Semesterarbeit

Version VFS

Erweiterung des virtuellen Dateisystems unter Linux um eine Versionsverwaltung

von

Stephan Müller und Sven Widmer

Universität Potsdam
Hasso-Plattner-Institut
Professur für Betriebssysteme und Middleware

Aufgabenstellung und Betreuung:

Prof. Dr. Andreas Polze
Dipl. Inf. Andreas Rasche

Potsdam
7. Dezember 2005

INHALTSVERZEICHNIS

1 Einleitung **1**
1.1 Dateisystem 1
1.1.1 Aufgaben von Dateisystemen 2
1.1.2 Dateisystemtypen 2
1.2 Dateiversionierung 4
1.3 Aufgabenstellung 6

2 Versionierung in VMS **7**
2.1 VMS-Dateisysteme 7
2.1.1 Verzeichnisstruktur 8
2.1.2 Laufwerksorganisation 8
2.1.3 Record Management Services 9
2.1.4 Physikalisches Layout – On-Disk Structure 9
2.1.5 Struktur der Verzeichnisse 10
2.1.6 Master File Directory 10
2.1.7 Indexdatei - INDEXF.SYS 10
2.2 Dateisystemsemantik 12

3 Dateisystemimplementationen in Linux **14**
3.1 Das virtuelle Dateisystem VFS 14
3.1.1 Common File Model 15
3.1.2 Abstrahierende Schnittstelle 16
3.1.3 Superblock 17
3.1.4 Inoden 19
3.1.5 Dentry-Cache 22
3.2 Arbeiten mit Dateien 23

4 Implementation **25**
4.1 Spezifikation der Anforderungen 25
4.2 Implementation 26

4.2.1 Entwicklungsumgebung 26
4.2.2 Level der Implementation 26
4.2.3 Verwaltung von DirMaxV 28
4.2.4 Darstellung und Interpretation der Versionsnummer 31
4.2.5 Darstellung der aktuellsten Version 31
4.2.6 Copy-On-Write 33
4.2.7 Copy-On-Rename 34
4.2.8 Usermode-Tools 35
4.3 Fazit 38

5 Performance-Messungen **39**
5.1 Messumgebung und -durchführung 39
5.2 Anzahl der Verzeichniseinträge 40
5.3 Dateigröße 41
5.4 Fazit 42

6 Ausblick und Fazit **45**
6.1 Ausblick 45
6.1.1 Mountoptionen 45
6.1.2 Metadatenverwaltung 45
6.1.3 Inkrementelle Versionierung 45
6.1.4 Thread-safe 46
6.1.5 Neueste Version 46
6.2 Fazit 46

A Quellcode **48**
A.1 Quelltexte der Kernelmodifikationen 48
A.1.1 Erweiterungen im VFS 48
A.1.2 Erweiterungen im Ext2-Dateisystem 64
A.2 Quelltexte der Usermode-Tools 68
A.2.1 Verwalten von DirMaxV 68
A.2.2 Aufräumen mithilfe von Purge 70

Abkürzungsverzeichnis **81**

Abbildungsverzeichnis **83**

Literaturverzeichnis **84**

KAPITEL 1

Einleitung

Neben der Speicher-, Prozess- und Geräteverwaltung gehört das Speichern und Verwalten von Daten zu den wichtigsten Aufgaben eines Betriebssystems. Diese werden in Form von Dateien organisiert und sind über einen Namen adressierbar. Der hierfür verantwortliche Code kann unter dem Begriff Dateisystem zusammengefasst werden.

In diesem Kapitel wird zunächst der Begriff des Dateisystems eingeführt. Dabei werden allgemeine Aufgaben definiert und eine Klassifizierung vorgenommen. Anschließend erfolgt eine Motivation für eine Versionsverwaltung durch das Betriebssystem sowie die Darlegung der Aufgabenstellung dieser Semesterarbeit.

1.1 Dateisystem

Das Dateisystem, als Bestandteil des Betriebssystems, dient der Abstraktion der Anwendungen gegenüber der Hardware.

Während zur Applikation hin eine möglichst einheitliche, abstrakte Schnittstelle zur persistenten Verwaltung von Informationen existieren sollte, so müssen auf Hardwareebene die unterschiedlichsten Speichermedien und Geräte unterstützt werden. Mechanisch bedingt, bringen diese Laufwerke verschiedenste Leistungseigenschaften mit und benötigen eine auf sie zugeschnittene Ansteuerung. Wie Abbildung 1.1 zeigt, ist dies jedoch nicht mehr Teil des Dateisystems, sondern wird von den Gerätetreibern erledigt. Die Hardware repräsentiert sich dabei gegenüber dem Betriebssystem als eine Ansammlung von Blöcken, wobei jeder Block mehrere Bytes zusammenfasst. Die Abbildung zeigt außerdem eine weitere Schicht, die Systembibliothek, die häufig benötigte Funktionen zur Verfügung stellt und über die die Anwendung mit Betriebssystem kommuniziert.

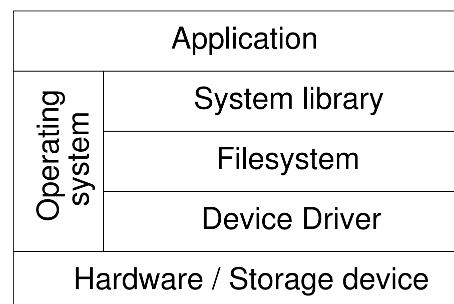


Abbildung 1.1: *Einordnung des Dateisystems*

Nach dieser Einordnung des Dateisystem innerhalb des Betriebssystems sollen nun die eigentlichen Aufgaben besprochen werden.

1.1.1 Aufgaben von Dateisystemen

Ein Dateisystem muss Dateien hinzufügen, modifizieren und löschen können. Dabei gilt es zunächst den vom Benutzer spezifizierten Name aufzulösen und den eigentlichen Daten zuzuordnen. Da Dateien eine unterschiedliche Länge besitzen und somit eine unterschiedliche Anzahl von Speicherblöcken belegen, können Daten und Dateiname nicht in einer einzigen Struktur gespeichert, sondern müssen separat verwaltet werden. Neben dieser Assoziation zwischen Bezeichner und belegten Speicherblöcken müssen aber auch die noch freien Blöcke bekannt sein, um sie nicht aufwändig suchen zu müssen.

Neben der direkten Dateimanipulation gehört die Verwaltung von Metainformationen zu den Aufgaben eines Dateisystems. So ist es üblich, den Zeitpunkt der letzten Veränderung sowie des letzten Zugriffs auf eine Datei zu speichern. Auf Mehrbenutzersystemen kommen zusätzliche Attribute wie Besitzer und Zugriffsrechte dazu. Selbstverständlich muss in diesem Fall das Dateisystem sicherstellen, dass nur autorisierte Nutzer Zugriff auf eine Datei erhalten.

Können in einem Betriebssystem mehrere Programme gleichzeitig laufen, so muss das Dateisystem verhindern, dass diese zeitgleich schreibend auf eine Datei zugreifen. Neben dieser Fehlerquelle kann es aber auch zu Inkonsistenzen kommen wenn z.B. während eines Schreibvorgangs die Stromversorgung zusammenbricht. Auch können ganze Blöcke auf dem Speichermedium ausfallen und deren Inhalt verloren gehen. Moderne Dateisysteme sind in der Lage, defekte Blöcke zu erkennen, und speichern wichtige Verwaltungsinformationen redundant ab.

Zusammenfassend können die folgenden Aufgaben eines Dateisystems definiert werden:

- Verwalten von freien und belegten Speicherblöcken
- Auslesen und Manipulieren von Dateien
- Verwaltung von dateispezifischen Metainformationen
- Zugriffskontrolle/Rechteverwaltung
- Ausschluss gleichzeitigen Zugriffs
- Erkennen und Ausgleichen von Inkonsistenzen sowie Fehlern der Hardware

1.1.2 Dateisystemtypen

Im Laufe der Zeit haben sich verschiedene Varianten von Dateisystemen herausgebildet, die neben den eben beschriebenen Anforderungen spezielle Aufgaben erfüllen und in diesem Abschnitt unterschieden werden sollen.

1.1.2.1 Hierarchische Dateisysteme

In heutigen Computersystemen existieren mehrere 1.000 bis 100.000 Dateien, wobei das Betriebssystem selbst einen großen Anteil leistet. Nach oben existiert keine Grenze. Aus Gründen der Übersicht und des Vermeidens von Nameskonflikten können diese nicht in einer Ebene verwaltet werden. Als Lösung hat sich eine hierarchische Verwaltung in Form eines Verzeichnisbaums durchgesetzt. Die Knoten werden durch Verzeichnisse

repräsentiert, die beliebig viele Unterknoten und Blätter – die Dateien mit den eigentlichen Daten – enthalten können. Der Pfad durch diesen Baum zusammen mit dem Dateinamen identifiziert eine Datei eineindeutig.

Innerhalb der hierarchischen Dateisysteme existieren eine Reihe von Implementationen, die verschiedenste Leistungsmerkmale besitzen. So speichern CDROM-Dateisysteme[1] Daten redundant, um einen gewissen Grad an Oberflächenbeschädigung ausgleichen zu können. Andere Systeme kennen z.B. Links[2] oder Access Control Lists (ACL), die eine feingranulare Einstellung von Zugriffsrechten ermöglichen.

Moderne Erweiterungen sind Journalfunktionen[3], Verschlüsselung und Komprimierung der Daten auf dem Speichermedium sowie die Begrenzung des Speicherplatzes pro Benutzer über so genannte Quota.

Neben dem Funktionsumfang können innerhalb der hierarchischen Dateisysteme noch die folgenden Untergruppen unterschieden werden:

- **Netzwerkdateisystem** – Ein Netzwerkdateisystem beherrscht den Zugriff auf Ressourcen über die Rechnergrenzen hinweg. Dabei wird von einem Server ein Verzeichnis exportiert und von einem Client importiert. Für den Benutzer gestaltet sich der Zugriff transparent[4], da die Schreib- und Leseaktionen vom Dateisystem auf Netzwerknachrichten abgebildet werden. Vertreter dieser Kategorie sind NFS, AFS oder SAMBA.

- **Virtuelles Dateisystem** - Diese Art von Systemen repräsentieren sich gegenüber dem Anwender als normales Dateisystem im Sinne einer Verzeichnishierarchie und einer Dateisemantik. Auf unterster Ebene kommunizieren sie jedoch nicht mit einem Speichermedium sondern Programmen unterschiedlichster Art.

  So existiert z.B. im Linuxkernel das `proc`-Dateisystem, das der Benutzer zur Abfrage von Statusinformationen und zur Steuerung von Betriebssystemfunktionen benutzen kann, wobei er lediglich Operationen auf einer Datei ausführt. Des Weiteren gibt es das Projekt FUSE [2], das eine einfache Implementation von Dateisystemen auf Basis bestehender Programme und Kommunikationstechniken ermöglicht. So existieren z.B. Exoten wie GMailFS [4], welches zur Speicherung von Daten einen kostenlosen GMail-Account [3] benutzt und diese über E-Mail Nachrichten transportiert.

1.1.2.2 Datenbank-Dateisysteme

Als weitere Kategorie sind Dateisysteme zu nennen, die ihre Daten in einer Datenbank ablegen. Den Daten zugehörig werden Metainformationen wie Dateigröße, Zeitstempel und Dateityp gespeichert. Die Abfrage geschieht mithilfe von SQL-Anfragen.

Datenbank-Dateisysteme bringen hervorragende Such- und Filtermöglichkeiten mit, da Dateiattribute indiziert werden können. Im Gegensatz zu hierarchischen Dateisystemen müssen so bei einer Suche nicht komplette Verzeichnisbäume durchlaufen werden.

[1] z.B. `ISO9660` und dessen Nachfolger `Joliet` und `Rockridge`

[2] Hierbei handelt es sich um Verzeichniseinträge, die einen eigenen Namen besitzen, aber nur auf eine andere Datei oder einen weiteren Link verweisen, und deren Auflösung vom Dateisystem übernommen wird.

[3] Während der Veränderung einer Datei werden alle Aktionen in einem Journal festgehalten. Konnte die Manipulation nicht vollständig durchgeführt werden, so können anhand dieser Logdatei alle Modifikationen rückgängig gemacht werden, so dass ein konsistenter Zustand gewährleistet ist.

[4] An dieser Stelle ist die Zugriffssemantik gemeint. Der Datendurchsatz und die Reaktionszeit wird im Allgemeinen schlechter im Vergleich zu einem lokalen Zugriff sein.

Die große Herausforderung stellt jedoch die Schnittstelle zum Benutzer dar. Einerseits sollten Abfragen in natürlicher Sprache möglich sein ohne eine komplizierte Syntax lernen zu müssen. Andererseits bedarf es beim Ablegen von Daten der Kooperation des Benutzers, welcher die zu speichernden Informationen klassifizieren muss.

Vertreter von Datenbank-Dateisystemen sind WinFS [11] und GNOME Storage [5]. Während WinFS auf dem hierarchischen Dateisystem NTFS aufsetzt und die Datenbankfunktionalitäten in übergeordneten Schichten nachbildet, benutzt GNOME Storge mit Postgres SQL eine echte Datenbank. Eine interessante Eigenschaft ist dabei z.B. die Netzwerktransparenz, die ohne zusätzlichen Aufwand zur Verfügung steht, da SQL Anfragen an lokale oder entfernte Datenbankserver gestellt werden können. Auch wurde eine Abwärtskompatibilität zu existierenden Anwendungen bedacht. Eine Kompatibilitätsschicht wandelt Dateien in ein strukturiertes Format um, wenn sie in den GNOME Storage kopiert werden und erzeugt in entgegengesetzter Richtung wieder herkömmliche Dateien.

1.1.2.3 Grid-Dateisysteme

Ein Grid ist laut Buyya [13] ein paralleles und verteiltes System, das es erlaubt geografisch verteilte und autonome Ressourcen zu teilen, auszuwählen und zusammenzufassen. Dies kann dabei dynamisch zur Laufzeit und zwar in Abhängigkeit von der Verfügbarkeit, Fähigkeiten, Leistung und Kosten geschehen.

Anders als bei einem Cluster befinden sich die beteiligten Rechner nicht in einer administrativen Domäne, werden also von unterschiedlichen Organisationen verwaltet. Dies führt zu einer stark heterogenen Infrastruktur. Auch müssen ganz andere Sicherheitsmechanismen entwickelt werden, da die Kommunikation zwischen Organisationen über unsichere Kanäle statt findet.

In diesem Kontext ein gridüberspannendes Dateisystem zu betreiben, stellt eine besondere Herausforderung dar. So wollen z.B. Benutzer einen eigenen, vor fremden Zugriffen geschützten Bereich im Grid haben, auf dem sie mit einer lokalen Zugriffssemantik arbeiten können. Der Zugriff soll dabei von überall möglich sein und den Ort der eigentlichen Datenspeicherung abstrahieren. Dies ist notwendig um z.B. bei der Migration eines Programms auf einen anderen Cluster bzw. Site eine ähnliche Ausführungsumgebung bereitstellen zu können, bei der die verwendeten Pfade übereinstimmen.

Im Unterschied zu den im Abschnitt 1.1.2.1 auf Seite 2 vorgestellten Netzwerkdateisystemen benötigt das Grid-Dateisystem einen globalen Namensraum, der alle Dateisysteme auf den beteiligten Clustern überspannt. Des Weiteren müssen Hürden wie Firewalls und Gateways überwunden werden. Auch wird wohl kaum ein Administrator ein NFS-Verzeichnis global exportieren.

Erste Schritte in Richtung eines Grid-Dateisystems stellen z.B. die Projekte GridNFS [7] und GridFTP, [6] welche auf der Globus-Infrastruktur [9] aufbauen.

1.2 Dateiversionierung

Im Abschnitt 1.1.2.1 auf Seite 2 erfolgte bereits ein Überblick über die Funktionsvielfalt von hierarchischen Dateisystemen. Jedoch wurde ein Merkmal, die Versionierung, noch nicht erwähnt. Wann immer Daten verändert werden, so kann es wüschenswert sein ältere Versionen wiederherzustellen bzw. den Verlauf der Veränderungen nachvollziehen zu können. Hierfür gibt es verschiedene Ansätze. So kann die Versionierung mithilfe

von externen Tools geschehen oder über Mechanismen realisiert werden, die das Dateisystem selbst bereitstellt.

Zu der ersten Kategorie gehören externe Systeme wie das Concurrent Versions System [1] und Subversion [8]. Sie verwalten ein zentrales Repository, welches auch über Netzwerkgrenzen hinweg angesprochen werden kann. Neben der Speicherung von verschieden Versionen einer Datei können u.a. Metainformationen wie Art der Veränderungen gespeichert, neue Versionszweige von älteren Versionen abgeleitet oder Unterschiede zwischen einer lokalen Kopie einer Datei und dem Repository abgeglichen werden. Sämtliche Aktionen müssen hierbei vom Benutzer mithilfe eines Programms explizit durchgeführt werden.

Bei der zweiten Gruppe, der Versionierung durch das Dateisystem, gibt es die folgenden Vorgehensweisen:

- Explizite Versionierung – Frühere Dateisysteme wie das Cedar File System [15] und 3DFS [14] unterstützten eine Dateiversionierung über spezielle Tools bzw. Kommandos.

  Im CFS musste der Benutzer die Datei, die er modifizieren wollte, auf einen lokalen Datenträger kopieren. Nach der Bearbeitung wurde sie zur Versionierung zurück auf den entfernten Server transferiert. Dieser erzeugte dann eine unveränderliche Version.

  3DFS stellt eine Bibliothek mit Funktionen zur Versionierung bereit. Als Beispielapplikation wird das Version Control System (VCS) mit einer kommandoähnlichen Syntax, etwa `checkin` oder `checkout`, verwendet. Verschiedene Versionen einer Datei werden intern in einem Verzeichnis verwaltet. Um zu entscheiden welche Version innerhalb des Verzeichnisses gerade bearbeitet wird, hält jeder Prozess eine Versionstabelle für die Abbildung auf die gewünschte Datei.

  Der größte Nachteil dieser Semantik besteht darin, dass sie nicht transparent gegenüber dem Benutzer ist. Entsprechende Kommandos müssen erlernt werden. Die Versionierung bedarf der Kooperation des Anwenders.

- Snapshots – Snapshots oder Checkpoints sind eine weitere Methode, um Versionierung innerhalb eines Dateisystems zu unterstützen. Diese werden hauptsächlich von Backupsystemen eingesetzt, wobei es zur periodischen Erstellung von Abbildern des Dateisystems kommt. Diese Snapshots werden dem Benutzer online bereitgestellt, wodurch ältere Versionen abrufbar sind.

  Systeme jener Art haben einige entscheidende Nachteile. Das größte Problem besteht darin, dass Änderungen zwischen zwei Snapshots nicht wiederherstellbar sind. Auch werden alle Dateien gleich behandelt, obwohl nicht jede Datei gleichhäufig benutzt wird. Wenn Speicher freigegeben werden muss, können nur komplette Snapshots gelöscht werden, was meist die Aufgabe eines Administrators ist.

  Systeme mit einem Snapshot-Mechanismus sind vor allem AFS [18], Plan-9 [19] und Ext3COW [21]. Ext3COW erweitert die Metadatenstruktur des Extended 3 File System (ext3) um eine Versionsunterstützung. Jeder Snapshot erhält eine Nummer vom Typ `unsigned long`, die `epoch`. Anhand dieser können frühere Versionen wiederhergestellt werden.

- Copy On Write (COW) – Eine weitere Variante ist die COW-Semantik, welche sich vollkommen transparent gegenüber dem Benutzer verhält. Hierbei wird bei jedem schreibenden Zugriff auf eine Datei zunächst eine weitere Version erstellt

und diese an Stelle der ursprünglichen Datei bearbeitet. Vertreter dieser Kategorie sind VMS [17], welches in Kapitel 2 noch ausführlich besprochen wird, sowie das Elephant File System [16]. Elephant FS erzeugt eine neue Version einer geöffneten Datei, wenn diese zum ersten mal geschrieben werden soll. Es werden auch mehrere Regeln zur Verwaltung der Versionen bereitgestellt. Die Regel `keep one` schaltet Versionierung ab, es wird nur eine Version bearbeitet. Durch `keep all` werden alle Versionen einer Datei behalten. Um Versionen über einen gewissen Zeitraum beizubehalten muss `keep safe` verwendet werden. Die letzte Regel `keep landmark` verwaltet nur wichtige Versionen einer Datei. Dazu kann der Benutzer oder das System spezielle Versionen markieren. Letzteres verwendet dazu eine Heuristik. Des Weiteren ist es möglich, Speicherplatz für die Versionierung festzulegen und somit indirekt auch eine Kontrolle über die Anzahl der Versionen zu haben.

Alle vorgestellten Varianten können außerdem dahingehend unterschieden werden, ob beim Anlegen der neuen Version eine komplette Kopie des Originals erstellt oder nur die Veränderungen gespeichert werden. Während beim ersten Verfahren mehr Speicherplatz und Zeit für das Erstellen der Kopie benötigt wird, so gestaltet sich der Zugriff und die Manipulation bei der inkrementellen Variante schwieriger. Wird z.B. eine Zwischenversion entfernt, so müssen alle Änderungen, die in dieser Variante enthalten waren, auf die nächst höhere Versionsstufe gebracht werden, um diese nicht zu verändern.

Abgesehen von den technischen Eigenschaften kann die Repräsentation der Versionsverwaltung gegenüber dem Benutzer unterschieden werden. So bieten externe Programme wie CVS eine riesige Funktionsvielfalt, deren Bedienung jedoch erst erlernt werden muss. Eine Versionierung durch das Dateisystem ist zwar unflexibel und auf wenige Funktionen beschränkt, bietet jedoch eine einfache Schnittstelle. Sicherlich unterstützen auch ausgewählte Anwendungen den Benutzer im Umgang mit einem CVS-System, so ist dennoch die Versionierung durch das Dateisystem generischer, da alle Programme, welche Dateien manipulieren, diese nutzen können.

1.3 Aufgabenstellung

Die Aufgabenstellung dieser Semesterarbeit besteht aus der Portierung der Versionssemantik des VMS Dateisystems für das Betriebssystem Linux. Dabei gilt es zunächst Strukturen und Mechanismen des Dateisystems von VMS zu analysieren und dessen Versionssemantik zu beschreiben. Anschließend soll die Dateisystemstruktur von Linux untersucht werden und davon ausgehend Umsetzungsmöglichkeiten diskutiert werden. Insbesondere muss geklärt werden, in welchem Moment eine neue Version angelegt wird und welche Möglichkeiten zur Speicherung der maximalen Anzahl von Versionsnummern bestehen.

Nach diesem Schritt hat die Implementation zu erfolgen. Sie umfasst die Erweiterungen im Dateisystem sowie das Erstellen von Programmen, die als Schnittstelle für den Benutzer dienen, um die maximale Anzahl von Versionen einer Datei festzulegen und gegebenenfalls ältere Versionen löschen zu können. Anschließend gilt es die Dateisystemerweiterungen einer Performanceanalyse zu unterziehen. Anhand von Messungen soll der Overhead bei lesenden und schreibenden Zugriffen ermittelt werden.

Nicht zuletzt ist die Erstellung dieser Dokumentation Teil der Aufgabenstellung.

KAPITEL 2

Versionierung in VMS

Nachdem im vorangegangenen Kapitel eine Einführung in Dateisysteme stattgefunden hat und die Aufgabenstellung erläutert wurde, soll nun die Dateiverwaltung des Betriebssystems VMS untersucht werden, dessen Konzepte und Mechanismen Grundlage für die weiteren Entwicklungen sind.

2.1 VMS-Dateisysteme

Files-11, On-Disk Structure, ist ein Dateisystem, welches in VMS (OpenVMS) und in einer einfachen Variante in RSX-11-Systemen Verwendung findet. Es gehört zu den hierarchischen Dateisystemen und besitzt Eigenschaften wie Zugriffsbeschränkungen über Access Control Lists, datensatzorientierte Ein- und Ausgabe[1], Zugriff auf verteilte Netzwerkressourcen sowie die Versionierung von Dateien.

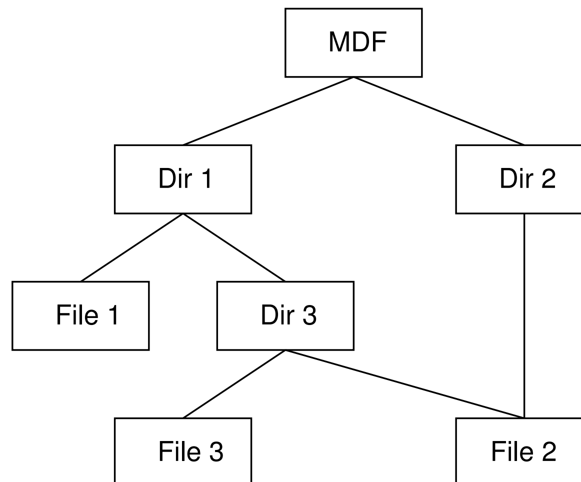
Unter der Bezeichnung Files-11 sind fünf verschiedene Dateisysteme zusammengefasst, die so genannten On-Disk Structures (ODS) Level 1 bis Level 5. Unterstützt werden sie über nebenläufige Prozesse, jeweils einen für jeden ODS Level.

- ODS-1 ist ein flaches Dateisystem. Dieses wurde von RSX-11 benutzt und durch ältere VAX-Systeme unterstützt, um die Kompatibilität zu RSX gewährleisten zu können.
- ODS-2 ist das Standard-Dateisystem für VMS.
- ODS-3 und ODS-4 wurden eingeführt um ISO 9660 und High-Sierra-Dateisysteme zu unterstützen.
- ODS-5 ist eine erweiterte Version von ODS-2. Es wurde im Zusammenhang mit dem „Windows NT Affinity Project“ entwickelt, um Dateizugriffe auch von Nicht-VMS-Systemen, wie Windows NT, zu ermöglichen.

[1] Im Gegensatz zur Ein- und Ausgabe auf Basis von Datenblöcken werden hier nicht einfache Bytestreams abgelegt. Vielmehr ist die typisierte Speicherung von Daten möglich.

2.1.1 Verzeichnisstruktur

Alle Dateien und Verzeichnisse eines Files-11-Dateisystems sind hierarchisch in einer baumartigen Struktur organisiert. Abbildung 2.1 zeigt einen möglichen Verzeichnisbaum. Der Wurzelknoten wird als Master File Directory (MFD) bezeichnet. Eine Datei bzw. ein Verzeichnis kann dabei in einem oder mehreren Verzeichnissen liegen. So befindet sich die Datei `File2` sowohl im Verzeichnis `Dir2` als auch in `Dir3`. Wird die Datei aus einem der Verzeichnisse gelöscht, so bleibt sie für andere Verzeichnisse erhalten.



Quelle: [17, S. 123]

Abbildung 2.1: *Verzeichnishierarchie*

2.1.2 Laufwerksorganisation

VMS kann mehrere Laufwerke verwalten, wobei jedes ein eigenes, unabhängiges Dateisystem besitzt. Dabei kann es sich um lokale oder in einem Netzwerk verteilte Platten handeln. Nicht private Platten werden mit allen Clusterknoten geteilt, private Platten sind hingegen nur für lokale Benutzer oder Prozesse auf einem Knoten verfügbar. Der Zugriff auf im Netzwerk verteilte Dateien wird durch den OpenVMS Distributed Lock Manager, als Bestandteil des Dateisystems, organisiert.

Festplatten werden über einen physikalischen oder benutzerdefinierten Namen identifiziert. Die Dateisysteme jeder Platte sind hierarchisch strukturiert. Eine Ausnahme bildet dabei ODS-1. Das Standardformat für einen Dateinamen besteht aus einem Namen für den Clusterknoten, Benutzername und Passwort, Gerätenamen, Verzeichnis- und Dateiname sowie Dateiendung und Versionsnummer.

```
NODE "user pass" ::device:[dir.subdir]filename.type;ver
```

Jede Datei besitzt eine Versionsnummer, welche initial 1 ist. Die Nummer wird bei jeder Schreiboperation inkrementiert. Alte Versionen einer Datei werden nur automatisch gelöscht wenn das Versionslimit erreicht wurde. Alte Versionen werden somit nicht überschrieben und können jederzeit wiederhergestellt werden. Das Limit für Versionsnummern definiert VMS mit 32767 ($2^{15} - 1$).

ODS-2 unterstützt maximal eine Tiefe von acht Unterverzeichnissen. Für den Dateinamen sowie für die Dateiendung sind maximal 39 alphanumerische, großgeschriebene

Zeichen zulässig. ODS-5 erweitert den Zeichensatz um Kleinschreibung und annähernd alle druckbaren ASCII-Zeichen, ISO Latin-1 und Unicode, sowie um eine unbegrenzte Tiefe von Unterverzeichnissen.

2.1.3 Record Management Services

Bei dem Record Management Services (RMS) handelt es sich um die Ein- und Ausgabeschicht des VMS-Betriebssystems. RMS bietet die Unterstützung für eine Verwaltung und Bearbeitung von strukturierten Dateien, wie etwa Record-basierte und indexbasierte Datenbankdateien. Das VMS-Dateisystem im Zusammenhang mit RMS erweitert damit den einfachen Zugriff auf Dateien via Bytestreams. Jede Datei innerhalb von Files-11 kann man sich als Datenbank, bestehen aus einer Vielzahl von Einträgen, vorstellen. Eine Textdatei ist zum Beispiel eine Liste von Einträgen, den Zeilen, getrennt durch einen Zeilenumbruch.
Folgende Formate von strukturierten Dateien sind definiert [17, S. 398]:

- Records fester Länge
- Records variabler Länge und einem Bytecount
- Records variabler Länge und einem terminalen Zeichen
- Streams

Für Record-basierte Datenbankdateien werden die folgenden vier Zugriffsarten unterschieden[17, S. 412ff]:

1. Sequentieller Zugriff – Unter Angabe eines initialen Datensatzes können alle weiteren Record-Einträge nacheinander bis zum Dateiende ausgelesen werden.
2. Relative Record Number – Es findet ein Zugriff auf den n-ten Eintrag relativ zum Dateianfang statt.
3. Record File Address (RFA) – In Abhängigkeit zum Dateianfang wird direkt über eine relative Adresse auf einen Datensatz zugegriffen.
4. Indexierter Zugriff – Unabhängig vom Dateianfang werden in einer so genannten Key-Value-Map Schlüsseleinträge verwaltet, die durch eine bijektive Abbildung einer physikalischen Adresse auf dem Datenträger zugeordnet sind.

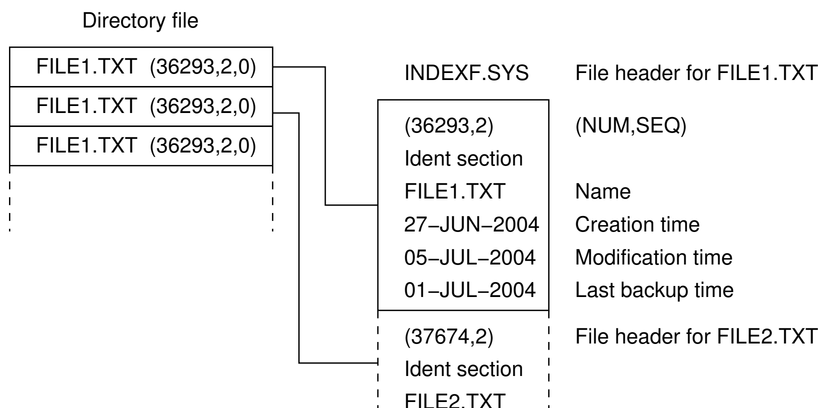
2.1.4 Physikalisches Layout – On-Disk Structure

ODS repräsentiert das Dateisystem als ein Feld von Blöcken. Die Blockgröße beträgt dabei 512 Byte auf dem physikalischen Datenträger. Die Blöcke werden zu Clustern zusammengefasst. Anfangs bestand ein Cluster aus drei Blöcken. Dies wurde mit dem Aufkommen von größeren Festplatten erhöht. Im Idealfall wird eine Datei in nebeneinander liegenden Blöcken abgelegt. Aufgrund von Fragmentation sowie Größe der Datei kann diese jedoch auch über mehrere nicht zusammenhängende Volumen verteilt werden. Durch die Möglichkeit mehrere Platten zu einem Volume Set zusammenzufassen, kann es auch vorkommen, dass Teile einer Datei auf mehreren physikalischen Platten abgelegt werden.

Jede Datei eines Files-11 Systems besitzt eine eindeutige Identifikationsnummer, die File Identification (FID). Sie setzt sich aus drei Nummern, der File Number (NUM), der File Sequence Number (SEQ) und der Relative Volume Number (RVN), zusammen. Die NUM gibt die Position innerhalb der `INDEXF.SYS` an, an der sich die Metadaten für die Datei befinden. Bei SEQ handelt es sich um eine Nummer, die inkrementiert wird, falls eine Datei gelöscht und eine andere Datei mit dem selben Eintrag in der `INDEXF.SYS` erzeugt wird. Die RVN gibt das Volume an, auf dem sich die Datei befindet, sofern ein Volume Set verwendet wird.

2.1.5 Struktur der Verzeichnisse

Verzeichnisse in ODS Volumes sind spezielle Dateien, die eine Liste von Dateinamen mit Versionsnummer und der dazugehörigen FID enthalten. Das Wurzelverzeichnis MFD, enthält alle auf einem Volume befindlichen Dateien direkt bzw. indirekt über Indizies.



Quelle: [17, S. 123]

Abbildung 2.2: *Abbildung von Verzeichniseinträgen in* `INDEXF.SYS`

2.1.6 Master File Directory

Oberstes Verzeichnis eines jeden ODS-Dateisystems ist das MFD, welches alle Top-Level-Verzeichnisdateien (sich selbst eingeschlossen) und spezielle Systemdateien zur Speicherung von Dateisysteminformationen enthält.

Abbildung 2.3 zeigt eine Übersicht der Systemdateien des MFD.

2.1.7 Indexdatei - INDEXF.SYS

Die Indexdatei `INDEXF.SYS` enthält einfache Informationen über ein Files-11 Volume Set. Block 1, der Boot-Block, speichert die Position des primären Bootabbilds. Er befindet sich auch auf nicht boot-fähigen Volumen. Nach dem Boot-Block kommt der Home-Block. Dieser beinhaltet den Volumennamen, den User Identification Code (UIC) des Volumens sowie Information über dessen Schutz. Der Home-Block wird redundant abgelegt, um dem Benutzer die Möglichkeit zu geben diesen wiederherzustellen. Der Rest der Indexdatei besteht aus Dateiheadern, welche Erweiterungen von Dateien, UICs und

Dateiname	Funktion
`INDEXF.SYS;1`	Index file
`BITMAP.SYS;1`	Storage bitmap file
`BADBLK.SYS;1`	Bad block file
`000000.DIR;1`	MFD file
`CORIMG.SYS;1`	Core image file
`VOLSET.SYS;1`	Volume set list file (ODS-2/5)
`CONTIN.SYS;1`	Continuation file (ODS-2/5)
`BACKUP.SYS;1`	Backup log file (ODS-2/5)
`BADLOG.SYS;1`	Pending bad block (ODS-2/5)
`SECURITY.SYS;1`	Volume security profile (ODS-2/5)
`QUOTA.SYS;1`	Quota file (ODS-2/5)

Abbildung 2.3: *VMS Systemdateien und ihre Funktion*

ACLs beschreiben. Jeder Block eines Dateiheaders besitzt eine feste Länge, kann aber aus Einträgen mit fester und variabler Länge bestehen.

- Header – Er beinhaltet die NUM und die SEQ, Sicherheitsinformationen sowie die Positionen anderer Header.
- Ident – Metadaten, wie Dateiname, Zeit der Erzeugung und Modifikation sowie die Zeit des letzten Updates werden hier gespeichert.
- Map – Map beschreibt die Abbildung von physikalischen auf virtuelle Blöcke einer Datei.
- ACL – Sie erlauben eine feingranulare Zugriffs- und Rechteverwaltung.
- Reserved – Speicherplatz am Ende eines Headers kann für benutzerspezifische Informationen verwendet werden.
- Die letzten 2 Byte – In den letzten zwei Bytes eines Headers wird eine PrÃ¼fsumme zur Validierung abgelegt.

Ident area

FH2$T_FILENAME	file name	20 bytes
FH2$W_REVISION	revision number	2 bytes
FH2$Q_CREDATE	creation date and time	8 bytes
FH2$Q_REVDATE	revision date and time	8 bytes
FH2$Q_EXPDATE	expiration date and time	8 bytes
FH2$Q_BAKDATE	backup date and time	8 bytes
FH2$T_FILENAMEEXT	file name extension	66 bytes

Quelle: [17, S. 123]

Abbildung 2.4: *Teil des INDEXF.SYS Headers*

Die Datei-Header der Erweiterungen, etwa ACL oder der Map, besitzen eine flexible Größe. Dadurch ist es notwendig Offsets zu definieren, um bestimmte Header ansprechen zu können. Die Offsets `IDOFFSET`, `MAPOFFSET`, `ACOFFSET` und `ROFFSET` befinden sich am Anfang des `INDEXF.SYS` Headers.

STRUCLEV verwaltet den aktuellen Structure Level des Dateisystems und dessen Version als Low-Byte. ODS-2 besitzt z.B. den Structure Level 2. Damit wird dem Entwickler die Möglichkeit gegeben, auf Abwärtskompabilität bei unterschiedlichen Versionen zu achten. Structure Level selbst sind nicht kompatibel untereinander. Die Einträge von W_FID* korrespondieren mit NUM, SEQ und RVN. Die Variablen EXT_FID* beschreiben die Position des nächsten Headers für Erweiterungen. FILECHAR gibt an wie eine Datei zu verarbeiten ist, z.B. DIRECTORY oder BADBLOCK. Mit ACCMODE und FILEPROT werden Zugriffs- und Benutzerrechte zugewiesen.

In der Ident Area wird der Name einer Datei mit der dazugehörigen Versionsnummer, im Eintrag FILENAME, abgelegt. Zusätzlich werden verschiedenste Zeitstempel verwaltet.

2.2 Dateisystemsemantik

Der Name einer Datei innerhalb eines VMS-Dateisystems besteht aus einem Dateinamen und der Dateiendung gefolgt von einem Separatorzeichen und der Versionsnummer. Bei dem Separator handelt es sich um das Semikolonzeichen. Es können maximal 32.767 Versionen einer Datei verwaltet werden.

Die maximale Anzahl von Versionen einer Datei in einem Verzeichnis wird beim Erstellen eines neuen Verzeichnisses spezifiziert. Dazu besitzt das DCL-Kommando[2] CREATE/DIRECTORY das Attribut /VERSION_LIMIT. Damit kann angegeben werden, wie viele Versionen einer Datei es geben soll. Wird eine 0 übergeben, besitzt das Verzeichnis kein Limit, unterliegt aber dem Maximum von 32.767 Versionen. Standardmäßig wird das Versionlimit des übergeordneten Verzeichnisses gewählt. Wenn die Einstellung geändert wird, gilt dies nur für Dateien, die nach der Änderung erzeugt werden; alte Versionen behalten das ursprüngliche Limit bei.

Eine Möglichkeit das Versionslimit explizit für eine Datei zu setzen, bietet die Änderung einer entsprechenden Dateieigenschaft. Dazu wird von VMS (OpenVMS) das DCL-Kommando SET FILE bereitgestellt. Im Zusammenhang mit der Option /VERSION_LIMIT kann das Versionslimit verändert werden.

Wenn der Benutzer eine Datei erzeugen will wird automatisch eine Versionsnummer vergeben. Existiert die Datei bereits vergibt das System die größtmögliche Versionsnummer, sonst 1. Wird eine modifizierte Datei gespeichert, so legt die Schreiboperation eine neue Datei mit der nächstmöglichen Versionsnummer an. Eine Ausnahme von diesem Verhalten bilden Editoren. Diese lesen den Inhalt der originalen Datei in einen Puffer und legen für ihn eine temporäre Datei an. Beim Speichern der Änderungen wird nun eine neue Datei mit der nächsten Versionsnummer erstellt.

Somit entspricht das Verhalten der Versionierung unter FILES-11 dem COW-Mechanismus. Die alte Datei wird unmodifiziert beibehalten. Außerdem kann jederzeit das Attribut NO_VERSION gesetzt werden. Dieser unterdrückt die Versionierung wodurch die Originaldatei überschrieben wird.

Der Zugriff auf Dateien kann explizit durch die Angabe von Dateinamen plus der Versionsnummer erfolgen. Es werden auch noch andere Adressierungsmöglichkeiten unterstützt, um bestimmte Versionen einer Datei zu öffnen.

- Bei Angabe eines Dateinamen ohne Versionsnummer wird die aktuellste Datei

[2] Die Digital Command Language (DCL) ist die Standard-Komandozeilen-Schnittstelle der meisten DEC-Betriebssysteme.

geöffnet. Dieses Verhalten kann auch erreicht werden, indem man `0` als Versionsnummer angibt.

- `-0` öffnet die älteste Version.
- Durch Angabe von `-1` wird die Vorgängerversion der aktuellsten Datei geöffnet, bei `-2` dessen Vorgänger und so weiter. Somit kann mit dieser Methode relativ auf die Dateien zugegriffen werden, ohne dass die eigentliche Versionsnummer bekannt ist.

Beim Kopieren einer versionierten Datei sind mehrere Fälle im Bezug auf eine neue Version zu beachten. Wenn beim Copy-Kommando keine Versionsnummer für Quell- und Zieldatei angegeben wird, erhält die Zieldatei die nächstgrößere Versionsnummer einer eventuell bestehenden Datei mit demselben Namen und gleicher Endung. Besitzt die Quelldatei eine Nummer und die Zieldatei nicht, so wird diese verwendet. Wenn die Versionsnummer der Zieldatei explizit angegeben wird, wird die Datei kopiert wenn eine Datei mit höherer Version existiert und eine Warnung ausgegeben. Existiert die Version der Datei schon bricht Copy mit einer Fehlermeldung ab und die Daten werden nicht kopiert.

VMS (OpenVMS) stellt DCL-Kommandos zur Verwaltung der Versionierung von Dateien zur Verfügung. Wichtigstes dieser Kommandos ist `PURGE`. Es entfernt ältere Versionen einer bestehenden Datei, löscht dabei aber niemals alle Versionen. Standardmäßig wird die aktuellste beibehalten. Wenn als Parameter keine Datei angegeben wird, werden alle sich im aktuellen Verzeichnis befindenden Dateien bearbeitet. Mit der Option `/KEEP=number-of-versions` kann man angeben, wie viele der letzten Versionen beibehalten werden sollen. Wird die Option nicht gesetzt entspricht dies dem Standardverhalten von `PURGE`.

Weitere Optionen mit denen festgelegt werden kann, welche Versionen gelöscht werden sollen sind `/SINCE` und `/BEFORE`. Sie beziehen sich auf Zeitangaben und arbeiten unabhängig von den Versionsnummern. Mit `/EXPIRED` werden alle Dateien selektiert deren Expiration-Date abgelaufen ist. Nachdem alle älteren Versionen von Purge gelöscht wurden, werden alle erhalten gebliebenen Dateien entsprechend ihrer Erstellungsreihenfolge beginnend mit 1 durchnummeriert.

KAPITEL 3 Dateisystemimplementationen in Linux

1987 wurde von Andrew S. Tanenbaum ein minimaler Unix-Klon, Minix, entwickelt. Die Motivation hinter der Schaffung von Minix [12] lag darin, ein für Lehrzwecke minimales Betriebssystem zu entwickeln. Ziel war es Konzepte von Betriebssystemen und deren Implementationen zu veranschaulichen.

Das erste von Linux unterstützte Dateisystem war das Minix-Dateisystem, da Linus Torvalds 1991 unter dem Betriebssystem Minix den Linux-Kernel entwickelte. Ein lauffähiges Minix-System war in erster Linie notwendig um Konfigurationen zu bearbeiten, den Kernel zu compilieren und Linux zu installieren. Auch musste in den ersten Versionen von Linux ein anderes Betriebssystem vorhanden sein, um von der Festplatte booten zu können. Aus diesem Grund finden sich Gemeinsamkeiten zwischen beiden Dateisystemen und bis zur Einführung des Extended File System (Ext FS) im April 1992 wurde das Minix FS als Standarddateisystem verwendet.

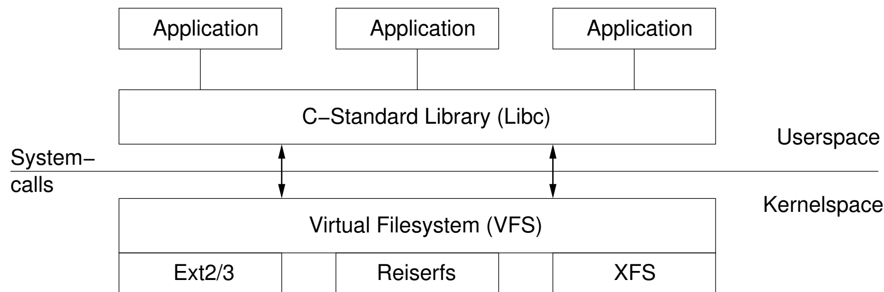
3.1 Das virtuelle Dateisystem VFS

Mit der Einführung von Ext FS für Linux wurde im Kernel eine Schnittstelle zwischen der Systembibliothek und der eigentlichen Dateisystemimplementation geschaffen, das Virtual File System (VFS). Abbildung 3.1 zeigt im oberen Teil, dem Userspace, Anwendungen die mit der Bibliothek Libc kommunizieren. Diese wendet sich bei Dateioperationen an das sich im Kernelspace befindliche VFS. Je nach Formatierung des Datenträgers, auf dem sich die Zieldatei befindet wird dateisystemspezifischer Code ausgeführt.

Durch die Schaffung des VFS konnte eine hohe Flexibilität erreicht werden. Verschiedene Dateisysteme sind über ein einheitlichen Schnittstelle ansprechbar.

Neben der Abstraktion besteht eine wesentliche Aufgabe des virtuellen Dateisystem darin generische Funktionen bereitzustellen. Sie bilden eine Basisimplementation der Schnittstelle und können von konkreten Dateisystemausprägungen als Ausgangspunkt benutzt werden. Wann immer der vorgegebene Code nicht ausreicht, um ein bestimmtes Verhalten zu realisieren, so können eigene Funktionen an Stelle der generischen benutzt werden. Abschnitt 3.1.2 auf Seite 16 wird die dafür notwendigen Mechanismen noch genauer erläutern.

Außerdem werden durch das VFS Optimierungsmechanismen bereit gestellt, die nur schwer oder mit erhöhtem Aufwand durch eine native Implementation eines Dateisys-



Quelle: [20, S. 351]

Abbildung 3.1: *VFS-Layer zur Abstraktion von Dateisystemen*

tems zu realisieren wären. Beispiel hierfür ist ein Dentry-Cache-Mechanismus[1]. Er gewährleistet das effiziente Auflösen eines Pfades zu den Dateistrukturen, den so genannten Path Lookup, und wird im Abschnitt 3.1.5 auf Seite 22 genauer behandelt.

Momentan unterstützt der Kernel mehr als 40 Dateisysteme aus den verschiedensten Betriebssystemen. Beispiele reichen von File Allocation Table (FAT) und ISO9660 bis hin zu Netzwerk- und Clusterdateisystemen, wie das Oracle Cluster File System 2 (OCFS2)[2].

3.1.1 Common File Model

In diesem Abschnitt soll ein Überblick über wichtige Dateisystemobjekte und -mechanismen gegeben werden. Eine Vertiefung liefern nachfolgende Abschnitte.

Eine der wichtigsten Designentscheidungen für Linux-Dateisysteme lautet:

„Alles ist eine Datei“

So bietet das VFS eine einheitliche Sicht auf auf die vom Dateisystem verwalteten Objekte. Egal, ob es sich dabei um Verzeichnisse, Pipes, Gerätedateien oder Links handelt, so werden diese dem Benutzer als Datei repräsentiert. Verzeichnisse zum Beispiel sind Dateien, die als Inhalt eine Liste von Dateien besitzen.

Benutzerprogramme müssen für die Arbeit mit Dateien Zugriff auf einen File Descriptor (FD) erlangen. Dieser ist eine Integer-Zahl, die durch den Kernel verwaltet wird und eine konkrete Instanz einer geöffneten Datei eindeutig identifiziert. Sie ist nur innerhalb eines Prozesses eindeutig, wodurch unterschiedliche Prozesse bei identischem File Descriptor auf verschiedenen Dateistrukturen arbeiten.

Eine wichtige Struktur innerhalb des Kernels ist die Inode. Über sie können Dateien eindeutig identifiziert werden. Jede Datei besitzt genau eine Inode, welche aus zwei wesentlichen Abschnitten besteht:

- Den Metadaten zur Speicherung von z.B. Zugriffsrechten und Zeitstempeln sowie dem
- Datenabschnitt mit Adressen von Datenblöcken.

[1] Das Wort Dentry steht für Directory Entry. Es werden also mit diesem Mechanismus Verzeichniseinträge zwischengespeichert.

[2] `http://oss.oracle.com/projects/ocfs2/`

Die Verzeichnishierarchie des Dateisystems wird durch Inoden abgebildet. Hierbei befindet sich auf oberster Ebene der Wurzelknoten bzw. das Wurzelverzeichnis. Diesem zugeordnet ist die Root-Inode, die dem System stets bekannt ist. Im Datenabschnitt der Inode befinden sich die Verzeichniseinträge und Inodenummern des Wurzelverzeichnisses. Die Nummern dienen zur eindeutigen Identifikation der Inode. So wird eine Verbindung zwischen Inode und Dateiname realisiert.

Durch einfaches Ablaufen der Einträge kann ein Pfad aufgelöst werden, um eine gewünschte Datei zu suchen. Dieser Algorithmus wird Path-Lookup genannt und im VFS durch Caches unterstützt, um das Aufsuchen häufig verwendeter Dateien zu beschleunigen.

Um Verbindungen zwischen Objekten realisieren zu können, die nicht in der Baumhierarchie beschreibbar sind, können Links verwendet werden. Es werden dabei zwei Arten unterschieden:.

- Symbolische Links – Im Datenabschnitt der Inode wird eine Zeichenkette abgelegt, die den relativen Pfad zur Zieldatei angibt. Dadurch sind Link und Zieldatei nicht fest miteinander verbunden. Wenn die Zieldatei entfernt wird bleibt der Link erhalten.
- Harte Links – Hierbei handelt es sich um Verzeichniseinträge, die eine gemeinsame Inode besitzen.

  Dabei taucht jedoch das Problem auf, dass Hardlink und Originaldatei nicht mehr voneinander unterschieden werden können. Würde beim Löschen des harten Links einfach die Inode entfernt, so wäre auch die Zieldatei verloren, da beide dieselbe Inodenummer besitzen.

  Zur Umgehung dieser Problematik wurde ein Linkcount, eingeführt der immer erhöht wird, wenn ein harter Link auf eine Datei erstellt wird. Beim Löschen wird dieser dekrementiert. Erst wenn der Zähler auf Null steht ist sichergestellt, dass die Inode nicht mehr verwendet wird und somit gelöscht werden kann.

3.1.2 Abstrahierende Schnittstelle

Um den Zugriff innerhalb der VFS-Schicht auf das darunter liegende Dateisystem zu abstrahieren, können keine festen Funktionen verwendet werden. Diese würden das Konzept des abstrahierenden Layers aufbrechen. So wurde sich bei der VFS-Implementation der Technik der Funktionszeiger bedient.

Dazu werden Strukturen von Operationen gebildet, die die Prototypen der Funktionen enthalten. Die Aufgabe der Dateisystemimplementation ist es, für diese Operationsstrukturen die zugrunde liegenden Implementationen bereitzustellen. Dabei müssen die Signaturen der Funktionen übereinstimmen. Als Beispiel hierfür sollen die Operationen auf einer Datei dienen. In der Headerdatei `<fs.h>` ist die Struktur `struct file_operations` definiert.

Listing 3.1: *Struktur* `struct file_operations` *in der Datei* `include/linux/fs.h`

```
1 struct file_operations {
2
3       struct module *owner;
4
5       loff_t (*llseek) (struct file *, loff_t, int);
6       ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
7       ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
```

```
8
9       int (*readdir) (struct file *, void *, filldir_t);
10      int (*open) (struct inode *, struct file *);
11      int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
12
13      [ 20 additional lines ]
14 };
```

Wie Auflistung 3.1 zeigt, werden eine Vielzahl von Funktionsprototypen zur Verwaltung und Bearbeitung definiert. Dazu muss eine Instanz der Struktur erstellt werden. Die einzelnen Funktionszeiger werden dann an die vom Dateisystem bereitgestellten Funktionen gebunden.

```
struct inode {
        [ ... ]
        struct file_operations  *i_fop;
        [ ... ]
};

void ext2_read_inode (struct inode * inode) {
        [ ... ]
        inode->i_fop = &ext2_file_operations;
        [ ... ]
};
```

Wie beim Vergleich der Zeilenanzahl von Auflistung 3.2 und 3.1 auffällt, ist es nicht nötig, alle Operationen zu implementieren. Des Weiteren kann man auch auf die generischen Funktionen zurückgreifen, die vom VFS bereitgestellt werden.

Listing 3.2: *Struktur* `struct ext2_file_operations` *in der Datei* `fs/ext2/file.c`

```
1  struct file_operations ext2_file_operations = {
2       .llseek   = generic_file_llseek,
3       .read     = generic_file_read,
4       .write    = generic_file_write,
5
6       [ 2 additional lines ]
7
8       .ioctl    = ext2_ioctl,
9
10      [ 7 additional lines ]
11 };
```

3.1.3 Superblock

Verschiedene Dateisysteme werden mithilfe der Superblocks verknüpft. Diese halten neben zentralen Informationen über das eigentliche Dateisystem auch Funktionszeiger zur Manipulation der Inoden. Der Kernel enthält eine Liste von Superblöcken aller im System befindlichen und aktiven Dateisysteme. Es können dabei auch mehrere Instanzen eines Dateisystemtyps abgelegt sein.

Zentrales Element eines Superblocks, der im Hauptspeicher abgelegt ist, ist eine Liste aller modifizierten Inoden. Dabei handelt es sich um Inoden, bei denen das Dirty-Flag gesetzt ist. Dadurch kann man feststellen welche Datei verändert wurde und auf

den assoziierten Datenträger geschrieben werden muss. Es wird die Liste der markierten Inoden nach einem bestimmten Zeitintervall durchlaufen und die Veränderungen durchgeschrieben.
Ein im Kernel registriertes Dateisystem besitzt einen Eintrag in der Liste vom Typ `file_system_type`. Die Struktur enthält zum Beispiel den Namen des Dateisystemtyps als String, den Besitzer sowie einen Zeiger auf ein Modul. Dieser ist nur gültig, wenn das Dateisystem als Modul geladen wurde und nicht im Kernel fest einkompiliert ist. Wesentliche Elemente sind der Zeiger auf den Kopf einer Liste von Superblöcken des Dateisystemtyps `struct list_head fs_supers` und der Funktionspointer `get_sb` zum Einlesen des Superblocks eines Dateisystems.

Beim Einhängen eines Dateisystems wird im Speicher ein neues Objekt vom Typ `struct super_block` angelegt. Dabei wird über den Funktionspointer `get_sb` die Struktur vom Datenmedium eingelesen.

Listing 3.3: *Struktur* `struct super_block` *in der Datei* `include/linux/fs.h`

```
1  struct super_block {
2        struct list_head s_list;  /* Keep this first */
3
4        unsigned long  s_blocksize;
5        unsigned long long s_maxbytes; /* Max file size */
6
7        struct file_system_type s_type;
8        struct super_operations s_op;
9        struct dentry     *s_root;
10
11       struct list_head s_inodes; /* all inodes */
12       struct list_head s_dirty; /* dirty inodes */
13       struct list_head s_files;
14
15       void           *s_fs_info; /* Filesystem private info */
16 };
```

Die Auflistung 3.3 zeigt einen Auszug aus der Struktur des Superblocks.

- `s_list` ist der Kopf auf eine Liste aller sich im System befindlichen Superblöcke. Des Weiteren gibt es noch eine Liste, die alle Superblöcke eines Dateisystemtyps zusammenfasst.
- `s_blocksize` enthält die Blockgröße des Dateisystems in Kilobyte. Der Wert ist abhängig von der Implementation des zugrunde liegenden Dateisystems.
- `s_maxbytes` gibt die maximale Größe einer Datei an, die das native Dateisystem verwalten und verarbeiten kann.
- `s_type` zeigt auf Typinformationen des Dateisystems.
- `s_root` ist ein Zeiger auf den Eintrag des globalen Wurzelverzeichnisses.
- `s_inodes` ist eine Liste aller Inoden des eingetragen Dateisystems.
- `s_dirty` ist die Liste aller modifizierten Inoden, die noch auf den Datenträger geschrieben werden müssen.
- `s_files` ist eine Liste vom Typ `struct file` und verwaltet geofffnete Dateien die zum assoziierten Dateisystem gehören. Die Liste wird verwendet um beim Aushängen des Dateisystems festzustellen, ob dieses noch benutzt wird.

Ein wichtiges Element ist `s_op`. Dieser Zeiger verweist auf eine Struktur von Funktionszeigern 3.1.2. Sie stellen Operationen zum Lesen und Schreiben von Inoden bereit. Außerdem weist die Struktur Methoden auf, um Datenträger einbinden bzw. entfernen zu können.

Listing 3.4: *Struktur* `struct super_operations` *in der Datei* `include/linux/fs.h`

```
1  struct super_operations {
2        struct inode *(*alloc_inode)(struct super_block *sb);
3        void (*destroy_inode)(struct inode *);
4
5        void (*read_inode) (struct inode *);
6
7        void (*dirty_inode) (struct inode *);
8        int (*write_inode) (struct inode *, int);
9        void (*put_inode) (struct inode *);
10       void (*drop_inode) (struct inode *);
11
12       [ 12 additional lines ]
13 };
```

Die Auflistung 3.4 zeigt einen Auszug aus der Struktur der Operationen eines Superblocks.

- `read_inode` und `write_inode` dienen zum Lesen und Schreiben von Inodedaten. Diese Methoden werden durch das darunter liegende Dateisystem implementiert.
- `dirty_inode` markiert eine Inode als verändert und zeigt somit an, dass diese modifiziert wurde. Sie wird auch in die Liste der modifizierten Inoden `s_dirty` im Superblock eingetragen.

3.1.4 Inoden

Wie schon im Abschnitt zum Common File Model (3.1.1) festgestellt, wird die Verzeichnishierarchie über Inoden modelliert. Zwischen Inoden und Dateien besteht eine eindeutige Beziehung, zu jeder Datei gibt es eine Inode mit einer Inodenummer, die eindeutig ist. Eine Ausnahme bilden die auf Seite 16 vorgestellten Hardlinks.

Die Struktur der Inode, wie sie im VFS Layer implementiert wurde, ist für die Ablage und Bearbeitung im Hauptspeicher entwickelt worden. Somit besitzt sie nur die nötigsten Elemente und unterscheidet sich stark von einer speziellen Implementation einer Inode für ein natives Dateisystem, wie etwa die `struct ext2_inode <ext2_fs.h>` von Ext2.

Listing 3.5: *Struktur* `struct inode` *in der Datei* `include/linux/fs.h`

```
1  struct inode {
2        struct hlist_node i_hash;
3        struct list_head  i_list;
4        struct list_head  i_dentry;
5        unsigned long     i_ino;
6        atomic_t          i_count;
7        umode_t           i_mode;
8        unsigned int      i_nlink;
9        loff_t            i_size;
10
11       struct timespec   i_atime;
12       struct timespec   i_mtime;
13       struct timespec   i_ctime;
```

```
14
15        [ 3 additional lines ]
16
17        unsigned long      i_blocks;
18
19        [ 5 additional lines ]
20
21        struct inode_operations *i_op;
22        struct file_operations *i_fop; /* former ->i_op->default_file_ops */
23        struct super_block *i_sb;
24
25        [ 26 additional lines ]
26 };
```

Auflistung 3.5 zeigt einen Auszug aus der Struktur der Inoden wie sie im Hauptspeicher abgelegt werden.

- Wesentliches Merkmal einer Inode ist die dazu gehörige Nummer, `i_ino`, die eindeutige Kennzahl.
- `i_count` ist ein Zähler, der angibt wie viele Prozesse auf die Inode zugreifen.
- `i_nlink` ist der Zähler der die Anzahl der Hardlinks angibt.
- Um eine Inode einer Datei von einer Inode eines Verzeichnisses zu unterscheiden hält `i_mode` den Modus der Inode.
- Viele Elemente einer Inode dienen zur Verwaltung von Statuselementen. `i_atime, i_mtime` und `i_ctime` speichern zum Beispiel Zeitstempel über den letzten Zugriff auf die assoziierte Datei, die Modifikation der Datei und die letzte Änderung an Attributen der Inode.
- Die Größe einer Datei bzw. die Dateilänge, wird in den Elementen `i_size` und `i_blocks` abgelegt. Dabei wird der Wert von `i_size` in Byte und der Wert von `i_blocks` in Blöcken gemessen.

Um auf Dateien arbeiten zu können stellt der Kernel wichtige Funktionen zur Verfügung. Die eigentliche Manipulation der Dateien muss durch das native Dateisystem geschehen. Dazu werden wieder Funktionszeiger verwendet, die eine Spezialisierung der Operationen je nach Dateisystemtyp ermöglichen.

Auch im Fall der Inode-Operationen werden vom VFS-Layer generische Funktionen bereitgestellt. Der Zeiger auf die `struct inode_operations` bzw. die `struct file_operations` Struktur befindet sich in der Inode in Form der Variablen `i_op` und `i_fop`. Viele der Bedeutungen der Funktionen lassen sich direkt aus dem Namen herleiten und bedürfen daher keiner näheren Erläuterung.

Als erstes Beispiel soll der Funktionszeiger für die Operation `rename` betrachtet werden.

```
int (*rename) (struct inode *, struct dentry *,
               struct inode *, struct dentry *);
```

Die Operation zum Umbenennen von Dateien besitzt als Parameter zwei Inoden und zwei Dentry-Strukturen, jeweils für Ziel und Quelle. Bei den Inode-Strukturen handelt es sich um die Verzeichnisse, in denen sich die Objekte befinden, die umbenannt werden sollen. Die Dentry-Strukturen geben das eigentlich zu bearbeitende Objekt an. Dabei handelt es sich um einen Eintrag aus dem Directory-Cache.

```
void (*truncate) (struct inode *);
```

Die Operation `truncate` schneidet gewisse Teile einer Datei ab. Der Funktion wird nur ein Parameter übergeben, die Datenstruktur der zu modifizierenden Inode. Die neue Größe der Inode muss dabei vor dem Aufruf über die Variable `i_size` gesetzt werden.

```
struct dentry * (*lookup) (struct inode *,struct dentry *,
                           struct nameidata*);
```

`Lookup` wird verwendet um Inoden eines Objektes anhand des Namens zu ermitteln. In den Funktionsprototypen der Operationen wird häufig die Struktur `struct dentry` verwendet. Sie ist eine einheitliche Datenstruktur, die sowohl Dateien als auch Verzeichnisse widerspiegeln kann und wird im Teilabschnitt 3.1.5 genauer beschrieben.

Ein weiteres Element in der Inode-Struktur 3.5 ist der Zeiger `i_list`. Er verweist auf den Kopf einer der durch den Kernel verwalteten Inode-Listen. Sie repräsentieren verschiedene Zustände, die eine Inode annehmen kann:

- Die Inode liegt im Speicher vor, ist aber mit keiner Datei verknüpft und wird nicht mehr aktiv verwendet. Diese Inoden werden in der Liste `inode_unused` abgelegt.
- Die Struktur befindet sich im Speicher und wird verwendet. Die Zähler `i_count` und `i_nlink` müssen größer als Null sein. Seit der letzten Synchronisierung dürfen der Dateiinhalt und die Metadaten nicht modifiziert worden sein. Hierfür wird die Liste `inode_in_use` verwendet.
- Die Inode wird aktiv verwendet und ist modifiziert worden. Die Inode ist mit dem Dirty-Flag markiert und wurde in der bereits im Abschnitt 3.1.3 vorgestellten Liste `s_dirty` abgelegt.

Neben diesen spezifischen Listen werden die Inoden noch zusätzlich in einer Hash-Tabelle abgelegt. Ein schneller Zugriff kann durch Inodenummer und Superblock realisiert werden.

Um eine Inode anhand eines gegebenen Dateinamens zu finden, wird ein `Lookup`-Mechanismus verwendet, der als Ergebnis sowie zur Übergabe von Parametern die Datenstruktur `nameidata` verwendet.

Listing 3.6: *Struktur* `struct nameidata` *in der Datei* `include/linux/namei.h`

```
1 struct nameidata {
2
3       struct dentry *dentry;
4       struct vfsmount *mnt;
5       struct qstr last;
6       unsigned int flags;
7
8       [ 8 additional lines ]
9 };
```

- `Dentry` und `mnt` enthalten nach einem Lookup die Daten des gesuchten Dateisystemeintrags, die aufgelösten Einhängepunkte des Dateisystems sowie den Eintrag im Dentry-Cache.
- `Last` gibt den Namen an, der nachgeschlagen werden soll.

Um einen Pfad- oder Dateinamen nachzuschlagen, wird die Funktion `path_lookup` vom Kernel verwendet.

3.1.5 Dentry-Cache

Durch langsame Speichermedien kann es relativ lange dauern, bis ein Dateiname aufgelöst wurde und die dazugehörige Inode ermittelt ist. Der Mechanismus um Ergebnisse von Lookup-Operationen zwischenzuspeichern nennt sich Dentry-Cache. Nachdem Daten von einer Datei oder einem Verzeichnis geladen wurden, wird eine Instanz der Struktur `dentry (<dcache.h>)` zum Cachen erzeugt. Die Instanzen von `dentry` bilden dabei die Struktur des Dateisystems ab. Dazu besitzt ein Dentry-Eintrag den Kopf einer Liste von weiteren, ihm untergeordneten Einträgen des Typs `dentry`.

Listing 3.7: ***Struktur*** `struct dentry` ***in der Datei*** `include/linux/dcache.h`

```
1  struct dentry {
2
3        [ 3 additional lines ]
4
5        struct inode *d_inode;
6        struct hlist_node d_hash; /* lookup hash list */
7        struct dentry *d_parent; /* parent directory */
8        struct qstr d_name;
9
10       [ 12 additonal lines ]
11 };
```

Auflistung 3.7 zeigt einen Auszug aus der Struktur `dentry`.

- `d_inode` ist ein Zeiger auf die assoziierte Instanz einer Inode. Der Zeiger kann auch den Wert `NULL` annehmen, wenn eine Dentry-Instanz für einen nicht vorhandenen Dateinamen existiert. Dieses Verhalten wurde implementiert, um das Auflösen nicht vorhandener Dateien zu beschleunigen.
- Der Name einer Datei wird durch `d_name` repräsentiert.
- `d_parent` ist ein Zeiger auf das übergeordnete Verzeichnis. Beim Wurzelverzeichnis verweist der Zeiger auf sich selbst.

Alle aktiven Instanzen der Dentry-Struktur werden mithilfe einer Hash-Tabelle verwaltet. Des Weiteren hält der Kernel eine zweite Liste von Dentry-Instanzen, `dentry_unused`, ähnlich der Zustandslisten in der die Inoden verwaltet werden. Ziel des Dentry-Caches ist es, die Arbeit mit dem Dateisystem zu vereinfachen und zu beschleunigen indem die Kommunikation zwischen dem VFS-Layer und dem Dateisystem auf ein Minimum reduziert wird. Jede Anfrage an ein Dateisystem resultiert in einem neuen Dentry-Eintrag, der danach zwischengespeichert wird, um den nächsten ähnlichen Aufruf zu beschleunigen. Der Dentry-Cache besteht aus zwei Listen.

1. Eine Hash-Tabelle in der sich alle Dentry-Instanzen befinden.
2. Eine LRU-Liste, bestehend aus Dentry-Einträgen, die nicht mehr verwendet werden. Diese werden nach einer bestimmten Zeit gelöscht. Einträge die sich in der LRU-Liste befinden, sind auch in der Hash-Tabelle vorhanden. Dadurch ist ein effizientes Auffinden der Einträge möglich.

3.2 Arbeiten mit Dateien

Abschnitt 3.1.1 beschrieb bereits die Systemaufrufe als Schnittstelle zwischen dem User- und Kernelspace. Im Folgenden wird auf Systemaufrufe eingegangen, die bei der Arbeit mit Dateien zum Einsatz kommen.

Bei dem Öffnen einer Datei durch den Systemaufruf `open` wird zur eindeutigen Identifikation ein Dateideskriptor zurückgegeben. Alle folgenden Operationen wie das Lesen oder Schreiben über die Aufrufe `read` und `write` beziehen sich auf diesen Identifikator. Dabei wird selten die gesamte Datei auf einmal modifiziert. Vielmehr werden Veränderungen bzw. Dateiinhalte in einen Puffer begrenzter Länge geschrieben und dem jeweils anderen Adressraum übergeben. Lese- bzw. Schreiboperation sind so oft zu wiederholen, bis die nötigen Daten bezogen wurden bzw. das Ende einer Datei erreicht ist.

Im Kernelspace wird für jeden Prozess eine Instanz der Struktur `struct files_struct` verwaltet. Diese enthält ein Feld `fd` aus Zeigern auf Strukturen vom Typ `struct file`. Dieses Feld dient zur Verwaltung von Informationen von allen geöffneten Dateien. Der Dateideskriptor, der im Userspace verwendet wird, wird vom Kernel an dieser Stelle als Index für das Feld verwendet. So kann die zugehörige `file`-Struktur ermittelt werden. Die von Ihr verwalteten Informationen soll nun erläutert werden.

Listing 3.8: *Struktur* `struct file` *in der Datei* `include/linux/fs.h`

```
1  struct file {
2          struct list_head f_list;
3          struct dentry   *f_dentry;
4          struct vfsmount  *f_vfsmnt;
5          struct file_operations *f_op;
6          atomic_t      f_count;
7          unsigned int  f_flags;
8          mode_t        f_mode;
9          loff_t        f_pos;
10
11         [ 14 additional lines ]
12 };
```

- `f_dentry` zeigt auf eine Instanz der Struktur `struct dentry`.
- Die benötigten Funktionzeiger der Operationen um Dateien zu manipulieren werden in `f_op`, einer Instanz von `struct file_operations` abgelegt.
- `f_flags` ist der beim Öffnen der Datei angegebene Zugriffsmodus, etwa `O_RDONLY` oder `O_CREAT`.
- Die aktuelle Position des Dateizeigers wird durch `f_pos` festgehalten und gibt einen Offset in Bytes vom Dateianfang an.

Abschließend sollen wesentliche Operationen auf Dateien, aus der Struktur `file_operations`, erläutert werden.

```
size_t (*read) (struct file *, char __user *, size_t, loff_t *);
```

Diese Funktion dient, wie der Name schon angibt, zum Einlesen von Daten aus einer Datei. Als erster Parameter wird eine Instanz vom Typ `file` übergeben. Diese repräsentiert die Datei innerhalb des aktuellen Prozesses. Weitere Parameter sind ein Puffer zum Zwischenspeichern der gelesenen Daten, die Anzahl der gelesenen Bytes sowie ein

Offset, welcher die Position innerhalb der Datei angibt, ab der die Daten gelesen werden sollen. Analog existiert eine Funktion `write`, die Daten aus dem übergebenen Puffer in die Datei schreibt.

```
int (*open) (struct inode *, struct file *);
```

Die Operation öffnet eine Datei, und bindet die assoziierte Inode an die Instanz vom Typ `struct file`.

```
int (*readdir) (struct file *, void *, filldir_t);
```

`Readdir` liest den Inhalt einer Verzeichnisdatei aus. Diese Operation ist daher nur für Dateien zulässig, bei denen der Modus in der Inodestruktur `i_mode` auf ein Verzeichnis gesetzt ist.

Der erste Parameter ist das zu untersuchende Verzeichnis in Form eines Zeigers auf eine `file`-Struktur. An zweiter Stelle wird ein untypesierter Zeiger auf einen Puffer übergeben. In Ihm wird das Ergebnis gespeichert. Als letzter Parameter wird ein Zeiger auf eine Funktion mit der Signatur `typedef int (*filldir_t)(void *, const char *, int, loff_t, ino_t, unsigned)` spezifiziert. Diese Funktion wird für jede Datei in einem Verzeichnis aufgerufen und unterscheidet sich unter den Dateisystemimplementationen. So kann spezifisch festgelegt werden, welche Verzeichniseinträge im Puffer aufgenommen werden und wie dieser zu formatieren ist.

KAPITEL 4

Implementation

Nachdem im vorangegangenen Kapitel eine Einführung in die Dateisystemimplementierungen von Linux gegeben wurde, sollen nun die genauen Anforderungen der angestrebten Versionsverwaltung spezifiziert und mögliche Umsetzungen diskutiert werden. Anschließend erfolgt eine detaillierte Erläuterung der konkreten Implementation.

4.1 Spezifikation der Anforderungen

Ziel der Implementation ist die Schaffung eines Dateisystems für Linux mit einer Versionsverwaltung nach dem Vorbild der Files-11-Familie. Dabei wird es nicht gelingen alle Merkmale zu übernehmen, da die Kompatibilität zu bestehenden Anwendungen gewahrt werden muss. Umgesetzt werden sollen zumindest die folgenden Konzepte:

1. Anzahl der Versionen einer Datei – Pro Verzeichnis soll es möglich sein, die maximale Anzahl von Versionen (DirMaxV), die für eine Datei existieren dürfen, zu spezifizieren. Abweichend zu VMS wird jedoch das Überschreiben dieser Vorgabe für einzelne Dateien nicht unterstützt. Dafür soll es jedoch möglich sein die Versionierung abzuschalten bzw. nur für gewisse Verzeichnisse zu aktivieren, indem DirMaxV auf Null gesetzt wird.

   Zur Erfüllung dieser Aufgabe muss zunächst ein geeigneter Ort innerhalb der Verzeichnisstrukturen gefunden werden, wo DirMaxV abgelegt werden kann. Ferner ist eine Schnittstelle zwischen dem Dateisystem und dem Userspace zu entwickeln, um dem Anwender die Möglichkeit zur Manipulation zu geben.

2. Copy-On-Write – Ziel ist es bei jedem schreibenden Zugriff auf eine Datei eine Kopie mit gleichem Namen und einer um eins erhöhten Versionsnummer anzulegen. Dabei gilt es einen geeigneten Separator von Dateiname und Versionsnummer zu finden. Das in VMS benutzte Semikolon kann nicht verwendet werden, da in gängigen Unix-Shells dieses Zeichen für das Separieren von Befehlen verwendet wird.

   Ferner wird mit Null die kleinste und mit 65000 die höchstmögliche Versionsnummer definiert. Wird mit einer neuen Version DirMaxV überschritten, so muss die älteste Version gelöscht werden. Als Einschränkung ist zu beachten, dass nur reguläre Dateien versioniert werden können. Versionierung von Pipes und Device Nodes ergibt keinen Sinn, da sich Prozesse und Geräte auf diesem Weg nicht

duplizieren lassen. Verzeichnisse scheiden ebenfalls aus, da sie nicht schreibend geöffnet werden können und mit dem Verzeichnis auch alle in ihm enthaltenen Dateien kopiert werden müssten.

3. Copy-On-Rename – Viele Programme arbeiten auf temporären Dateien. Sind die Modifikationen abgeschlossen, so wird die temporäre Datei in die ursprüngliche Datei umbenannt. Auch an dieser Stelle muss eine Versionierung vorgenommen werden.
4. Darstellung der aktuellsten Version – VMS bietet wie bereits in Abschnitt 2.2 auf Seite 13 beschrieben die Möglichkeit unter Angabe des reinen Namens ohne Versionsnummer auf die aktuellste Datei zu zugreifen. Ein ähnliche Semantik soll auch hergestellt werden.
5. Purge – Neben dem Usermode-Programm zum Setzen von DirMaxV ist das Kommando Purge aus VMS nachzubilden. So soll ein Aufruf unter Angabe des Dateinamens alle Dateiversionen bis auf die aktuellste löschen und die Versionsnummer auf Null zurücksetzen. Unter Angabe weiterer Parameter können die n neuesten beibehalten bzw. alle Versionen innerhalb eines Bereiches entfernt werden.

4.2 Implementation

Im folgenden sollen Lösungsmöglichkeiten diskutiert und die konkrete Umsetzung beschrieben werden. Zuvor jedoch noch ein paar Worte zu der Entwicklungsumgebung.

4.2.1 Entwicklungsumgebung

Entwickelt wurde auf einem Gentoo-Linux-System mit einem aktuellen 2.6er Kernel und dem Compiler GCC in der Version 3.3.6. Weitere benutzte Programme waren Vim, Make und CVS. Während sich die Implementation der Userspace-Programme einfach gestaltet, so ist das Testen von Kernelcode aufwendiger, da jedesmal ein komplettes Betriebssystem gebootet werden muss.
Aus diesem Grund wurde auf ein Usermode Linux (UML) zurückgegriffen. Spezielle UML-Kernelquellen werden dabei mit der Option `make ARCH=um` übersetzt. Die resultierende Binärdatei kann z.B. wie folgt aufgerufen werden:

```
./linux mem=128M devfs=mount ubd0=./rootfs ubd1=./swapfs umid=name
```

Zuvor wurde eine komplette Betriebssysteminstallation in die Datei `rootfs` vorgenommen. Mehr Informationen zum Thema UML bietet die Homepage des Usermode-Linux-Projekts [10]. Auf diesem Weg lässt sich ein Linux-System in weniger als zehn Sekunden neu starten, was einen enormen Vorteil bedeutet, da sich Segmentation-Faults und Dead-Locks nicht immer vermeiden lassen.

4.2.2 Level der Implementation

Im vorangegangenen Kapitel 3 wurde das VFS als Abstraktionsschicht eingeführt, dessen erweiterter Funktionsumfang jedoch von den verschiedensten Dateisystemimplementationen genutzt wird. Somit stellt sich die Frage, ob die Erweiterungen um eine Dateiversionierung in einem separaten Dateisystem unterhalb des VFS oder in der VFS-Schicht selbst vorgenommen werden sollen.

Die Implementation in einem eigenständigen Dateisystem bietet die Möglichkeit einer klaren Abgrenzung gegenüber anderen Dateisystemen. Veränderungen beeinflussen nicht andere Entwicklungen. Das neue Dateisystem kann als separates Modul geladen und entladen werden. Auch sollte sich die Implementation einfacher gestalten, da das VFS die in Abschnitt 3.1.2 auf Seite 16 vorgestellten Funktionszeiger als explizite Schnittstelle für dateisystemspezifische Implementationen bereit stellt. Generische Funktionen sind hierbei durch Aufrufe zur ersetzen, die eine Versionierung übernehmen.

So wurde im ersten Ansatz in der Tat die Versionsverwaltung in einem separaten Dateisystem angestrebt mit der Prämisse keine Veränderungen in anderen Bereichen des Kernels, insbesondere dem VFS, durchzuführen. Jegliche Modifikationen können in anderen Kernelversionen nicht vorausgesetzt werden. Im ersten Schritt wurde eine Kopie des Dateisystem Ext2 unter dem Namen Ext2v angelegt. Die Wahl für diese Vorlage beruht auf der Tatsache, dass Ext2 einen minimalen Funktionsumfang bietet und somit einen überschaubaren Quellcode besitzt. Außerdem konnte sichergestellt werden, dass es zu keinen ungewollten Veränderungen auf dem Hostsystem bzw. der UML-Testumgebung kommt, da beide Ext3 als Dateisystem benutzten.

Die Entwicklungen schritten erfolgreich voran. Jedoch zeigte sich zunehmend, dass das VFS der angestrebten Versionsverwaltung im Weg stand, da generischer Code bereits Aufgaben erledigt und nur zwischendurch die dateisystemspezifischen Funktionen über die genannten Funktionszeiger aufruft. Code vor und nach dem Aufruf der spezifischen Implementation kann nicht verändert werden. War es bis dato noch möglich alle Probleme zu umgehen, so bedeutete der folgende Konflikt das Ende der angestrebten separaten Dateisystemimplementation.

Wie Abschnitt 4.2.7 auf Seite 34 noch erläutern wird, ist es unter Umständen notwendig, beim Umbenennen einer versionierten Datei die Quelle nicht zu löschen, sondern eine Kopie unter einem neuen Namen anzulegen.

Listing 4.1: *Methode* `vfs_rename_other` *in der Datei* `fs/namei.c`

```
1  static int vfs_rename_other(struct inode *old_dir, struct dentry *old_dentry,
2                              struct inode *new_dir, struct dentry *new_dentry)
3  {
4     [ 14 additional lines ]
5
6        error = old_dir->i_op->rename(old_dir, old_dentry, new_dir, new_dentry);
7     if (!error) {
8        /* The following d_move() should become unconditional */
9        if (!(old_dir->i_sb->s_type->fs_flags & FS_ODD_RENAME))
10          d_move(old_dentry, new_dentry);
11       security_inode_post_rename(old_dir, old_dentry, new_dir, new_dentry);
12    }
13    if (target)
14       up(&target->i_sem);
15    dput(new_dentry);
16    return error;
17 }
```

Auflistung 4.1 zeigt einen Teil der Funktion `vfs_rename_other`, welche über `vfs_rename` und `do_rename` von dem Syscall `sys_rename` aufgerufen wird.

In Zeile 6 wird die dateisystemspezifische Implementation dieser Funktion aufgerufen, die in der Inodeoperations-Struktur registriert ist. Zeile 10 nimmt mit der Funktion `d_move` die Änderungen im Dentry-Cache vor, wodurch diese erst wirksam werden. Um zu verhindern, dass das Original durch jene Funktion gelöscht wird, könnte in

der dateisystemspezifischen Variante ein temporärer Dateieintrag erstellt und dieser zum Löschen bereit gestellt werden. Da jedoch nur ein direkter Zeiger auf die Dentry-Struktur übergeben wird, kann dieser nicht für die übergeordnete Funktion verändert werden[1].

Somit schien eine Veränderung des VFS unumgänglich. Dadurch war jedoch der Vorteil einer unabhängigen Implementation nicht mehr gegeben. Alle anderen Dateisysteme hätten ebenfalls angepasst werden müssen.

Nach dieser ernüchternden Feststellung wurde der Ansatz einer separaten Implementation verworfen und eine Veränderung der VFS-Schicht selbst angestrebt. Neben den beschriebenen Nachteilen bietet diese Variante jedoch auch, einmal implementiert, das Potential einer universellen Versionsunterstützung für alle Dateisysteme.

Da jedoch die Kompatibilität nicht zu allen Dateisystemen gewährleistet werden kann, wird ein konservativer Ansatz gewählt. Die Versionierung soll standardmäßig deaktiviert sein und nur, falls explizit vom Dateisystem unterstützt, aktivierbar sein. Beispielhaft ist dies für Ext2 zu gewährleisten.

4.2.3 Verwaltung von DirMaxV

4.2.3.1 Ablegen von DirMaxV

Die Speicherung der maximalen Dateiversionen für ein Verzeichnis kann auf vielen Wegen erfolgen. Denkbar ist eine versteckte Datei, die diese Zahl aufnimmt. Um sicherzustellen, dass jene Datei nicht wie jede andere angezeigt wird, muss die dateisystemspezifische Implementation zur Anzeige aller Dateien in einem Verzeichnis angepasst werden. Diese verbirgt sich hinter dem Funktionszeiger `readdir` in der Struktur `file_operations` und heißt im Fall von Ext2 `ext2_readdir`. Außerdem gilt es einen Namen zu benutzen, der wahrscheinlich niemals verwendet wird. Da jedoch ein Konflikt nicht völlig ausgeschlossen werden kann, sollte beim Versuch eine Datei mit diesem reservierten Namen anzulegen eine passende Fehlermeldung geliefert werden.

Neben einer Datei kann DirMaxV auch in der Verzeichnis-Inode selbst abgespeichert werden, wodurch eventuelle Namenskonflikte ausgeschlossen sind. Dabei muss jedoch sichergestellt werden, dass sich die Position von Variablen innerhalb der Struktur nicht verändert, da ansonsten jegliche Kompatibilität zu bestehenden Daten verloren ginge. In der Ext2-Referenzimplementation konnte diese Variante gewählt werden.

Listing 4.2: *Struktur* `ext2_inode` *in der Datei* `include/linux/ext2_fs.h`

```
1  struct ext2_inode {
2     [ 27 additional lines ]
3     union {
4        struct {
5           __u8  l_i_frag;   /* Fragment number */
6           __u8  l_i_fsize;  /* Fragment size */
7           __u16 i_pad1;
8           __le16 l_i_uid_high; /* these 2 fields */
9           __le16 l_i_gid_high; /* were reserved2[0] */
10          __u32 l_i_reserved2;
11       } linux2;
12       struct {
13          __u8  h_i_frag;   /* Fragment number */
14          __u8  h_i_fsize;  /* Fragment size */
15          __le16 h_i_mode_high;
```

[1]Nötig wäre ein Zeiger auf einen Zeiger auf die Struktur (`dentry**`).

```
16          __le16 h_i_uid_high;
17          __le16 h_i_gid_high;
18          __le32 h_i_author;
19       } hurd2;
20       [ 6 additional lines ]
21    } osd2;                 /* OS dependent 2 */
22 };
```

Auflistung 4.2 zeigt einen Ausschnitt der Ext2-Inodestruktur. Im zweiten betriebssystemspezifischen Bereich existieren, zumindest für das Betriebssystem Linux, vier Byte reservierter Speicherplatz (Zeile 10). Im Abschnitt 4.1 wurde mit 65000 die höchste Version festgelegt, wodurch niemals mehr als 65001 verschiedene Versionen existieren können. Somit reichen zwei Byte zur Speicherung von DirMaxV aus. In der modifizierten Version dieser Struktur wurde die Zeile 10 durch die beiden 16 bit breiten Variablen `l_i_max_versions` sowie `l_i_reserved2` ersetzt.

Dieser reservierte Speicherplatz kann jedoch nicht für jedes Dateisystem vorausgesetzt werden. Aus diesem Grund wurde die Schnittstelle der Inodeoperationen um zwei Funktionen erweitert, welche Auflistung 4.3 zeigt.

Listing 4.3: *Struktur* `inode_operations` *in der Datei* `include/linux/fs.h`

```
1 struct inode_operations {
2    [ 21 additional lines ]
3    int (*getmaxversions) (struct inode *);
4    int (*setmaxversions) (struct inode *, int);
5 };
```

Während `getmaxversions` unter Angabe eines Zeigers auf eine Inode DirMaxV liefert, so kann über `setmaxversions` dieser Wert gesetzt werden. Der Vorteil dieser Variante liegt klar auf der Hand. Jedes Dateisystem kann selbst festlegen, wie es DirMaxV verwaltet. Außerdem kann über diesen Funktionszeiger überprüft werden, ob eine Versionsunterstützung überhaupt vorliegt. Dazu wurde in der Datei `include/linux/fs.h` das Makro `VERSION_SUPPORT` definiert, welches prüft, ob eine Implementation für den Funktionszeiger `getmaxversions` hinterlegt wurde.

Anhang A.10 auf Seite 66 zeigt die dateisystemspezifischen Funktionen `ext2_get_max_versions` und `ext2_set_max_versions` sowie deren Bindung an die Funktionszeiger in der Struktur `ext2_dir_inode_operations`. Interessant ist der Aspekt, dass die Veränderungen nicht auf der eigentlichen Ext2-Inode[2], sondern auf einer weiteren Struktur, der `ext2_inode_info` durchgeführt werden. Diese kapselt alle zusätzlichen, Ext2-spezifischen Inode-Attribute und kann modifiziert werden, ohne die physikalische Struktur auf dem Datenträger zu ändern. Ihre Anpassung ist im Anhang A.8 auf Seite 65 zu finden. Eine Zuordnung zwischen generischer Inode und der Info-Inode geschieht über die Funktion `EXT2_I`. Ferner werden die Modifikationen nicht sofort auf den Datenträger geschrieben, sondern die Inode über die Funktion `mark_inode_dirty` als verändert markiert.

Der Kontakt zur Raw-Inode wird erst in der Datei `inode.c` (Anhang A.9 auf Seite 65) hergestellt. Dabei mussten die Funktionen `ext2_read_inode` und `ext2_update_inode` erweitert werden, die die Info-Inode aus der Raw-Inode einlesen bzw. Veränderungen auf die Raw-Inode übertragen.

[2] In dem Zusammenhang wird auch von der Raw-Inode gesprochen.

4.2.3.2 Schnittstelle zum Userspace

Die Manipulation von DirMaxV soll dem Anwender ermöglicht werden. Somit muss der Kernel eine Schnittstelle zum Userspace zur Verfügung stellen. Im Abschnitt 3.2 auf Seite 23 wurden die Systemaufrufe vorgestellt. Eine Lösung wäre es, zwei weitere Systemaufrufe einzuführen, die das Setzen und Auslesen von DirMaxV übernehmen. Jedoch existieren hiervon je nach Plattform weit über 200. Jede Erweiterung muss mit anderen Projekten abgestimmt sein, da jedem Systemaufruf eine eindeutige Nummer zugeordnet ist, die zusammen mit weiteren Parametern der Systembibliotheksfunktion `syscall` übergeben wird.

Eine weitere Möglichkeit stellt der spezielle Systemaufruf `ioctl` dar, auf den über eine gleichnamige Systemfunktion zugegriffen werden kann. Dieser wurde eingeführt um spezielle Gerätedateien ansteuern zu können[3], wobei die Datei mittels zuvor geöffnetem Dateideskriptor spezifiziert wird.

Bei der Umsetzung wurde diese zweite Variante gewählt und dafür die Ioctl-Kommandos `FIGETMAXVERSIONS` und `FISETMAXVERSIONS` in der Datei `include/linux/fs.h` (Anhang A.2 auf Seite 48) definiert. Außerdem wurde die Funktion `vfs_ioctl` der Datei `fs/inode.c` (Anhang A.4 auf Seite 50) erweitert. So werden von ihr nun zusätzlich die oben genannten Kommandos behandelt und gegebenenfalls die Funktionen `getmaxversion` bzw. `setmaxversion` aufgerufen.

Ein weiteres Augenmerk soll auf die Überprüfung der Rechte gelenkt werden. Die Standardprozedur ist es bereits beim Öffnen der Datei durch den Systemaufruf `sys_open` Schreib- oder Leserechte anzufordern und sie bei der Verarbeitung des Ioctl-Aufrufs zu überprüfen. Dazu muss lediglich das Vorkommen des Datei-Flags `O_RDONLY` bzw. `O_WRONLY` betrachtet werden. Das Setzen von DirMaxV ist jedoch eine verzeichnisspezifische Operation. Verzeichnisse können aber nur lesend geöffnet werden, da Veränderungen nur durch Modifikation der durch sie verwalteten Dateien/Verzeichnisse auftreten können. Aus diesem Grund muss auch der modifizierende Aufruf `FISETMAXVERSIONS` auf einem mit Leserechten geöffneten Dateideskriptor erfolgen. Um dennoch Veränderungen an DirMaxV nur von autorisierten Personen zuzulassen wird das Schreibrecht explizit über die Funktion `generic_permission` und den Parameter `MAY_WRITE` überprüft.

Zur Erlangung eines besseren Überblicks werden durch Abbildung 4.1 wichtige beteiligte Funktionen aufgezeigt. Zur Erläuterung der Syntax der Diagramme ist zu sagen, dass jedes Kästchen einen Funktionsaufruf darstellt. Findet in einer Routine ein weiterer Aufruf statt, so wird dieser eingerückt dargestellt. Sequentielle Ausführungen im selben Codeblock werden auf gleicher Höhe gezeichnet. Funktionen, die bereits im Vanilla-Kernel existieren, sind grün hinterlegt. Neu geschaffene Aufrufe werden gelb dargestellt.

So ist auf der linken Seite das Lesen von DirMaxV abgebildet. Hierbei wird der Wert bereits beim Öffnen des Verzeichnisses in die Info-Inode-Struktur eingelesen. Der nachfolgende Ioctl-Aufruf stellt keinen Kontakt zum Datenträger her. Die rechte Seite der Abbildung zeigt das Setzen von DirMaxV. Auch hier modifiziert das Ioctl-Kommando den Datenträger nicht direkt. Die Veränderungen werden erst beim regelmäßigen oder expliziten Synchronisieren geschrieben.

[3]Nach dem Grundsatz „Alles ist eine Datei“ könnte die Ansteuerung von Geräten auch durch das Schreiben von speziellen Zeichenketten in die entsprechende Gerätedatei geschehen. Dabei könnten Konflikte jedoch nie völlig ausgeschlossen werden. Auch stellt die Einführung eines Systemaufrufs für jede gerätespezifische Aktion keine Alternative dar.

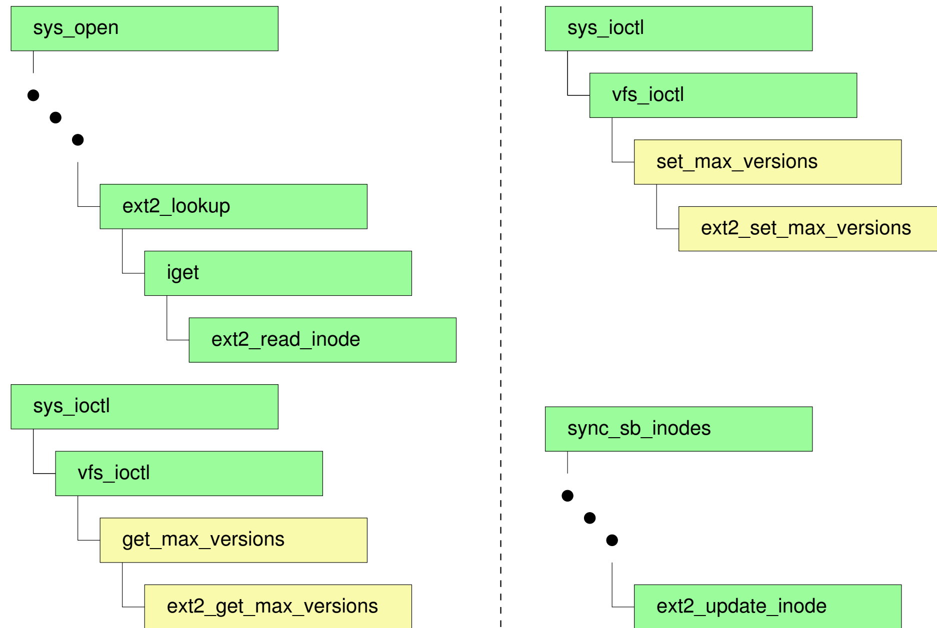


Abbildung 4.1: *Lesen und Setzen von DirMaxV*

4.2.4 Darstellung und Interpretation der Versionsnummer

Wie auch im Vorbild VMS soll die Versionsnummer im Namen der jeweiligen Datei enthalten sein. Dafür muss ein Trennzeichen gefunden werden, welches nicht von der Shell interpretiert wird, gleichzeitig jedoch aussagekräftig genug ist und üblicherweise nicht in einem Dateinamen vorkommt.

Die Wahl fiel auf die Raute bzw. das Doppelkreuz (#), welches unter anderem als Abkürzung für „Anzahl“ steht. Um flexibel gegenüber Änderungen zu sein, wurde in der Datei `include/linux/fs.h` (Anhang A.2 auf Seite 48) das Makro `VERSION_SEPARATOR` definiert.

Für die Interpretation der Versionsnummer ist die Funktion `separate_name_and_version` verantwortlich, die in `fs/namei.c` (Anhang A.5 auf Seite 51) implementiert ist. Sie durchsucht den angegebenen Dateinamen rückwärts, beginnend von der letzten Position nach dem ersten Vorkommen des Separators. Alle Zeichen rechts von der gefundenen Position werden versucht als Zahl zu interpretieren. Tabelle 4.2 zeigt Beispiele für gültige und ungültige Versionsnummern.

gültig	ungültig
`foo#1`	`foo`
`foo#23`	`foo#`
`foo#bar#42`	`foo#1a`

Abbildung 4.2: *Dateinamen mit gültigen / ungültigen Versionsnummern*

4.2.5 Darstellung der aktuellsten Version

Existieren mehrere Versionen einer Datei, so soll auf die aktuellste Datei über den Dateinamen ohne Angabe einer Versionsnummer zugegriffen werden können. Dies könnte dadurch erreicht werden, dass dem Userspace eine Datei bereit gestellt wird, die physikalisch nicht existiert. Dazu müsste z.B. der Syscall `sys_getdents` in der Datei `fs/readdir.c` verändert werden. Zusätzlich zu allen Verzeichniseinträgen muss er nun auch noch diese virtuelle Datei zurückliefern. Außerdem sind alle Zugriffe auf eine solche Datei zu erkennen und entsprechend umzulenken.

```
lrwxrwxrwx  1 root root  8 Oct 20 19:35 foobar -> foobar#2
lrw-r--r--  1 root root  5 Oct 20 19:25 foobar#0
lrw-r--r--  1 root root 10 Oct 20 19:29 foobar#1
lrw-r--r--  1 root root 15 Oct 20 19:35 foobar#2
```

Abbildung 4.3: *Darstellung der aktuellsten Dateiversion über Softlinks*

Eine einfachere und für den Benutzer weitaus transparente Möglichkeit bilden die Softlinks, welche im Abschnitt 3.1.1 auf Seite 16 vorgestellt wurden. So zeigt Abbildung 4.3 die Ausgabe des Befehls `ls -l`. Zu sehen sind drei Versionen einer Datei sowie ein Softlink, der auf die größte Version zeigt. Theoretisch hätten auch Hardlinks benutzt werden können, jedoch fällt so die Unterscheidung zwischen Original und Link einfacher.

Selbstverständlich muss das Dateisystem dafür sorgen, dass sich der Link stets auf dem neuesten Stand befindet. Wird z.B. die aktuellste Version gelöscht, so muss das Verzeichnis nach der nun größten Nummer durchsucht und der Link entsprechend gesetzt werden. Abbildung 4.4 zeigt das Vorgehen in diesem Fall. Die aufgeführten Funktionen sind im Anhang A.5 zu finden. Im Syscall `sys_unlink` wird zunächst geprüft, ob sich die zu löschende Datei in einem versionierten Verzeichnis befindet. Dies geschieht über das Makro `VERSION_SUPPORT_ENABLED`, welches überprüft, ob DirMaxV größer als Null ist. In diesem Fall wird an Stelle der Funktion `vfs_unlink` `version_unlink` aufgerufen.

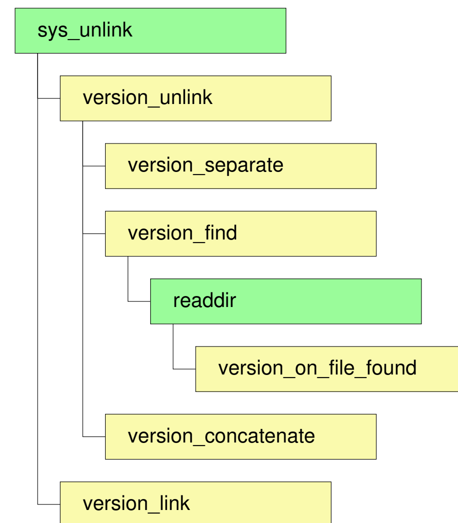


Abbildung 4.4: *Löschen der aktuellsten Version*

Nachdem Dateiname und Versionsnummer durch `version_separate` getrennt wurden, kann der Name dem Eintrag `name` der Struktur `name_and_version` aus Auflistung 4.4 zugewiesen werden. Dieser wird dem Aufruf `version_find` sowie `readdir` übergeben und erfüllt gleich mehrere Funktionen. So wird der Funktion `version_on_file_found` [4] der gesuchte Bezeichner mitgeteilt. Gleichzeitig kann in ihr die größte, zweitgrößte und kleinste Version sowie die Gesamtzahl aller Versionen festgehalten werden. Nachdem alle Verzeichniseinträge überprüft sind, kann die größte gefundene Version über die Funktion `version_concatenate` mit dem Dateinamen verknüpft werden. Anschließend erstellt `version_link` mithilfe dieser Zeichenkette den aktualisierten Softlink.

Listing 4.4: *Struktur* `name_and_version` *in der Datei* `include/linux/fs.h`

```
1 struct name_and_version {
2    char * name;       /* Reference name to search for. */
3    int max_version; /* Largest version found */
4    int pre_max_version; /* The second largest version found */
5    int min_version; /* Smallest version found */
6    int version_count; /* Number of different versions found */
7 };
```

[4]Diese Funktion ist vom Typ `filldir_t` und wird für jeden gefundenen Verzeichniseintrag aufgerufen. Die genaue Semantik hierfür wurde bereits im Abschnitt 3.2 auf Seite 24 beschrieben.

4.2.6 Copy-On-Write

Beim schreibenden Zugriff wird eine neue Version erstellt, ohne das Original bzw. das eigentliche Schreibziel zu verändern.

```
root:/test# ls -l
total 3
lrwxrwxrwx  1 root root  8 Oct 20 19:35 foobar -> foobar#2
lrw-r--r--  1 root root  5 Oct 20 19:25 foobar#0
lrw-r--r--  1 root root 10 Oct 20 19:29 foobar#1
lrw-r--r--  1 root root 15 Oct 20 19:35 foobar#2
root:/test# echo "1234" >> foobar
root:/test# ls -l
total 3
lrwxrwxrwx  1 root root  8 Oct 20 19:37 foobar -> foobar#3
lrw-r--r--  1 root root 10 Oct 20 19:29 foobar#1
lrw-r--r--  1 root root 15 Oct 20 19:35 foobar#2
lrw-r--r--  1 root root 20 Oct 20 19:37 foobar#3
```

Abbildung 4.5: *Anlegen einer neuen Version*

Abbildung 4.5 zeigt ein Verzeichnis, bei dem DirMaxV auf drei Versionen festgelegt wurde. Wie der erste `ls`-Aufruf zeigt existieren bereits drei Versionen. Die aktuellste Version trägt den Namen `foobar#2` und ist 15 Byte groß. Nun werden weitere fünf Byte[5] geschrieben. Wie die nächste `ls`-Ausgabe zeigt, lautet die nun neuste Version `foobar#3` und ist 20 Byte groß. Die Datei `foobar#2` wurde nicht verändert, die älteste Version dagegen gelöscht.

Bei der Implementation dieses COW-Verhaltens konnte jedoch nicht, wie der Name vielleicht vermuten lässt, auf den aus Abschnitt 3.2 auf Seite 23 bekannten Systemaufruf `write` zurückgegriffen werden. Dieser Aufruf wird beim Schreiben in eine Datei je nach Datenmenge und Puffergröße mehrmals wiederholt, wodurch jedes mal eine neue Version erzeugt würde. Vielmehr gilt es zwischen den Aufrufen `open` und `close` nur einmal zu versionieren. So wäre es denkbar, erst beim Schließen einer Datei die neue Version anzulegen. In diesem Fall muss jedoch verhindert werden, dass Schreiboperationen die Originaldatei verändern. Eine Lösung bestünde in einer temporären Datei zur Aufnahme aller Veränderungen. Diese müsste jedoch bereits beim Öffnen erstellt werden, wodurch sowohl `open` und `close` zu modifizieren wären.

Aus diesem Grund wurde die Versionierung ausschließlich für den Moment des Öffnens einer Datei umgesetzt. Abbildung 4.6 zeigt das genaue Vorgehen.

Im Systemaufruf `sys_open` wird durch die Funktion `get_unused_fd` ein freier Dateideskriptor bestimmt. Anschließend liefert die Funktion `filp_open` die kernelspezifische Repräsentation der geöffneten Datei in Form eines Zeigers auf eine `file`-Struktur. Dateideskriptor und `file`-Struktur werden durch `fd_install` miteinander verknüpft und prozessspezifisch abgelegt.

In `filp_open` wird zunächst die Funktion `open_namei` aufgerufen. Sie löst den Namen über die Funktion `lookup_hash` in eine `nameidata`-Struktur auf und legt einen Eintrag, falls noch nicht vorhanden, im Dentry-Cache (vgl. 3.1.5 auf Seite 22) an. Anschließend wird die Funktion `may_open` aufgerufen, die neben der Überprüfung der Rechte noch die Funktion `do_truncate` aufruft, falls das Leeren der Datei vor Schreibbeginn gewünscht ist. An dieser Stelle sind die ersten Modifikationen vorzunehmen, da bei angeschalteter Versionierung niemals das Original verändert werden darf. Aus diesem Grund wird vor dem Aufruf von `may_open` das Flag `O_TRUNC` entfernt, sofern

[5] Der `echo`-Aufruf schreibt neben den vier Zeichen noch einen Zeilenumbruch (`\n`)

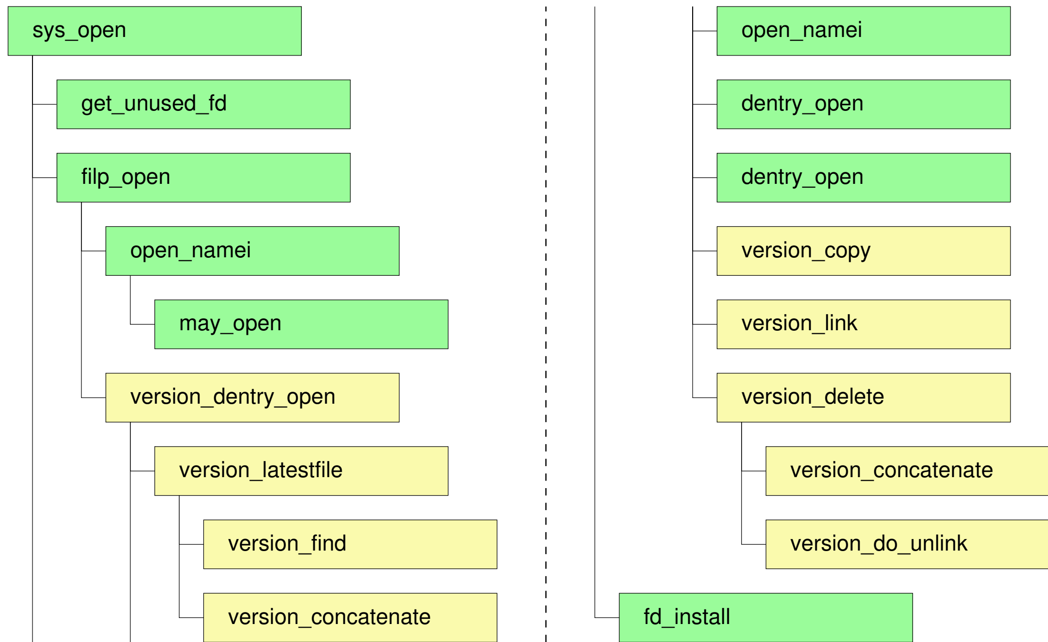


Abbildung 4.6: *Der veränderte Systemaufruf* `sys_open`

dieses gesetzt ist:

```
flag &= ~O_TRUNC
```

Um dennoch die gewünschte Semantik beizubehalten, wird später auf das Kopieren des Originals verzichtet und eine leere Datei angelegt.

Als zweite Funktion ruft `filp_open` standardmäßig `dentry_open` auf, welche eine `file`-Struktur befüllt und einen Zeiger darauf zurück liefert. Sie wurde durch `version_dentry_open` ersetzt, welche die größten Erweiterungen beinhaltet. So wird zuerst der neue Dateiname mit der neuesten Versionsnummer über die Funktion `version_latestfile` bestimmt. Anschließend wird auch für diesen Namen ein Dentry-Eintrag angelegt (`open_namei`). Der doppelte Aufruf von `dentry_open` liefert jeweils eine `file`-Struktur für die alte und neue Version. Diese werden `version_copy` übergeben, die das Kopieren des Originals übernimmt.

Abschließend wird nun noch der Versionslink mit `version_link` aktualisiert und gegebenenfalls die älteste Version mit `version_delete` gelöscht. Rückgabewert von `version_dentry_open` ist die `file`-Struktur der neuen Version.

4.2.7 Copy-On-Rename

Viele Programme und Tools unter Linux arbeiten auf Puffern oder benutzen temporäre Dateien um Ergebnisse zwischenzuspeichern. Dabei kann man im Wesentlichen zwei Verhaltensmuster der Programme identifizieren.

1. Ein mögliches Verhalten von Programmen ist es temporäre Dateien zu benutzen, auf denen Modifikationen durchgeführt werden. Beim Speichern durch die Anwendung wird diese Datei in die originale Datei umbenannt. Somit geht das Original und damit alle Daten der Version verloren. Man kann nicht mehr von einem konsistenten Zustand der Versionierung sprechen, da die Änderungen nicht in chronologischer Reihenfolge nachvollzogen werden können.

2. Eine weitere Möglichkeit, die vor allem bei Texteditoren beobachtet werden kann, ist das Arbeiten auf Puffern. Beim Öffnen einer Datei mit dem Flag `O_RDONLY` werden Daten in einen Puffer gelesen. Auf diesem werden alle folgenden Modifikationen der Daten durchgeführt. Sollen nun die Änderungen gespeichert werden, so wird als erstes die Originaldatei in eine andere Datei umbenannt, welche als Sicherheitskopie dient. Eine neue leere Datei mit dem originalen Dateinamen wird angelegt und der Puffer in diese geschrieben. War die Schreiboperation erfolgreich so wird die Sicherheitskopie entfernt.

   Für die Versionierung bedeutet dies, dass mit dem Umbenennen der originalen Datei die aktuellste Version verloren geht. Bei dem Open-Aufruf zur Erzeugung einer leeren Datei wird nur die Vorgängerversion gefunden. Dadurch kommt es zur Ermittlung einer falschen, bereits verwendeten Versionsnummer.

Um dieses Verhalten der verschiedenen Programme berücksichtigen zu können, muss ein Copy-On-Rename-Mechanismus implementiert werden, welcher auch eine Versionierung beim Verschieben bzw. Umbenennen von Dateien unterstützt. Der zu verändernde Systemaufruf heißt unter Linux `sys_rename`.

Abbildung 4.7 zeigt das entsprechende Vorgehen. Der Systemaufruf besitzt zwei Parameter `sys_rename(const char __user * oldname, const char __user * newname)`, jeweils einen Zeiger auf den Namen der Quelldatei `oldname` und der Zieldatei `newname`. Zu diesen werden in der Funktion `do_rename` die dazugehörigen Verzeichniseinträge ermittelt. Mittels der Einträge kann entschieden werden, ob die entsprechenden Verzeichnisse Versionierung unterstützten.

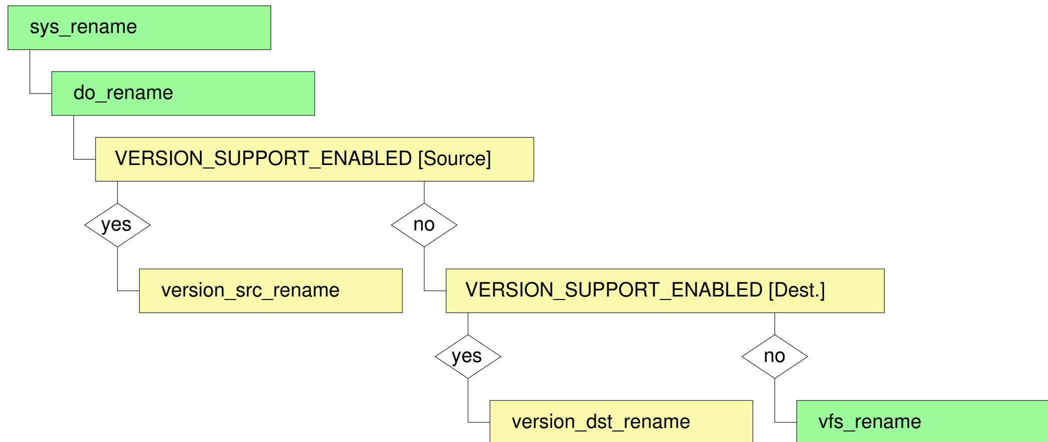
Je nachdem ob das Quell- oder Zielverzeichnis versioniert ist, wird der entsprechende Codefluss in verschiedene Pfade geteilt. Unterliegt das Quellverzeichnis einer Versionierung, so wird die Funktion `version_src_rename` aufgerufen. Analog dazu wird bei versioniertem Zielverzeichnis `version_dst_rename` ausgeführt. Ist weder das Quell- noch das Zielverzeichnis versioniert, so wird die ursprüngliche Funktion `vfs_rename` angesteuert.

Der Fall von `version_src_rename` ähnelt, wie Abbildung 4.6 zeigt, dem Systemaufruf `sys_open`. Dabei wird bei aktivierter Versionierung des Quellverzeichnisses durch `version_dentry_open` eine neue Version erzeugt. Das Kopieren des Dateiinhalts der vormals aktuellsten Version in die neue Datei wird deaktiviert, da der Inhalt der Zieldatei nicht aus dem Vorgänger dieser Datei bestehen soll. Vielmehr wird im Anschluss von `version_dentry_open` der Inhalt der zu verschiebenden Datei explizit über die Funktion `version_copy` kopiert.

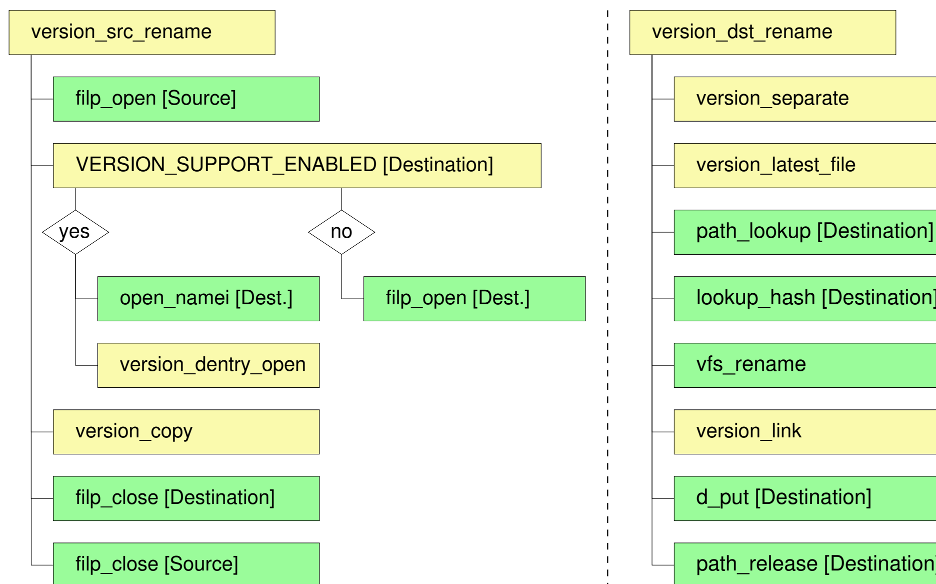
In Abbildung 4.7(b) ist die Funktion `Version_dst_rename` schematisch dargestellt. Dabei wird zunächst die Versionsnummer vom Dateinamen separiert. Anschließend bestimmt die Funktion `version_latestfile` einen Dateinamen mit der aktuellsten, noch nicht verwendeten Version. Die originale Datei bekommt nun diesen Namen. Abschließend wird der Versionslink auf den neuesten Stand gebracht.

4.2.8 Usermode-Tools

Dem Anwender werden die Programme `dirmaxv` und `purge` zur Verfügung gestellt, mit denen er Einfluss auf die Versionierung nehmen kann.



(a) Varianten der Versionierung



(b) Details der Veränderungen

Abbildung 4.7: *Der veränderte Systemaufruf* `sys_rename`

4.2.8.1 Setzen der maximalen Anzahl von Versionen mit `dirmaxv`

Zum Auslesen und Setzen der maximalen Anzahl von Version für ein Verzeichnis wurde das Programm `dirmaxv` mit folgender Aufrufsemantik geschrieben:

```
dirmaxv [-s number] <directory>
```

Wird beim Aufruf als einziger Parameter das gewünschte Verzeichnis angegeben, so liefert `dirmaxv` die maximale Anzahl von Dateiversionen. Erfolgt die Ausführung mit der Option `-s` gefolgt von einer Zahl und dem Verzeichnis, so wird der entsprechende Wert gesetzt.

Die Implementation gestaltete sich recht einfach und kann im Anhang A.2.1 auf Seite 68 eingesehen werden. Im Wesentlichen sind nur die Aufrufe `open`, `ioctl` und `close` interessant, die jeweils einen gleichnamigen Systemaufruf durchführen.

Der erste von den dreien liefert somit einen Dateideskriptor für die angegebene Datei. Um ihn auch auf einem Verzeichnis ausführen zu können, müssen die Flags `O_RDONLY`

und O_DIRECTORY gesetzt sein. Während O_RDONLY zum POSIX-Standard gehört, ist O_DIRECTORY Linux-spezifisch, weshalb das Makro _GNU_SOURCE definiert werden muss.

Die zweite Funktion ioctl wird zum eigentlichen Auslesen bzw. Setzen von DirMaxV benutzt. Als Parameter erhält sie den Dateideskriptor, die aus Abschnitt 4.2.3 bekannten Ioctl-Kommandos FIGETMAXVERSIONS bzw. FISETMAXVERSIONS sowie einen Zeiger auf eine Integervariable. Diese Zahl enthält nach dem Aufruf bzw. übergibt für den Aufruf die maximale Anzahl von Dateiversionen.

Nach erfolgter Ausführung wird der Dateideskriptor durch close geschlossen.

4.2.8.2 Aufräumen mit purge

Während der Arbeit mit dem versionierten Dateisystem werden immer wieder neue Dateiversionen angelegt. Auch wenn die maximale Anzahl über dirmaxv eingestellt werden kann, so wächst dennoch mit jedem schreibenden Zugriff die Verisonsnummer. Zum Aufräumen im Verzeichnis wird deshalb das aus VMS bekannte Programm purge zur Verfügung gestellt.

```
purge [[-k keep versions] | [-d delete versions]] <file>
```

Erfolgt der Aufruf nur unter Angabe einer Datei, so werden alle Dateiversionen bis auf die Neueste gelöscht. Zusätzlich können über den Parameter -k (keep) die Anzahl von Versionen angegeben werden, die erhalten bleiben sollen. Der Aufruf von -k 1 ist somit mit dem Aufruf ohne Parameter identisch. Wird -k 0 angegeben, so werden alle Dateiversionen gelöscht. Die Option -d (delete) erlaubt das Löschen konkreter Dateiversionen. Dabei kann eine einzelne Zahl oder ein Bereich angegeben werden.

Neben dem Löschen von gewissen Versionen übernimmt purge die Aufgabe alle Versionsnummern zurückzusetzen, wobei die älteste noch übrige Version die Nummer 0 bekommt. Abbildung 4.8 zeigt am Beispiel wie ein Verzeichnis vor und nach dem Kommando aussehen könnte.

```
root:/test# ls -l
total 3
lrwxrwxrwx  1 root root   8 Oct 22 18:45 foobar -> foobar#6
lrw-r--r--  1 root root   5 Oct 22 18:03 foobar#3
lrw-r--r--  1 root root  10 Oct 22 18:34 foobar#4
lrw-r--r--  1 root root  15 Oct 22 18:45 foobar#6
root:/test# purge -k 2 foobar
root:/test# ls -l
total 2
lrwxrwxrwx  1 root root   8 Oct 22 18:45 foobar -> foobar#1
lrw-r--r--  1 root root  10 Oct 22 18:34 foobar#0
lrw-r--r--  1 root root  15 Oct 22 18:45 foobar#1
```

Abbildung 4.8: *Beispiel eines* purge *Aufrufs*

Für die Implementation dieses Verhaltens wurde zunächst eine doppelt-verkettete Liste definiert. Im ersten Schritt durchsucht purge das im angegebenen Dateinamen enthaltende Verzeichnis nach allen Dateien gleichen Namens und trägt die gefundenen Versionsnummern mithilfe der Funktion list_insert sortiert in die Liste ein.

Sollen die k neuesten Versionen beibehalten werden, so wird die Liste rückwärts durchlaufen. Wurde eine entsprechende Schleife k mal durchlaufen, so werden alle folgenden

Versionen gelöscht. Der dafür notwendige Dateiname muss zuvor aus Versionsnummer und Basisname zusammengesetzt werden.

Beim Löschen eines bestimmten Versionsbereichs wird ähnlich verfahren, wobei die Liste diesmal vorwärts durchlaufen wird und nur einige Dateien entfernt werden, deren Versionsnummern in diesem Bereich liegen.

Wichtig für die weitere Arbeit ist bei beiden Varianten, dass für jede gelöschte Datei auch deren Listeneintrag entfernt wird. So kann beim Zurücksetzen der Versionsnummern sofort mit der Liste weitergearbeitet werden. Diese wird nun in einer neuen Schleife vorwärts durchlaufen. Dabei wird eine Variable mitgeführt, die beginnend mit 0 in jedem Durchlauf erhöht wird. Sie stellt die neue Version dar. Aus der Versionsnummer im aktuellen Listeneintrag und ihr kann zusammen mit dem Basisnamen ein Quell- und Zielpfad erstellt werden. Anschließend erledigt der Aufruf `rename` das Umbenennen bzw. das Zurücksetzen der Version.

4.3 Fazit

Für eine Versionsverwaltung im VFS mussten zahlreiche Erweiterungen am Linux-Kernel vorgenommen werden. So wurden zwei neue Ioctl-Kommandos zum Auslesen und Setzen der maximalen Anzahl von Dateiversionen geschrieben. DirMaxV wird beispielhaft für Ext2 in der Inodestruktur gespeichert. Weitere Veränderungen wurden an den Systemaufrufen `sys_open`, `sys_rename` und `sys_unlink` durchgeführt, um eine Copy-On-Write- und Copy-On-Rename-Semantik zu implementieren. Die jeweils aktuellste Version einer Datei wird über einen Softlink zugänglich gemacht.

Neben den Erweiterungen im Kernel kam es zu der Implementation von zwei Usermode-Programmen. Über `dirmaxv` kann die maximale Anzahl von Dateiversionen ausgelesen und gesetzt werden. `purge` löscht ältere Versionen und setzt die Versionsnummern zurück.

Das nächste Kapitel wird nun die soeben beschriebenen Implementationen hinsichtlich ihrer Performance untersuchen.

KAPITEL 5 Performance-Messungen

In diesem Kapitel wird die Implementation hinsichtlich ihrer Leistung untersucht. Dabei werden die VVFS-Erweiterungen einem Vanilla-Kernel[1] gegenübergestellt, um so den Overhead ermitteln zu können. Hierbei wird insbesondere eine Verschlechterung der Zugriffszeit mit steigender Anzahl von Verzeichniseinträgen und wachsender Dateigröße erwartet, da mehr Dateinamen verglichen bzw. eine größere Datenmenge kopiert werden muss. Vor der Analyse soll jedoch die Messumgebung beschrieben werden.

5.1 Messumgebung und -durchführung

Der Messrechner besaß eine Intel Pentium 4 CPU mit 3.06GHz und 512 KB Level 2 Cache, einen 512 MB großen Arbeitsspeicher sowie eine 80 GB SATA-Festplatte. Als Kernel wurden die 2.6.12er Vanilla-Quellen, sowohl mit als auch ohne VVFS-Patch, benutzt.

Das Testsystem war ein Gentoo-Linux, bei dem alle unnötigen und automatisch laufenden Dienste wie z.B. der Cron-Daemon sowie SSH- und Webserver abgeschaltet und sämtliche Netzwerkverbindungen gekappt wurden. So kann ausgeschlossen werden, dass die Tests unterbrochen werden bzw. sich die Ressourcen mit anderen Prozessen geteilt werden müssen.

Für das genaue Ermitteln von Zeiten wurde auf die Instruktion Read Time Stamp Counter (RDTSC) zurückgegriffen, die sich besonders für Benchmarks eignet. Sie wurde für Pentium CPUs eingeführt und schreibt die Anzahl von CPU-Zyklen seit dem Start bzw. Neustart als 64-Bit-Zahl in die Register EDX und EAX. Auflistung 5.1 zeigt das Makro `get_time`, das das Auslesen übernimmt.

Listing 5.1: *Benchmark-Instruktion* `rdtsc`

```
1 typedef struct
2 {
3    unsigned long hi;
4    unsigned long lo;
5 } time_586;
6
7 #define get_time(x) \
```

[1] Hierbei handelt es sich um den Referenz- oder auch Standard-Kernel ohne distributionsspezifische Erweiterungen. Er kann von `http://kernel.org` bezogen werden.

```
8    __asm__ __volatile__( "rdtsc" : "=d" ( x.hi ), "=a" ( x.lo ) )
```

Außerdem wurden bei allen Testfällen stets 24 Wiederholungen mit den gleichen Parametern durchgeführt und die Ergebnisse in ein Feld geschrieben. Das erste Auffinden einer Datei findet noch in Interaktion mit dem Datenträger statt. Da das Ergebnis jedoch im Dentry-Cache (vgl. 3.1.5 auf Seite 22) gespeichert wird, können alle folgenden Zugriffe eine enorme Leistungsverbessung erfahren. Werden im ersten Fall noch 70μs für das Öffnen einer Datei benötigt, so gelingt mit Hilfe des Zwischenspeicherns ein Zugriff in ca. 0,8μs.

Um gleiche Bedingungen für alle Versuche zu schaffen, könnte einerseits versucht werden durch das Aus- und Wiedereinhängen des Dateisystems den Dentry-Cache zu leeren. Die resultierenden Ergebnisse entsprechen zwar der Praxis, variieren jedoch untereinander sehr stark, da die Hardware ebenfalls Zwischenspeicher besitzt. Außerdem verursacht gerade die Hardware die größte Latenz, wodurch die Unterschiede, die durch die Veränderung der Software entstehen, weniger genau gemessen werden können.

Aus diesem Grund wurde eine bewusste Entscheidung für das Messen auf der Basis des Dentry-Cache getroffen. Alle Messungen profitieren im gleichen Maße von ihm. Um jedoch nicht die Durchschnittsbildung der 24 Wiederholungen zu verfälschen werden stets die beiden ersten Versuche ignoriert und als Aufwärmphase gedeutet. Anschließend wird die restliche Liste sortiert und die beiden schlechtesten ebenfalls verworfen. So bleiben Wackler, die z.B. durch die Unterbrechung des Messprozesses durch das Betriebsystem entstehen können, ohne Auswirkungen auf die Messreihe. Das Endergebnis eines Testfalls entsteht somit aus dem Durchschnitt von 20 Wiederholungen.

5.2 Anzahl der Verzeichniseinträge

Vor dem Anlegen einer neuen Dateiversion werden alle Verzeichniseinträge durchlaufen, um die größte Version einer Datei zu finden. Aus diesem Grund wurde zunächst das Verhalten der Versionserweiterungen in Abhängigkeit von der Anzahl von Verzeichniseinträgen untersucht. Abbildung 5.1(a) zeigt die entsprechenden Ergebnisse.

Auf der x-Achse ist die Anzahl von Verzeichniseinträgen abgetragen, die zusätzlich zu der zu öffnenden Datei in dem Testverzeichnis existieren. Diese wurde von 0 bis 9998 in zweier Schritten variiert. Somit wurden pro Messreihe 5000 mal 24 Messungen durchgeführt. Die y-Achse zeigt in logarithmischer Darstellung die Latenz des Open-Aufrufs gemessen in Mikrosekunden.

Die unterste, pinke Messungen stellt ein Öffnen mit Schreibrechten im 2.6.12er Vanilla-Kernel da. Unabhängig von den Verzeichniseinträgen werden hierbei durchschnittlich 0,8μs benötigt. Dieser Messung ist ein 2.6.12er Kernel mit den Versionserweiterungen gegenübergestellt.

Die blaue Kurve zeigt das Öffnen in einem Verzeichnis mit ausgeschalteter Versionierung (DirMaxV = 0). Hierfür werden konstant 1.2μs benötigt. Die Diskrepanz von 0.4μs ist durch das Überprüfen von DirMaxV zu erklären. Hierfür muss die unter `getmaxversions` registrierte Funktion aufgerufen werden. Sie bestimmt die Raw-Inode des Verzeichnisses zu der generischen Inode und liest die maximale Anzahl von Versionen aus. Wie das Diagramm zeigt, ist auch dieser Mechanismus unabhängig von den Verzeichniseinträgen.

Anders verhalten sich die Kurven bei angeschalteter Versionierung, die bei 10.000 Einträgen drei Millisekunden überschreiten und somit um einen Faktor 3000 höher sind.

Dieses Verhalten erklärt sich dadurch, dass für jeden Verzeichniseintrag zwei Stringkopien, ein Stringvergleich, eine Suche innerhalb einer Zeichenkette und die Umwandlung einer Zeichenkette in eine Zahl durchgeführt wird.

Während auf der grünen Kurve das Anlegen einer neuen Version ohne das Überschreiten der höchsten Versionsnummer zu sehen ist, so zeigt die rote Kurve den Fall in dem zusätzlich noch die älteste Version gelöscht werden muss. Wie die Darstellung mit linearer y-Achse im Diagramm 5.1(b) auf Seite 43 zeigt, wachsen beide Messungen linear. Dabei übersteigt die Latenz der roten Kurve mit 3228μs die der grünen Kurve mit 3054μs bei 10.000 Verzeichniseinträgen deutlich.

Der Grund für die Scherung beider Messungen ist jedoch nicht in den Versionserweiterungen zu suchen, denn beim Auffinden der aktuellsten Version wird auch stets die älteste Version in der Struktur `name_and_version` abgelegt (vgl. 4.4 auf Seite 32), wodurch kein erneuter Suchlauf gestartet werden muss. Vielmehr ist dieses Ergebnis in der Messumgebung zu suchen. So besaß die zu öffnende Datei den Dateinamen `foobar`, die zusätzlichen Verzeichniseinträge wurden jedoch mit `dummy0` - `dummy9999` benannt. Der Aufruf von `version_delete` benutzt im Fall von `Ext2` die Funktion `ext2_find_entry`, die innerhalb der Verzeichnisdatei einen entsprechenden Eintrag sucht. Durch die Namensgebung und die alphabetische Sortierung ist dies jedoch stets der letzte Eintrag, wodurch diese Messungen zugleich eine Obergrenze für den versionierten Open-Aufruf darstellen.

Dennoch besteht bei der Bestimmung der aktuellsten Version enormer Optimierungsbedarf. Der Abschnitt 6.1 auf Seite 45 diskutiert eine mögliche Lösung.

5.3 Dateigröße

Bei dem Anlegen einer neuen Version muss der Inhalt der ursprünglichen Datei kopiert werden. Aus diesem Grund wurde die Latenz des Open-Aufrufs in Abhängigkeit von der Dateigröße gemessen. Die Ergebnisse zeigt die Abbildung 5.3 auf Seite 44.

Dabei stellt die x-Achse die Dateigröße dar. Da das Kopieren blockweise geschieht, wurde sie in Schritten der zehnfachen Blockgröße zwischen 0 und 990 variiert. Bei einer Blockgröße von 4 KB war die letzte Datei somit 3960 KB groß. Die y-Achse beschreibt erneut die Latenz gemessen in Mikrosekunden.

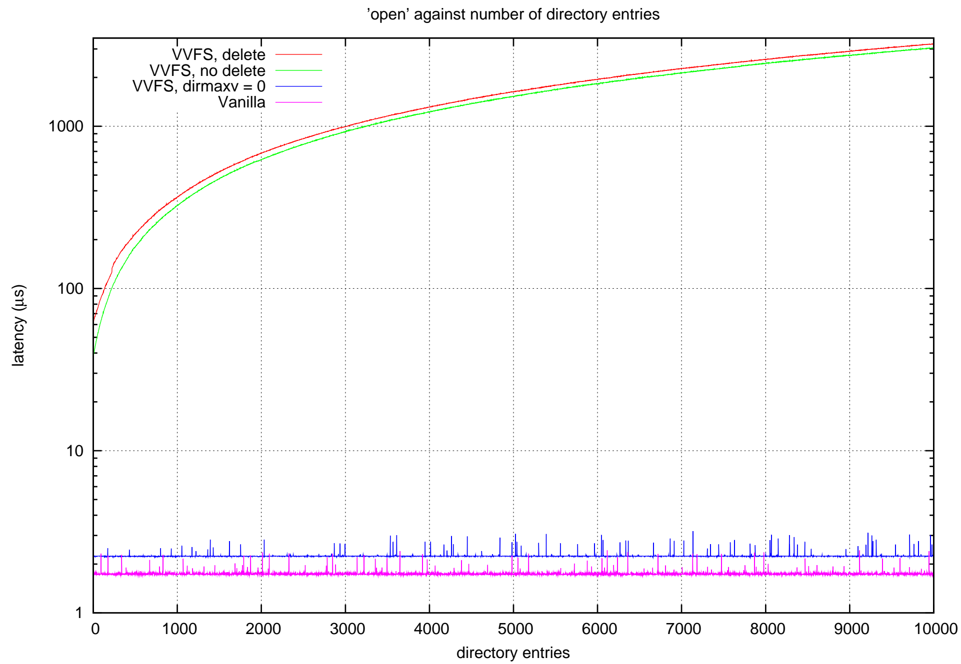
Die grüne Kurve zeigt das versionierte Öffnen einer Datei, jedoch ohne das Überschreiten von DirMaxV. Bei der maximal gemessenen Dateigröße benötigt der Aufruf 13998 Mikrosekunden. Dieser Messung wurde mit der roten Kurve ein Usermode-Programm gegenübergestellt, dass die gleiche Menge von Daten kopiert. Dabei wird in einer Schleife jeweils über die Funktion `read` ein Block aus einer Quelldatei gelesen und mittels `write` geschrieben. Liegen beide Messreihen bei kleiner Dateigröße dicht nebeneinander, so zeigen die häufigeren Kontextwechsel zwischen User- und Kernelmode mit wachsender Dateigröße bei dem Testprogramm bald ihre Wirkung.

Die vorgestellten Messergebnisse beruhen jedoch nur auf Operationen im Hauptspeicher und können nicht auf größere Dateien im Bereich von mehreren Gigabyte hochgerechnet werden. Reicht der Hauptspeicher nicht mehr aus, müssen die Daten zunächst auf den Datenträger durchgeschrieben werden. Bei den verwendenten SATA-Festplatten dürfte mit 60-90 MB/s die größte Durchsatzrate erreicht sein.

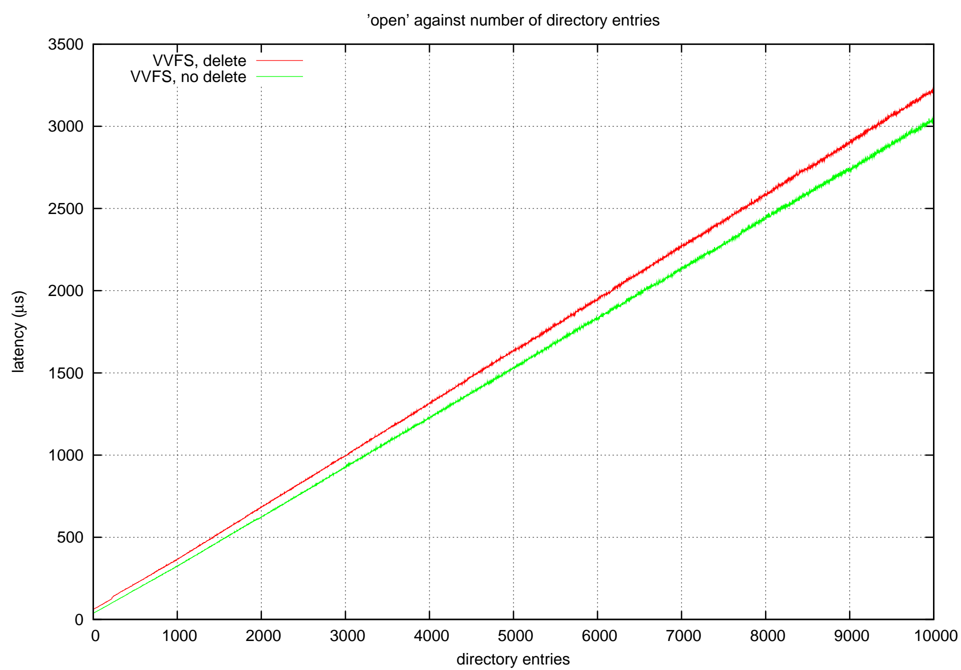
5.4 Fazit

Das Kopieren von Daten im Kernelspace zeigt leichte Geschwindigkeitsvorteile, da keine Kontextwechsel vorliegen. Beim Öffnen sehr großer Dateien kann ein Prozess längere Zeit blockieren. Wurde der Kernel auf einem Rechner mit nur einer CPU ohne die Option `CONFIG_PREEMPT = y` übersetzt, so blockiert das gesamte System für den Zeitraum des Kopierens.

Die Messergebnisse zeigen, das besonders bei der Ermittlung der neuen Versionsnummer noch erheblicher Optimierungsbedarf vorliegt.



(a) Logarithmische Darstellung



(b) Lineare Darstellung

Abbildung 5.1: *Zugriffszeit in Abhängigkeit von Verzeichniseinträgen*

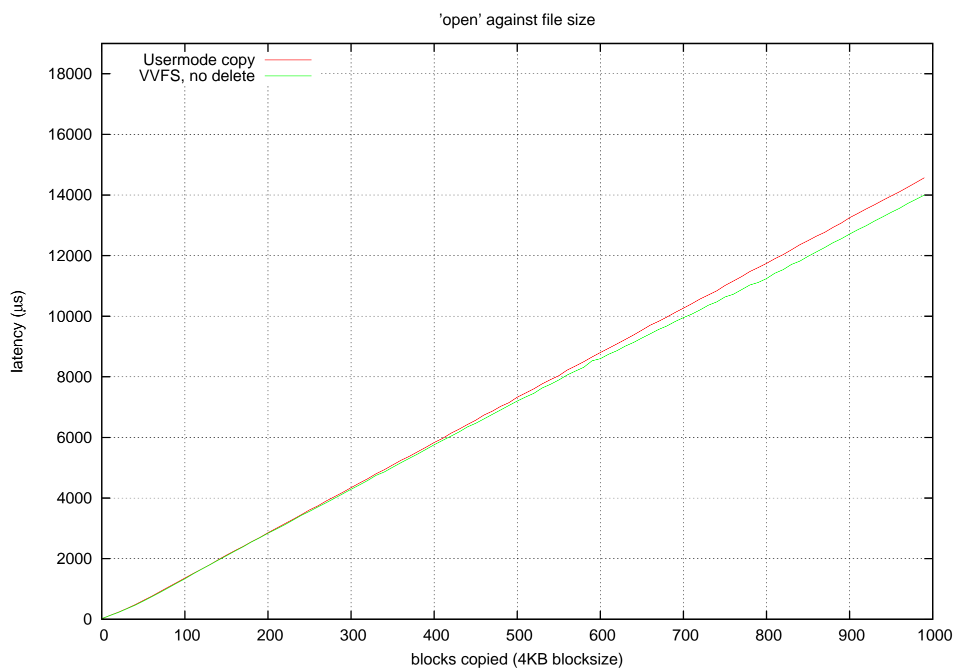


Abbildung 5.2: *Zugriffszeit in Abhängigkeit von der Dateigröße*

KAPITEL 6

Ausblick und Fazit

Abschließend soll durch dieses Kapitel eine Zusammenfassung der vorgestellten Arbeit gegeben werden. Zuvor beschreibt der Abschnitt Ausblick an welcher Stelle weitere Entwicklungsarbeit geleistet werden könnte.

6.1 Ausblick

Auch wenn diese Arbeit eine funktionierende Implementation beinhaltet, so sind dennoch zahlreiche Verbesserungen oder Erweiterungen denkbar.

6.1.1 Mountoptionen

Beim Einbinden eines Dateisystem könnte es wünschenswert sein, über so genannte Mountoptionen entscheiden zu können, ob generell eine Versionierung zur Verfügung gestellt wird. So könnte der Administrator festlegen, unterhalb welcher Mountpoints versionierte Dateien existieren dürfen.

6.1.2 Metadatenverwaltung

Die Implementation hat an mehren Stellen gezeigt, dass es Bedarf für eine generische Verwaltung von Metainformationen im virtuellen Dateisystem gibt, um beliebige zusätzliche Informationen dateispezifisch zu speichern. Mithilfe eines solchen Mechanismus könnte DirMaxV im VFS verwaltet werden, wodurch eine 100-prozentig generische Versionsverwaltung unabhängig von den einzelnen Dateisystemen möglich wäre.

Außerdem haben die Performancemessungen im vorherigen Kapitel gezeigt, dass es sehr aufwendig sein kann bei einer großen Anzahl von Verzeichniseinträgen die aktuellste oder die älteste Dateiversion zu bestimmen, da jeder Eintrag verglichen werden muss. Wäre es möglich unter einem Dateinamen diese Informationen zu registrieren, so sind enorme Performanceverbesserung zu erwarten.

6.1.3 Inkrementelle Versionierung

Bisher wird bei jedem Anlegen einer neuen Version eine vollständige Kopie der Vorgängerversion erstellt. Dies führt besonders bei großen Dateien mit kleinen Veränderungen

zwischen den Versionen zu einem sehr hohen, zusätzlichen Speicherplatzbedarf.

Abhilfe könnte das inkrementelle Abspeichern von Veränderungen schaffen. Ein mögliches Verfahren wäre es beim Open-Aufruf eine temporäre Datei zu erstellen, auf der alle Veränderungen durchgeführt werden. Wird die Datei geschlossen, so gilt es die Veränderungen zwischen dem Original und der Zwischenversion zu ermitteln und diese in einem geeigneten Format als neue Version abzulegen.

Gewinnt man durch diese Methode Speicherplatz, so ergeben sich jedoch auch zahlreiche Nachteile. Bei jedem Zugriff (auch der lesende) auf eine Datei muss zunächst eine temporäre Kopie erstellt werden, die bei Anwendung der Veränderungen aller Versionen entsteht. Dies kostet Zugriffszeit und ist nicht einfach zu implementieren. Wird z.B. eine Zwischenversion gelöscht, so dürfen ihre Veränderungen nicht verloren gehen, sondern müssen mit der nächst höheren Version verschmolzen werden.

6.1.4 Thread-safe

Die bisherige Implementation ist nicht Thread-safe. Ist der Kernel mit der Option `CONFIG_SMP` übersetzt und es existiert mehr als eine CPU, so kann es zu Inkonsistenzen kommen, falls ein Kernelthread gerade eine neue Version einer Datei erstellt und ein anderer Thread schreibend auf die Originaldatei zugreift. Das gleiche Problem kann auch nur mit einer CPU auftreten, sofern mit `CONFIG_PREEMPT` es dem System gestattet wird einen Kernelthread vorzeitig zu unterbrechen, um zunächst eine andere Aufgabe zu erledigen.

Wird z.B. eine versionierte Datei gelöscht, so erfolgt im Vanilla-Kernel zunächst eine Sperrung des Verzeichnisses, da dieses durch eine solche Operation verändert wird. Zu den Modifikationen dieses Projektes gehört es aber an dieser Stelle gegebenenfalls den Versionslink anzupassen, sofern die neueste Version gelöscht wurde. Auch hierbei handelt es sich um eine verzeichnisverändernde Operation, die ebenfalls eine Sperrung für andere Threads erfordert.

Ein Ausweg bestünde in der Definition eines neuen Locks oder der Übergabe eines Parameters speziell für diese Operation. So würde die Routine zum Umbenennen nur dann blockieren falls das Verzeichnis gesperrt ist, es sich jedoch bei dem Aufruf nicht um das Löschen einer versionierten Datei handelt.

6.1.5 Neueste Version

Die Darstellung der aktuellsten Version einer Datei wurde dem Verhalten von VMS nachempfunden. Auch dort kann die aktuellste Datei unter dem reinen Dateinamen oder dem Dateinamen mit der größten Versionsnummer angesprochen werden.

Eine abweichende und eventuell einfache Variante wäre es auf den Versionslink zu verzichten und alle Dateien bis auf die aktuellste mit Versionsnummern zu versehen. Der Open-Aufruf würde somit den Inhalt der neuesten Version (ohne Versionsnummer) in eine Datei mit der größten Versionsnummer kopieren. Das Verwalten des Versionslinks könnte eingespart werden.

6.2 Fazit

Im Rahmen dieser Semesterarbeit wurde ein Konzept für ein versionierendes Dateisystem unter Linux erarbeitet und durch eine Implementation umgesetzt. Der Grundstein

hierfür legte das Kapitel 2 mit der Erarbeitung der Mechanismen von VMS für eine Dateiversionierung im Dateisystem auf Copy-On-Write-Basis. Nach der Erläuterung des virtuellen Dateisystems von Linux im Kapitel 3 wurde die Implementation in Kapitel 4 beschrieben.

Dabei wurde unter anderem gezeigt, wie die maximale Anzahl von Versionen einer Datei gespeichert wird und welche Mechanismen beim Öffnen und Umbenennen einer Datei greifen. Diese wurden im Kapitel 5 hinsichtlich ihrer Performance untersucht.

Die Ergebnisse dieser Arbeit wurden in Form des Sourceforge[1]-Projekts `VVFS` veröffentlicht. Sämtliche Quellen können unter

`http://www.sourceforge.net/projects/vvfs`

aus einem CVS abgerufen werden. Zusätzlich stehen die Usermode-Programme sowie ein Patch für den 2.6.12er Vanilla-Kernel zum direkten Download bereit.

Weiterführende Arbeiten zur Verbesserung sind geplant, mit dem Ziel die Versionserweiterungen des VFS in den Mainstream-Kernel einfließen zu lassen.

[1]Sourceforge ist eine Plattform zur Verwaltung von Open-Source-Vorhaben mit über 100.000 Projekten und 1.000.000 registrierten Benutzern. Neben einem CVS-System zur Verwaltung des Sourcecodes und Speicher für eine Internetpräsenz werden nützliche Werkzeuge wie ein Bugtrackingsystem zur Verfügung gestellt.

ANHANG A
Quellcode

A.1 Quelltexte der Kernelmodifikationen

A.1.1 Erweiterungen im VFS

Listing A.1: `include/asm-generic/errno.h`

```
1  /* VERSION SUPPORT */
2
3  [ 107 additional lines ]
4
5  /* VERSION SUPPORT - BEGIN */
6  #define EVERSIONMAX 150 /* Maximum number of versions exceeded */
7  #define ENOTLNK  151 /* The file is not a link */
8  /* VERSION SUPPORT - END */
9
10 #endif
```

Listing A.2: `include/linux/fs.h`

```
1  /* VERSION SUPPORT */
2
3  [ 200 additional lines ]
4
5  /* VERSION SUPPORT - BEGIN */
6  #define FIGETMAXVERSIONS _IO(0x01,1) /* get number of versions */
7  #define FISETMAXVERSIONS _IO(0x01,2) /* set number of versions */
8
9  #define VERSION_SEPARATOR  '#'
10 #define VERSION_MAX     65000
11 #define VERSION_NAME_LEN  255
12 /* VERSION SUPPORT - END */
13
14 [ 716 additional lines ]
15
16 /* VERSION SUPPORT - BEGIN */
17 #define VERSION_SUPPORT(i)  (i && i->i_op && i->i_op->getmaxversions)
18 #define VERSION_SUPPORT_ENABLED(i) \
19    (VERSION_SUPPORT(i) && i->i_op->getmaxversions(i))
20
21 struct name_and_version {
22    char * name;      /* Reference name to search for. */
```

```
23     int max_version; /* Largest version found */
24     int pre_max_version; /* The second largest version found */
25     int min_version; /* Smallest version found */
26     int version_count; /* Number of different versions found */
27  };
28  /* VERSION SUPPORT - END */
29
30  [ 34 additional lines ]
31
32  struct inode_operations {
33
34     [ 21 additional lines ]
35
36     /* VERSION SUPPORT - START */
37     int (*getmaxversions) (struct inode *);
38     int (*setmaxversions) (struct inode *, int);
39     /* VERSION SUPPORT - END */
40  };
41
42  [ 289 additional lines ]
43
44  /* VERSION SUPPORT - BEGIN */
45  extern struct file *version_dentry_open(struct nameidata * nd,
46         const char *, int flags, int namei_flags, int mode, int copy);
47
48  /* fs/namei.c */
49
50  extern void version_separate(const char *, char *, int *);
51  extern void version_file_separate(const char *, char *, char *);
52  extern int version_delete(struct inode *, char *, struct name_and_version *);
53  extern int version_concatenate(char *, const char *, int);
54  extern int version_link(const char *, const char *);
55  extern int version_latestfile(char *, const char *, struct name_and_version *);
56
57  /* fs/file.c */
58
59  extern int version_copy(struct file *, struct file *);
60  /* VERSION SUPPORT - END */
61
62  [ 475 additional lines ]
```

Listing A.3: `fs/file.c`

```
 1  /* VERSION SUPPORT */
 2
 3  [ 15 additional lines ]
 4
 5  /* VERSION SUPPORT - BEGIN */
 6  //#include <linux/syscalls.h>
 7  #include <asm/uaccess.h>
 8
 9
10  /*
11   * This function copies data from one file pointer to another. Since
12   * generic_file_sendfile was restricted to copy data from a mm_file to a
13   * socket there is no other convinience function available.
14   */
15  int version_copy(struct file * src_file, struct file * dst_file)
16  {
17     unsigned long page = 0;
18     char * buffer;
19     mm_segment_t orgfs;
```

```
20     int read = 0;
21     int written = 0;
22     int retn = 0;
23     struct inode * src_inode;
24     struct inode * dst_inode;
25
26     /* Save the current address space */
27     orgfs = get_fs();
28
29     /*
30      * Since generic_file_read is build to return the bytes read to a user
31      * space pointer, we have to change this because the buffer lies in kernel
32      * space.
33      */
34     set_fs(KERNEL_DS);
35
36     /* Get one free memory page */
37     page = __get_free_page(GFP_KERNEL);
38
39     buffer = (char *) page;
40
41     while ((read = vfs_read(src_file, buffer, PAGE_SIZE,
42                 &src_file->f_pos)) > 0) {
43        written = vfs_write(dst_file, buffer, read, &dst_file->f_pos);
44
45        retn = -EFAULT;
46        if (!written)
47           goto __finally;
48
49        retn = written;
50        if (written < 0)
51           goto __finally;
52     }
53
54     src_inode = src_file->f_dentry->d_inode;
55     dst_inode = dst_file->f_dentry->d_inode;
56
57     dst_inode->i_mode = src_inode->i_mode;
58     dst_inode->i_ctime = src_inode->i_ctime;
59     dst_inode->i_mtime = CURRENT_TIME;
60     dst_inode->i_atime = CURRENT_TIME;
61
62     src_inode->i_atime = CURRENT_TIME;
63
64     mark_inode_dirty(src_inode);
65     mark_inode_dirty(dst_inode);
66
67     retn = 0;
68  __finally:
69     free_page(page);
70
71     /* Restore the original memory space */
72     set_fs(orgfs);
73
74     return retn;
75  }
76  /* VERSION SUPPORT - END */
77
78  [ 237 additional lines ]
```

Listing A.4: `fs/ioctl.c`

```
1 /* VERSION PATCH */
2
3 [ 78 additional lines ]
4
5 /*
6  * When you add any new common ioctls to the switches above and below
7  * please update compat_sys_ioctl() too.
8  *
9  * vfs_ioctl() is not for drivers and not intended to be EXPORT_SYMBOL()'d.
10  * It's just a simple helper for sys_ioctl and compat_sys_ioctl.
11  */
12 int vfs_ioctl(struct file *filp, unsigned int fd, unsigned int cmd, unsigned long
       arg)
13 {
14    unsigned int flag;
15    int on, error = 0;
16    /* VERSION SUPPORT - BEGIN */
17    struct inode * inode = filp->f_dentry->d_inode;
18    int max_versions = 0;
19    /* VERSION SUPPORT - END */
20
21    switch (cmd) {
22
23       [ 56 additional lines ]
24
25       /* VERSION SUPPORT - BEGIN */
26       case FIGETMAXVERSIONS:
27          // Read the number of versions allowed for the current dir.
28          if (!inode->i_op->getmaxversions)
29             return -EINVAL;
30          // Only directories have this attribute attached.
31          if (!S_ISDIR(inode->i_mode))
32             return -ENOTDIR;
33          return put_user(inode->i_op->getmaxversions(inode), (int __user *) arg);
34       case FISETMAXVERSIONS:
35          // Set the number of versions allowed for the current dir.
36          if (!inode->i_op->setmaxversions)
37             return -EINVAL;
38          // Only directories have this attribute attached.
39          if (!S_ISDIR(inode->i_mode))
40             return -ENOTDIR;
41          /* Check for write permissions */
42          if ((error = generic_permission(inode, MAY_WRITE, NULL)))
43             return error;
44          if (get_user(max_versions, (int __user *) arg ))
45             return -EFAULT;
46          return inode->i_op->setmaxversions(inode, max_versions);
47       /* VERSION SUPPORT - END */
48
49       [ 6 additional lines ]
50    }
51    return error;
52 }
53
54 [ 28 additional lines ]
```

Listing A.5: namei.c

```
1 /* VERSION SUPPORT */
2
3 [ 105 additional lines ]
```

```
4
5 /* VERSION SUPPORT - BEGIN */
6 int version_latestfile(char * filename, const char * basename,
7       struct name_and_version * extern_nav)
8 {
9    int error = -ENAMETOOLONG;
10   struct name_and_version tmp_nav;
11   struct name_and_version * nav = extern_nav == 0 ? &tmp_nav : extern_nav;
12
13   /* For the versioning extensions we need one additional charactor for the
14    * separator and five characters for the version number. */
15   if (strlen(basename) > VERSION_NAME_LEN - 7)
16      return error;
17
18   /* Search the current directory for all files matching the basename and
19    * determine the latest version */
20   nav->name = (char *)basename;
21   if ((error = version_find(nav)) < 0)
22      return error;
23
24   /* Increment the latest version by one since version numbers have to be
25    * increased on each write access. */
26   ++nav->max_version;
27
28   error = EVERSIONMAX;
29   if (nav->max_version > VERSION_MAX)
30      return error;
31
32   /* Build the new filename */
33   return version_concatenate(filename, basename, nav->max_version);
34 }
35
36 void version_separate(
37       const char *filename, /* complete name */
38       char *name,        /* file name without separator and version */
39       int *version_out) /* version */
40 {
41   char sep = VERSION_SEPARATOR;
42   char *sep_pos = NULL;
43   int ver_nr = 0;
44   char *endptr = NULL;
45   int tmp_version = 0;
46   int * version = version_out == 0 ? &tmp_version : version_out;
47
48   *version = -1;
49
50   /* Pointer the last occurance of the separator */
51   sep_pos = strrchr(filename, sep);
52
53   /* If no separator is found copy the whole name and return */
54   if (!sep_pos) {
55      strcpy(name, filename);
56      return;
57   }
58
59   /* Move one character behind the separator */
60   ++sep_pos;
61
62   /* Try to convert the digets behind the separator into a number */
63   ver_nr = simple_strtol(sep_pos, &endptr, 10);
64
65   /* If nothing follows the separator or the return value is different
```

```
66      * from NULL copy the whole name and return -1 */
67     if (!strlen(sep_pos) || *endptr != '\0') {
68        strcpy(name, filename);
69        return;
70     }
71
72     *version = ver_nr;
73     strncpy(name, filename, strlen(filename) - strlen(sep_pos) - 1);
74     return;
75  }
76
77  static int version_on_file_found(void * buf, const char * filename,
78        int length, loff_t offset, ino_t inode, unsigned int type)
79  {
80     int version = -1;
81     char tmp_filename[VERSION_NAME_LEN];
82     char tmp_name[VERSION_NAME_LEN];
83     struct name_and_version * nav = buf;
84
85     memset(tmp_filename, 0, VERSION_NAME_LEN);
86     memcpy(tmp_filename, filename, length);
87
88     /*
89      * Split the filename into the name and version parts.
90      */
91     memset(tmp_name, 0, VERSION_NAME_LEN);
92     version_separate(tmp_filename, tmp_name, &version);
93     if (version < 0)
94        return 0;
95
96     /*
97      * If the name part of the file name is equal to the basename we are
98      * looking for, then the version numbers are going to be compared.
99      * If a version bigger or smaller than the current extrem values is
100     * found, this version will be stored. Additionally the counter of overall
101     * versions is incremented.
102     */
103    if (!strcmp(nav->name, tmp_name)) {
104       ++nav->version_count;
105       if (version > nav->max_version) {
106          nav->pre_max_version = nav->max_version;
107          nav->max_version = version;
108       }
109       else if (version > nav->pre_max_version) {
110          nav->pre_max_version = version;
111       }
112       if (version < nav->min_version)
113          nav->min_version = version;
114    }
115
116    return 0;
117 }
118
119 int version_find(struct name_and_version * nav)
120 {
121    struct file * dir;
122
123    /*
124     * Find the newest version of the specified filename. If no version number
125     * is found at all the newest_version is set to -1.
126     */
127    if ((dir = filp_open(".", O_RDONLY, 0)) <= 0)
```

```
128       /* TODO welcher return Wert */
129       return -1;
130
131    nav->max_version = -1;
132    nav->pre_max_version = -1;
133    nav->min_version = VERSION_MAX;
134    nav->version_count = 0;
135
136    if (!dir->f_op || !dir->f_op->readdir)
137       /* TODO welcher return Wert */
138       return -1;
139
140    dir->f_op->readdir(dir, nav, version_on_file_found);
141    filp_close(dir, NULL);
142
143    return 0;
144 }
145
146 int version_concatenate(char * buf, const char * name, int version)
147 {
148    char version_string[6];
149    char sep[2] = {VERSION_SEPARATOR, '\0'};
150
151    /* Convert version number into a string */
152    snprintf(version_string, sizeof(version_string), "%d", version);
153    memset(buf, 0, VERSION_NAME_LEN);
154
155    /* Concatenate filename, separator and version_string */
156    strcpy(buf, name);
157    strcat(buf, sep);
158    strcat(buf, version_string);
159
160    return 0;
161 }
162
163 /* The function is a copy of 'sys_unlink' except there is no version code in
164  * here */
165 int version_do_unlink(const char * name)
166 {
167    int error = 0;
168    struct dentry *dentry;
169    struct nameidata nd;
170    struct inode *inode = NULL;
171
172    error = path_lookup(name, LOOKUP_PARENT, &nd);
173    if (error)
174       goto exit;
175    error = -EISDIR;
176    if (nd.last_type != LAST_NORM)
177       goto exit1;
178    //down(&nd.dentry->d_inode->i_sem);
179    dentry = lookup_hash(&nd.last, nd.dentry);
180    error = PTR_ERR(dentry);
181
182    if (!IS_ERR(dentry)) {
183       /* Why not before? Because we want correct error value */
184       if (nd.last.name[nd.last.len])
185          goto slashes;
186
187       inode = dentry->d_inode;
188       if (inode)
189          atomic_inc(&inode->i_count);
```

```
190
191         error = vfs_unlink(nd.dentry->d_inode, dentry);
192     exit2:
193         dput(dentry);
194     }
195
196     //up(&nd.dentry->d_inode->i_sem);
197     if (inode)
198         iput(inode); /* truncate the inode here */
199 exit1:
200     path_release(&nd);
201 exit:
202     return error;
203
204 slashes:
205     error = !dentry->d_inode ? -ENOENT :
206         S_ISDIR(dentry->d_inode->i_mode) ? -EISDIR : -ENOTDIR;
207     goto exit2;
208 }
209
210 /* Deletes the oldest version if the maximum number of version for one file
211  * was exceeded */
212 int version_delete(struct inode * dir, char * basename,
213         struct name_and_version * nav)
214 {
215     char filename[VERSION_NAME_LEN];
216
217     if (!dir->i_op || !dir->i_op->getmaxversions)
218         return -EVERSIONMAX;
219
220     /* If there exists more versions then specified by the directory inode,
221      * the oldest version must be deleted. */
222     if (nav->version_count >= dir->i_op->getmaxversions(dir)) {
223         /* Compose the filename of the basename and the oldest version. */
224         version_concatenate(filename, basename, nav->min_version);
225         version_do_unlink(filename);
226     }
227
228     return 0;
229 }
230
231 /* Determines the targetname the link is pointing to if the following
232  * conditions are fulfilled:
233  *   - dentry is a link
234  *   - link count == 1 (direct target)
235  *   - link dir and target dir are identical
236  *   - link name == basename of target */
237 int version_followlink(struct dentry * den_link, char * targetname)
238 {
239     struct inode * inode = den_link->d_inode;
240     struct nameidata nd;
241     char basename[VERSION_NAME_LEN];
242     char tmpname[VERSION_NAME_LEN];
243     int error = 0;
244
245     if (!S_ISLNK(inode->i_mode))
246         return -ENOTLNK;
247     nd.depth = 0;
248     if ((error = inode->i_op->follow_link(den_link, &nd)))
249         return error;
250     /* The regular file must be a direct link target */
251     if (nd.depth != 0)
```

```
252         return -ENOTLNK;
253     memset(tmpname, 0, VERSION_NAME_LEN);
254     strcpy(tmpname, nd.saved_names[0]);
255 
256 /* if ((error = path_lookup(tmpname, LOOKUP_PARENT, &nd )))
257         goto exit;
258     error = -ENOTLNK;
259     if (den_link->d_parent != nd.dentry)
260         goto exit; */
261 
262     memset(basename, 0, VERSION_NAME_LEN);
263     version_separate(tmpname, basename, 0);
264     error = -ENOTLNK;
265     if (strcmp(den_link->d_name.name, basename))
266         goto exit;
267     if (targetname)
268         strcpy(targetname, tmpname);
269 
270     error = 0;
271 exit:
272     /* path_release(&nd); */
273     return error;
274 }
275 /* VERSION SUPPORT - END */
276 
277 [ 1309 additional lines ]
278 
279 int open_namei(const char * pathname, int flag, int mode, struct nameidata *nd)
280 {
281 
282     [ 99 additional lines ]
283 
284 ok:
285     /* VERSION SUPPORT - BEGIN */
286     /* If Version support is enabled for the current directory, instead of
287      * truncating a file we will not copy the content in version_dentry_open */
288     if (VERSION_SUPPORT_ENABLED(nd->dentry->d_parent->d_inode))
289         flag &= ~O_TRUNC;
290     /* VERSION SUPPORT - END */
291     error = may_open(nd, acc_mode, flag);
292 
293     [ 54 additional lines ]
294 }
295 
296 [ 293 additional lines ]
297 
298 /* VERSION SUPPORT - BEGIN */
299 /* This function is only called by sys_unlink and distinguishes the following
300  * cases:
301  * 1. 'sys_unlink' was called on a link which points to a file with a version
302  *    number.
303  *    -> change the dentry to this file and delete the link
304  *    -> check whether the latest version is going to be deleted, if this is
305  *       the case determine the previous version and build a filename for
306  *       later link creation
307  * 2. 'sys_unlink' was called on a file with a version number
308  *    -> check whether the latest version is going to be deleted, if this is
309  *       the case determine the previous version and build a filename for
310  *       later link creation
311  * 3. 'sys_unlink' was called on a regular file without a version number or a
312  *     ordenary link
313  *     -> do nothing at all
```

```
314  */
315 static int version_unlink(struct dentry ** dentry, char * basename,
316       char * versionname, int * link)
317 {
318    struct dentry * file_dentry = *dentry;
319    struct nameidata nd;
320    int error = 0;
321    int version;
322    char targetname[VERSION_NAME_LEN];
323    struct name_and_version nav;
324    int is_link = 0;
325
326    if (!file_dentry->d_inode)
327       return 0;
328    memset(targetname, 0, VERSION_NAME_LEN);
329    memset(basename, 0, VERSION_NAME_LEN);
330    memset(versionname, 0, VERSION_NAME_LEN);
331
332
333    /* The file which is going to be deleted is a version link */
334    if (version_followlink(file_dentry, targetname) == 0) {
335       is_link = 1;
336
337       /* Change the dentry to the version file so that the code in
338        * 'sys_unlink' can delete it */
339       if ((error = path_lookup(targetname, LOOKUP_PARENT, &nd)))
340          return error;
341
342       dput(*dentry);
343       *dentry = lookup_hash(&nd.last, nd.dentry);
344       path_release(&nd);
345       error = PTR_ERR(*dentry);
346       if (IS_ERR(*dentry))
347          return error;
348
349       /* Delete the version link */
350       if ((error = version_do_unlink(file_dentry->d_name.name)))
351          return error;
352    } else {
353       /* Save the original file name */
354       strcpy(targetname, file_dentry->d_name.name);
355    }
356
357    version_separate(targetname, basename, &version);
358
359    /* If the file has no version number attached stop here */
360    if (version < 0)
361       return 0;
362
363    /* Search the current directory for all files matching the basename and
364     * determine the latest version */
365    nav.name = basename;
366    if ((error = version_find(&nav)))
367       return error;
368    version = (version == nav.max_version) ?
369       nav.pre_max_version : nav.max_version;
370
371    /* If there are no further versions, no new link has to be created */
372    if (version < 0) {
373       /* Special case: 'sys_unlink' was called directly on the lastest
374        * version and no further versions exists
375        * -> the version link pointing to this file has to be deleted
```

```
376          *    as well */
377
378         /* TODO: Check wheter the link exists. If this is the case return the
379          * error code of version_do_unlink
380         if (!is_link && error = version_do_unlink(basename)))
381            return error; */
382         if (!is_link)
383            version_do_unlink(basename);
384         return 0;
385      }
386
387      /* Build a file name which can be used as a link target later on */
388      if ((error = version_concatenate(versionname, basename, version)))
389         return error;
390
391      *link = 1;
392      return 0;
393   }
394
395   int version_link(const char * oldname, const char * newname)
396   {
397      int retn = 0;
398      mm_segment_t orgfs;
399
400      retn = version_do_unlink(newname);
401
402      /* Save the current address space */
403      orgfs = get_fs();
404      set_fs(KERNEL_DS);
405
406      retn = sys_symlink(oldname, newname);
407
408      /* Restore the original memory space */
409      set_fs(orgfs);
410
411      return retn;
412   }
413   /* VERSION SUPPORT - END */
414
415   /*
416    * Make sure that the actual truncation of the file will occur outside its
417    * directory's i_sem. Truncate can take a long time if there is a lot of
418    * writeout happening, and we don't want to prevent access to the directory
419    * while waiting on the I/O.
420    */
421   asmlinkage long sys_unlink(const char __user * pathname)
422   {
423      int error = 0;
424      char * name;
425      struct dentry *dentry;
426      struct nameidata nd;
427      struct inode *inode = NULL;
428      /* VERSION SUPPORT - BEGIN */
429      char basename[VERSION_NAME_LEN];
430      char versionname[VERSION_NAME_LEN];
431      int link = 0;
432      /* VERSION SUPPORT - END */
433
434      name = getname(pathname);
435      if(IS_ERR(name))
436         return PTR_ERR(name);
437
```

```
438     error = path_lookup(name, LOOKUP_PARENT, &nd);
439     if (error)
440        goto exit;
441     error = -EISDIR;
442     if (nd.last_type != LAST_NORM)
443        goto exit1;
444     down(&nd.dentry->d_inode->i_sem);
445     dentry = lookup_hash(&nd.last, nd.dentry);
446     error = PTR_ERR(dentry);
447     if (!IS_ERR(dentry)) {
448        /* VERSION SUPPORT - BEGIN */
449        if (VERSION_SUPPORT_ENABLED(nd.dentry->d_inode)) {
450           if ((error = version_unlink(&dentry, basename, versionname,
451                       &link)) < 0)
452              goto exit2;
453        }
454        /* VERSION SUPPORT - END */
455        /* Why not before? Because we want correct error value */
456        if (nd.last.name[nd.last.len])
457           goto slashes;
458        inode = dentry->d_inode;
459        if (inode)
460           atomic_inc(&inode->i_count);
461        error = vfs_unlink(nd.dentry->d_inode, dentry);
462     exit2:
463        dput(dentry);
464     }
465     up(&nd.dentry->d_inode->i_sem);
466     if (inode)
467        iput(inode); /* truncate the inode here */
468     /* VERSION SUPPORT - BEGIN */
469     /* TODO: This code must be moved into the locked area if the
470      * implementation of version_link makes no use of the big sys_calls which
471      * leads to a deadlock */
472     if (link)
473        error = version_link(versionname, basename);
474     /* VERSION SUPPORT - END */
475  exit1:
476     path_release(&nd);
477  exit:
478     putname(name);
479     return error;
480
481  slashes:
482     error = !dentry->d_inode ? -ENOENT :
483        S_ISDIR(dentry->d_inode->i_mode) ? -EISDIR : -ENOTDIR;
484     goto exit2;
485  }
486
487  [ 289 additional lines ]
488
489  /* VERSION SUPPORT - BEGIN */
490  static int version_src_rename(const char * old_name,
491        const struct nameidata * new_nd)
492  {
493     struct inode * new_inode = new_nd->dentry->d_inode;
494     const char * new_name = new_nd->last.name;
495     struct file * old_filp;
496     struct file * new_filp;
497     int flags = O_WRONLY | O_CREAT;
498     int namei_flags;
499     struct nameidata tmp_nd;
```

```
500     int error = 0;
501
502     printk( KERN_EMERG "version_src_rename␣0\n" );
503
504     printk( KERN_EMERG "oldname␣:␣%s\n", old_name);
505
506     printk( KERN_EMERG "version_src_rename␣1\n" );
507
508     old_filp = filp_open(old_name, O_RDONLY, 0);
509     error = PTR_ERR(old_filp);
510     if (IS_ERR(old_filp))
511        goto exit0;
512
513     printk( KERN_EMERG "version_src_rename␣3\n" );
514
515     if (VERSION_SUPPORT_ENABLED(new_inode)) {
516        namei_flags = flags;
517        if ((namei_flags + 1) & O_ACCMODE)
518           namei_flags++;
519
520        printk( KERN_EMERG "version_src_rename␣4\n" );
521
522        if ((error = open_namei(new_name, namei_flags, 0, &tmp_nd)) < 0)
523           goto exit1;
524
525        printk( KERN_EMERG "version_src_rename␣5\n" );
526
527        new_filp = version_dentry_open(&tmp_nd, flags, namei_flags,
528              namei_flags, 0);
529     } else {
530        new_filp = filp_open(new_name, O_WRONLY | O_CREAT, 0);
531     }
532
533     error = PTR_ERR(new_filp);
534     if (IS_ERR(new_filp)) {
535        path_release(&tmp_nd);
536        goto exit1;
537     }
538
539     printk( KERN_EMERG "version_src_rename␣6\n" );
540
541     if ((error = version_copy(old_filp, new_filp)) < 0)
542        goto exit2;
543
544     printk( KERN_EMERG "version_src_rename␣7\n" );
545  exit2:
546     filp_close(new_filp, NULL);
547  exit1:
548     filp_close(old_filp, NULL);
549  exit0:
550     printk( KERN_EMERG "␣␣error␣:␣%d\n", error );
551     return error;
552  }
553
554  static int version_dst_rename(struct inode * srcdir_inode,
555        struct dentry * src_dentry, const char * name)
556  {
557     struct nameidata dst_nd;
558     struct dentry * dst_dentry;
559     int error = 0;
560     char dst_name[VERSION_NAME_LEN];
561     char basename[VERSION_NAME_LEN];
```

```
562
563     printk( KERN_EMERG "version_dst_rename\n" );
564     printk( KERN_EMERG "␣name␣␣␣␣␣:␣%s\n", name );
565
566     version_separate(name, basename, 0);
567
568     printk( KERN_EMERG "␣basename␣:␣%s\n", basename );
569
570     if ((error = version_latestfile(dst_name, basename, 0)))
571        return error;
572
573     printk( KERN_EMERG "␣dst_name␣:␣%s\n", dst_name );
574
575     if ((error = path_lookup(dst_name, LOOKUP_PARENT, &dst_nd)))
576        return error;
577
578     dst_dentry = lookup_hash(&dst_nd.last, dst_nd.dentry);
579     error = PTR_ERR(dst_dentry);
580     if (IS_ERR(dst_dentry))
581        goto exit;
582
583     printk( KERN_EMERG "␣dentry␣␣␣:␣%s\n", dst_dentry->d_name.name );
584
585     if ((error = vfs_rename(srcdir_inode, src_dentry,
586           dst_nd.dentry->d_inode, dst_dentry)))
587        goto exit1;
588
589     error = version_link(dst_dentry->d_name.name, basename);
590
591  exit1:
592     dput(dst_dentry);
593  exit:
594     path_release(&dst_nd);
595     return error;
596  }
597  /* VERSION SUPPORT - END */
598
599  static inline int do_rename(const char * oldname, const char * newname)
600  {
601     int error = 0;
602     struct dentry * old_dir, * new_dir;
603     struct dentry * old_dentry, *new_dentry;
604     struct dentry * trap;
605     struct nameidata oldnd, newnd;
606     /* VERSION SUPPORT - BEGIN */
607     int do_unlock = 1;
608     /* VERSION SUPPORT - END */
609
610     [ 51 additional lines ]
611
612     /* VERSION SUPPORT - BEGIN */
613     if (VERSION_SUPPORT_ENABLED(oldnd.dentry->d_inode) ||
614        VERSION_SUPPORT_ENABLED(newnd.dentry->d_inode))
615     {
616        printk( KERN_EMERG "****************************************\n" );
617        printk( KERN_EMERG "␣␣old␣:␣%s␣␣␣␣new␣:␣%s\n", oldname, newname );
618
619        /* The source must be either a regular file or a version link */
620        if (S_ISREG(old_dentry->d_inode->i_mode) ||
621           !version_followlink(old_dentry, 0))
622        {
623           printk( KERN_EMERG "do_rename␣0\n" );
```

```
624         /* This is a dirty hack and has to be changed when the version
625          * code should become smp save. */
626         unlock_rename(new_dir, old_dir);
627         printk( KERN_EMERG "do_rename␣1\n" );
628         do_unlock = 0;
629 
630         if (VERSION_SUPPORT_ENABLED(oldnd.dentry->d_inode))
631            error = version_src_rename(oldname, &newnd);
632         else
633            error = version_dst_rename(oldnd.dentry->d_inode,
634                  old_dentry, newname);
635         goto exit5;
636      }
637   }
638   /* VERSION SUPPORT - END */
639 
640   error = vfs_rename(old_dir->d_inode, old_dentry,
641            new_dir->d_inode, new_dentry);
642 exit5:
643   dput(new_dentry);
644 exit4:
645   dput(old_dentry);
646 exit3:
647   /* VERSION SUPPORT - BEGIN */
648   if (do_unlock)
649   /* VERSION SUPPORT - END */
650   unlock_rename(new_dir, old_dir);
651 exit2:
652   path_release(&newnd);
653 exit1:
654   path_release(&oldnd);
655 
656 exit:
657   return error;
658 }
659 
660 [ 193 additional lines ]
```

Listing A.6: open.c

```
1 /* VERSION SUPPORT */
2 
3 [ 737 additional lines ]
4 
5 /* VERSION SUPPORT - BEGIN */
6 struct file *version_dentry_open(struct nameidata * nd, int flags,
7       int namei_flags, int mode, int copy)
8 {
9    int error = 0;
10   struct nameidata new_nd;
11   struct dentry * dentry = nd->dentry;
12   struct inode * inode = dentry->d_inode;
13   struct inode * current_dir_inode = dentry->d_parent->d_inode;
14   int filesize = inode->i_size;
15   char basename[VERSION_NAME_LEN];
16   char new_filename[VERSION_NAME_LEN];
17   int current_version = 0;
18   struct name_and_version nav;
19   struct file * filp = 0;
20   struct file * tmp_filp = 0;
21 
22   /* New versions of files are only created if the file is opened with write
```

```
23       * access. */
24      if ((flags & O_ACCMODE) == O_RDONLY)
25         goto out_no_versioning;
26
27      /* Versioning is only allowed for regular files. If the current inode is
28       * not a file we can stop here. */
29      if (!S_ISREG(inode->i_mode))
30         goto out_no_versioning;
31
32      /* Versioning support for hidden files will be disabled since many
33       * programms use this files for temporary data. */
34      if (dentry->d_name.name[0] == '.')
35         goto out_no_versioning;
36
37      /* Separates the filename from the version number */
38      memset(basename, 0, VERSION_NAME_LEN);
39      version_separate(dentry->d_name.name, basename, &current_version);
40
41      /* Versioning is sensless for empty files, which already exists */
42      if (filesize == 0 && (!(flags & O_TRUNC)) && current_version >= 0)
43         goto out_no_versioning;
44
45      if ((error = version_latestfile(new_filename, basename, &nav)))
46         goto error;
47
48      /* Open the new filename */
49      if ((error = open_namei(new_filename, namei_flags, mode, &new_nd)) < 0)
50         goto error;
51      filp = dentry_open(new_nd.dentry, new_nd.mnt, flags);
52      if (filp <= 0)
53         return filp;
54
55      /* If there is no content there is no reason create a new version */
56      if (filesize == 0 || copy == 0) {
57         path_release(nd);
58         goto out_versioning;
59      }
60
61      /* Copy the content of the original adressed file to the latest version */
62      tmp_filp = dentry_open(nd->dentry, nd->mnt, O_RDONLY);
63      if (tmp_filp <= 0)
64         return filp;
65      error = version_copy(tmp_filp, filp);
66      filp_close(tmp_filp, NULL);
67
68      if (error < 0)
69         goto error;
70
71   out_versioning:
72      /* Create a symlink from the basename to the latest version */
73      if ((error = version_link(new_filename, basename)) < 0)
74         goto error;
75      /* Delete the oldest version if the maximum version number was exceeded */
76      if ((error = version_delete(current_dir_inode, basename, &nav)) < 0)
77         goto error;
78      return filp;
79
80   out_no_versioning:
81      return dentry_open(nd->dentry, nd->mnt, flags);
82
83   error:
84      if (filp > 0)
```

```
85          filp_close(filp, NULL);
86      return ERR_PTR(error);
87  }
88  /* VERSION SUPPORT - END */
89
90
91  /*
92   * Note that while the flag value (low two bits) for sys_open means:
93   * 00 - read-only
94   * 01 - write-only
95   * 10 - read-write
96   * 11 - special
97   * it is changed into
98   * 00 - no permissions needed
99   * 01 - read-permission
100  * 10 - write-permission
101  * 11 - read-write
102  * for the internal routines (ie open_namei()/follow_link() etc). 00 is
103  * used by symlinks.
104  */
105 struct file *filp_open(const char * filename, int flags, int mode)
106 {
107     int namei_flags, error;
108     struct nameidata nd;
109     /* VERSION SUPPORT - BEGIN */
110     int copy = 1;
111     /* VERSION SUPPORT - END */
112
113     namei_flags = flags;
114     if ((namei_flags+1) & O_ACCMODE)
115         namei_flags++;
116     if (namei_flags & O_TRUNC)
117         namei_flags |= 2;
118
119     error = open_namei(filename, namei_flags, mode, &nd);
120     /* VERSION SUPPORT - BEGIN */
121     if (!error && VERSION_SUPPORT_ENABLED(nd.dentry->d_parent->d_inode)) {
122         /* If version support is enabled for the directory a file will never
123          * be truncated. If a clean file was requestet the copying of data
124          * between the versions will be switched of instead. */
125         copy = flags & O_TRUNC ? 0 : 1;
126         flags &= ~O_TRUNC;
127         return version_dentry_open(&nd, flags, namei_flags, mode, copy);
128     }
129     /* VERSION SUPPORT - END */
130     if (!error)
131         return dentry_open(nd.dentry, nd.mnt, flags);
132
133     return ERR_PTR(error);
134 }
135
136 [ 306 additional lines ]
```

A.1.2 Erweiterungen im Ext2-Dateisystem

Listing A.7: `include/linux/ext2_fs.h`

```
1 /* VERSION SUPPORT */
2
3 [ 206 additional lines ]
```

```
4
5 /*
6  * Structure of an inode on the disk
7  */
8 struct ext2_inode {
9
10    [ 27 additional lines ]
11    union {
12       struct {
13          __u8  l_i_frag; /* Fragment number */
14          __u8  l_i_fsize; /* Fragment size */
15          __u16 i_pad1;
16          __le16 l_i_uid_high; /* these 2 fields */
17          __le16 l_i_gid_high; /* were reserved2[0] */
18          /* VERSION SUPPORT - BEGIN */
19          __u16 l_i_max_versions;
20          __u16 l_i_reserved2;
21          /* __u32 l_i_reserved2; */
22          /* VERSION SUPPORT - END */
23       } linux2;
24
25       [ 14 additional lines ]
26
27    } osd2;        /* OS dependent 2 */
28 };
29
30 [ 10 additional lines ]
31
32 /* VERSION SUPPORT - BEGIN */
33 #define i_max_versions osd2.linux2.l_i_max_versions
34 /*#define i_reserved2 osd2.linux2.l_i_reserved2*/
35 /* VERSION SUPPORT - END */
36
37 [ 278 additional lines ]
```

Listing A.8: `fs/ext2/ext2.h`

```
1 /* VERSION SUPPORT */
2
3 [ 2 additional lines ]
4
5 /*
6  * second extended file system inode data in memory
7  */
8 struct ext2_inode_info {
9
10    [ 36 additional lines ]
11
12    /* VERSION SUPPORT - BEGIN */
13    __u16 i_maximum_versions;
14    /* VERSION SUPPORT - BEGIN */
15
16    [ 16 additional lines ]
17 };
18
19 [ 100 additional lines ]
```

Listing A.9: `fs/ext2/inode.c`

```
1 /* VERSION SUPPORT */
2
3 [ 1042 additional lines ]
```

```
4
5 void ext2_read_inode (struct inode * inode)
6 {
7     struct ext2_inode_info *ei = EXT2_I(inode);
8     ino_t ino = inode->i_ino;
9     struct buffer_head * bh;
10    struct ext2_inode * raw_inode = ext2_get_inode(inode->i_sb, ino, &bh);
11    int n;
12
13    [ 53 additional lines ]
14
15    /* VERSION SUPPORT - BEGIN */
16    ei->i_maximum_versions = raw_inode->i_max_versions;
17    /* VERSION SUPPORT - END */
18
19    [ 46 additional lines ]
20
21    return;
22 }
23
24 static int ext2_update_inode(struct inode * inode, int do_sync)
25 {
26    struct ext2_inode_info *ei = EXT2_I(inode);
27    struct super_block *sb = inode->i_sb;
28    ino_t ino = inode->i_ino;
29    uid_t uid = inode->i_uid;
30    gid_t gid = inode->i_gid;
31    struct buffer_head * bh;
32    struct ext2_inode * raw_inode = ext2_get_inode(sb, ino, &bh);
33    int n;
34    int err = 0;
35
36    [ 47 additional lines ]
37
38    /* VERSION SUPPORT - BEGIN */
39    raw_inode->i_max_versions = ei->i_maximum_versions;
40    /* VERSION SUPPORT - END */
41
42    [ 36 additional lines ]
43
44    mark_buffer_dirty(bh);
45    if (do_sync) {
46       sync_dirty_buffer(bh);
47       if (buffer_req(bh) && !buffer_uptodate(bh)) {
48          printk ("IO error syncing ext2 inode [%s:%08lx]\n",
49             sb->s_id, (unsigned long) ino);
50          err = -EIO;
51       }
52    }
53    ei->i_state &= ~EXT2_STATE_NEW;
54    brelse (bh);
55    return err;
56 }
57
58 [ 90 additional lines ]
```

Listing A.10: fs/ext2/namei.c

```
1 /* VERSION SUPPORT */
2
3 /* VERSION SUPPORT - BEGIN */
4
```

```
5 extern struct ext2_inode *ext2_get_inode(
6       struct super_block *, ino_t, struct buffer_head **);
7
8 /*
9  * Read the number of versions allowed for this directory from the extended
10  * inode struct entry 'i_maximum_version'
11  */
12 static int ext2_get_max_versions(struct inode * inode)
13 {
14    struct ext2_inode_info * ei = EXT2_I(inode);
15
16    /* Only a directory inode holdes the disired information */
17    if (!S_ISDIR(inode->i_mode))
18       return -ENOTDIR;
19
20    return ei->i_maximum_versions;
21 }
22
23 /*
24  * Write the number of versions allowed for this directory to the extended
25  * inode struct entry 'i_maximum_versions'.
26  */
27 static int ext2_set_max_versions(struct inode * inode, int max_versions)
28 {
29    struct ext2_inode_info * ei = EXT2_I(inode);
30
31    /* Setting this attribute is only allowed for directories. */
32    if (!S_ISDIR(inode->i_mode))
33       return -ENOTDIR;
34
35    ei->i_maximum_versions = max_versions;
36
37    /* The inode was modified so update the inode change time. */
38    inode->i_ctime = CURRENT_TIME_SEC;
39
40    /* Mark the inode as dirty to make sure the changes will be written */
41    mark_inode_dirty(inode);
42
43    return 0;
44 }
45
46 /* VERSION SUPPORT - END */
47
48 [ 346 additional lines ]
49
50 struct inode_operations ext2_dir_inode_operations = {
51
52    [ 17 additional lines ]
53
54    /* VERSION SUPPORT - BEGIN */
55    .getmaxversions = ext2_get_max_versions,
56    .setmaxversions = ext2_set_max_versions,
57    /* VERSION SUPPORT - END */
58 };
59
60 [ 10 additional lines ]
```

A.2 Quelltexte der Usermode-Tools

A.2.1 Verwalten von DirMaxV

Die folgende Auflistung zeigt den Quellcode des Usermode-Programms `dirmaxv` mit dessen Hilfe die maximale Anzahl von Versionen einer Datei pro Verzeichnis gesetzt und abgerufen werden können.

Listing A.11: `dirmaxv`

```
/*************************************************************************
 * This application comes to you under the terms of the
 * GNU General Public License.
 *
 * Copyright (C) 2005 Stephan Mueller and Sven Widmer
 ************************************************************************/

#define _GNU_SOURCE

CHANGE ME:
/*************************************************************************
 * This application needs some defines from the patched kernel file
 * 'include/linux/fs.h'. You can either copy the pathed kernel header to your
 * kernel header dir (e.g. /usr/include/linux) and uncommend the first include
 * line or you have to correct the path in the second line. */

/* #include <linux/fs.h> */
#include "/usr/src/PATH_TO_PATCHED_KERNEL_SOURCES/include/linux/fs.h"
/*************************************************************************/

#include <stdio.h>
#include <sys/ioctl.h>
#include <sys/file.h>
#include <stdlib.h>
#include <sys/types.h>
#include <string.h>
#include <unistd.h>

void printUsage( char *appName )
{
   printf( "usage: %s [-s number] <directory>\n", appName );
   exit( 1 );
}

int my_atoi( const char * nptr, int * value )
{
   char *endptr;

   if( nptr == 0 )
   {
      return 1;
   }

   *value = ( int ) strtol( nptr, &endptr, 10 );

   return nptr == endptr ? 1 : 0;
}


int main( int argc, char **argv )
{
```

```
52     int fd;
53     int num = -1;
54     char *dirName;
55
56     if( ( argc != 2 ) && ( argc != 4 ) )
57     {
58        printUsage( argv[ 0 ] );
59     }
60
61     if( argc == 2 )
62     {
63        dirName = argv[ 1 ];
64
65        if( ( fd = open( dirName, O_RDONLY | O_DIRECTORY ) ) < 0 )
66        {
67           perror( dirName );
68           exit( 1 );
69        }
70
71        if( ioctl( fd, FIGETMAXVERSIONS, &num ) < 0 )
72        {
73           perror( dirName );
74           close( fd );
75           exit( 1 );
76        }
77
78        printf( "%d\n", num );
79     }
80     else
81     {
82        if( strcmp( argv[ 1 ], "-s" ) )
83        {
84           printUsage( argv[ 0 ] );
85        }
86
87        dirName = argv[ 3 ];
88
89        if( ( fd = open( dirName, O_RDONLY | O_DIRECTORY ) ) < 0 )
90        {
91           perror( dirName );
92           exit( 1 );
93        }
94
95        if( my_atoi( argv[ 2 ], &num ) )
96        {
97           printf( "invalid␣number\n" );
98           close( fd );
99           printUsage( argv[ 0 ] );
100       }
101
102       if( ioctl( fd, FISETMAXVERSIONS, &num ) < 0 )
103       {
104          perror( dirName );
105          close( fd );
106          exit( 1 );
107       }
108    }
109
110    close( fd );
111    exit( 0 );
112 }
```

A.2.2 Aufräumen mithilfe von Purge

Die folgende Quellcodezeilen zeigen das Programm `purge`. Es dient zum Löschen älterer Versionen und zum Zurücksetzen der Dateiversionen.

Listing A.12: `purge`

```
/****************************************************************************
 * This application comes to you under the terms of the
 * GNU General Public License.
 *
 * Copyright (C) 2005 Stephan Mueller and Sven Widmer
 ***************************************************************************/

#define _GNU_SOURCE

CHANGE ME:
/****************************************************************************
 * This application needs some defines from the patched kernel file
 * 'include/linux/fs.h'. You can either copy the pathed kernel header to your
 * kernel header dir (e.g. /usr/include/linux) and uncommend the first include
 * line or you have to correct the path in the second line. */

/* #include <linux/fs.h> */
#include "/usr/src/PATH_TO_PATCHED_KERNEL_SOURCES/include/linux/fs.h"
/****************************************************************************/

#include <stdio.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <sys/file.h>
#include <dirent.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <linux/fs.h>

#define NAME_LEN VERSION_NAME_LEN
#define SEPARATOR VERSION_SEPARATOR

typedef struct list_entry lentry;
struct list_entry
{
   int version;
   lentry *prev;
   lentry *next;
};

void printUsage( char *appName )
{
   printf( "usage: %s [[-k keep_versions] | [-d delete_versions]] <file>\n",
       appName );
   exit( 1 );
}

int my_atoi( const char * nptr, int * value )
{
   char * endptr = NULL;

   if( nptr == 0 )
   {
```

```
55          return 1;
56       }
57
58       *value = ( int ) strtol( nptr, &endptr, 10 );
59
60       return nptr == endptr ? 1 : 0;
61    }
62
63    int concatenate(char * buf, const char * name, int version)
64    {
65       char version_string[ 6 ];
66       char sep[ 2 ] = { SEPARATOR, '\0' };
67
68       /* Convert version number into a string */
69       snprintf( version_string, sizeof( version_string ), "%d", version );
70       memset( buf, 0, NAME_LEN );
71
72       /* Concatenate filename, separator and version_string */
73       strcpy( buf, name );
74       strcat( buf, sep );
75       strcat( buf, version_string );
76
77       return 0;
78    }
79
80    static void version_separate(
81          const char *filename, /* complete name */
82          char *basename,    /* file name without separator and version */
83          int *version)      /* version */
84    {
85       char sep = SEPARATOR;
86       char *sep_pos = NULL;
87       int ver_nr = 0;
88       char *endptr = NULL;
89
90       memset( basename, 0, NAME_LEN + 1 );
91       *version = -1;
92
93       /* Pointer the last occurance of the separator */
94       sep_pos = strrchr( filename, sep );
95
96       /* If no separator is found copy the whole name and return */
97       if( !sep_pos )
98       {
99          strcpy( basename, filename );
100         return;
101      }
102
103      /* Move one character behind the separator */
104      ++sep_pos;
105
106      /* Try to convert the digets behind the separator into a number */
107      ver_nr = strtol( sep_pos, &endptr, 10 );
108
109      /* If nothing follows the separator or the return value is different
110       * from NULL copy the whole name and return -1 */
111      if( ( ! strlen( sep_pos ) ) || ( *endptr != '\0' ) )
112      {
113         strcpy( basename, filename );
114         return;
115      }
116
```

```
117     *version = ver_nr;
118     strncpy( basename, filename, strlen( filename ) - strlen( sep_pos ) - 1);
119     return;
120 }
121
122 int file_separate(
123        const char *path, /* dir and file name */
124        char *dir,        /* dir name without a trailing slash */
125        char *file)       /* filename */
126 {
127     char *sep_pos = NULL;
128
129     memset( dir, 0, NAME_LEN + 1 );
130     memset( file, 0, NAME_LEN + 1 );
131
132     /* Pointer the last occurance of the separator */
133     sep_pos = strrchr( path, '/' );
134
135     /* If no separator is found copy the whole path to the filename and set
136      * the directory to '.' */
137     if( !sep_pos )
138     {
139        dir[0] = '.';
140        strcpy( file, path );
141        return 0;
142     }
143
144     /* Move one character behind the separator */
145     ++sep_pos;
146
147     /* If nothing follows the separator no filename was specified */
148     if( strlen( sep_pos ) == 0 )
149     {
150        return 1;
151     }
152
153     strncpy( dir, path, strlen( path ) - strlen( sep_pos ) - 1);
154     strncpy( file, sep_pos, strlen( sep_pos ) );
155     return 0;
156 }
157
158 int interval_separate( const char * interval, int * begin, int * end )
159 {
160     char *sep_pos = NULL;
161
162     /* Pointer the last occurance of the separator */
163     sep_pos = strrchr( interval, '-' );
164
165     /* If no separator is found copy the whole path to the filename and set
166      * the directory to '.' */
167     if( ! sep_pos )
168     {
169        return my_atoi( interval, begin );
170     }
171
172     /* Move one character behind the separator */
173     ++sep_pos;
174
175     if( my_atoi( interval, begin ) )
176     {
177        return 1;
178     }
```

```
179
180     return my_atoi( sep_pos, end );
181  }
182
183  void list_print( lentry * head )
184  {
185     if( head )
186     {
187        fprintf( stderr, "%d␣", head->version );
188        list_print( head->next );
189     }
190  }
191
192  void list_reverse_print( lentry * tail )
193  {
194     if( tail )
195     {
196        fprintf( stderr, "%d␣", tail->version );
197        list_reverse_print( tail->prev );
198     }
199  }
200
201  int list_insert( lentry ** head, lentry ** tail, int version )
202  {
203     lentry *new;
204     lentry *iter = *head;
205
206     new = ( lentry * ) malloc( sizeof( lentry ) );
207     if( !new )
208     {
209        return -1;
210     }
211
212     new->prev = NULL;
213     new->next = NULL;
214     new->version = version;
215
216     /* No entry yet */
217     if( !iter )
218     {
219        *head = new;
220        *tail = new;
221        return 0;
222     }
223
224     /* New entry becomes the head of the list */
225     if( version < iter->version )
226     {
227        iter->prev = new;
228        new->next = iter;
229        *head = new;
230        return 0;
231     }
232
233     /* Find the greatest entry, which is smaller than or equal to the current
234      * one */
235     while( iter->next && iter->next->version <= version )
236     {
237        iter = iter->next;
238     }
239
240     new->prev = iter;
```

```
241     new->next = iter->next;
242     if( iter->next )
243     {
244        iter->next->prev = new;
245     }
246     iter->next = new;
247
248     /* If the new entry was inserted at the end of the list change the tail */
249     if( ! new->next )
250     {
251        *tail = new;
252     }
253
254     return 0;
255  }
256
257  int list_delete( lentry * lentry )
258  {
259     if( lentry == 0 )
260     {
261        return -1;
262     }
263
264     if( lentry->prev != 0 )
265     {
266        lentry->prev->next = lentry->next;
267     }
268
269     if( lentry->next != 0 )
270     {
271        lentry->next->prev = lentry->prev;
272     }
273
274     free( lentry );
275
276     return 0;
277  }
278
279  void list_free( lentry * head )
280  {
281     if( head )
282     {
283        list_free( head->next );
284        free( head );
285     }
286  }
287
288  /* Deletes all file versions except the 'num' newest ones and removes the
289   * corresponding entries from the list. */
290  int keep( lentry ** list_head, lentry ** list_tail, int num,
291        const char * file_name )
292  {
293     char path[ NAME_LEN + 1 ];
294     lentry * entry = 0;
295     lentry * iter = *list_tail;
296     lentry * new_head = 0;
297     int i = num;
298     int error = 0;
299
300     if( iter == 0 )
301     {
302        return -1;
```

```
303     }
304
305     /* Iterate 'num' times beginning at the end of the list in order to spare
306      * the 'num' newest list entries from erasement */
307     while( ( iter != 0 ) && ( i-- > 0 ))
308     {
309        iter = iter->prev;
310     }
311
312     /* If more versions should be kept than versions exists, do nothing at all
313      * */
314     if( iter == 0 )
315     {
316        return 0;
317     }
318
319     /* Remember the oldest version which should be kept in order to correct
320      * the head of the list later on. */
321     new_head = iter->next;
322
323     /* Delete the rest list entries. */
324     while( iter != 0 )
325     {
326        /* Build the relative path */
327        concatenate( path, file_name, iter->version );
328        /* printf( "deleteing file: %s", path ); */
329
330        /* Remove the file */
331        if( remove( path ) )
332        {
333           fprintf( stderr, "%s␣%d:␣remove()␣failed␣(%s)\n",
334              __FILE__, __LINE__, strerror( errno ) );
335           error = -1;
336           break;
337        }
338        /* printf( " done.\n"); */
339
340        entry = iter;
341        iter = iter->prev;
342        /* Delete the list entry */
343        list_delete( entry );
344     }
345
346     /* If no version should be kept modify the list tail. The tail can't be
347      * set to zero generally cause an error might be occured when delete the
348      * file */
349     if( num == 0 )
350     {
351        *list_tail = iter;
352     }
353
354     /* If no error occured correct the list head */
355     if( error == 0 )
356     {
357        *list_head = new_head;
358     }
359
360     return error;
361  }
362
363  /* Deletes all file versions within the specified range and removes them from
364   * the version list. */
```

```
365 int delete( lentry ** list_head, lentry ** list_tail, int from, int upto,
366       const char * file_name )
367 {
368    char path[ NAME_LEN + 1 ];
369    lentry * last_valid = 0;
370    lentry * iter = *list_head;
371    lentry * lentry = 0;
372    int error = 0;
373    int stop = 0;
374    int change_head = 0;
375 
376    if( ( iter == 0 ) || ( ( upto >= 0 ) && ( upto < from ) ) )
377    {
378       return -1;
379    }
380 
381    while( iter != 0 )
382    {
383       lentry = iter;
384 
385       if( iter->version >= from )
386       {
387          if( ( upto >= 0 ) && ( iter->version > upto ) )
388          {
389             break;
390          }
391 
392          /* Build the relative path */
393          concatenate( path, file_name, iter->version );
394          /* printf( "deleteing file: %s", path ); */
395 
396          /* Remove the file */
397          if( remove( path ) )
398          {
399             fprintf( stderr, "%s␣%d:␣remove()␣failed␣(%s)\n",
400                __FILE__, __LINE__, strerror( errno ) );
401             error = -1;
402             break;
403          }
404          /* printf( " done.\n"); */
405 
406          /* If the list head was deleted correct the head pointer later on
407           * */
408          if( iter == *list_head )
409          {
410             change_head = 1;
411          }
412 
413          /* If no upper border was specified delete just the version
414           * specified by 'from'. */
415          if( upto < 0 )
416          {
417             stop = 1;
418          }
419 
420          /* Delete the list entry */
421          list_delete( lentry );
422       }
423       else
424       {
425          last_valid = iter;
426       }
```

```
427
428         iter = iter->next;
429
430         if( stop == 1 )
431         {
432            break;
433         }
434      }
435
436      /* If end of the list was reached and no error occured or no element
437       * remains in the list, correct the tail */
438      if( ( ( iter == 0 ) && ( error == 0 ) ) || ( last_valid == 0 ) );
439      {
440         *list_tail = last_valid;
441      }
442
443      /* If the head was deleted or no element remains in the list, correct the
444       * head */
445      if( ( change_head == 1 ) || ( last_valid == 0 ) )
446      {
447         *list_head = last_valid;
448      }
449
450      return 0;
451   }
452
453   /* Purges all versions of a file by removing all gaps within the ordered list
454    * of version numbers. The version number '0' is assigned to the oldest
455    * version. Any succeeding file version is incremented by one.
456    * E.g.
457    * Files before : foo#2 foo#3 foo#23 foo#42 foo#43
458    * Files after : foo#0 foo#1 foo#2 foo#3 foo#4
459    */
460   int purge( lentry ** list_head, lentry ** list_tail, const char * file_name,
461         const char * dir_name )
462   {
463      char old_path[ NAME_LEN + 1 ];
464      char new_path[ NAME_LEN + 1 ];
465      lentry * new_head = 0;
466      lentry * new_tail = 0;
467      lentry * iter = *list_head;
468      int i = 0;
469      int link = 0;
470      int retn = 0;
471      int max_versions = 0;
472      int fd = 0;
473      int zero = 0;
474
475      if( ( fd = open( dir_name, O_RDONLY | O_DIRECTORY ) ) < 0 )
476      {
477         fprintf( stderr, "%s␣%d:␣open()␣failed␣(%s)\n",
478            __FILE__, __LINE__, strerror( errno ) );
479         return -1;
480      }
481
482      if( ioctl( fd, FIGETMAXVERSIONS, &max_versions ) < 0 )
483      {
484         fprintf( stderr, "%s␣%d:␣ioctl()␣failed␣(%s)\n",
485            __FILE__, __LINE__, strerror( errno ) );
486         return -1;
487      }
488
```

```
489    if( ( max_versions > 0 ) &&
490       ( ioctl( fd, FISETMAXVERSIONS, &zero ) < 0 ) )
491    {
492       fprintf( stderr, "%s␣%d:␣ioctl()␣failed␣(%s)\n",
493          __FILE__, __LINE__, strerror( errno ) );
494       return -1;
495    }
496
497    while( iter != 0 )
498    {
499       if( iter->version != i )
500       {
501          /* Build the relative paths */
502          concatenate( old_path, file_name, iter->version );
503          concatenate( new_path, file_name, i );
504          /* printf( "renameing file '%s' to '%s'", old_path, new_path ); */
505
506          if( rename( old_path, new_path ) )
507          {
508             fprintf( stderr, "%s␣%d:␣rename()␣failed␣(%s)\n",
509                __FILE__, __LINE__, strerror( errno ) );
510             goto error;
511          }
512          /* printf( " done.\n"); */
513
514          list_insert( &new_head, &new_tail, i );
515          link = 1;
516       }
517
518       ++i;
519       iter = iter->next;
520    }
521
522    if( link )
523    {
524       /* printf( "deleting old link: '%s'", file_name ); */
525       if( ( unlink( file_name ) ) && ( errno != ENOENT ) )
526       {
527
528          fprintf( stderr, "%s␣%d:␣unlink()␣failed␣(%s)\n",
529             __FILE__, __LINE__, strerror( errno ) );
530          goto error;
531       }
532       /* printf( " done.\n"); */
533
534       /* printf( "creating symlink from '%s' to '%s'", new_path, file_name ); */
535       if( symlink( new_path, file_name ) )
536       {
537          fprintf( stderr, "%s␣%d:␣symlink()␣failed␣(%s)\n",
538             __FILE__, __LINE__, strerror( errno ) );
539          return -1;
540       }
541       /* printf( " done.\n"); */
542    }
543
544    list_free( *list_head );
545    *list_head = new_head;
546    *list_tail = new_tail;
547
548    goto finally;
549
550 error:
```

```
551    retn = -1;
552
553 finally:
554    if( ( max_versions > 0 ) &&
555       ( ioctl( fd, FISETMAXVERSIONS, &max_versions ) < 0 ) )
556    {
557       fprintf( stderr, "%s␣%d:␣ioctl()␣failed␣(%s)\n",
558          __FILE__, __LINE__, strerror( errno ) );
559       return -1;
560    }
561
562    return retn;
563 }
564
565 int main( int argc, char **argv )
566 {
567    DIR *dir;
568    struct dirent *dirent;
569    char *path;
570    char dir_name[ NAME_LEN + 1 ];
571    char file_name[ NAME_LEN + 1 ];
572    char base_name[ NAME_LEN + 1 ];
573    int keep_versions = 0;
574    int delete_from = -1;
575    int delete_upto = -1;
576    int current_version = -1;
577    int error = 0;
578    lentry *list_head = 0;
579    lentry *list_tail = 0;
580
581    switch( argc )
582    {
583       /* Only the newest version is kept */
584       case 2:
585          path = argv[ 1 ];
586          keep_versions = 1;
587          break;
588       case 4:
589          path = argv[ 3 ];
590
591          /* Keep the k newest versions */
592          if( strcmp( argv[ 1 ], "-k" ) == 0 )
593          {
594             if( my_atoi( argv[ 2 ], &keep_versions ) )
595             {
596                fprintf( stderr, "%s␣%d:␣invalid␣number␣(%d)\n",
597                   __FILE__, __LINE__, keep_versions );
598                printUsage( argv[ 0 ] );
599             }
600             break;
601          }
602
603          /* Delete the d oldest versions */
604          if( strcmp( argv[ 1 ], "-d" ) == 0 )
605          {
606             if( interval_separate( argv[ 2 ], &delete_from, &delete_upto ) )
607             {
608                fprintf( stderr, "%s␣%d:␣invalid␣interval␣(%s)\n",
609                   __FILE__, __LINE__, argv[ 2 ] );
610                printUsage( argv[ 0 ] );
611             }
612             break;
```

```
613         }
614         printUsage( argv[ 0 ] );
615         break;
616
617      default:
618         printUsage( argv[ 0 ] );
619         break;
620   }
621
622   /* path name to long */
623   if( strlen( path ) > NAME_LEN )
624   {
625      fprintf( stderr, "%s␣%d:␣filename␣to␣long␣(max␣%d)\n",
626            __FILE__, __LINE__, NAME_LEN );
627   }
628
629   /* separate path into dir and file component */
630   if( file_separate( path, dir_name, file_name ) )
631   {
632      fprintf( stderr, "%s␣%d:␣invalid␣path␣(%s)\n",
633            __FILE__, __LINE__, path);
634   }
635
636   /* open the dir */
637   if( ! (dir = opendir( dir_name ) ) )
638   {
639      fprintf( stderr, "%s␣%d:␣opendir()␣failed␣(%s)\n",
640            __FILE__, __LINE__, strerror( errno ) );
641   }
642
643   /* determine all versions of the specified file and insert them into a
644    * list */
645   while( ( dirent = readdir( dir ) ) != 0 )
646   {
647      version_separate( dirent->d_name, base_name, &current_version );
648
649      if( ( current_version >= 0 ) &&
650         ( strcmp( base_name, file_name ) == 0 ) )
651      {
652         list_insert( &list_head, &list_tail, current_version );
653      }
654   }
655
656   /* list_print( list_head );
657   list_reverse_print( list_tail ); */
658
659   memset( file_name, 0 , NAME_LEN );
660   strcpy( file_name, dir_name );
661   file_name[ strlen( dir_name ) ] = '/';
662   strcat( file_name, base_name );
663
664   if( keep_versions > 0 )
665   {
666      error = keep( &list_head, &list_tail, keep_versions, file_name );
667      /* list_print( list_head );
668      list_reverse_print( list_tail ); */
669   }
670
671   if( error )
672   {
673      exit( 1 );
674   }
```

```
675
676     /* fprintf( stderr, "from: %d, upto: %d\n", delete_from, delete_upto ); */
677
678     if ( delete_from >= 0 )
679     {
680        error = delete( &list_head, &list_tail, delete_from, delete_upto,
681              file_name );
682        /* list_print( list_head );
683        list_reverse_print( list_tail ); */
684     }
685
686     if( error )
687     {
688        exit( 1 );
689     }
690
691     error = purge( &list_head, &list_tail, file_name, dir_name );
692     /* fprintf( stderr, "purge return: %d\n", error ); */
693
694     /* list_print( list_head );
695     list_reverse_print( list_tail ); */
696
697     if( error )
698     {
699        exit( 1 );
700     }
701
702     closedir( dir );
703     list_free( list_head );
704
705     exit( 0 );
706  }
```

ABKÜRZUNGSVERZEICHNIS

ACL .... Access Control List
AFS .... Andrew File System
CFS .... Cedar File System
COW .... Copy On Write
DCL .... Digital Command Language
DirMaxV .... Maximale Anzahl von Versionen einer Datei innerhalb eines Verzeichnisses
Ext FS .... Extended File System
ext3 .... Extended 3 File System
FAT .... File Allocation Table
FD .... File Descriptor
FID .... File Identification
GCC .... GNU Compiler Collection
LRU .... Least Recently Used
MFD .... Master File Directory
NFS .... Network File System
NTFS .... New Technology File System
NUM .... File Number
OCFS2 .... Oracle Cluster File System 2
ODS .... On-Disk Structure
RDTSC .... Read Time Stamp Counter
RFA .... Record File Address
RMS .... Record Management Services
RVN .... Relative Volume Number
SEQ .... File Sequence Number
SQL .... Structured Query Language
UIC .... User Identification Code
UML .... Usermode Linux
VCS .... Version Control System
VFS .... Virtual File System
Vim .... Vi IMproved

ABBILDUNGSVERZEICHNIS

1.1 Einordnung des Dateisystems . . . 1

2.1 Verzeichnishierarchie . . . 8
2.2 Abbildung von Verzeichniseinträgen in `INDEXF.SYS` . . . 10
2.3 VMS Systemdateien und ihre Funktion . . . 11
2.4 Teil des INDEXF.SYS Headers . . . 11

3.1 VFS-Layer zur Abstraktion von Dateisystemen . . . 15

4.1 Lesen und Setzen von DirMaxV . . . 31
4.2 Dateinamen mit gültigen/ungültigen Versionsnummern . . . 31
4.3 Darstellung der aktuellsten Dateiversion über Softlinks . . . 32
4.4 Löschen der aktuellsten Version . . . 32
4.5 Anlegen einer neuen Version . . . 33
4.6 Der veränderte Systemaufruf `sys_open` . . . 34
4.7 Der veränderte Systemaufruf `sys_rename` . . . 36
4.8 Beispiel eines `purge` Aufrufs . . . 37

5.1 Zugriffszeit in Abhängigkeit von Verzeichniseinträgen . . . 43
5.2 Zugriffszeit in Abhängigkeit von der Dateigröße . . . 44

LITERATURVERZEICHNIS

[1] Concurrent Version System.
URL http://www.cvshome.org

[2] FUSE.
URL http://fuse.sourceforge.net

[3] GMail.
URL http://gmail.google.com

[4] GMailFS.
URL http://richard.jones.name/google-hacks/gmail-filesystem/gmail-filesystem.html

[5] Gnome Storage.
URL http://www.gnome.org/~seth/storage

[6] Grid FTP.
URL http://www.globus.org/grid_software/data/gridftp.php

[7] Grid NFS.
URL http://www.citi.umich.edu/projects/gridnfs/GridNFS-SoW.pdf

[8] Subversion.
URL http://subversion.tigris.org

[9] The Globus Alliance.
URL http://www.globus.org

[10] The User-mode Linux Kernel Home Page.
URL http://user-mode-linux.sourceforge.net

[11] Win FS.
URL http://msdn.microsoft.com/data/winfs

[12] Albert S. Woodhull Andrew S. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice Hall, second edition, 1997.

[13] Dr. Rajkumar Buyya. What is a Grid?
URL http://www.gridcomputing.com/gridfaq.html

[14] D. G. Korn und E. Krell. The 3-D File System. In *Proceedings of the USENIX Summer Conference*, pages 147–156, 1989.

[15] D. K. Gifford, R. M. Needham und M. D. Schroeder. The Cedar File System. In *Communication of the ACM*, pages 288–298, 1988.

[16] D. S. Santry, M. J. Feeley, N. C. Hutchinson, A. C.Veitch, R.W. Carton, und J. Ofir. Deciding When to Forget in the Elephant File System. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 110–123. ACM Press, 1999.

[17] David Donald Miller. *OpenVMS Operating System Concepts*. Digital Press, United States of America, second edition, 1997.

[18] J. J. Kistler und M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 213–225. ACM Press, 1991.

[19] S. Quinlan. A Cached WORM File System. In *Software Practice and Experience*, pages 1289–1299, 1991.

[20] Wolfgang Mauerer. *Linux Kernelarchitektur*. Hanser, München Wien, 2004.

[21] Z. N. J. Peterson and R. C. Burns. Ext3cow: The design, Implementation, and Analysis of Metadata for a Time-Shifting File System. In *Technical Report HSSL-2003-03, Computer Science Department, The Johns Hopkins University*, 2003. http://hssl.cs.jhu.edu/papers/peterson-ext3cow03.pdf