNELBASE**
	user stack user data	
	user text	
0x00002000 →		
0x00000000	invalid	

Bild 8: Virtuelle Speicherbelegung einer Sun 3/50 unter SunOS 4.0

Um die Überlegungen, die zum Entwurf der Speicherverwaltung für den *Gnode*-Cache geführt haben, besser erläutern zu können, werden die einzelnen Bereiche des obigen Bildes diskutiert.

Der Bereich der Virtuellen Adressen 0 bis *Kernelbase* ist für die Benutzerprogramme reserviert. Bei einem *Pagefault* in diesem Adreßbereich wird nur die Adreßraumbeschreibung des gerade aktiven Benutzerprogramms durchsucht. Eine Nutzung dieses Bereichs für den *Gnode*-Cache scheidet daher aus.

Im Adreßbereich direkt darüber befindet sich eine Seite als Sicherheitsabstand zum Benutzerprogramm und ein Puffer, in dem die Ausgaben der Routine *printf()* des Kerns gepuffert werden. Es folgen die Trap-Tabelle, das Text-Segment, sowie initialisierte und nichtinitialisierte Daten des Kerns. Dieser Bereich reicht bis *Syslimit* und hat eine Größe von ca. 8 MBytes. Wenn der Kern, wie bei SunOS 4.0 ca. 1 MByte belegt, dann liegt hinter dem *bss*-Segment des Kerns innerhalb des eben beschriebenen Bereichs ein ungenutzter Bereich von ca. 6 MBytes. Da der Kern für den gesamten eben beschriebenen Bereich nur einen Adress Segment-Descriptor verwendet, ist es nicht möglich den ungenutzten Bereich von 6 MBytes zum Mappen von anonymen Speicher zu nutzen.

Der dann folgende Bereich von der virtuellen Adresse 0x0F808000 bis 0x0F908000 wird für das Umkopieren der Argumente beim Systemaufruf *execve(2)* verwendet. Dort könnte man bis zu 1 MByte zum kontinuierlichen Mappen von anonymen Seiten verwenden, jedoch auf Kosten der maximal zulässigen Anzahl von Argumenten beim Systemaufruf *execve(2)*.

Das *generic mapping segment* wird für das kurz- bis mittelfristige Mappen von Bereichen der Filesysteme verwendet. Dieser Bereich eignet sich daher am besten für die Zwecke des *Gnode*-Caches.

SEGTEMP und *SEGTEMP2* sind virtuelle Bereiche, die für temporäre Zwecke benutzt (wie den Zugriff auf Pagetable-Entries und das Synchronisieren des virtuellen Adress-Caches) werden.

Die folgenden Bereiche werden für Monitor, Kerneldebugger, DMA und Hardwareregister benutzt.

Der große, mit *redzone* bezeichnete Bereich, ist wegen eines Design-Fehlers bei der MMU der Sun 3/50 nicht zu benutzen, denn die Adreßdekodierung erfolgt bei dieser MMU unvollständig.

4.5.4. Möglichkeiten der Verwendung von anomymen Speicher in *SunOS* 4.0

Für den *Gnode*-Cache wäre es optimal, wenn man ein großes, zusammenhängendes Stück Speicher bekommen könnte. Da, wie schon beschrieben, der Speicherbedarf des *Gnode*-Caches bei 4 Mbytes und mehr liegt,[34] ist dieser Wunsch nach zusammenhängendem

[34] Die für das *Worm*-Filesystem vorgesehene Platte ist ca. 400 MBytes groß, sie verfügt über ca. 200 000 Sektoren zu 2048 Bytes. Würde man diese 200 000 Sektoren vollständig mit *Gnodes* füllen, d.h alle Dateien hätten die Größe 0 und es wären mehrere *Gnodes* in einem Sektor, dann könnte man

Speicher nicht zu erfüllen, denn ein genügend großes, nicht für andere Zwecke benutztes Segment im virtuellen Adreßraum, kann nicht gefunden werden.

Die einzige realisierbare Methode, Speicher zu benutzen, der dem *Paging* unterzogen werden kann, besteht darin, einzelne Seiten auf Anforderung in das *Generische Mapping Segment* zu mappen. Durch diese Methode erscheinen die betreffenden Seiten aber auf wechselnden virtuellen Adressen. Deshalb ist es nicht möglich, diesen Speicher für die Verzeigerung der Baumstruktur des *Gnode*-Caches zu verwenden.

Um dennoch einen größeren Teil des *Gnode*-Caches dem *Paging* unterziehen zu können, werden die *Gnodes* des Caches in einen Anteil von ca. 20 Bytes, der nur die Zeiger enthält und einen Anteil von ca. 80 Bytes, der die restlichen Daten enthält, aufgeteilt. Dann muß nur der Anteil von ca. 20 Bytes, der für die Verzeigerung der *Gnodes* benötigt wird aus realem Speicher bestehen.[35]

4.5.5. Die Verwaltung des anonymen Speichers

Um den anonymen Speicher verwalten zu können, wird für jede anonyme Seite eine in realem Speicher stehende Struktur angelegt, sie wird *struct ahead* genannt. Diese für die Verwaltung angelegten Strukturen werden in zwei doppelt verketteten Listen geführt. Eine Liste enthält alle aktiven Elemente in der Reihenfolge des Entstehens, die andere Liste ist nach der Häufigkeit des Gebrauchs sortiert.

Jede anonyme Seite ist durch einen *struct anon* eindeutig beschrieben. Wenn sie sich im virtuellen Adreßraum des Kerns befindet, ist die virtuelle Adresse der Seite in der zur Verwaltung der anonymen Seiten verwendeten Struktur *ahead* gültig und zeigt auf die anomyme Seite. In einem Feld mit *Flag*-Bits ist vermerkt, ob sich die anonyme Seite im virtuellen Adreßraum des Kerns befindet und ob sie modifiziert wurde, seit sie das letzte Mal vom Hintergrundspeicher in den virtuellen Adreßraum des Kerns geholt wurde.

Um eine Relation von den *Gnode*-Headern auf die dazugehörigen *Gnode*-Daten in den anonymen Seiten zu bekommen, enthält jeder *Gnode*-Header einen Zeiger auf die Struktur *ahead* für die Seite, in der sich die *Gnode*-Daten befinden, und den Offset, auf dem die *Gnode*-Daten innerhalb der betreffenden anonymen Seite liegen. Wenn man auf die Daten eines Eintrags im *Gnode*-Cache zugreifen will, ruft man eine Funktion aus der Verwaltung der anonymen Seiten des *Gnode*-Caches auf, die sicherstellt, daß sich die anonyme Seite, auf der die *Gnode*-Daten liegen, im virtuellen Adreßraum des Kerns befindet. Danach kann man die im *struct ahead* vermerkte virtuelle Adresse der anonymen Seite und den Offset aus dem *struct ghead* addieren und bekommt die virtuelle Adresse der *Gnode*-Daten.

3,2 Millionen *Gnodes* auf dem Medium unterbringen. Bei einer mittleren Dateigröße von 4096 Bytes wären es noch 100 000 *Gnodes;* bei 100 Bytes pro *Gnode* würde der *Gnode*-Cache für diese 100 000 *Gnodes* 10 MBytes belegen.

[35] Prinzipiell ist es auch möglich die für die Verzeigerung nötigen Daten in pagebarem Speicher zu halten. Da das aber die Verwaltung des *Gnode*-Caches stark erschweren würde, wurde in der ersten Implementierung die einfachere, oben beschriebene Variante implementiert.

Wenn sich eine konfigurierbare Anzahl von anonymen Seiten gleichzeitig im virtuellen Adreßraum des Kerns befindet und eine weitere benötigt wird, dann wird in der Liste, die nach der Häufigkeit des Gebrauchs sortiert ist, der älteste Eintrag gesucht und die dazugehörige anonyme Seite freigegeben. In einem der *Flag*-Bits ist vermerkt, ob dazu der Inhalt der Seite vorher auf den Hintergrundspeicher zu schreiben ist, weil er modifiziert wurde.

Liste von *Gnode* Header-Strukturen

next next next
cont cont cont
offset offset offset
ahead ahead ahead

Liste von *Anonymous* Header-Strukturen

next next next
prev prev prev
buf buf buf

anonyme Seite (gemappt)
anonyme Seite (nicht gemappt)

Gnode-Rumpf

Bild 9: Die Verwaltung des *Gnode*-Caches unter Verwendung von anonymen Speicher.

4.5.6. Nebenläufigkeitsprobleme in der Verwaltung

Ein großes Problem stellen die *Gnodes* im Cache dar. Immer wenn die Baumstruktur des *Gnode*-Caches durchsucht wird, muß auf Daten aus dem Bereich der anonymen Seiten zugegriffen werden. Da UNIX ein multitasking Betriebssystem ist, ist es denkbar, daß mehrere Prozesse quasi gleichzeitig auf den *Gnode*-Cache, und daher gleichzeitig auf unterschiedliche anonyme Seiten aus diesem Cache zugreifen wollen.

Daher ist es denkbar, daß ein Prozess unterbrochen wird, nachdem er sich eine Seite aus dem anonymen Speicher gemappt hat, wenn keinen besonderen Maßnahmen dagegen unternommen werden. Bevor dieser Prozess wieder aktiviert wird, kann nun ein weiterer Prozess versuchen, eine weitere anonyme Seite aus dem Cache zu mappen und muß dafür eventuell wegen knapper Resourcen eine Seite aus dem anonymen Speicher freigeben. Wenn dafür nun die vom ersten Prozess gerade gemappte Seite gewählt wurde, ist, nachdem der erste Prozess wieder aktiviert wurde, die von ihm gewünschte anonyme Seite nicht verfügbar.

Man könnte dieses Problem dadurch umgehen, daß jeder Prozess, der auf den anonymen Speicher zugreift, den Zeitraum zwischen dem Mappen und dem Zugriff auf die soeben gemappten Daten durch Erhöhen der Interrupt-Priorität des Prozessors absichert. Diese Methode erscheint jedoch nicht empfehlenswert, denn der Zeitraum, der auf diese Weise abzudecken wäre, ist bei aufwendigeren Arbeiten an den sich auf der anonymen Seite befindlichen Daten zu groß.

Eine bessere Methode zur Lösung des Problems besteht darin, einen Referenzzähler einzuführen. Er befindet sich in jeder Struktur zur Verwaltung einer Seite aus dem anonymen Speicher. Wenn dieser Referenzzähler größer als Null ist, dann darf die dazugehörige Seite nicht freigegeben werden. Bei dieser Methode kann es allerdings passieren, daß die gewünschte Freigabe einer Seite nicht erfolgen kann, weil ein anderer Prozess die Resourcen benötigt. Dann muß man den Prozess, der als erster die Resourcen angefordert hat, bevorzugen und den anderen, solange der erste Prozess die Resourcen benötigt, blockieren.

4.6. Überblick über die verwendeten Datenstrukturen

`wofs.h` definiert folgende Datenstrukturen:

```
#define WOBBSIZE        8192            /* Boot block size */
#define WOSBSIZE        8192            /* Super block size */
#define WOBBOFF         ((daddr_t)(0))  /* Position of Boot block */
#define WOSBOFF         ((daddr_t)(WOBBOFF + WOBBSIZE / DEV_BSIZE))

#define WOROOTINO       ((ino_t)2)    /* i number of all roots */
#define WOLOSTFOUNDINO  (WOROOTINO +1)/* lost+found ino (fixed for now)*/

#define MAXWOMNTLEN     512

#define WOFS_CURVERSION 0x000000
#define WOFS_MAGIC      0x270355

#define MWRITE  0x0001                  /* filesys is rewritable */
#define COMPACT 0x0002                  /* create compacted gnodes only*/

#define wofsbtodb(fs, b)        ((b) << (fs)->wofs_fsbtodb)
#define dbtowofsb(fs, b)        ((b) >> (fs)->wofs_fsbtodb)
```

Bild 10: Konstanten für die Positionierung des primären Superblocks, sowie Makros zum Umrechnen der Blockadressen des Mediums in Blockadressen für den Gerätetreiber.

```
struct wofs {
        struct  wofs *wofs_link;/* linked list of file systems */
        struct  wofs *wofs_rlink;
        time_t  wofs_time;      /* last time written */

/* the next fields are derived from the hardware */
        daddr_t wofs_sblkno;    /* addr of super-block in filesys */
        long    wofs_size;      /* number of sectors in fs */
        long    wofs_sbsize;    /* actual size of super block */
        long    wofs_fsize;     /* medias sector size (frag) */
        long    wofs_kbps;      /* contiguous read speed (kBytes/s) */

/* the next fields are configuration parameters */
        long    wofs_sblkcnt;   /* # of super-blocks in segment */
        long    wofs_ngnull;    /* # of null sectors past last gnode*/
        long    wofs_minfree;   /* minimum percentage of free blocks*/
        long    wofs_cspare[8]; /* constants for future expansion */

/* the next fields can be computed from above */
        long    wofs_dsize;     /* number of data sectors in fs */
        long    wofs_fsbtodb;   /* wofsbtodb and dbtowofsb shift */
        long    wofs_gnopf;     /* max # of gnodes per frag */
        long    wofs_nspf;      /* number of DEV_BSIZE sectors/frag */
        long    wofs_dspare[8]; /* constants for future expansion */

/* the next fields can be recomputed from a run of fsck */
        daddr_t wofs_sblkjump;  /* addr of jump super-block */
        daddr_t wofs_sblknext;  /* addr of next super-block */
        daddr_t wofs_gno_hi;    /* addr of oldest active gnode */
        daddr_t wofs_gno_lo;    /* addr of youngest active gnode */
        daddr_t wofs_dblkno;    /* addr of first data in filesys */
        daddr_t wofs_dblkno_hi; /* addr of last allocated data */
        long    wofs_ino_hi;    /* highest used inode-number */
        long    wofs_ndir;      /* number of directories */
        long    wofs_nfile;     /* number of files */
        long    wofs_ngfree;    /* number of free gnodes */
        long    wofs_nffree;    /* number of free data blocks */

        long    wofs_xspare[16];/* constants for future expansion */
        char    wofs_fmod;      /* super block modified flag */
        char    wofs_clean;     /* filesystem is clean flag */
        char    wofs_ronly;     /* mounted read-only flag */
        char    wofs_flags;     /* flags */
        long    wofs_id[2];     /* file system id */
        long    wofs_version;   /* version # of wofs */
        long    wofs_magic;     /* magic number */
        short   wofs_chksum;    /* xor chksum on sbsize/2 short's */
        char    wofs_fsmnt[MAXWOMNTLEN];/* name mounted on */
};
```

Bild 11: Der Superblock auf dem *Worm*-Filesystem.

`wofs_node.h` definiert folgende Datenstrukturen:

```
#define ONDADDR 3               /* Direct blocks (currently only one) */
#define ONIADDR 1               /* Indirect blocks (currently unused) */

/*
 * disk g-node
 */
struct gnode {
        u_short gn_mode;        /*  0: mode and type of file */
        short   gn_nlink;       /*  2: number of links to file */
        uid_t   gn_uid;         /*  4: owner's user id */
        gid_t   gn_gid;         /*  6: owner's group id */
        quad    gn_qsize;       /*  8: number of bytes in file */
#ifdef KERNEL
        struct timeval gn_atime;/* 16: time last accessed */
        struct timeval gn_mtime;/* 24: time last modified */
        struct timeval gn_ctime;/* 32: last time inode changed */
#else
        time_t  gn_atime;       /* 16: time last accessed */
        long    gn_atspare;
        time_t  gn_mtime;       /* 24: time last modified */
        long    gn_mtspare;
        time_t  gn_ctime;       /* 32: last time inode changed */
        long    gn_ctspare;
#endif
#define gn_hino gn_db[0]        /*     files's i-number for hardlinks */
        daddr_t gn_db[ONDADDR]; /* 40: disk block addresses */
        daddr_t gn_ib[ONIADDR]; /* 52: indirect blocks (unused) */
        long    gn_flags;       /* 56: flags see below */
        long    gn_blocks;      /* 60: blocks actually held */
        long    gn_ino;         /* 64: inode number gnode refers to */
        long    gn_parent;      /* 68: parent inode number */
        long    gn_gen;         /* 72: generation number */
        long    gn_spare[3];    /* 76: reserved, currently unused */
        short   gn_magic;       /* 88: magic it's a gnode */
        short   gn_chksum;      /* 90: xor chksum for gn_size/2 short's*/
        short   gn_gsize;       /* 92: actual size of gnode */
        short   gn_namelen;     /* 94: number of chars in gn_name */
/*      char    gn_name[0];     /* 96: name of file */
        char    gn_name[32];    /* 96: name of file */
};                              /* 128: end if sizeof(gn_name) == 32 */
```

Bild 12: Der *Gnode*, wie er auf dem Medium verwendet wird.

```
/*
 * cached g-node
 */
struct gcnode {
        u_short gc_mode;        /*  0: mode and type of file */
        short   gc_nlink;       /*  2: number of links to file */
        uid_t   gc_uid;         /*  4: owner's user id */
        gid_t   gc_gid;         /*  6: owner's group id */
        quad    gc_qsize;       /*  8: number of bytes in file */
        struct timeval gc_atime;/* 16: time last accessed */
        struct timeval gc_mtime;/* 24: time last modified */
        struct timeval gc_ctime;/* 32: last time inode changed */
#define gc_hino gc_db[0]        /*     files's i-number for hardlinks */
        daddr_t gc_db[1];       /* 40: disk block addresses */
        long    gc_flags;       /* 44: flags see below */
        long    gc_blocks;      /* 48: blocks actually held */
        long    gc_ino;         /* 52: inode number gnode refers to */
        long    gc_parent;      /* 56: parent inode number */
        long    gc_gen;         /* 60: generation number */
        short   gc_gsize;       /* 62: actual size of gnode */
        short   gc_namelen;     /* 64: number of chars in gn_name */
        char    gc_name[1];     /* 68: name of file */
};
```

Bild 13: Der *Gnode*, wie er im Cache verwendet wird.

```
#define GN_MAGIC        0x474D /* G-node magic number */
#define LN_MAGIC        0x4C4D /* Linkname Magic number */

/*
 * G-node Flags
 */
#define GDATA           0x0001 /* G-node and Data is in one sector */
#define GCOMPACT        0x0002 /* Compacted n G-nodes into one sector */
#define GHARD           0x0004 /* G-node is ino-relative link */
#define GCACHE          0x0008 /* Cache G-node during rebuilt of tree */
#define GTMP            0x0010 /* G-node is tmp during rebuilt of tree */
```

Bild 14: *Magic Numbers* für *Gnode* und Erweiterungs-Sektoren bei langen Linknamen, sowie im *Gnode* gebrauchte *Flag*-Bits.

```
/*
 * header for cached g-nodes
 */
typedef struct ghead {
        struct ghead *next;    /* next entry in directory contents */
        struct ghead *prev;    /* previous entry in chain */
        struct ghead *cont;    /* first entry in contents chain */
        struct ahead *hp;      /* pointer to anon page containing gcnode*/
        short   hindex;        /* gcnode offset in page */
} GHEAD;
```

Bild 15: Die Struktur *ghead* befindet sich im realen Memory des Kerns und enthält die Zeiger des *Gnode*-Caches, die die Baumstruktur eines Filesystems repräsentieren.

`wofs_anon.h` definiert folgende Datenstrukturen:

```
typedef struct ahead {
        struct  ahead *next;    /* the next header in the linked list */
        struct  ahead *prev;    /* the last header in the linked list */
        struct  ahead *more;    /* the next most recently used header */
        struct  ahead *less;    /* the next least recently used header */

        struct  anon    *ap;    /* anon info for this page */

        caddr_t buf;            /* where in memory it is, 0 if not */

        short   flags;          /* flags see below */
        short   count;          /* reference count for this buffer */
} AHEAD;

/*
 * flags
 */
#define INMEMORY        0x0001
#define MODIFIED        0x0002
```

Bild 16: Die Struktur *ahead* wird für die Verwaltung einer anonymen Seite aus dem virtuellen Memory des Kerns verwendet.

5. Diskussion und Ausblick

Als Ergebnis dieser Arbeit verfügen wir nun über ein funktionsfähiges Filesystem, das auch auf *Worm*-Medien benutzbar ist und gute Performance-Eigenschaften in Bezug auf die Benutzung in multimedialen Anwendungen zeigt. Die notwendigen Einschränkungen gegenüber dem UNIX-Filesystem sind bei der beabsichtigten Anwendung nicht schwerwiegend. Das neue Filesystem verfügt sogar über Eigenschaften, die es wünschenswert erscheinen lassen, es auch für andere Anwendungen mit nahezu ausschließlichem Lesezugriff, wie z.B. dem `/usr`-Filesystem zu verwenden.

Im Laufe der Entwicklung des *Worm*-Filesystems wurden weite Entwicklungsmöglichkeiten deutlich, deren Verwirklichung dem Filesystem in einigen Punkten noch bessere Eigenschaften geben würde.

5.1. Kompression der *Gnodes*

Wie im Kapitel *Datenstrukturen auf dem Medium* erwähnt, finden bei typischer Anwendung weit über 90% der *Gnodes* in der Minimalgröße des *Gnodes* von 128 Bytes Platz. Da es für die Sicherheit der Filesystemkonsistenz auf dem Medium notwendig ist, jeden *Gnode*-Update baldmöglichst nach einem *close(2)* auf eine modifizierte Datei, auf das Medium zu schreiben, muß ein *Gnode*-Update im allgemeinen mindestens einen Sektor eines *Worm*-Mediums belegen. Bei einer Sektorgröße von 512 Bytes bleiben dadurch jedoch 75% des Bereichs der *Gnode*-Updates auf dem Medium ungenutzt. Bei einer Sektorgröße von 2048 Bytes erhöht sich der ungenutzte Bereich innerhalb der *Gnode*-Updates auf 93,75%.

Durch diese Methode des *Gnode*-Updates ergibt sich außer der Tatsache, daß ein relativ großer Bereich des Mediums von den *Gnodes* belegt wird, noch ein weiterer unerwünschter Effekt: Die Zeit, die zum Füllen des *Gnode*-Caches beim Systemaufruf *mount(2)* benötigt wird, ist sowohl durch den Rechenaufwand für die Verzeigerung des *Gnodes*-Caches, als auch durch die Zeit für das Einlesen der gesamten Datenmenge aus dem Bereich der aktiven *Gnodes* auf dem Medium bedingt. Dies ist insbesondere bei Medien von Interesse, die nur über eine geringe Datentransfergeschwindigkeit verfügen.

Um dies zu verdeutlichen, sei ein Beispiel angeführt:

Bei dem *Worm*-Laufwerk RXT-800S der Firma Maxtor lassen sich, bei einer Größe des Transferpuffers von 63 kBytes, Datentransferraten von 80 kBytes/s erreichen. Bei der dort verwendeten Sektorgröße von 2048 Bytes lassen sich also nur 40 *Gnodes*/s einlesen. Da bei einer Filesystemgröße von 400 MBytes und einer mittleren Dateigröße von 10 kBytes mit ca. 40 000 Dateien[36] auf dem Medium zu rechnen ist, werden ca. 1000 Sekunden für das Einlesen des *Gnodes* benötigt. Dadurch werden für den Systemaufruf *mount(2)* unter diesen Randbedingungen mehr als 16 Minuten benötigt. Diese Zeit ist offensichtlich untragbar hoch.

[36] Ohne Berücksichtigung der durch die *Gnodes* bedingten Verringerung des für die Dateien zur Verfügung stehenden Bereichs aus dem Medium.

Wenn man mehrere *Gnodes* in einem Sektor unterbrächte, dann würden bei dem obigen Beispiel 16 *Gnodes* in einen Sektor passen. Damit würde die Zeit, die zum Einlesen der *Gnodes* benötigt wird, auf ca. 1 Minute sinken. Diese Zeit ist in der gleichen Größenordnung, wie die Rechenzeit, die bei einer Sun 3/50 zum Aufbau des *Gnode*-Caches benötigt wird.[37]

5.2. Methoden zur Verringerung des von den *Gnodes* belegten Bereichs

Eine Methode, den von den *Gnodes* belegten Bereich auf dem Medium zu verringern, wäre die Verwendung eines Programms, das aktiviert wird, wenn das Filesystem nicht montiert ist. Dieses Programm könnte den Bereich der aktiven *Gnodes* auf dem Medium einlesen und einen neuen *Gnode*-Bereich anlegen. Dieser Bereich würde dann nur noch die aktuellen Versionen der *Gnodes* enthalten, von denen dann soviel wie möglich in einem Sektor untergebracht wären.

Damit man erkennen kann, daß sich in einem Sektor mehr als ein *Gnode* befindet, kann man solche *Gnodes* mit einem *Flag*-Bit markieren oder jeden *Gnode* mit einer Angabe für die Gesamtgröße versehen. Wenn die im *Gnode* verzeichnete Gesamtgröße geringer als die Sektorgröße ist, dann befinden sich in dem Sektor noch weitere *Gnodes.*

Um einen neuen *Gnode*-Bereich auf dem Medium anzulegen, muß man hinter dem logischen Ende des aktuellen Bereichs der *Gnodes* auf dem Medium in Wachstumsrichtung die gewünschten *Gnodes* schreiben. Danach wird ein Superblock-Update, in dem die neuen Werte für den Anfang und das Ende des gültigen *Gnode*-Bereichs auf dem Medium vermerkt sind, geschrieben.

Durch diese Methode läßt sich zwar das Einlesen der *Gnodes* in den Cache beschleunigen, sie hat jedoch den Nachteil, daß dabei ein relativ hoher Anteil der Kapazität des Mediums verbraucht wird, denn jeder *Gnode* wird zweimal geschrieben. Beim ersten Mal wird er so geschrieben, daß er einen ganzen Sektor auf dem Medium belegt, beim zweiten Mal geschieht dies in der komprimierten Form. Da bei fortlaufender modifizierender Benutzung des Filesystems der Bereich der *Gnodes* wieder wachsen würde, müßte man diese Prozedur immer dann wiederholen, wenn das Montieren des Filesystems zu lange dauert.

Besser wäre es, wenn man die Kompression der *Gnodes* im laufenden Betrieb vornehmen könnte. Um das zu erreichen, könnte man immer dann, wenn ein *Gnode*-Update auf das Medium geschrieben werden muß, soviele der ältesten aktiven *Gnodes,* wie zusammen mit dem neuen *Gnode*-Update in einem Sektor unterzubringen sind, gemeinsam mit dem neuen *Gnode*-Update schreiben.

---

[37] Für den Fall, daß das Einlesen der *Gnodes* asynchron erfolgt, können das Einlesen der *Gnodes* und der Aufbau der Verkettung des *Gnode*-Caches gleichzeitig erfolgen. Damit liegt die Gesamtzeit, die für das Montieren des Filesystems benötigt wird nur unwesentlich über der Zeit, die für das Einlesen der *Gnodes* benötigt wird.

Bei dieser Methode der Kompression von *Gnodes* wird insgesamt soviel Platz auf dem Medium verbraucht, wie ohne Anwendung einer Kompressionsmethode benötigt würde, denn bei jedem *Gnode*-Update wird mindestens ein Sektor auf dem Medium verbraucht. Der von den *Gnodes* auf dem Medium belegte Bereich wäre, gegenüber der Anwendung des oben beschriebenen Kompressionsprogramms, nur um den Anteil nicht mehr aktiver *Gnode*-Versionen größer. Der Anteil der nicht mehr aktiven *Gnode*-Versionen würde sich jedoch in Grenzen halten, denn dadurch, daß die ältesten aktiven *Gnodes* mit an das aktuelle Ende des *Gnode*-Bereichs geschrieben werden, nicht mehr aktive Versionen der *Gnodes* jedoch nicht kopiert werden, verlassen die nicht mehr aktiven Versionen den zum Füllen des *Gnode*-Caches zu lesenden Bereich auf dem Medium.

5.3. Das *Worm*-Filesystem auf wiederbeschreibbaren Medien

Eine wiederbeschreibbares Medium, auf dem sich noch kein *Worm*-Filesystem befunden hat, läßt sich problemlos mit dem *Worm*-Filesystem benutzen. Ohne Modifikationen am Algorithmus des Filesystems ist es jedoch, wie schon im Kapitel *Datenstrukturen auf dem Medium* beschrieben, nur dann möglich, ein wiederbeschreibbares Medium wieder mit einem *Worm*-Filesystem zu versehen, wenn vorher das gesamte Medium mit einem Datenmuster überschrieben wurde.

Wenn man ohne besondere Vorkehrungen ein wiederbeschreibbares Medium mehrmals mit einem *Worm*-Filesystem versehen will, dann müssen die im Kapitel *Datenstrukturen auf dem Medium* erwähnten Modifikationen am Filesystemcode vorgenommen werden:

- Immer dann, wenn ein Bereich für Superblock-Updates neu angelegt wird, muß dieser Bereich genullt werden.
- Beim Schreiben von *Gnode*-Updates ist dafür zu sorgen, daß ein ausreichend großer Bereich hinter dem logischen Ende der *Gnode*-Updates genullt wird.

Dazu muß im Superblock ein *Flag*-Bit vorgesehen werden, mit dessen Hilfe man die bei wiederbeschreibbaren Medien notwendigen Modifikationen des *Worm*-Filesystemcodes aktiviert.

Werden keine weiteren Modifikationen am *Worm*-Filesystem vorgenommen, dann hat man ein Filesystem, das auf wiederbeschreibbaren Medien die gleichen Eigenschaften aufweist, wie auf *Worm*-Medien. Wenn man das *Worm*-Filesystem auf Magnetplattenlaufwerken betreibt, dann profitiert man selbstverständlich von der höheren Schreibgeschwindigkeit diese Laufwerke. Bei Filesystemen mit geringer Schreibaktivität und häufigem Lesezugriff ist das *Worm*-Filesystem besser geeignet als das UNIX-Filesystem. Daher eignet sich das *Worm*-Filesystem besonders für das `/usr`-Filesystem von UNIX. Mit zusätzlichen Maßnahmen läßt sich das *Worm*-Filesystem jedoch noch besser an wiederbeschreibbare Medien anpassen.

Da es bei wiederbeschreibbaren Medien keine Probleme mit dem Update von Daten gibt, kann man auf die besonderen Methoden, die für Update von Daten auf *Worm*-Medien entwickelt wurden, verzichten. Wiederbeschreibbare Medien bieten daher folgende Vorteile:

- Der Superblock-Update kann beliebig häufig an der selben Position des Mediums erfolgen. Dafür kann die Position des primären Superblocks verwendet werden.
- *Gnode*-Updates können beliebig häufig an der Position vorgenommen werden, die der betreffende *Gnode* bei seiner Erzeugung hatte.

Wenn es keine Beschränkungen für die Frequenz des Updates von Superblock und *Gnodes* gibt, dann kann der Update dieser Strukturen bei jedem *sync(2)* erfolgen. Dadurch wird die Integrität des Filesystems und die Sicherheit bei Systemzusammenbrüchen wesentlich erhöht.

Um den Zeitraum der aktiven Benutzung des *Worm*-Filesystems auf wiederbeschreibbaren Medien zu erhöhen, könnte man mit Hilfe eines Programmes, das aktiviert wird, wenn das Filesystem nicht montiert ist, die Datenbereiche der aktiven Dateien auf dem Medium so umkopieren, daß die Datenbereiche der gelöschten Dateien und der alten Versionen von Dateien wiederverwendet werden.

5.4. Daten kleiner Dateien innerhalb von *Gnodes*

Eine alternative Möglichkeit zur Nutzung des von einem *Gnode* im Sektor nicht genutzten Raumes besteht darin, den Dateiinhalt, falls er vollständig in diesen Raum paßt, mit im *Gnode* unterzubringen. Diese Methode, den ungenutzten Raum in dem den *Gnode* tragenden Sektor zu nutzen, scheint aber nur dann sinvoll, wenn man nicht die oben beschriebenen Methoden zur Kompression der *Gnodes* anwendet: Die im Bereich der *Gnodes* untergebrachten Dateiinhalte müssen beim Einlesen des *Gnode*-Bereichs zum Füllen des *Gnode*-Caches mitgelesen werden. Dadurch erhöht sich die Zeit, die zum Montieren eines solchen Filesystems benötigt wird gegenüber der Zeit, die bei einem Filesystem mit komprimierten *Gnodes* benötigt wird.

Trotzdem scheint es sinnvoll zu überlegen, ob man Dateien, die nur wenige Bytes enthalten, ähnlich wie die Linknamen von Symbolischen Links hinter den assoziierten *Gnodes* unterbringt. Wenn man als Grenze dafür z.B. 128 Bytes (die Minimalgröße eines *Gnodes*) ansetzt, dann lassen sich immer noch mehrere *Gnodes* in einem Sektor unterbringen und die Zeit zum Montieren eines so angelegten Filesystems bleibt immer noch unter der beim Montieren eines Filesystems, das einen *Gnode* pro Sektor trägt, benötigten Zeit.

5.5. Das Anbringen einer vorwärts verketteten Struktur auf dem Medium

Da die Baumstruktur des *Worm*-Filesystems in einer der normalen Benutzung entgegengesetzten Weise auf dem Medium notiert ist, kann man das *Worm*-Filesystem nur dann effektiv benutzen, wenn man einen Cache im Betriebssystem vorsieht. Wenn das Filesystem nahezu voll ist könnte man den restlichen Platz auf dem Filesystem dazu nutzen, eine vorwärts verkettete Struktur zu schreiben. Danach könnte man das Filesystem auch ohne den Cache benutzen.

5.6. Erhöhen der Schreibgeschwindigkeit bei magneto-optischen Medien

Bei magneto-optischen Laufwerken liegt die Schreibgeschwindigkeit deutlich unter der Lesegeschwindigkeit. Das liegt daran, daß bei diesen Medien vor einer Schreiboperation das Medium auf dem gewünschten Sektor zunächst gelöscht werden muß. Wenn man einzelne Bereiche nicht mehrmals beschreiben will, kann man bei dem SONY SMO-C501 Laufwerk diese Bereiche auch mit Hilfe eines speziellen *SCSI*-Kommandos vor der Benutzung löschen. Wenn man dann zum Zeitpunkt der gewünschten Schreiboperation ein herstellerspezifisches *SCSI*-Kommando verwendet, das keine Löschoperation auslöst, dann ist die Schreiboperation genauso schnell, wie eine Leseoperation.

Da das *Worm*-Filesystem auch bei wiederbeschreibbaren Medien die Bereiche des Mediums, die Dateiinhalte tragen, nur einmal beschreibt, könnte das obengenannte Verfahren auf die Dateiinhalte angewendet werden. Dann hätte man ein Filesystem, das bei magneto-optischen Medien eine Schreibgeschwindigkeit liefert, die identisch mit der Lesegeschwindigkeit ist.

6. Literatur

[Berkeley 4.2 SMM Kap. 14]
Marshall Kirk McKusick, William N. Joy, Samuel J. Leffler, Robert S. Fabry, *A Fast Filesystem for UNIX* Berkeley 4.2 System Manager's Manual Kapitel 14 vom 18. Februar 1984

[Berkeley 4.2 SMM Kap. 5]
Marshall Kirk McKusick, T. J. Kowalski *The UNIX File System Check Program* Berkeley 4.2 System Manager's Manual Kapitel 5 vom 16. Juli 1985

[Gingell *et al.* 1987]
R.A. Gingell, J.P. Moran, & W.A. Shannon, *Virtual Memory Architecture in SunOS* USENIX Association Conference Proceedings

[Sandberg 1985]
R. Sandberg, *The Design and Implementation of the Sun Network Filesystem* USENIX Association Conference Proceedings 1985

[Leffler *et al.* 1989]
Samuel J. Leffler, Marshall Kirk McKusick, Michael J. Karels, John S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System* Addison-Wesley Publishing Company

[Bach 1986]
M.J. Bach, *The Design of the UNIX Operating System* Prentice-Hall, Englewood Cliffs, NJ 1986

[Lions 1977]
J. Lions, *A Commentary on the UNIX Operating System* Department of Computer Science, The University of New South Wales 1977 Second Printing

[Lions 1977]
J. Lions, *UNIX Operating System Source Code level six* Department of Computer Science, The University of New South Wales 1977 Second Printing

Die selbstständige und eigenhändige Anfertigung versichert an Eides Statt:

Berlin, den 23. Mai 1991

(Jörg Schilling)

Inhaltsverzeichnis