Kurzfassung

In den letzten Jahren hat das Konzept der objektorientierten Programmierung mit der Sprache Java^{TM} als einem populären Vertreter mehr und mehr an Bedeutung gewonnen. Die Einsicht, dass die traditionelle funktionale Programmierung bei sehr großen und komplexen Projekten an ihre Grenzen stößt, hat dazu nicht wenig beigetragen. Da ein Betriebssystem durchaus umfangreich ist und eine hohe Komplexität aufweisen kann, liegt es nahe, beim Design und bei der Implementierung auf Techniken der objektorientierten Programmierung zurückzugreifen. Dabei wird zwangsläufig von den Vorteilen der OOP profitiert werden; als Beispiele seien hier Reduktion der Komplexität, Abstraktion, Modularität und Wiederverwendbarkeit des Quellcodes genannt.

Im Laufe dieser Studienarbeit wurde eine Festplattensteuerung für die IDE-Schnittstelle und darauf aufsetzend ein Dateisystem in der Sprache Java implementiert, wobei viel Wert darauf gelegt wurde, das Dateisystem „offen", d.h. erweiter- und austauschbar zu gestalten. Das System beinhaltet ein virtuelles Dateisystem und eine konkrete Implementierung, die vollständig und funktionsfähig ist. Das Festplatten-Layout wurde dabei kompatibel zum weit verbreiteten Linux-Dateisystem *Ext2* gehalten. Mit Linux stand eine mächtige Testumgebung zur Verfügung, die darüber hinaus die Möglichkeit zum Performance-Vergleich bietet. Der Festplattentreiber wurde komplett in Java realisiert und greift auf einen kleinen Satz von Routinen zurück, die direkt mit der Hardware kommunizieren und deshalb in einer maschinennahen Sprache verfasst sind. Er erkennt und identifiziert vorhandene Controller und Festplatten und steuert den Datentransfer zwischen diesen Komponenten und dem Hauptspeicher.

Die Festplattensteuerung und das Dateisystem sind Teil des Betriebssystems *metaXaOS*, das die sichere Ausführung von Java-Programmen erlaubt, bei denen konformes Verhalten nicht gewährleistet ist. *metaXaOS* besitzt einen vertrauenswürdigen Kern, der als robuste Basis für Betriebssystemkomponenten und Applikationen dient.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einführung

1.1 Zielsetzung und Rahmenbedingungen

Diese Studienarbeit ist Teil eines größeren Projekts, dessen Ziel es ist, ein Betriebssystem (*metaXaOS*) unter der Verwendung der Programmiersprache Java zu implementieren. Als Basis dient ein minimaler, in C geschriebener Betriebssystemkern, der unter besonderer Beachtung der Kriterien Stabilität und Sicherheit entwickelt wurde. Dieser Teil des Betriebssystems soll nur maschinennahe Operationen realisieren und *native*-Funktionen zur Verfügung stellen, d.h. C-Funktionen, die als Methoden in Java aufrufbar sind. Die darauf aufsetzenden Java-Klassen sind als isolierte, austauschbare Module (*packages* oder Pakete in Java) konzipiert, die die eigentlichen Betriebssystemfunktionen implementieren (Speicherverwaltung, Gerätesteuerung, Scheduler, Dateisystem, Netzwerkprotokollstack usw.).

Die Zielsetzung des in C verfassten Kerns ist also, eine minimale und sichere Grundlage zu schaffen, die den Zugriff auf die Hardware durchführt und kontrolliert. Minimal bedeutet, dass soviel Funktionalität wie möglich in die Java-Klassen ausgelagert wird; nur Zugriffe auf die Hardware sollen ausschließlich im Kern stattfinden. Mit sicherer Grundlage ist gemeint, dass die C- bzw. *native*-Funktionen stabil und verlässlich laufen.

Der in Java geschriebene Teil des Betriebssystems soll unter Verwendung der *native*-Funktionen alle Aspekte eines Betriebssystems realisieren. Die Klassen sind in abgeschlossene Module zu gruppieren, die austausch- und erweiterbar sind. Dadurch hat der Anwender die Möglichkeit, den Betriebssystemkern individuell zusammenzustellen und an eine bestimmte Umgebung bzw. Applikation anzupassen. Die Programmiersprache Java bietet mehrere Vorteile gegenüber C: der Programmtext ist leicht wartbar und einfacher zu lesen als C. Durch die Verwendung der mächtigen Java-Klassenbibliothek von Sun ist eine schnelle und effiziente Entwicklung möglich. Deshalb ist es ein wesentliches Ziel des Projekts, den C-Teil möglichst klein zu halten. Die Verwendung von Java-Klassen und -Paketen ermöglicht eine objektorientierte, modulare Gestaltung eines Betriebssystems mit allen damit verbundenen Vorteilen (Übersichtlichkeit und Wartbarkeit durch Hierarchiebildung und Kapselung von Implementationsdetails).

Natürlich ist der Einsatz von Java mit Performanceeinbußen verbunden, die aber wegen des experimentellen Charakters des Projekts sicher entschuldbar ist. Der aktuelle Stand der Implementierung bietet noch Raum für Optimierungen (z.B. der Einsatz eines Just-in-time Compilers).

Als Praxisteil der Studienarbeit wurde ein integraler Bestandteil eines Betriebssystems in Java realisiert: das Dateisystem. Einige herausragende Merkmale dieses Dateisystems sind: kompatibel zu *Ext2* von Linux, Aufteilung in Blockgruppen, Partitions- und Dateigröße nahezu unbegrenzt, automatische Erstellung von Kopien wichtiger Verwaltungsdaten und Einbettung in ein virtuelles Dateisystem mit den aus Unix bekannten Eigenschaften (homogener Verzeichnisbaum, Erweiterbarkeit). Das Dateisystem besitzt im Gegensatz zu Journaling- oder Logorientierten Dateisystemen ein traditionelles Design, wodurch es zwar langsamer ist und bei Ausfällen einen inkonsistenten Zustand erreichen kann, aber auch wesentlich einfacher und übersichtlicher implementiert werden konnte. Als Basis für das Dateisystem dient ein ebenfalls in Java geschriebener IDE-Festplattentreiber, der als besondere Merkmale die Unterstützung von Busmaster-DMA (und damit Ultra-DMA) für die Ein-/Ausgabe und die automatische Erkennung vorhandener IDE-Controller, Festplatten und darauf enthaltener Partitionen aufweisen kann. Ein minimaler C-Kern stellt Funktionen für den Zugriff auf Register und Arbeitsspeicher zur Verfügung und ermöglicht die Verwendung von Hardware-Interrupts. Der Treiber verhindert eine fehlerhafte Bedienung der Controller und dient als Monitor für den Zugriff auf diese Ressource.

Folgende Kapitel stellen einen kurzen Überblick über die Rahmenbedingungen dieser Arbeit dar.

1.1.1 Die Programmiersprache Java

Die Programmiersprache Java wird von Sun Microsystems seit 1990 entwickelt und von den Erfindern folgendermaßen beschrieben ([Sunb]):

> „Java is simple, object-oriented, distributed, interpreted, robust, secure, architecture-neutral, portable, high-performance, multithreaded and dynamic.“

Java besitzt nur einen kleinen Satz von Sprachkonstrukten; die Syntax gleicht weitgehend der von C++. Um die Sprache einfach zu gestalten und Fehlerquellen zu beseitigen, wurde bei Java auf Zeiger, Mehrfachvererbung und dem Überladen von Operatoren verzichtet. Programme werden während des Kompilierens und zur Laufzeit überprüft. Um die Speicherverwaltung ebenfalls einfach zu gestalten, existiert ein Garbage Collector, der nicht mehr referenzierte Objekte erkennt und löscht; der Programmierer muss sich nicht um die Freigabe des Speichers kümmern.

Java ist plattform-unabhängig; Programme werden interpretiert und laufen in einer virtuellen Maschine, die an die jeweilige zugrundeliegende Architektur angepasst ist; sie werden in einen Bytecode übersetzt, um eine schnellere Ausführungsgeschwindigkeit zu erreichen.

Threadunterstützung ist in die Sprache bereits eingebaut; es besteht sogar die Möglichkeit, mit Objekten, Methoden oder auch nur Teilen von Methoden als Monitor gegenseitigen Ausschluss herzustellen. Außerdem sind Mechanismen vorgesehen, um verteilte Programmierung zu ermöglichen (Objektserialisation, Remote Method Invocation). Schließlich existiert eine mächtige und umfassende Klassenbibliothek, das *Application Programmer Interface* (API).

1.1.2 Die virtuelle Maschine *metaXa* und *metaXaOS*

Die Java Virtual Machine (JVM) *metaXa* setzt nicht auf einem Betriebssystem auf, d.h. Java-Programme in dieser Ablaufumgebung können auf die Hardware zugreifen. Damit ist es möglich, ein

Betriebssystem fast komplett in Java zu realisieren. Direkte Zugriffe auf die Hardware geschehen über *native*-Funktionen; die gesamte Funktionalität des Systems kann in Java implementiert werden. Im Gegensatz zu anderen Betriebssystem-Projekten wie JOS oder JavaOS ([JOS], [Suna]), die einen monolithischen Kern besitzen, ist *metaXa* modular aufgebaut und ermöglicht Reflexion und Metaprogrammierung in Java. Mit diesen Mechanismen lässt sich das Ausführungsverhalten von Objekten über eine Metaebene beeinflussen.

Das Betriebssystem *metaXaOS* setzt sich aus einem vertrauenswürdigen Kern (*iCore* und Kernhülle *iShell*) und nicht vertrauenswürdigen Komponenten, die in getrennten *Domains* ablaufen, zusammen. Der Kern soll gewährleisten, dass es Betriebssystemkomponenten nicht möglich ist, das System in einen instabilen Zustand zu bringen. Damit lassen sich „fremde“ Module einsetzen, deren Verhalten u.U. nicht konform ist, ohne den Ablauf des Systems gefährden zu können. Abbildung 1.1 ist eine schematische Darstellung des Systems zum Zeitpunkt dieser Arbeit und stimmt u.U. nicht mit der endgültigen Version überein.

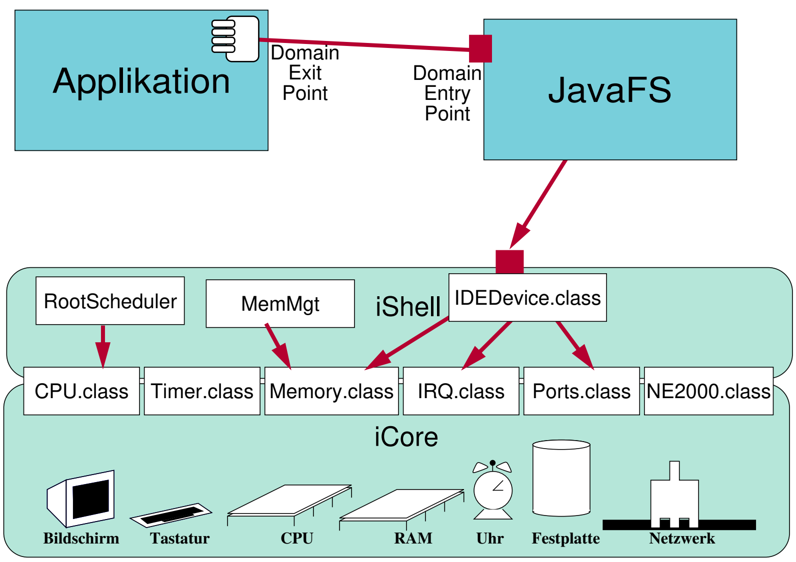


Abbildung 1.1: Aufbau des *metaXaOS*

1.1.3 *OSKit*

Das *OSKit* ist eine Umgebung für die Konstruktion von Betriebssystemkernen. Es besteht aus einem Satz von modularisierten Komponenten bzw. Bibliotheken, vergleichbar mit einem „Baukasten“ für Betriebssysteme. Die einzelnen Module können je nach Zielsetzung kombiniert und als Ergänzung eigener Entwicklungen verwendet werden. *OSKit* besteht aus einzelnen Bibliotheken, die die Infrastruktur eines Betriebssystems zur Verfügung stellen.

Innerhalb des *metaXaOS*-Projekts diente *OSKit* als Basis für die Entwicklung des Betriebssystems und wurde nach und nach durch eigene Implementierungen ersetzt. Dieser „modulare Abbau“ wird durch die Architektur des *OSKit* mit seinen austauschbaren Komponenten sehr begünstigt und erfordert nur geringe Änderungen am Quelltext.

1.2 Kapitelübersicht

Im nächsten Kapitel wird die Ansteuerung der IDE-Controller bzw. -Festplatten behandelt, wobei auf wichtige Konzepte bei der Treiberprogrammierung und auf besondere Merkmale der IDE-Spezifikation eingegangen wird. Das dritte Kapitel führt den Begriff des Dateisystems ein, wobei kurz die (historische) Entwicklung und die wesentlichen Eigenschaften traditioneller Dateisysteme dargelegt werden, bevor die im Laufe dieser Studienarbeit entwickelte Implementierung näher betrachtet wird. Anschließend wird in Kapitel 4 kurz auf die Schnittstelle zwischen Java und anderen Programmiersprachen, die bei der Entwicklung der maschinennahen Betriebssystemebene verwendet wurden, eingegangen. Es folgen in Kapitel 5 eine Beschreibung des Linux-Testsystems und abschließende Betrachtungen. Mit der Beschreibung einer Implementierung, die auf die Funktionalität des Dateisystems zurückgreift, und der Dokumentation der wichtigsten Klassen schließt der Anhang diese Arbeit ab.

1.3 Zusammenfassung

Die Rahmenbedingungen und Zielsetzungen der Arbeit wurden erläutert, wobei kurz auf Java als Implementierungssprache, *metaXa* als Ablaufumgebung und dem *OSKit* als Basis der Entwicklung des Betriebssystems eingegangen wurde. Schließlich wurde in einer kurzen Kapitelübersicht der Aufbau dieser Arbeit skizziert.

Kapitel 2

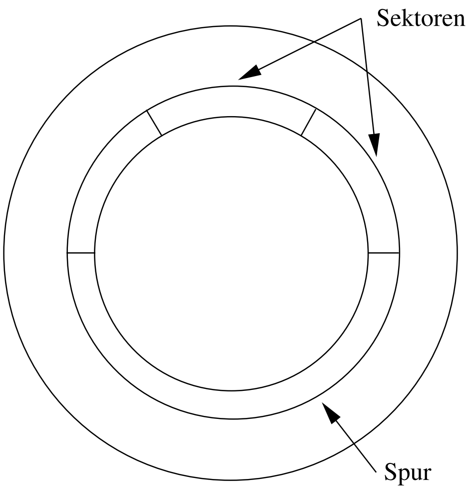
Die Festplattenansteuerung

In den folgenden Kapiteln wird auf Theorie und Praxis eines IDE-Festplattentreibers eingegangen. Dazu werden verschiedene Konzepte der Gerätesteuerung und als konkretes Beispiel die Implementierung für das *metaXaOS* vorgestellt.

2.1 Der Aufbau einer Festplatte

Eine Festplatte besteht aus mehreren kreisförmigen Platten, die mit einer konstanten Geschwindigkeit rotieren und die über Schreib-/Leseköpfe angesteuert werden (meistens zwei Köpfe pro Platte, je einer für eine Seite). Die Bewegung der Köpfe wird von einem *Controller* gesteuert, einem Gerät, das über einen standardisierten Satz von Befehlen den Zugriff auf die Festplatte ermöglicht.

Abbildung 2.1 zeigt schematisch den Aufbau einer Festplatte.



Jede Platte ist in Kreise mit unterschiedlichen Radien eingeteilt, die als Spuren oder Tracks bezeichnet werden. Da in einem Laufwerk mehrere Platten vorhanden sind, werden die Spuren mit gleichem Radius (deren Zahl entspricht der Anzahl Köpfe) zusammengefasst; eine solche Einheit nennt man Zylinder. Die Kombination aus Kopf- und Zylindernummer beschreibt also eindeutig eine bestimmte Spur. Die Spuren werden noch feiner in *Sektoren* (in Form von Kreissektoren) unterteilt (s. nebenstehende Grafik). Die kleinste adressierbare Einheit einer IDE-Festplatte ist somit ein Sektor mit einer Kapazität von normalerweise 512 Byte.

Die bisher beschriebenen Eigenschaften einer Festplatte (Anzahl Köpfe, Zylinder und Sektoren) bilden die *(Laufwerks)Geometrie*. Heute erhältliche Festplatten weisen u.U. keine konstante Anzahl von Sektoren pro Spur auf, sondern besitzen am äußeren Rand der Platten, wo mehr Platz vorhanden ist, mehr Sektoren als in der Nähe des Zentrums.

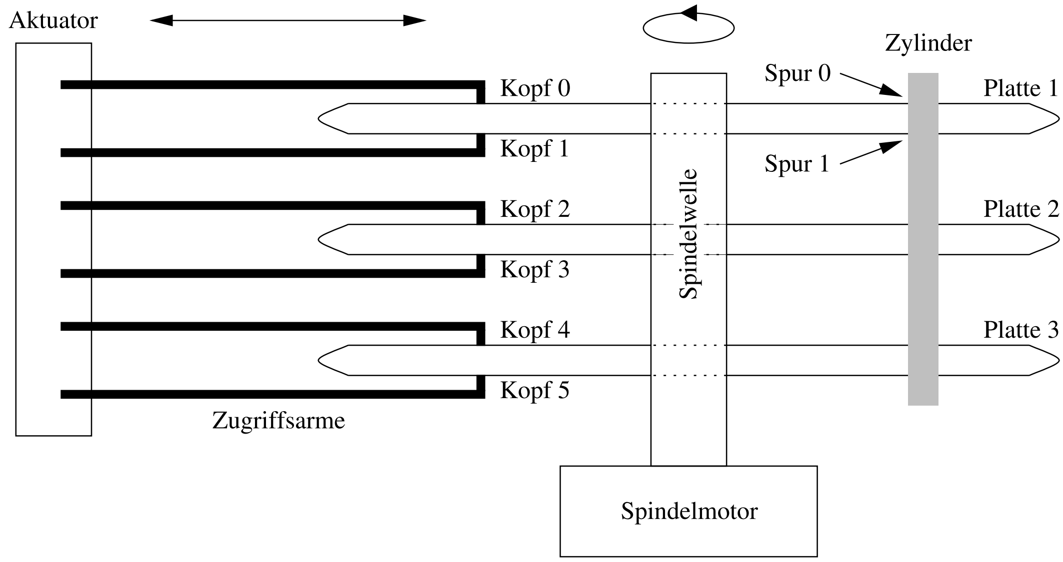


Abbildung 2.1: Der Aufbau einer Festplatte

2.2 Standards

Anfang der 80er Jahre wurden von Seagate die Festplatten-Modelle ST506 und ST412 entwickelt, deren Schnittstellenkonzeption damals von IBM für den PC übernommen wurde und als ST412/506-Schnittstelle bekannt wurde. Maxtor veröffentlichte 1983 ESDI als eine Weiterentwicklung des ST-412/506-Standards. Bei der nächsten Generation von Laufwerken wurden Controller und Festplatte integriert, was zur Schaffung der AT-Bus-Schnittstelle führte (Compaq, Western Digital, Imprimis und Seagate) und schließlich 1989 in den ATA-Standard (*AT-Attachment*) mündete. Dieser Standard hat unter der Bezeichnung IDE (*Integrated Drive Electronics*) seine heutige Popularität erreicht, IDE und ATA sind synonym zu gebrauchen. Im Gegensatz zu ST412/506 oder ESDI, die die physikalische Schnittstelle zwischen Festplatte und Controller festlegen, stellen diese beiden Komponenten bei IDE eine untrennbare Einheit dar; die Ansteuerung der Festplatte ist dem jeweiligen Hersteller überlassen. IDE bildet eine logische Schnittstelle zwischen System und Controller (Festplatte), für die ein Satz von Befehlen vorgesehen ist (z.B. Sektoren lesen, Laufwerksparameter einstellen) ([Mes98]).

Der ATA-Standard wurde in den folgenden Jahren (und wird noch!) weiterentwickelt. 1995 entstand ATA2, das schnellere Übertragungsmodi und eine neue Methode zur Adressierung von Sektoren (LBA) einführte. LBA vereinfacht die Adressierung, erhöht aber nicht die ansprechbare Kapazität (s. Kapitel 2.5). Um auch die Ansteuerung von anderen Geräten als Festplatten in den Standard zu integrieren, wurde das *ATA Packet Interface* (ATAPI) entwickelt, das die Spezifikationen für CD-ROMs, Band- und Diskettenlaufwerke für den Anschluss an eine gewöhnliche ATA-/IDE-Schnittstelle festlegt. Western Digital entwarf daraufhin EIDE (*Enhanced IDE*), das im Wesentlichen auf ATA-2 und ATAPI aufsetzt. Als Antwort auf EIDE entwickelten Seagate und Quantum *Fast-ATA*, das nur ATA-2 als Grundlage besitzt. 1997 wurde ATA-3 verabschiedet, das zwar keine schnelleren Übertragungsmodi, dafür aber eine Vielzahl von Verbesserungen vorsieht: höhere Zuverlässigkeit, besseres Powermanagement, frühzeitiges Erkennen von Fehlern durch das Laufwerk usw. Als eine Erweiterung des ATA-3 Standards mit einem schnelleren DMA-Modus (33 MB/s) hat sich Ultra-ATA durchgesetzt. Die neuste Entwicklung, ATA-4, verschmelzt den ATA- und den ATAPI-Standard und definiert höhere Übertragungsmodi (unter anderem den DMA/33-Modus von Ultra-DMA) ([dH00]).

Falls die Vielzahl der Standards und Weiterentwicklungen verschiedener Firmen verwirrend erscheint, sei darauf hingewiesen, dass sich seit der Einführung des AT-Task-Files, das den Einsatz eines ST412/506-Controllers im AT ermöglichte, an der Registerbelegung und Programmierung nichts geändert hat. Zu den ursprünglichen Befehlen sind seit Anfang der 90er Jahre zwar etliche hinzugekommen, durch die Verwendung des AT-Task-Files sind aber alle ATA- und IDE-Standards abwärtskompatibel.

2.3 Konzepte der Gerätesteuerung

Um eine Kommunikation zwischen dem Prozessor und einem Gerätecontroller zu ermöglichen, werden vom Controller Register bereitgestellt. Der Prozessor kann diese Register mit speziellen I/O-Befehlen (z.B. `inb` und `outb` bei der x86-Architektur) auslesen und setzen.

Um einen Befehl zu initiieren, schreibt der Prozessor in Register des Controllers. Der Controller benutzt wiederum Register, um über den Status der Operation und eventuell aufgetretene Fehler Auskunft zu erteilen. In diesem Fall liest der Prozessor die Register aus.

Eine weitere Kommunikationsform stellt *memory-mapped I/O* dar, bei der die Geräteregister in den Adressraum des Prozessors abgebildet sind und mit den gewöhnlichen Instruktionen zum Datentransfer (z.B. `move`) angesprochen werden können. Grafikkarten verwenden *memory-mapped I/O*, da sie schnell große Mengen von Daten kopieren müssen.

Die Koordination der beiden Parteien kann auf zwei Arten geschehen ([Mes98]):

- Der Controller setzt bestimmte Bits in seinen Registern (z.B. das „Busy"- oder das „Data Request"-Bit), die vom Prozessor laufend abgefragt werden (*Polling*), um auf eine Veränderung reagieren zu können.
- Der Controller teilt dem Prozessor das Ende einer (Teil)Operation mit, indem er über eine spezielle Hardware-Leitung einen *Interrupt* auslöst. Daraufhin wird eine spezielle Routine angesprungen, der *Interrupt-Handler*.

In den folgenden Kapiteln soll mit Bezug auf [SG94] genauer auf die beiden Mechanismen eingegangen werden. Dabei werden Programmfragmente des Treibers als Beispiele aus der Praxis herangezogen. Durch diese Vorwegnahme von Implementierungsdetails soll ein tieferes Verständnis der Konzepte erreicht werden. Anschließend wird eine Technik vorgestellt, die den Datentransfer zwischen Hauptspeicher und Gerätecontroller ohne Einflussnahme des Prozessors durchführen kann.

2.3.1 Polling

Bei der Kommunikation über Register ist ein minimales Protokoll notwendig, um die Operationen von Prozessor und Controller zu koordinieren; dieser Mechanismus wird auch als *Handshaking* bezeichnet. Der Controller setzt ein bestimmtes Bit in seinem Registersatz, das anzeigt, dass er gerade arbeitet und keine Befehle entgegen nehmen kann („Busy"-Bit). Der Prozessor arbeitet dabei im *Polling*- oder *busy-waiting*-Betrieb, d.h. er liest das entsprechende Register in einer Schleife immer wieder aus, um eine Änderung seines Inhalts erkennen zu können. Sobald das Busy-Bit vom Controller gelöscht wird, kann der Prozessor durch Schreiben in ein Befehlsregister des Controllers einen neuen Befehl ausführen lassen. Der Controller setzt daraufhin das Busy-Bit wieder, um anzuzeigen, dass er den Befehl abarbeitet.

Falls der Controller schnell ist, stellt Polling eine vernünftige Alternative zu anderen Koordinationsmethoden dar, da kaum Overhead auftritt. Bei einem langsamen Controller wird allerdings der Prozessor für die gesamte Dauer der Operation zum Lesen der Register beansprucht, was sich negativ auf die Gesamtperformance des Systems auswirkt. Der Anwender muss auf das Ende der Operation warten, auch wenn das gar nicht nötig ist.

Am Beispiel der Routine `waitFor` in der Klasse `Controller` soll die Verwendung von Polling verdeutlicht werden. Dort wird das Statusregister des IDE-Controllers in einer Schleife abgefragt und auf einen bestimmten Wert überprüft. Der Vergleich `Clocks.getTicks() < timeout` begrenzt die Wartezeit für den Fall, dass ein Fehler aufgetreten ist. Die Routine kehrt zurück, sobald der Status den Wert `good` annimmt, der beim Aufruf festgelegt werden muss.

```
boolean waitFor(Drive drive, int good, int bad, long timeout) {

    ....

    timeout += Clocks.getTicks();
    while (Clocks.getTicks() < timeout) {
        if ((getStatus() & (good|bad)) == good)
            return false;
    }
    return true;
}
```

2.3.2 Interrupts

Ein charakteristisches Merkmal einer Festplattenoperation ist der im Vergleich zu einem Speicherzugriff um den Faktor 1000 langsamere Zugriff auf die Festplatte. Hier würde die Abfrage durch Polling erhebliche Performance-Einbußen mit sich bringen. Um dem Controller eine Möglichkeit zu geben, den Prozessor aktiv über eine Zustandsänderung benachrichtigen zu können, existieren Leitungen zwischen den Gerätecontrollern und dem Prozessor, die *Interrupt Request Lines*, die vom Prozessor nach jeder Instruktion abgefragt werden. Wenn an einer dieser Leitungen ein Signal anliegt, wird der normale Programmfluss unterbrochen und eine spezielle Routine, der *Interrupt-Handler*, angesprungen. Nach Abarbeitung des Interrupt-Handlers wird das unterbrochene Programm (bzw. Prozess) wieder fortgesetzt. Die Zahl der Interruptleitungen ist begrenzt; den ersten zwei IDE-Controllern wurden die *IRQs* 14 und 15 zugewiesen. Weitere Geräte, die Daten mit dem Hauptspeicher austauschen und auf *Interrupt-getriebene Kommunikation* zurückgreifen, sind der Disketten-Controller, die parallele Schnittstelle, die Tastatur und Netzwerkkarten.

Der Overhead, der durch das Aufsetzen des Interrupt-Handlers verursacht wird, ist gerechtfertigt, da damit asynchrone Operationen möglich sind; der Prozessor muss auf das Ende einer Operation nicht (aktiv) warten.

Das folgende Programmfragment zeigt den Interrupt-Handler eines IDE-Controllers, der schließlich abhängig von der durchgeführten Operation (Lesen, Schreiben) eine spezielle Handler-Routine aufruft (`operation.handler()`, s. Kapitel 2.7.5).

```
class IDEIntrHandler implements IrqHandler {

    ...

    public void interruptHandler() {
        Operation operation = controller.operation;
        if (operation != null) {
            controller.stopTimeout();
            operation.handler();
        } else {
            Debug.out.println("unerwarteter Interrupt");
        }
    }
}
```

2.3.3 Direct Memory Access

Gerätecontroller besitzen Register, die für den Datentransfer verwendet werden. Daten werden vom Hauptspeicher zum Controller übertragen, indem der Prozessor sie Byte für Byte in das Datenregister schreibt; diese Art der Übertragung wird als *programmed I/O* (PIO) bezeichnet ([SG94]).

Beim direkten Speicherzugriff (*Direct Memory Access* oder *DMA*) wird ein Datenkanal zwischen Peripheriegeräten und dem Hauptspeicher verwendet, über den ein spezieller Controller ohne Mitwirkung des Prozessors auf den Hauptspeicher zugreifen und Daten zu Peripheriegeräten übertragen kann; der Prozessor wird dadurch entlastet. Diese Aufgabe wird in der PC-Architektur vom DMA-Controller 8237A übernommen.

Bei der Einführung des PCI-Local-Bus wurde das Konzept des direkten Speicherzugriffs erweitert durch das *Busmaster*-Konzept. Neu dabei ist, dass sich der Busmaster (der DMA-Controller, der die Bussteuersignale erzeugt), nicht mehr auf dem Motherboard befindet, wo er mit einer I/O-Einheit an einem Adapter kommuniziert. Vielmehr stellt jeder Adapter einen eigenen Busmasterchip zur Verfügung, der alle Bussteuersignale selbst erzeugen und somit in beliebiger Weise den I/O- und Speicheradressraum ansprechen kann. Diese Architektur impliziert aber eine kompliziertere Busarbitrierung über den Erweiterungsbus hinweg; bei DMA musste die Arbitrierung nur zwischen dem

Prozessor und dem Motherboard-DMA-Controller durchgeführt werden. Der Vorteil des Busmaster-Konzeptes ist jedoch, dass es wesentlich flexibler ist; der externe Busmaster auf dem Adapter kann als DMA-Controller realisiert sein, der Daten schnell zwischen dem Hauptspeicher und dem Adapter austauscht ([Mes98]).

2.4 Die IDE-Schnittstelle

Bevor auf die Bedienung der Schnittstelle eingegangen wird, sei noch mal betont, dass die Bezeichnungen IDE, AT-Bus und ATA (AT-Attachment) weitgehend synonym gebraucht werden können. Auf die Unterschiede wird nicht eingegangen, sie sind für die folgenden Betrachtungen nicht von Bedeutung.

2.4.1 Ansteuerung

Der Zugriff auf den Controller einer IDE-Festplatte findet über mehrere Daten- und Steuerregister statt, die zusammen als *AT-Task-File* bezeichnet werden. Das AT-Task-File setzt sich aus zwei Registergruppen mit den Port-Basisadressen 0x1f0 und 0x3f0 für den ersten, bzw. 0x170 und 0x370 für den zweiten Controller zusammen. Die Register befinden sich im I/O-Adressraum, auf den über Ports mit den Maschinenbefehlen `in`, `out` usw. byte-, wort- oder doppelwortweise zugegriffen werden kann.

Register	**Adresse**	**R/W**	**Bemerkung**
Datenregister	0x1f0	R/W	Programmierte Ein-/Ausgabe von Daten
Fehlerregister	0x1f1	R	Fehlerinformationen zum zuletzt aktiven Befehl
Vorkompensation	0x1f1	W	(wird ignoriert, Kompatibilität zum Original-AT)
Sektorenzahl	0x1f2	R/W	Zahl der zu lesenden / zu schreibenden Sektoren
Sektornummer	0x1f3	R/W	Startsektor für einen Lese/Schreib-Befehl
Zylinder LSB	0x1f4	R/W	niederwertige 8 Bit der 10 Bit-Zylindernummer
Zylinder MSB	0x1f5	R/W	höherwertige 2 Bit der 10 Bit-Zylindernummer
Laufwerk/Kopf	0x1f6	R/W	Adressierung von Laufwerk und Startkopf
Statusregister	0x1f7	R	Statusinformationen zum zuletzt aktiven Befehl
Befehlsregister	0x1f7	W	Übergabe von Befehlscodes
2. Statusregister	0x3f6	R	(äquivalent zum Statusregister 0x1f7)
Dig. Ausgabereg.	0x3f6	W	Reset, Interrupt (de)aktivieren
Laufwerkadresse	0x3f7	R	gegenwärtig aktives Laufwerk und aktiver Kopf

Abbildung 2.2: Die Register im AT-Task-File

Abbildung 2.2 zeigt eine Übersicht über das AT-Task-File des ersten IDE-Controllers. In [SG94] werden vier Registertypen unterschieden: Status-, Steuer-, Dateneingabe- und Datenausgaberegister. Die mit „R“ gekennzeichneten Register können nur gelesen werden, stellen also Statusregister dar; „W“ bedeutet Steuerregister. Das Datenregister 0x1f0 dient sowohl zur Dateneingabe als auch -ausgabe. Register, die gelesen und geschrieben werden können („R/W“) können als Steuerregister (z.B. Festlegen der Sektorenzahl) und als Statusregister (Abfrage der Zahl der übertragenen Sektoren) angesprochen werden. Das Datenregister ist 16 Bit breit; alle anderen Register 8 Bit.

Das Format des Laufwerk/Kopf-Registers wird in Abbildung 2.3 und das Statusregister in Abbildung 2.4 dargestellt (nur die wichtigen Bits werden erklärt).

7	6	5	4	3	2	1	0
1	LBA	1	DRV	HD_3	HD_2	HD_1	HD_0

DRV:	Laufwerk (1 = Slave, 0 = Master)
HD_3 - HD_0:	Kopfnummer (Binärzahl, 0 - 15)
LBA:	LBA-Modus (1 = aktiv)

Abbildung 2.3: Die Belegung des Laufwerk/Kopf-Registers (0x1f6)

7	6	5	4	3	2	1	0
BSY	RDY	WFT	SKC	DRQ	COR	IDX	ERR

BSY:	Laufwerk belegt
RDY:	Laufwerk bereit
WFT:	Schreibfehler
DRQ:	Daten können übertragen werden bzw. werden erwartet
ERR:	Fehler (Fehlerregister enthält weitere Informationen)

Abbildung 2.4: Die Belegung des Statusregisters (0x1f7)

2.4.2 Programmierung und Befehlsphasen

Die Programmierung und Ausführung der IDE-Befehle läuft in drei Phasen ab:

Befehlsphase Der Prozessor füllt die Parameterregister mit den gewünschten Werten und übergibt den Befehlscode, woraufhin die Ausführung startet.

Datenphase Das Laufwerk positioniert gegebenenfalls die Schreib-/Leseköpfe und überträgt Daten zwischen Hauptspeicher und Festplatte.

Ergebnisphase Der Controller stellt in den Statusregistern Informationen zum ausgeführten Befehl bereit und löst eventuell über seine IRQ-Leitung einen Hardware-Interrupt aus.

Die Befehls- und Datenregister des Controllers werden vom Prozessor geschrieben und gelesen. Die Sektor- und Formatdaten werden in den entsprechenden Registern übergeben. Der Controller löst zu verschiedenen Zeiten über seine IRQ-Leitung einen Hardware-Interrupt aus, um sich mit dem Prozessor für den Datenaustausch zu synchronisieren.

Sektor lesen Für jeden Sektor, der von der Festplatte gelesen wurde und im Sektorpuffer bereitliegt, um vom Prozessor ausgelesen zu werden, aktiviert der Controller seine IRQ-Leitung. Die Zahl der Hardware-Interrupts stimmt mit der Zahl der gelesenen Sektoren überein.

Sektor schreiben Der Controller erzeugt immer dann einen Interrupt, wenn er bereit ist, Sektordaten vom Prozessor zu übernehmen. Eine Ausnahme stellt der erste Sektor dar, dessen Inhalt unmittelbar nach Aktivierung des Befehls übertragen wird. Den Beginn der Ergebnisphase signalisiert der Controller über seine IRQ-Leitung, so dass die Zahl der Hardware-Interrupts mit der Zahl der geschriebenen Sektoren übereinstimmt.

alle anderen Befehle Der Controller löst zu Beginn der Ergebnisphase einen Hardware-Interrupt aus.

Die IRQ-Leitung wird deaktiviert, sobald der Prozessor das Statusregister (0x1f7) liest. Werden die Statusinformationen über das alternative Statusregister (0x3f6) abgefragt, bleibt die Leitung aktiv.

Die Programmierung des IDE-Controllers soll an einem Beispiel verdeutlicht werden: es sind 7 Sektoren ab Zylinder 654, Kopf 3, Sektor 21 des Laufwerks 0 zu schreiben. Zuerst die Registerbelegung für den allgemeinen Fall:

AT-Task-File-		Bit							
Register		7	6	5	4	3	2	1	0
Sektorenzahl	(0x1f2)	Zahl der zu schreibenden Sektoren							
Sektornummer	(0x1f3)	S_7	S_6	S_5	S_4	S_3	S_2	S_1	S_0
Zylinder LSB	(0x1f4)	Z_7	Z_6	Z_5	Z_4	Z_3	Z_2	Z_1	Z_0
Zylinder MSB	(0x1f5)	0	0	0	0	0	0	Z_9	Z_8
Laufwerk/Kopf	(0x1f6)	1	0	1	DRV	HD_3	HD_2	HD_1	HD_0
Befehl	(0x1f7)	0	0	1	1	0	0	L	R

Die Belegung in unserem Beispiel:

AT-Task-File-		Bit							
Register		7	6	5	4	3	2	1	0
Sektorenzahl	(0x1f2)	0	0	0	0	0	1	1	1
Sektornummer	(0x1f3)	0	0	0	1	0	1	0	1
Zylinder LSB	(0x1f4)	1	0	0	0	1	1	1	0
Zylinder MSB	(0x1f5)	0	0	0	0	0	0	1	0
Laufwerk/Kopf	(0x1f6)	1	0	1	0	0	0	1	1
Befehl	(0x1f7)	0	0	1	1	0	0	0	1

Unmittelbar nachdem das Befehlsbyte geschrieben worden ist, in diesem Fall „00110001“, beginnt der Controller mit der Befehlsausführung — der Datenphase. Im Statusregister wird das BSY-Bit gesetzt, um anzuzeigen, dass der Befehl decodiert wird und der Sektorpuffer zur Aufnahme der Daten (512 Byte) vorbereitet wird. Sobald der Controller fertig ist, wird das BSY-Bit wieder gelöscht und das DRQ-Bit (Data Request) im Statusregister gesetzt. Damit wird dem Prozessor mittgeteilt, dass er die Sektordaten jetzt übergeben kann. Ist das geschehen, wird das BSY-Bit erneut gesetzt, das DRQ-Bit gelöscht und der Controller beginnt damit, die Daten auf die Festplatte zu schreiben.

Sobald der erste Sektor geschrieben worden ist, wird vom Controller über seine IRQ-Leitung ein Interrupt ausgelöst. Der Prozessor übergibt nun die nächsten 512 Byte in gleicher Weise über das Datenregister an den Controller. Dieser Vorgang wird insgesamt siebenmal wiederholt, bis alle Sektoren geschrieben worden sind.

Der letzte Interrupt leitet die Ergebnisphase ein. Das Fehler-Register (0x1f1) und das Status-Register (0x1f7) enthalten dann Informationen über den Erfolg bzw. über aufgetretene Fehler bei der Abarbeitung des Befehls. Im Register „Sektorenzahl“ steht die Zahl der noch zu schreibenden Sektoren, bei einer fehlerfreien Übertragung also 0.

2.5 Die Adressierung der Sektoren einer IDE-Festplatte

In der Einführung dieses Kapitels wurde die *(Laufwerks)Geometrie* als die Anzahl der Köpfe, Zylinder und Sektoren definiert. Jedem Sektor der Festplatte ist eindeutig ein Tripel bestehend aus Nummer des Kopfes, des Zylinders und des Sektors innerhalb der durch Kopf und Zylinder bezeichneten Spur zugeordnet: der *CHS*-Wert (Cylinder/Head/Sector).

Die Ansteuerung der ersten Festplatten/Controller-Systeme (ST412/506, später ESDI) erfolgte über das BIOS durch Angabe von Kopf-, Zylinder- und Sektornummer (des angegebenen Zylinders). Damit wurde eine eindeutige Adressierung der Sektoren ermöglicht, die direkt auf die zugrundeliegende Hardware abgebildet werden konnte. Bei der heute gebräuchlichen (E)IDE- oder AT-Bus-Schnittstelle orientiert sich der Registersatz und damit auch die Adressierung immer noch am alten Standard. Die physikalische Geometrie ist allerdings nur dem integrierten Controller auf der Festplatte bekannt, nach außen (für die Kommunikation mit dem IDE-Host-Adapter) wird die so genannte logische Geometrie verwendet. Bei den ersten Festplatten stimmte diese Geometrie mit der tatsächlichen überein, bei modernen Systemen ist das normalerweise nicht mehr der Fall (so ist eine unterschiedliche Zahl von Sektoren pro Spur möglich).

Es existieren zwei unterschiedliche Beschränkungen für die logische Geometrie einer Festplatte (die tatsächliche Geometrie ist davon nicht betroffen, kann also beliebig gestaltet werden):

1. Das Software-Interface des BIOS und die CHS-Felder in den Partitionstabellen verwenden 10 Bit für das Zylinderfeld (max. 1024), 8 Bit für das Kopffeld (max. 256) und 6 Bit für das Sektorfeld (max. 63 Sektoren, da ab 1 gezählt wird). Die maximale Plattenkapazität, die über dieses Interface angesprochen werden kann, ist also 8 GByte (1024*256*63 512 Byte-Sektoren).

2. Der ATA/IDE-Standard schreibt folgende Längen vor ([TCT97a]): 16 Bit für das Zylinderfeld (max. 65536), 4 Bit für das Kopffeld (max. 16) und 8 Bit für das Sektorfeld (max. 256). Diese

Werte spiegeln sich im AT-Task-File (s. Kapitel 2.4.1) wieder. Die maximale Plattenkapazität beträgt 128 GByte (65536*16*256 512-Byte-Sektoren).

Die Kombination dieser Beschränkungen ergibt max. 1024 Zylinder, 16 Köpfe und 63 Sektoren, woraus die berühmte 504 MByte-Grenze folgt. Ein BIOS, das die logische Geometrie ohne Modifikationen zur Adressierung verwendet, kann also nur die ersten 504 MByte einer IDE-Festplatte ansprechen.

Um diese Grenze zu umgehen, wurden zwei Verfahren eingeführt:

1. Beim erweiterten CHS findet eine zweite Umsetzung der Geometrieinformationen innerhalb des BIOS statt. Der Vorteil besteht darin, dass bei der ersten Umsetzung nur die Beschränkungen des IDE-Standards bestehen (max. 16 Köpfe); die Zylinderzahl muss nicht auf 1024 begrenzt sein. Bei der zweiten Umsetzung kann auf die korrekte Adressierung des IDE-Standards verzichtet werden, da der Festplattencontroller ja nicht direkt angesprochen wird; hier gelten nur die BIOS-Regeln (1024 Zylinder). Da die beiden Adressierungsarten mit ihren Beschränkungen nicht gleichzeitig auftreten, kann die max. BIOS-Kapazität von 8 GByte angesprochen werden.

2. Bei der logischen Blockadressierung (LBA) werden die Sektoren der Festplatte über eine einzelne Blocknummer adressiert; die Sektoren werden linear, beginnend mit der Nummer 0 für Zylinder 0, Kopf 0, Sektor 1, hochgezählt. Die Verwendung der CHS-Nummern ist also nur noch beim Zugriff über die BIOS-Routinen notwendig, die diese dann in LBA-Nummern konvertieren. Wird der Festplattencontroller über das AT-Task-File programmiert, können die LBA-Nummern zur Adressierung ohne Umsetzung in CHS-Nummern verwendet werden; in diesem Fall besteht die BIOS-Beschränkung auf 8 GByte nicht mehr.

Die Ansteuerung des Festplattencontrollers erfolgt bei IDE über das AT-Task-File, das (unter anderem) ein 8 Bit-Register für die Sektornummer, zwei 8 Bit-Register für die Zylindernummer und 4 Bit für die Kopfnummer bereitstellt. Soll ein Sektor über CHS angesprochen werden, können die drei Werte unverändert (bei einer Kopfnummer < 16) in die entsprechenden Register geschrieben werden. Eine Kopfnummer > 16 erfordert eine Transformation der Werte (u.U. mehrmaliges Verdoppeln der Sektorzahl mit gleichzeitiger Halbierung der Kopfzahl). Bei einer Ansteuerung mittels LBA wird die Blocknummer auf die Register verteilt: Bits 0 bis 7 auf das Register für die Sektornummer; Bits 8 bis 23 auf die beiden Zylinderregister; Bits 24 bis 27 auf das Laufwerk/Kopfregister. Die LBA-Nummer muss also nicht in einen CHS-Wert umgerechnet werden, das erledigt der Controller. Es stehen also 28 Bit für die LBA-Nummer zur Verfügung, was der schon erwähnten max. Kapazität von 128 GByte entspricht (mittels LBA lässt sich diese Beschränkung also nicht umgehen, dazu müsste die Struktur des AT-Task-Files verändert oder erweitert werden).

Festplatten, die mehr als 16514064 Sektoren (8 GByte) enthalten, können nur über LBA angesprochen werden. In diesem Fall gibt die Festplatte an, 16383 Zylinder, 63 Sektoren, 16 Köpfe und eine LBA-Sektorzahl von mindestens 16514064 (die tatsächliche Größe) zu besitzen (s. Kapitel 2.7.3, Identify-Kommando). Der Festplattencontroller erkennt an dieser Kombination, dass er es mit einer Platte von mehr als 8 GByte zu tun hat und den LBA-Modus verwenden muss. Bei Verwendung des BIOS-Software-Interface ist es nicht möglich, mehr als 8 GByte anzusprechen, als einzige Alternative bleibt die Programmierung des AT-Task-Files. Es existiert allerdings eine Erweiterung des BIOS-Standards von IBM und Microsoft, die LBA verwendet und die 8 GByte-Grenze überschreiten kann.

2.6 Der Festplattentreiber

Der IDE-Treiber hat zum einen die Aufgabe, nach vorhandenen Controllern und Festplatten zu suchen, diese zu identifizieren und für den Betrieb zu konfigurieren. Zum anderen hat er einen Mechanismus vorzusehen, mit dem Daten von und zu einer Festplatte transferiert werden können, wobei gegenseitiger Ausschluss beim Zugriff auf die Hardware, ein angemessenes Fehlermanagement und die korrekte und effiziente Ansteuerung der Controller gewährleistet sein muss.

Die Initialisierungsphase (auch als *probing* bezeichnet) ermittelt die vorhandenen IDE-Controller und daran angeschlossene Festplatten. Der IDE-Standard legt die maximale Anzahl Festplatten pro Controller auf 2 fest. Normalerweise befinden sich zwei IDE-Controller in einem handelsüblichen PC. Die Identifikation der vorhandenen Festplatten erfolgt entweder über das BIOS oder über ein spezielles Kommando des Controllers.

2.7 Die Implementierung

2.7.1 Überblick

Im Folgenden wird näher auf die Implementierung des Festplattentreibers in Java eingegangen. Die Schnittstelle nach außen stellt die Klasse `IDEDevice` dar, die Methoden zur Datenübertragung (`bufferheadIO`) und zum Ermitteln der vorhandenen Festplatten und Partitionen (`isPresent`) und ihrer Kapazität (`getCapacity`) bietet. Anhang B enthält eine mit *javadoc* generierte Klassendokumentation.

Die zwei Hauptaufgaben des Treiber bestehen in der Suche nach vorhandenen Controllern und Festplatten und deren Initialisierung und in der Abwicklung der Datenübertragung. Dafür wurde ein Satz von Klassen entwickelt, die zum einen durch Objekte der realen Welt (`Controller`, `Drive`), zum anderen durch Abstraktionen von Abläufen (`Operation` mit Unterklassen, `Timeout`) motiviert wurden. Die Klasse `IDEDevice` führt die Hardwareerkennung und Initialisierung durch und koordiniert den Zugriff auf die Hardware (über die zentrale Methode `bufferheadIO`). Von dieser Klasse aus werden die anderen Objekte des Treibers instantiiert und angesprochen. `IDEDevice` stellt eine Klasse nach dem *Singleton*-Muster dar ([Lar97]). Es existiert genau eine Instanz, die als Monitor für die IDE-Controller dient.

2.7.2 Integration in das Betriebssystem

Die Klassen des Treibers befinden sich im Paket `devices.ide` in `metaxa.os`. Der Treiber greift für den Zugriff auf die Hardware auf Methoden der Klassen `Memory`, `devices.Ports` und `de-`

`vices.pci.PCIBus` zurück. Für die Installation eines Interrupt-Handlers wird die Klasse `IRQ` verwendet. Kapitel 2.8 geht näher auf die verwendeten Klassen mit *native*-Methoden ein. Abbildung 2.5 gibt einen Überblick über die Struktur der Pakete und Klassen.

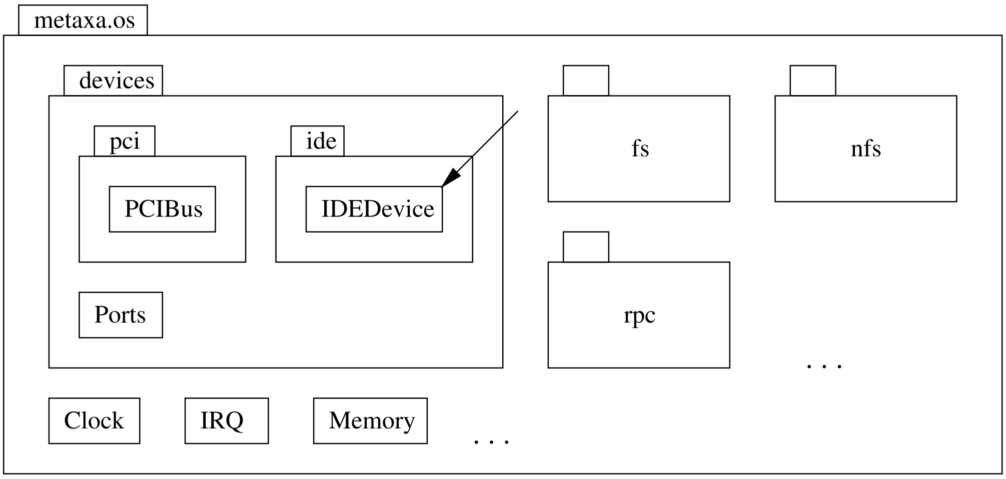


Abbildung 2.5: Integration des Festplattentreibers in *metaXaOS*

2.7.3 Die Initialisierungsphase

Zu Beginn der Initialisierungsphase werden die `Controller`-Objekte erzeugt, die die IDE-Controller im System repräsentieren. Diese wiederum erzeugen die `Drive`-Objekte für die am Controller angeschlossenen Festplatten.

Anschließend wird versucht zu ermitteln, welche IDE-Controller und -Festplatten vorhanden sind und welche physikalischen Eigenschaften die einzelnen Platten aufweisen (Kapazität, DMA-Fähigkeit). Außerdem wird der Chipsatz des Motherboards abgefragt, um festzustellen, ob das System Busmaster-DMA unterstützt und wenn ja, welche Art (Multiword-DMA oder Ultra-DMA). Dieser auch als *probing* bezeichnete Vorgang wird in der Methode `init` in der Treiber-Klasse `IDEDevice` durchgeführt. Dabei werden nacheinander folgende Systemkomponenten untersucht und initialisiert:

`PCI und Busmaster-DMA`

Der PCI-Bus wird nach einer PCI-ISA-IDE-Bridge (*PIIX* für *PCI ISA IDE Xcelerator*) von Intel abgefragt, die (bis vor kurzem) den am häufigsten eingesetzten Chipsatz darstellte. Gesucht wird nach dem Modell 82371FB (PIIX) und seinen Nachfolgern 82371SB (PIIX3) und 82371AB (PIIX4). Dieses PCI-Gerät erfüllt zwei Aufgaben: die einer PCI/ISA-Bridge mit DMA-, Timer- und Interrupt-Logik und die einer IDE-Schnittstelle. Es werden die Funktionen zweier Original-DMA-Controller 8237 des AT emuliert ([Mes98]). Der Zugriff auf den PCI-Bus erfolgt über die Methoden der Klasse `IDEBus` im Paket `devices.ide`. Falls ein Intel-Chipsatz gefunden wurde, werden die unterstützten DMA-Modi ermittelt (Multiword-, ab PIIX4 auch Ultra-DMA). Die DMA-Basisregister werden ausgelesen und die IDE-Controller für Busmaster-DMA konfiguriert (in der Methode `initTriton`). Für jeden Controller wird eine DMA-Tabelle angelegt, die die Adressen der zu übertragenden Daten aufnimmt.

`CMOS/BIOS`

Der Identify-Befehl (0xec), der benutzt wird, um Informationen über ein Laufwerk abzufragen, ist im AT-Standard als optional gekennzeichnet und beim Controller im Original-AT nicht vorhanden ([Mes98]). Um dennoch wenigstens die Kapazität eines alten Laufwerks bestimmen zu können, wird auf das BIOS bzw. den CMOS-Speicher zurückgegriffen. Der *Interrupt Vector Table* des BIOS enthält ab Adresse 0 für alle Interrupts eine Tabelle mit Zeigern (im Segment:Offset-Format) auf Interruptroutinen. Eine kleine Zahl von Interrupts erbringen keinen bestimmten Dienst; ihre Adressen verweisen vielmehr auf Tabellen im ROM mit Informationen über die Hardware. Die Adressen der Interrupts 0x41 und 0x46 zeigen auf die *Fixed Disk Parameter Tables* für die zwei Laufwerke des ersten Controllers. Dort sind die CHS-Werte gespeichert ([Pho91]).

Das CMOS-Ram enthält an der Position 0x12 ein Byte, das angibt, welche Laufwerke des ersten Controllers vorhanden sind. Ausgelesen wird der Wert, indem in das Register 0x70 die Position geschrieben wird, woraufhin über Register 0x71 der Inhalt des CMOS an der gewünschten Adresse gelesen werden kann.

`Controller`

Jeweils ein Register der ersten zwei Controller wird ausgelesen, verändert wieder gesetzt und erneut ausgelesen. Wurde die Änderung übernommen, ist der Controller vorhanden und wird nach Laufwerken abgefragt. Für jeden Controller wird ein `Timeout`-Objekt erzeugt; das ist ein Thread, der eine zu lange dauernde Operation nach einer bestimmten Zeitspanne abbricht (z.B. 10 Sekunden beim Lesen von Sektoren). Damit wird verhindert, dass ein Controller, der nicht mehr reagiert, das gesamte System blockiert.

`Laufwerke`

Mittels des Identify-Befehls wird das Laufwerk dazu veranlasst, im Sektorpuffer zahlreiche Informationen über sich bereitzustellen. Unter anderem kann damit die Modellbezeichnung, die Zahl der Köpfe, Sektoren und Zylinder, DMA- und LBA-Fähigkeiten, DMA-Modus und -Zykluszeiten abgefragt werden. Falls das Laufwerk den LBA-Modus nicht unterstützt, wird die Kapazität aus den CHS-Werten berechnet, ansonsten wird die vom Identify-Befehl gelieferte LBA-Kapazität übernommen. Bei Festplatten, deren Kapazität die 8 GByte-Grenze übersteigt, muss auf den LBA-Wert zurückgegriffen werden. Anschließend wird versucht, den schnellsten unterstützten PIO- (Methode `EnablePIOOperation`) bzw. DMA-Modus (`EnableDMAOperation`) zu aktivieren. ATAPI-Geräte (CD-ROMs, Bandlaufwerke, Diskettenlaufwerke usw.) werden zwar erkannt, aber nicht unterstützt.

`Interrupthandler`

Schließlich wird für jeden vorhandenen Controller ein Interrupt-Handler installiert, der IRQ 14 für den ersten Controller bzw. IRQ 15 für den zweiten Controller überwacht.

2.7.4 Auslesen der Partitionstabellen

Im Folgenden wird das Partitionierungssystem von MS-DOS erklärt, wie es heute für fast alle PCs verwendet wird. Sowohl unter Windows als auch Linux existiert das Programm `fdisk`, mit dem sich Änderungen an der Partitionstabelle vornehmen lassen können.

Der erste Sektor einer Festplatte, der auch als *Master Boot Record* (MBR) bezeichnet wird, enthält immer die Partitionstabelle. Sie besteht aus vier Einträgen mit je 16 Byte, gefolgt von einer Signatur (0x55aa), die den Sektor als MBR kennzeichnet. Jeder Eintrag der Tabelle besteht aus den CHS-Werten des ersten und letzten Sektors, dem Partitionstyp, dem Startsektor im LBA-Format und der Partitionsgröße in Sektoren. Um mehr als nur vier Partitionen zu erlauben, existiert als Typ auch die Kennung „erweiterte Partition", die auf einen Sektor verweist, der wiederum eine Partitionstabelle enthält. Die Partitionen, die im MBR verzeichnet sind, werden als „primäre", Partitionen innerhalb einer erweiterten Partition als „logische" (oder sekundäre) Partitionen bezeichnet. Diese zweite Partitionstabelle besteht aus höchstens zwei Einträgen: eine logische Partition und evtl. eine erweiterte Partition. Eine Festplatte kann also maximal 4 primäre oder 3 primäre und beliebig viele logische Partitionen enthalten, die ähnlich einer verketteten Liste verwaltet werden ([Lan96], [Sla98]).

Das Auslesen der Partitionstabelle übernimmt die (private) Methode `readPartitionTable` in der Klasse `Drive`. Wie bei Linux werden die vier Festplatten des ersten und zweiten Controllers mit `hda`, `hdb`, `hdc` und `hdd` bezeichnet, als Partitionskennung wird eine Zahl angehängt. Dabei bezeichnet 0 die gesamte Festplatte, 1 – 4 die primären und eine evtl. erweiterte Partition, Nummern ab 5 die logischen Partitionen. Lesen und Schreiben von Sektoren ist nur für primäre und logische Partitionen erlaubt.

2.7.5 Die Ansteuerung der Controller

Für die Kommunikation mit den Festplatten-Controllern wurde die `Operation`-Klasse konzipiert. Wie in Abbildung 2.6 zu sehen ist, besitzt `Operation` eine Vielzahl von Unterklassen, die teilweise während der Initialisierung verwendet werden. In der folgenden Übersicht werden die Unterklassen kurz beschrieben.

- **Initialisierung:** die Operationen werden in der angegebenen Reihenfolge ausgeführt. Falls eine PCI-ISA-IDE-Brücke mit Busmaster-Unterstützung gefunden wurde und die `Identify-Operation` dem Laufwerk DMA-Fähigkeit bescheinigt, wird es für DMA programmiert (`EnableDMAOperation`), ansonsten für *programmed I/O* (`EnablePIOOperation`).

  `IdentifyOperation` fordert vom Laufwerk Informationen an wie Kapazität, DMA-Fähigkeit usw.

  `RecalibrateOperation` setzt das Laufwerk zurück und positioniert den Schreib-/Lesekopf neu.

  `EnableDMAOperation` (falls DMA unterstützt wird) stellt für das Laufwerk den schnellsten DMA-Modus ein (*Multiword-DMA* oder *Ultra-DMA* 0, 1 oder 2).

EnablePIOOperation konfiguriert das Laufwerk auf den schnellsten PIO-Modus (3, 4 oder höher).

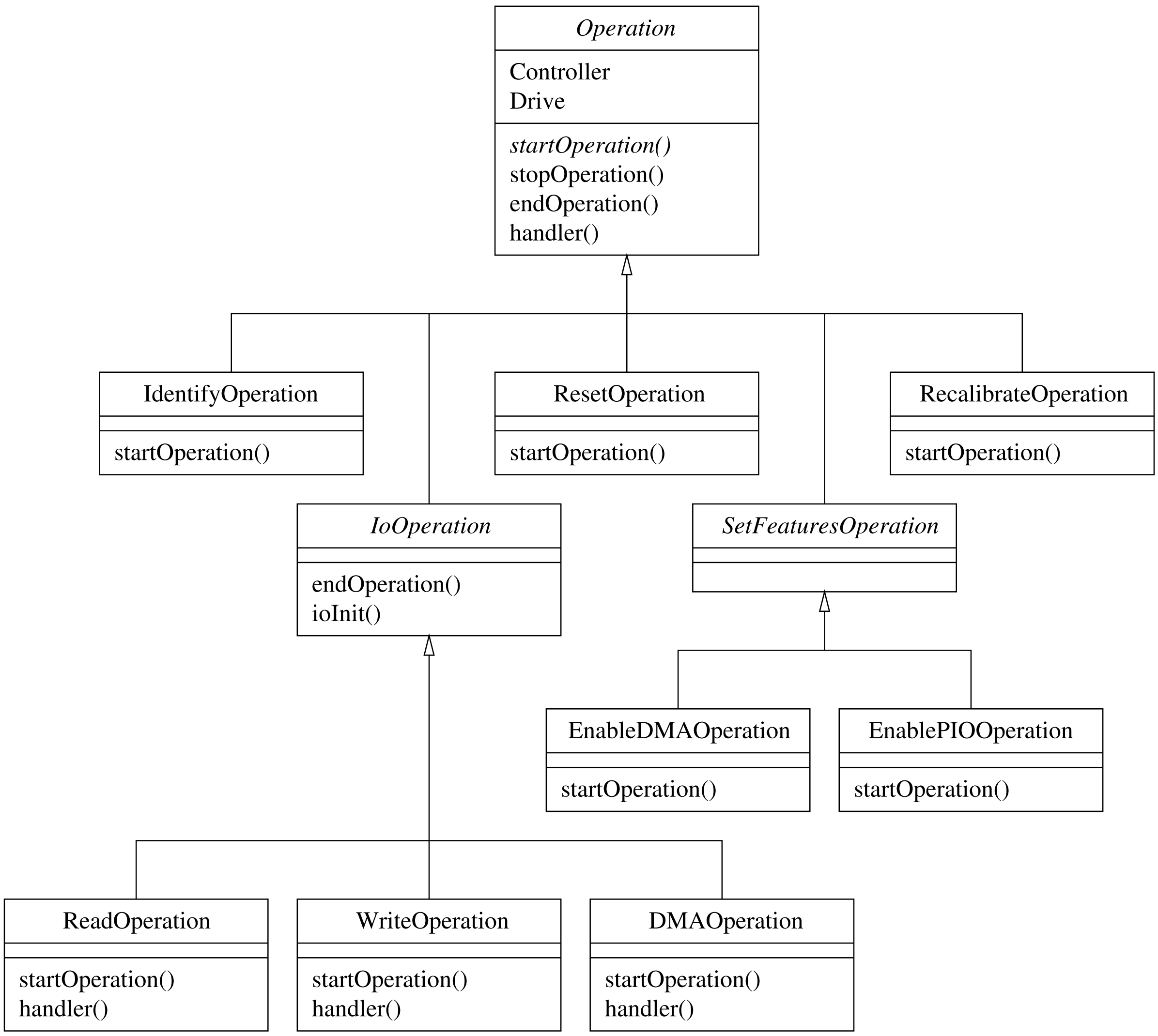


Abbildung 2.6: `Operation` und Unterklassen

- **Datenübertragung:** die `Operation`-Objekte werden mit einem BufferHead initialisiert, das ein Speicherobjekt für die Daten enthält. Je nach Richtung des Datentransfers bzw. der DMA-Fähigkeit des Laufwerks wird die entsprechende Operation durchgeführt. Die gemeinsame Superklasse `IoOperation` überlädt die Methode `endOperation` mit einer Version, die das Ende der asynchronen Übertragung im BufferHead vermerkt (s. nächstes Kapitel).

  ReadOperation liest den Inhalt von Festplattensektoren.

`WriteOperation` schreibt den Inhalt von Festplattensektoren.

`DMAOperation` liest bzw. schreibt Festplattensektoren im Busmaster-DMA-Betrieb.

- **Fehlermanagement:**

  `ResetOperation` setzt das Laufwerk zurück. Diese Maßnahme wird ergriffen, falls der Zugriff auf einen Sektor mehr als viermal missglückt ist, der Controller einen Fehler meldet oder nicht mehr reagiert (Timeout). Die aktuelle Operation wird nicht abgebrochen.

Bevor der Controller aktiviert wird, wird das ihm zugewiesene `Timeout`-Objekt initialisiert, das der Abarbeitung des Befehls ein Zeitlimit setzt. Durch Aufruf der Methode `startOperation` wird der Registersatz des entsprechenden Festplatten-Controllers initialisiert (Festlegen der erforderlichen Parameter, z.B. Anzahl zu übertragender Sektoren) und das eigentliche Kommando übermittelt (durch Schreiben in das Befehlsregister (0x1f7), wodurch der Controller veranlasst wird, den Befehl auszuführen.

Im Treiber wird nicht auf das Ende der Operation gewartet, sondern ein Interrupt-Handler festgelegt (Klasse `IDEIntrHandler`), der angesprungen wird, wenn der Controller die (Teil)Operation beendet hat oder wenn ein Fehler aufgetreten ist. Jedes `Controller`-Objekt verwaltet eine Warteschlange von Aufträgen; das Feld `operation` verweist auf die `Operation`, die gerade ausgeführt wird. Der Interrupthandler des Controllers hat somit Zugriff auf das aktive `Operation`-Objekt und ruft dort die `handler`-Methode auf, die z.B. den nächsten Sektor liest oder das Kommando über die Methode `endOperation` abschließt. In diesem Fall wird in der Warteschlage des `Controllers` zum nächsten `Operation`-Objekt gewechselt (falls vorhanden).

2.7.6 Die Datenübertragung

Bei Aufruf der Methode `bufferheadIO` im Treiber wird aus der Gerätekennung des `BufferHead`-Objekts (einer Struktur für die Aufnahme der Sektordaten) Controller und Laufwerk extrahiert. In der Kennung ebenfalls enthalten ist die Partition, die dazu benutzt wird, die relative Blocknummer des `BufferHeads` (bezüglich des ersten Sektors der Partition) in eine absolute Sektornummer der Festplatte umzurechnen. Es kann nur auf primäre und logische Partitionen zugegriffen werden, nicht auf die gesamte Festplatte oder auf eine erweiterte Partition. Anschließend wird ein neues `Operation`-Objekt für die Datenübertragung erzeugt. Die `IoOperation`-Unterklassen lassen einen Vergleich mit der `request`-Struktur in Linux zu.

Falls die Datenübertragung zwischen Hauptspeicher und IDE-Controller mittels *programmed I/O* (PIO) geschieht, werden die Befehle `inl` und `outl` verwendet, die so oft aufgerufen werden, bis alle Daten transferiert wurden. Die Methoden `inputData` bzw. `outputData` in der Klasse `Ports` erledigen diese Aufgabe. Bei einer DMA-Operation wird eine Tabelle mit den Speicheradressen der zu übertragenden Daten angelegt, diese dem DMA-Controller mitgeteilt, die Übertragungsrichtung (Lesen oder Schreiben) festgelegt und die Operation gestartet.

Wurden alle Sektoren übertragen, sorgt die Methode `endOperation` für den Abschluss des Kommandos und ruft die Methode `endIo` im `BufferHead`-Objekt auf, die dort vermerkt, ob Fehler aufgetreten sind. Bei einer synchronen Operation wird der wartende Thread aufgeweckt.

2.7.7 Fehlermanagement

Bei der Umsetzung von Block- in Sektornummern in der Methode `bufferheadIO` wird überprüft, ob die zu übertragenden Sektoren nicht die Kapazitätsgrenze der Festplatte übersteigen. Beim Schreiben des Registersatzes wird die Sektorzahl auf max. 256 beschränkt.

Während der Durchführung einer Festplattenoperation zeigt der Controller im Statusregister an, ob ein Fehler aufgetreten ist und stellt im Fehlerregister weitere Informationen bereit. Falls ein Controller nicht mehr reagiert, sorgt sein `Timeout`-Objekt dafür, dass die Operation abgebrochen wird. Bei einem Fehler wird die aktuelle Operation maximal 8-mal wiederholt, bevor sie abgebrochen wird; meldet der Controller einen defekten Sektor oder einen nicht korrigierbaren Datenfehler, auch sofort. Nach jedem vierten Versuch wird mittels einer `ResetOperation` ein Laufwerks-Reset durchgeführt; in diesem Fall wird DMA für dieses Laufwerk abgeschaltet.

2.8 Überblick über die verwendeten *native*-Funktionen

Der Festplattentreiber muss in bestimmten Situationen mit der Hardware kommunizieren, um seine Aufgaben erledigen zu können. Von der Software werden neben den IDE-Controllern der PCI-Bus, die Systemuhr, Prozessor und Speicher angesprochen. Der Zugriff auf die Geräte kann nicht direkt in Java implementiert werden; es besteht aber die Möglichkeit, *native*-Funktionen in einer maschinennahen Programmiersprache zu definieren, die in Form von Java-Methoden zugänglich sind. Kapitel 4 geht näher auf diesen Mechanismus ein. Die Klassen in den Paketen `metaxa.os` und `metaxa.os.devices` stellen geeignete Methoden für den Zugriff auf die Hardware bereit.

In der folgenden Aufstellung werden die Klassen und Methoden erläutert, die auf *native*-Funktionen aufsetzen oder als solche implementiert sind und vom Festplattentreiber verwendet werden (s. auch Abbildung 2.5):

Clock repräsentiert die Systemuhr. Die Methode `getTicks` liefert die Zahl der Ticks (10 ms), die seit Systemstart verstrichen sind. Der Treiber verwendet die Klasse, um notwendige kurze Verzögerungen bei der Ansteuerung der Controller zu realisieren.

IRQ bietet die Möglichkeit, einen Handler für Hardware-Interrupts zu installieren (Methode `installHandler`). Die Routine, die bei einem Interrupt angesprungen wird, ist in Assembler implementiert. Für jeden IDE-Controller wird ein Handler-Objekt erzeugt; normalerweise wird Interrupt 14 vom ersten und 15 vom zweiten Controller verwendet. Die Klasse enthält außerdem Funktionen, um Interrupts zu sperren (`cli`) und freizuschalten (`sti`).

Memory ist die Klasse der Speicherobjekte, die für den Datentransfer zwischen Festplattentreiber und Dateisystem verwendet wird. Die wichtigsten Methoden sind `get8/16/32`, `set8/16/32` usw. für den Zugriff auf Speicherstellen, `alloc`, um Speicher anzufordern und `getStartAddress`, die die Adresse des Speicherbereichs liefert. Der Festplattentreiber benötigt diese Funktionalität, um Daten zu kopieren; beim DMA-Transfer ist die Kenntnis der Speicheradressen notwendig.

devices.Ports enthält Methoden für den Zugriff auf den Register-Adressraum. Die Befehle `inb`, `outb` usw. entsprechen den gleichnamigen Assembler-Anweisungen und werden verwendet, um das AT-Task-File, die Register der IDE-Controller, anzusprechen. Um Daten zwischen Hauptspeicher und Controller-Puffer zu kopieren, dienen die Kommandos `inputData` und `outputData`.

devices.pci.PCIBus wird verwendet, um auf Geräte am PCI-Bus zuzugreifen. Im Falle des Festplattentreibers wird der DMAController, der von einer PCI-ISA-IDE-Brücke emuliert wird, für die Datenübertragung verwendet (sofern vorhanden). Mit den Kommandos `readConfig` und `writeConfig` können die Register eines PCI-Gerätes gelesen und geschrieben werden.

2.9 Zusammenfassung

Es wurde eine Übersicht über den Aufbau und die Programmierung einer Festplatten/Controller-Einheit gegeben. Neben einer Darstellung verschiedener Konzepte der Gerätesteuerung wurde auf die Besonderheiten der IDE-Schnittstelle eingegangen. Die Aufgaben eines Festplattentreibers — Hardwareerkennung und -konfiguration sowie Datenübertragung — wurden anhand der Implementierung detailliert ausgeführt. Weitere wichtige Eigenschaften — Kontrolle des Zugriffs, Fehlermanagement und effiziente Bearbeitung von Aufträgen — wurden ebenfalls erläutert.

Kapitel 3

Das Dateisystem

Die folgenden Kapitel geben einen Überblick über die Aufgaben und das Design eines Dateisystems. Ausgehend von den ersten Unix-Dateisystemen wird die Entwicklung bis hin zu *Ext2* von Linux aufgezeigt, das als Vorbild für das *metaXaOS*-Dateisystem *JavaFS* diente. Die Implementierung wird ausführlich erläutert, wobei zahlreiche Grafiken im UML-Format helfen sollen, Zusammenhänge zu verdeutlichen.

3.1 Einführung

3.1.1 Überblick und Definitionen

Um eine der wichtigen Aufgaben eines Betriebssystems — die dauerhafte Sicherung und Verwaltung von Daten ([Vah96]) — zu bewältigen, wurde bereits zur Zeit der ersten Computersysteme das Konzept des Dateisystems entwickelt. Das Dateisystem ist ein wesentlicher Teil des Betriebssystem; damit verbunden wurden wichtige Abstraktionen eingeführt, um dem Benutzer eine komfortable und effiziente Schnittstelle für die Arbeit mit dem Dateisystem zur Verfügung zu stellen. Die Umsetzung dieser Abstraktionen auf konkrete Objekte ist die Aufgabe des Dateisystems ([Kle99a]). Die grundlegenden Abstraktionen sind:

Datei Eine abgeschlossene Menge von Daten, die aus der Sicht des Benutzers eine Einheit bilden.

Verzeichnis Eine spezielle Art von Datei, die es ermöglicht, Dateien zu gruppieren und organisieren (vergleichbar einer Schublade oder eines Ordners). Verzeichnisse *enthalten* Dateien und Verzeichnisse.

Dateisystem Menge der Dateien (und Verzeichnisse), die sich auf einem bestimmten Gerät (Datenträger, Partition) befinden, und der Operationen auf diesen Objekten. Ein Dateisystem übernimmt Aufgaben der Verwaltung und kontrolliert den Zugriff auf die Dateien und Verzeichnisse.

Wird das Betriebssystem aus der Sicht der objektorientierten Programmierung gesehen, dann dient ein Dateisystem zur Realisierung der Objektpersistenz. Dies ist die Eigenschaft eines Objekts, seinen

Zustand und Informationen zur Klassenzugehörigkeit dauerhaft zu sichern. Dieses Konzept beinhaltet einen zeitlichen (das Objekt existiert auch, nachdem das Programm beendet wurde, das es erzeugt hat) und einen räumlichen Aspekt (das Objekt kann den Adressraum verlassen, in dem es erzeugt wurde) ([Boo94]). Objektpersistenz ist Teil des in [Boo94] definierten Objektmodells, das außerdem die Elemente Abstraktion, Kapselung, Modularität, Hierarchiebildung, Typisierung und Nebenläufigkeit enthält.

In den Anfangszeiten der Betriebssysteme war nur ein Dateisystem pro Betriebssystemkern möglich, später wurde das Konzept des virtuellen Dateisystems eingeführt, das das gleichzeitige Vorhandensein verschiedener Dateisysteme erlaubt und dem Benutzer eine einheitliche Sicht über Dateisystemgrenzen hinweg bietet. Im Kapitel 3.1.4 wird dieses Thema ausführlicher behandelt.

Durch die Verwendung der Abstraktionen *Datei*, *Verzeichnis* und *Dateisystem* wird eine klare Schnittstelle zwischen Betriebssystem und Benutzer definiert; verschiedene Dateisysteme implementieren diese Schnittstelle und präsentieren sich dem Benutzer somit alle auf dieselbe Weise. (Natürlich existieren einige Unterschiede, die erst bei genauerer Betrachtung auffallen, z.B. verschiedene Beschränkungen der Länge eines Dateinamens. Auf diese Unterschiede soll hier nicht eingegangen werden.)

Die eingeführten Konzepte werden im Folgenden näher betrachtet.

Eine *Datei* ist eine logische (nicht notwendigerweise physikalische) Gruppierung von Daten, ein „Behältnis“ für Daten. Benutzer haben die Möglichkeit, Dateien anzulegen, in ihnen Daten abzulegen (zu *schreiben*) und wieder auszulesen (sequentiell oder an einer beliebigen Position innerhalb der Datei). Der Inhalt wird vom Dateisystem nicht interpretiert, sondern lediglich als Folge von Bytes angesehen.

Dateien werden in einer hierarchischen, baumähnlichen Struktur organisiert. Durch die Zuordnung von Namen zu Dateien entsteht ein hierarchischer Namensraum. Die Bildung dieser Hierarchie wird durch die Verwendung von *Verzeichnissen* ermöglicht, die weitere Dateien und Verzeichnisse enthalten können. Der „Verzeichnisbaum“ beginnt am *Wurzelverzeichnis* und enthält Dateien und Verzeichnisse mit Dateien als Blätter.

Ein Verzeichnis verwaltet die Namen der Dateien und Verzeichnisse, die es enthält; in einem Unix-Betriebssystem wird das Wurzelverzeichnis über „ / “ angesprochen. Innerhalb eines Verzeichnisses sind Dateien mit gleichem Dateinamen nicht erlaubt; um eine Datei eindeutig zu identifizieren, muss der gesamte *Pfad* angegeben werden. Ein (absoluter) Pfadname enthält alle Komponenten des Pfades (Verzeichnisse) beginnend mit dem Wurzelverzeichnis; das Ende wird vom Dateinamen gebildet. Die Komponenten werden durch „ / “ getrennt. „ / “ dient also als Kennung für das Wurzelverzeichnis und als Trennzeichen für Teile des Pfades. In Unix besitzt jeder Prozess ein *aktuelles Verzeichnis*; Dateinamen ohne Pfadangabe werden relativ zu diesem Verzeichnis gesetzt. Für das Konzept des aktuellen Verzeichnisses wichtig sind zwei spezielle Verweise, die jedes Verzeichnis besitzt: „ . “ auf sich selbst und „ .. “ auf das übergeordnete Verzeichnis. Beim Wurzelverzeichnis zeigt „ .. “ ebenfalls auf sich selbst.

Eine Datei ist innerhalb des Betriebssystems ein Objekt, dem noch kein Name zugeordnet ist (vielmehr eine eindeutige Nummer). Die Zuordnung Name zu Datei (wie sie dem Benutzer präsentiert wird) entsteht erst durch die Verzeichniseinträge. Der Inhalt eines Verzeichnisses wird als Paare von Namen und Verweisen auf Dateien bzw. Verzeichnisse (durch Angabe ihrer Nummer) gebildet. Soll eine Datei z.B. umbenannt oder in der Verzeichnisstruktur verschoben werden, muss lediglich der Verzeichniseintrag geändert bzw. verschoben werden, die Datei wird nicht berührt und behält ihre Position (auf dem Datenträger).

3.1.2 Die Attribute einer Datei

Neben den Daten (dem Inhalt einer Datei) und ihrem Namen werden verschiedene weitere Attribute mit einer Datei gespeichert (im Folgenden auch als Verwaltungs- oder Metadaten bezeichnet). Diese Daten werden nicht in den Verzeichniseinträgen, sondern an anderer Stelle in der *Inode*-Struktur (*Index node*) gesichert. Der Inhalt dieser Struktur unterscheidet sich von Dateisystem zu Dateisystem, besteht aber normalerweise aus folgenden Daten:

- Dateityp (Datei, Verzeichnis, symbolischer Link)
- Größe der Datei in Byte
- Kennung des Datenträgers (Partition), der die Inode enthält
- Nummer der Inode: Durch die Kombination von Inodenummer und Kennung des Datenträgers lassen sich Inodes eindeutig identifizieren.
- Benutzer- und Gruppen-ID (UID und GID) kennzeichnen den Besitzer der Datei
- Zeitstempel (letzter Zugriff, letzte Änderung des Inhalts, letzte Änderung der Attribute)
- Rechte (lesen, schreiben, ausführen) für den Dateizugriff, differenziert nach Besitzer, Gruppe und andere Benutzer

3.1.3 Hardlinks und symbolische Links

Ein Verzeichniseintrag (der ja die Inode einer Datei nicht enthält, sondern nur aus einem Verweis auf diese besteht) wird auch als *hard link* bezeichnet. Eine Datei kann auch mehr als einen Verweis auf sich besitzen, sie scheint dann zwar in verschiedenen Verzeichnissen oder mit verschiedenen Namen in einem Verzeichnis öfters vorhanden zu sein, existiert tatsächlich aber nur einmal (sie wird nur öfters referenziert). Der Name ist also kein Attribut einer Datei. Reduziert sich die Anzahl der Verweise auf 0, wird die Datei gelöscht. *Hard links* besitzen allerdings einige Nachteile:

- Da Verzeichnisse Dateien nur über ihre Inodenummer ansprechen, sind diese auch nur innerhalb eines Dateisystems gültig, ein *hard link* darf Dateisystemgrenzen nicht überschreiten.
- Werden *hard links* auf Verzeichnisse angelegt, besteht die Gefahr, Zyklen im Verzeichnisbaum zu erzeugen (durch einen Verweis auf ein Unterverzeichnis, das ja damit indirekt den Verweis auf sich selbst enthält).

- Außerdem ist es für den Benutzer ohne Kenntnis der Inodenummer unmöglich, festzustellen, ob eine gelöschte Datei wirklich nicht mehr existiert, oder ob es irgendwo noch einen Verweis auf sie gibt, die Datei also noch vorhanden ist.

Mit 4.2BSD wurde das Konzept des *symbolischen Links* (oder *soft link*) eingeführt. Dies ist nichts anderes als eine spezielle Datei, deren Datenbereich einen Pfadnamen enthält, der auf eine andere Datei verweist. Der Pfadname kann relativ (zum Verzeichnis, das den Verweis enthält) oder absolut sein und muss nicht auf dasselbe Dateisystem verweisen. Durch *symbolische Links* werden viele der Nachteile von *hard links* vermieden. Das Dateisystem sorgt dafür, dass ein *symbolischer Link* aufgelöst wird, wenn auf ihn zugegriffen wird.

3.1.4 Das virtuelle Dateisystem

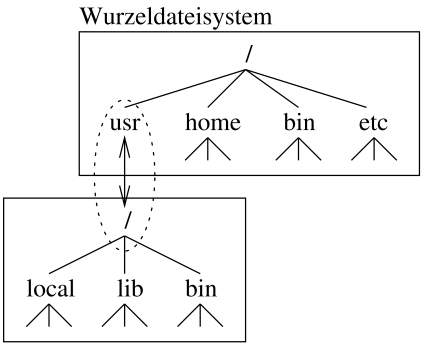
Mit der Zeit wurden verschiedene Dateisysteme für Unix entwickelt, darunter auch Spezialanwendungen (z.B. Systeme, die für Datenbankzugriffe optimiert sind). Der wachsende Einsatz von Netzwerken machte die Entwicklung verteilter Dateisysteme wie AT&Ts *Remote File Sharing RFS* oder Sun Microsystems *Network File System NFS* notwendig. Außerdem stieg der Bedarf an Unterstützung von nicht-Unix Dateisystemen (z.B. *MS-DOS*), zumindest für den Zugriff auf Disketten. Ein traditioneller Unix-Kernel mit einem monolithischen Aufbau erlaubte nur den Einsatz **eines** Dateisystems. Um diese Einschränkung zu umgehen, wurden verschiedene Lösungen entwickelt. Als de facto Standard kristallisierte sich die *vnode/vfs*-Architektur von Sun Microsystems heraus, die als Vorbild vieler heute existierender Implementierungen gedient hat und im Folgenden kurz erläutert wird.

Eingeführt wurden die Konzepte einer virtuellen Inode (*vnode*) und eines virtuellen Dateisystems (*vfs*), die dem Benutzer trotz unterschiedlicher Dateisysteme mit variierenden Eigenschaften eine einheitliche Sicht auf die Verzeichnisstruktur ermöglichen. Die Tatsache, dass mehrere Dateisysteme existieren, bleibt auf der Ebene des virtuellen Dateisystems vor dem Benutzer verborgen.

Um diese Ideen zu realisieren, bieten sich objektorientierte Konzepte und Techniken an ([Vah96]). Die Abstraktionen *vnode* und *vfs* können beide als abstrakte Basisklassen verstanden werden. Eine Inode eines bestimmten Dateisystems muss von der *vnode*-Klasse abgeleitet werden und implementiert die Methoden der Super-Klasse. Die Inode-Klasse wird innerhalb des *vfs* als *vnode* angesprochen, die dateisystemspezifischen Details der abgeleiteten Klasse sind hier nicht von Bedeutung. Durch die *vnode* als gemeinsame Basis der Inode-Klassen verschiedener Dateisysteme wird eine einheitliche Schnittstelle geschaffen. In der Sprache C, die normalerweise für Betriebssystem-Implementationen verwendet wird, werden Objekte durch Strukturen repräsentiert. Eine Vererbungshierarchie kann durch das Aufnehmen der Struktur der Super-Klasse als erstes Element in die eigene Struktur geschaffen werden; über eine Cast-Operation kann dann die Super-Klasse angesprochen werden.

3.1.5 Die *mount*-Operation

Die Verzeichnishierarchie eines Unix-Systems setzt sich aus einem oder mehreren unabhängigen Verzeichnisbäumen zusammen, wovon jeder Baum einem Dateisystem (mit Wurzelverzeichnis) angehört. Ein Dateisystem in der Hierarchie ist als *Wurzeldateisystem* ausgezeichnet, dessen Wurzelverzeichnis ist das *Systemwurzelverzeichnis* ([Vah96]). Die anderen Dateisysteme werden der bestehenden Verzeichnisstruktur durch die *mount*-Operationen hinzugefügt; das Wurzelverzeichnis eines Dateisystems überlagert dann ein Verzeichnis des bereits existierenden Baums. Die Wurzelverzeichnisse überdecken bzw. verbergen die überlagerten Verzeichnisse. Jeder Zugriff auf ein überlagertes Verzeichnis wird vom virtuellen Dateisystem in einen Zugriff auf das Wurzelverzeichnis des entsprechenden Dateisystems umgewandelt.



Im Beispiel wird das Verzeichnis `/usr` verdeckt; ein Zugriff auf `/usr` wird übersetzt in einen Zugriff auf das Verzeichnis `/` des untenstehenden Dateisystems. `/usr` wird auch als *mount point* bezeichnet. Beim Ablaufen eines Pfades müssen *mount points* vom Betriebssystem erkannt und umgesetzt werden. Alle Dateien und Unterverzeichnisse eines überlagerten Verzeichnisses (wie `/usr`) sind nicht erreichbar, solange das Dateisystem auf das Verzeichnis montiert ist.

Durch das Konzept der *mount*-Operation werden Details der Dateisystemorganisation vor dem Benutzer verborgen und der Namensraum kann homogen gestaltet werden. Der Benutzer braucht beim Zugriff auf eine Datei nicht deren genaue Datenträger/Partitionskennung zu wissen.

3.2 Design und Entwicklung der UNIX-Dateisysteme

3.2.1 Das System V Dateisystem *s5fs*

Zwei Beispiele für Dateisysteme, wie sie in traditionellen Unix Betriebssystemen häufig gefunden werden können, sind das „System V file system“ *s5fs* und das „Berkeley Fast File System“ *FFS*. Die Version *s5fs* ist das ursprüngliche Dateisystem für Unix, *FFS* wurde später mit dem Berkeley Unix in der Version 4.3BSD eingeführt und bietet bessere Performance, Sicherheit und Funktionalität als *s5fs* ([Vah96]). Die grundsätzliche Struktur eines Dateisystems wird im Folgenden an *s5fs* erklärt, das einen einfachen Aufbau besitzt (s. Abbildung 3.1).

Abbildung 3.1: Layout des Unix-Dateisystems *s5fs*

Eine Festplatte ist in eine oder mehrere Partitionen unterteilt; jede Partition kann höchstens ein Dateisystem enthalten. Jedes Dateisystem ist abgeschlossen und besitzt eine Verzeichnisstruktur (mit Wurzelverzeichnis), Daten und Verwaltungsdaten. Jede Partition kann als lineare Folge von Blöcken angesehen werden, die alle die gleiche Größe haben (z.B. 1024 Byte). Ein Block stellt die kleinste ansprechbare Einheit dar, sowohl für die Übertragung von und zur Festplatte, als auch für die Operationen des Dateisystems. Jeder Block erhält eine eindeutige (linear aufsteigende) Nummer, über die er innerhalb des Dateisystems angesprochen wird. Beim Schreiben und Lesen eines Blocks wird diese Nummer in ein für die Festplattenansteuerung erforderliches Format umgewandelt (z.B. in eine LBA-Nummer oder ein Zylinder/Kopf/Sektor-Tripel).

Der erste Block einer Festplatte ist der Bootbereich, der den Startupcode eines Betriebssystems oder einen Bootloader enthält. Der nächste Block ist der so genannte Superblock, in dem die Verwaltungsdaten gesichert werden, die das ganze Betriebssystem betreffen und für den Betrieb erforderlich sind (z.B. Anzahl der Blöcke, Nummer des ersten Datenblocks usw.).

Konzepte wie Dateien und Verzeichnisse, die von der Anwendersicht eines Dateisystems bekannt sind, werden in so genannten *Inodes* gespeichert. Das sind Datenstrukturen, die Informationen zu einem Objekt (Datei, Verzeichnis) enthalten, die für die Verwaltung notwendig sind (z.B. Rechte und Größe einer Datei). Die eigentlichen Daten (z.B. der Inhalt einer Datei oder die Einträge eines Verzeichnisses) werden in den Datenblöcken (s. unten) gehalten. Dem Superblock folgen Blöcke mit Inodes, die Inodeliste. Die Inodes werden ebenfalls mit linear aufsteigenden Nummern adressiert und besitzen eine feste Größe, die normalerweise kleiner als die Blockgröße ist, so dass pro Block mehrere Inodes gesichert werden können. Eine Inodenummer kann somit in Blocknummer und Offset innerhalb des Blocks umgerechnet werden. Die Zahl der Inodes bestimmt die max. Anzahl an Dateien und Verzeichnissen in einem Dateisystem.

Der Superblock und die Inodes sind die so genannten Verwaltungsdaten oder Metadaten eines Dateisystems. Sie belegen nur einen relativ kleinen Teil der Partition. Der übrige Platz wird von den Datenblöcken ausgemacht, die als Behälter für den Inhalt der Dateien dienen. Der Inhalt einer Datei mit einer Größe von 2500 Byte ist bei einer Blockgröße von 1024 Byte also auf drei Blöcke aufgeteilt, wobei der letzte nicht voll belegt ist. In den Inodes sind die Nummern dieser Blöcke gesichert. Dort können wegen der beschränkten Größe der Inodestruktur allerdings nur eine kleine Anzahl von Blöcken referenziert werden; besitzt die Datei mehr Blöcke, so werden Datenblöcke reserviert, die weitere Blocknummern aufnehmen (oder Nummern von Datenblöcken mit weiteren Blocknummern). Beim Auslesen des Inhalts einer Datei werden die Blöcke der Reihe nach in den Speicher geladen; die Fragmentierung in Blöcke bleibt vor dem Benutzer verborgen.

Im Superblock werden außerdem zwei Listen unterhalten, die auf die freien Blöcke und freien Inodes (Inodenummern) verweisen. Diese Listen sind durch verkettete Blöcke realisiert, die die Nummern der freien Blöcke bzw. Inodes enthalten.

3.2.2 Das Berkeley Fast File System

Es soll nur kurz auf die Unterschiede zum System V Dateisystem eingegangen werden, da viele Aspekte der Implementierung des *FFS* in das Dateisystem *JavaFS* eingeflossen sind und noch ausführlich in den folgenden Kapiteln behandelt werden.

Die Menge der Blöcke und Inodes wurde beim *FFS* in Gruppen eingeteilt, deren Position und Größe von den Zylindern der Festplatte bestimmt wurden; eine *Zylindergruppe* umfasst einen kleinen Satz benachbarter Zylinder. Die Operationen dieses Dateisystems versuchen, Zugriffe möglichst sequentiell zu gestalten und sie auf jeweils eine Zylindergruppe zu beschränken. Damit konnten die

Bewegungen der Schreib-/Leseköpfe optimiert und somit eine bessere Performance erreicht werden. Zum Beispiel werden neue Dateien bevorzugt in der Zylindergruppe ihres Verzeichnisses angelegt; ein Lesen des Verzeichnisinhalts findet dann nur innerhalb der Zylindergruppe des Verzeichnisses statt.

Außerdem werden Informationen, die beim *s5fs* im Superblock gehalten wurden, teilweise auf die Zylindergruppen verteilt (Liste der freien Blöcke, Inodes). Vom Superblock existieren Kopien in einigen Zylindergruppen, um diese wichtige Datenstruktur besser schützen zu können.

Weitere bedeutende Neuerungen beinhalten die Einführung von fragmentierten Blöcken, um den zur Verfügung stehenden Speicherplatz effizienter ausnutzen zu können, längere Dateinamen, symbolische Links und einen *rename*-Systemcall ([Vah96]). Die dynamische Verwaltung der Dateinamen wurde später vom Dateisystem *Ext2* übernommen.

3.2.3 Das Dateisystem *Ext2* von Linux

Das Dateisystem *Ext2* wird unter der Leitung von Theodore Ts'o seit 1993 entwickelt und stellt zur Zeit das Standard-Dateisystem für Linux dar. Es hat sehr viel gemeinsam mit dem *Fast File Fystem* von Berkeley. Im *Ext2* sind zwar auch Fragmente vorgesehen, werden aber z.Z. nicht verwendet; die Blockgröße lässt sich auf 1, 2 oder 4 KByte einstellen. Die *Blockgruppen* im *Ext2* entsprechen in etwa den *Zylindergruppen* des *FSS*. Das Aufteilen in Gruppen wird bei modernen Festplatten nicht mehr gemäß den Zylindergrenzen durchgeführt, da die physikalische Struktur nicht unbedingt mit der von außen sichtbaren logischen Struktur übereinstimmen muss. Eine homogene Belegung des Dateisystems versucht *Ext2* dadurch zu erreichen, dass neue Verzeichnisse möglichst gleichmäßig über die Gruppen verteilt werden ([SG94], [CTT94]).

Ext2 verwaltet die freien Blöcke und Inodes nicht in Listen, sondern unterhält Bitmaps. Jede Blockgruppe besitzt genau einen Block für die Bitmap der freien Blöcke (Blockbitmap) und einen für die Bitmap der freien Inodes (Inodebitmap). Hat z.B. Bit Nr. 5 in der Blockbitmap den Wert 1, bedeutet das: der Block mit der Nummer 5 relativ zum ersten Block der Gruppe ist bereits belegt. Mit diesem System ist es auf einfache Weise möglich, Blöcke im Voraus zu reservieren, indem statt nach einem freien Bit ein freies Byte in der Blockbitmap gesucht wird (*preallocation*). Dadurch wird das Fragmentieren von Dateien verhindert.

Eine weitere Beschreibung der Eigenschaften soll hier zurückgestellt werden, da sie weitgehend auch für das *JavaFS* von *metaXaOS* zutreffen (s. nächstes Kapitel). Abbildung 3.7 zeigt das Festplatten-Layout des Dateisystems *Ext2*.

3.3 Die Implementierung

3.3.1 Aufbau

Das Dateisystem wurde komplett in Java geschrieben, wie auch Teile des Festplattentreibers. Es soll noch einmal kurz auf die Anforderungen an die Implementierung eingegangen werden: Der in C verfasste Teil des Treibers sollte so minimal wie möglich gehalten werden und nur Funktionen bereitstellen, die den Zugriff auf die Hardware ermöglichen. Das Dateisystem soll „offen“ implementiert werden, d.h. es soll nicht auf die Implementierung eines bestimmten Dateisystemtyps oder die Verwendung von Festplatten als Medium beschränkt sein (so dass z.B. ein RAM-Dateisystem ebenso möglich ist wie der Zugriff auf CD-ROMs oder der Einsatz eines Journaling-Dateisystems). Das Layout auf der Festplatte sollte kompatibel zum Linux Standard-Dateisystem *Ext2* gehalten werden, um möglichst einfach Konsistenz- und Geschwindigkeitstests durchführen zu können.

Das System besitzt einen hierarchischen Aufbau, der in Abbildung 3.2 dargestellt und anschließend erklärt wird.

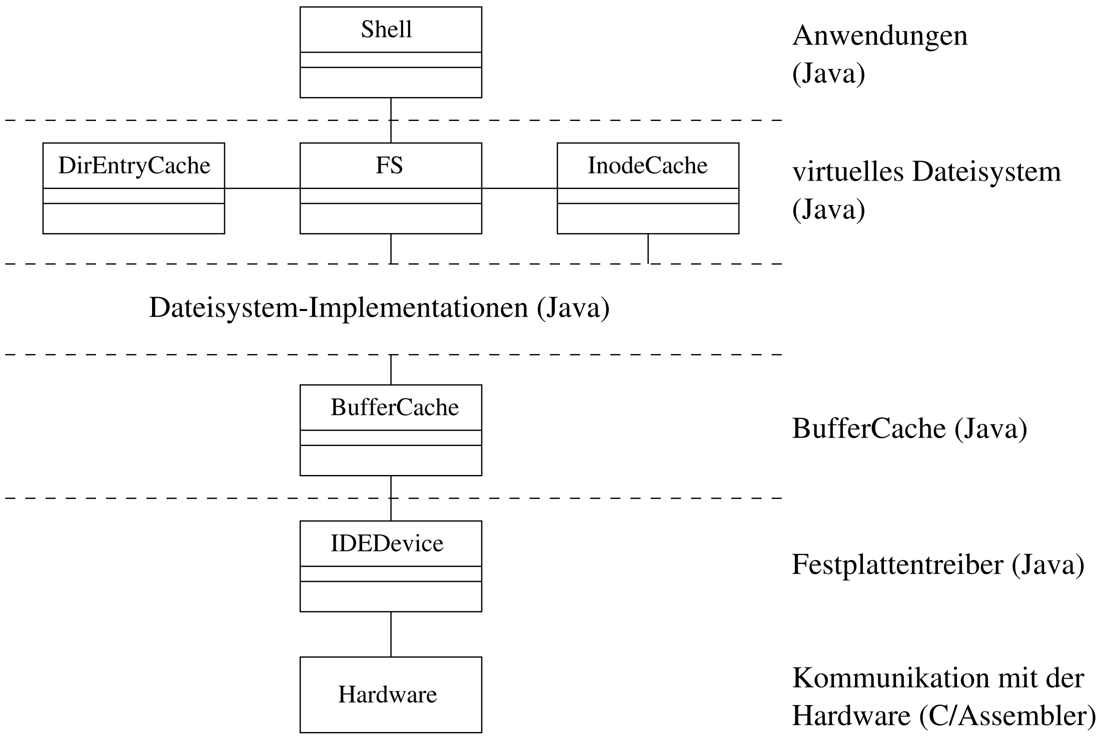


Abbildung 3.2: Der Aufbau des Dateisystems

Die Klasse `IDEDevice` kontrolliert den Zugriff auf die Ressource Festplatte und setzt Lese- und Schreiboperationen auf eine Partition in Zugriffe auf einen Registersatz eines IDE-Controllers um. Die Klasse `BufferCache` verwaltet die `BufferHead`-Objekte, die Dateisystem-Blöcke (ein oder mehrere Festplattensektoren) repräsentieren und durch die Angabe einer Partitionskennung, einer Blocknummer und -größe eindeutig identifiziert werden. Darauf aufsetzend sind die Dateisysteme wie das *JavaFS* implementiert. Auf dieser Ebene werden Dateisystemoperationen über *Inodes* durchgeführt, verschiedene Dateisysteme werden also durch ihre `Inode`-Klassen repräsentiert. Das virtuelle Dateisystem verwendet die Dateisystem-Implementierungen, um einen homogenen Verzeichnisbaum zu realisieren und bietet schließlich mit der Klasse `FS` eine Schnittstelle zur Anwendungsebene, wobei als Abstraktionen Dateien und Verzeichnisse, identifiziert durch ihre Pfadnamen, verwendet werden. Um Zugriffe auf Inodes zu beschleunigen, dienen die Klassen `DirEntryCache` und `InodeCache`, die Verzeichniseinträge bzw. Inodes verwalten und zwischenspeichern. Als Anwendung wurde eine `Shell` programmiert, die ähnlich einer UNIX-Shell Dateisystem-Kommandos wie `mkdir`, `ls` oder `cd` als Tastaturkommandos entgegennimmt.

3.3.2 Integration in das Betriebssystem

Die Klassen des Dateisystems befinden sich im Paket `fs` in `metaxa.os`. Für den Zugriff auf die Festplatte wird der IDE-Treiber im Paket `devices.ide` verwendet. Der Zugriff findet ausschließlich über die Methode `rwBlock` im `BufferCache`-Objekt statt. Das virtuelle Dateisystem wird von den Klassen im Paket `fs` realisiert, konkrete Dateisystemimplementationen sind in Paketen innerhalb von `fs` untergebracht (`metaxa.os.fs.javafs`). Die Objekte der Klasse `BufferHead`, die für den Datentransfer verwendet werden, greifen auf die Klasse `metaxa.os.Memory` zurück. Als Schnittstelle nach außen dienen die Klassen `FS` und `Inode`, über die die gesamte Funktionalität des Dateisystems erreichbar ist.

Abbildung 3.3 gibt einen Überblick über die Struktur der Pakete und Klassen.

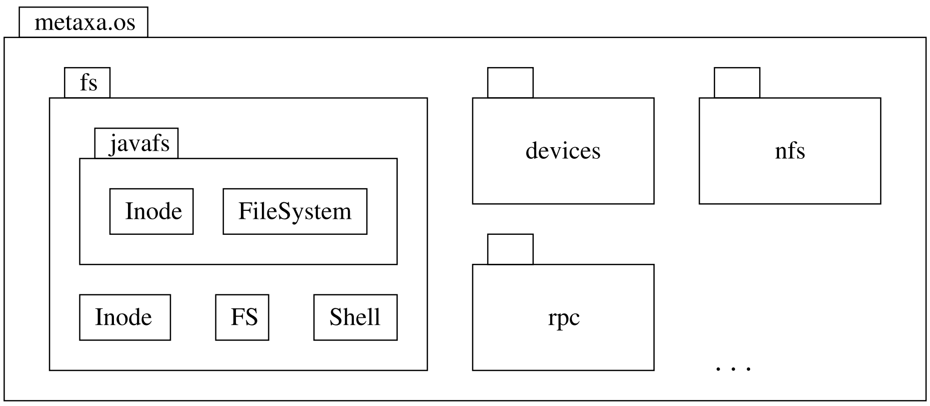


Abbildung 3.3: Integration des Dateisystems in *metaXaOS*

3.4 Das virtuelle Dateisystem

3.4.1 Überblick

Die Klassen des Java-Dateisystems befinden sich im Paket `metaxa.os.fs.javafs`. Um die Existenz und den (gleichzeitigen) Einsatz mehrerer Dateisysteme zu ermöglichen, wurden die Klassen im Paket `metaxa.os.fs` entworfen, die eine einheitliche Schnittstelle zwischen dem Betriebssystem und den Dateisystemen schaffen. Die Klasse `FS` stellt grundlegende Dateisystem-Operationen wie *mkdir* oder *unlink* zu Verfügung, wobei sie auf die Klasse `Inode` zurückgreift. Diese ist als *abstract* definiert und dient als Basisklasse für die Inode-Klassen der verschiedenen Dateisysteme;

die Implementierung der abstrakten Methoden ist den Dateisystemen überlassen. Durch diese Vorgehensweise wird das Konzept eines virtuellen Dateisystems realisiert, wie es z.B. von Unix bekannt ist. Dabei entsteht durch die Verwendung einer abstrakten Basisklasse als gemeinsame Schnittstelle kaum Overhead — Inode-Klassen, die von `fs.Inode` abgeleitet wurden, können direkt von der Klasse `FS` verwendet werden. Die Klasse `fs.Inode` lässt sich mit dem von Unix bekannten Konzept der `vnode` vergleichen. Eine ausführliche Dokumentation der Klassen `Inode` und `FS` befindet sich im Anhang B.

Folgende Tabelle zeigt die Methoden der Inode-Basisklasse, die vom virtuellen Dateisystem verwendet werden.

`isFile`	liefert *true*, falls es sich um eine Datei handelt.
`isDirectory`	liefert *true*, falls es sich um ein Verzeichnis handelt.
`isWritable`	liefert *true*, falls der Inhalt der Datei geändert werden kann (bei Verzeichnissen immer *false*).
`isReadable`	liefert *true*, falls die Datei gelesen werden kann (bei Verzeichnissen immer *false*).
`readdirNames`	liefert die Einträge des Verzeichnisses als `Vector`.
`getInode`	liefert die Inode zum angegebenen Namen (falls vorhanden).
`mkdir`	erzeugt ein neues Verzeichnis.
`rmdir`	löscht das angegebene Verzeichnis.
`create`	erzeugt eine neue Datei.
`unlink`	löscht die angegebene Datei.
`read`	liest aus der Datei.
`write`	schreibt in die Datei.
`available`	liefert den freien Platz auf dem Dateisystem (in Byte).
`getLength`	liefert die Größe der Datei/des Verzeichnisses.
`releaseInode`	gibt die Inode im InodeCache frei.

An dieser Stelle sei auf eine wichtige Eigenschaft der Implementation (oder allgemein des *vnode*-Konzepts) hingewiesen, die im objektorientierten Design begründet ist: die Funktionen des Dateisystems sind allesamt Methoden der die Datei bzw. das Verzeichnis repräsentierenden Inode. Um beispielsweise eine Datei zu löschen, muss die Inode des Verzeichnisses ermittelt und ihr mitgeteilt werden, dass die Datei entfernt werden soll; der entsprechende Verzeichniseintrag wird gelöscht und die darin referenzierte Inode veranlasst, die belegten Blöcke freizugeben. Die Klasse `FS` setzt diesen objektorientierten Mechanismus um auf Methoden, die mit Pfadnamen umgehen können; diese Herangehensweise hat mehr funktionalen Charakter, vergleichbar mit grundlegenden Systemkommandos unter Unix (`rm`, `mkdir` usw.).

Schließlich existiert noch das Interface `FileSystem`, das Operationen zur Verfügung stellt, die das Dateisystem als Ganzes betreffen. Folgenden Tabelle gibt einen Überblick über diese Methoden (s. auch Anhang B):

`init`	initialisiert das Dateisystem auf der angegebenen Partition (wird vom `mount`-Befehl der Klasse `FS` benötigt).

`release`	gibt die vom Dateisystem belegten Ressourcen frei und schreibt Blöcke mit Änderungen, die sich noch im BufferCache befinden, auf die entsprechende Partition.
`build`	erzeugt auf der angegebenen Partition ein neues Dateisystem.
`check`	überprüft das Dateisystem auf der angegebenen Partition auf Konsistenz und repariert es gegebenenfalls.

Dieses Interface zusammen mit Methoden aus `FS` (z.B. `mount`) entspricht also weitgehend dem `vfs`-Konzept in UNIX.

3.4.2 Der BufferCache

Der BufferCache erfüllt zwei Aufgaben: zum einen dient er als Pool für `BufferHead`-Objekte, die während einer laufenden I/O-Operation angefordert werden, zum anderen als Cache für abgeschlossene I/O-Operationen.

Im BufferCache werden `BufferHead`-Objekte, die Blöcke des Dateisystems repräsentieren, zwischengespeichert. Dazu wird eine Hashtabelle und eine Liste unterhalten. BufferHeads, die neu erzeugt wurden und noch nicht initialisiert sind, kommen zuerst in die Liste. Wird ein BufferHead angefordert (Methoden `getblk`, `bread`), der sich nicht im Cache (in der Hashtabelle) befindet, wird das erste Objekt von der Liste entfernt, neu initialisiert und in der Hashtabelle eingehängt. Jeder BufferHead protokolliert, ob er noch verwendet wird. Wird ein BufferHead nicht mehr benötigt, d.h. greift niemand mehr auf ihn zu, wird er ans Ende der Liste gehängt und wandert im Lauf der Zeit nach vorne. Die Liste enthält also BufferHeads in einer „Least Recently Used“ Reihenfolge. BufferHeads, deren Inhalt gültig ist, die aber nicht mehr benötigt werden, haben so die Chance, nochmal angesprochen zu werden, wenn auf den entsprechenden Block zugegriffen wird. Sie werden in diesem Fall wieder aus der Liste entfernt bzw. am Ende eingehängt. Durch diesen Mechanismus werden teure Festplattenzugriffe gespart, da der Inhalt eines BufferHeads u.U. oft gelesen und geändert wird, bevor er tatsächlich auf Festplatte geschrieben wird. Der Cache besitzt eine initiale Größe, kann aber bis zu einem bestimmten Umfang erweitert werden. Wird diese Grenze erreicht, und werden weitere BufferHeads angefordert, ohne dass sich Einträge in der LRU-Liste befinden, blockiert der Cache solange, bis wieder BufferHeads freigegeben werden. Der BufferCache greift auf den Festplattentreiber zurück, um Blöcke zu lesen und zu schreiben (Methode `rwBlock`).

Die Dokumentation im Anhang B gibt einen Überblick über die öffentlichen Methoden der Klasse `BufferCache`.

3.4.3 Der InodeCache

Die Klasse `InodeCache` beschleunigt den Zugriff auf häufig angesprochene Inodes, indem sie sie — ähnlich dem BufferCache — in einer Hashtabelle zwischenspeichert. Die Methode `getInode` der Klasse `Inode`, die von `lookup` aufgerufen wird, sucht zuerst im Cache nach der gewünschten Inode. Konnte sie dort nicht gefunden werden, wird die Inode neu angelegt (z.B. über `new DirInode()`) und dem Cache hinzugefügt. Wird die Inode nicht mehr benötigt, sollte der Cache durch Aufruf der Methode `releaseInode` davon benachrichtigt werden. Diese Vorgehensweise wird innerhalb der

FS-Klasse praktiziert. Falls das Freigeben der Inode vergessen wird, sorgt die finalize-Methode der Inode-Klasse dafür.

Der InodeCache besitzt eine Hashtabelle, in der alle im Cache befindlichen Inodes referenziert sind. Jede Inode wird zusätzlich in einer von zwei Listen gehalten, inode_dirty für die als „dirty“ markierten Inodes, die noch geschrieben werden müssen, inode_in_use für die Übrigen. Aus der Inodenummer und der Partitionskennung (die Inodenummern sind nur innerhalb einer Partition/eines Dateisystems eindeutig) wird ein Hashschlüssel generiert, der für den Zugriff auf die Hashtabelle notwendig ist. Jede Inode besitzt einen Nutzungszähler, der erhöht wird, wenn die Inode angefordert wird (Methode iget) und erniedrigt, wenn sie freigegeben wird (iput). Einmal registrierte Inodes werden normalerweise nicht aus dem Cache entfernt, außer dieser ist zu voll (in diesem Fall werden Inodes mit einem Nutzungszähler von 0 entfernt) oder ein Dateisystem wird abgemeldet, so dass die Inodes dieses Dateisystems (über die Methode invalidateInodes) freigegeben werden müssen. Eine ausführliche Dokumentation der Methoden befindet sich im Anhang B.

3.4.4 Der DirEntryCache

Die Klasse DirEntryCache speichert Verzeichniseinträge in Form von Pfad-Inode-Paaren. Die Methode FS.lookup, die die Inode zu einem angegebenen Pfad liefert, fragt als Erstes den Cache ab, ob der angegebene Pfad dort vermerkt ist. Der Cache stellt dazu die Methode getEntry(String pfad) bereit. Falls der Eintrag noch nicht im Zwischenspeicher ist, oder falls eine neue Datei (create) bzw. ein neues Verzeichnis (mkdir) erzeugt wird, kann der Pfad und die im zugeordnete Inode über addEntry(String pfad, Inode inode) dem Cache hinzugefügt werden. Analog zum Erzeugen existieren Methoden zum Löschen und Umbenennen von Einträgen.

Wie beim InodeCache werden Verzeichniseinträge normalerweise nicht aus dem Cache entfernt, außer der Cache ist voll oder ein Dateisystem wird abgemeldet. In diesem Fall können die Einträge über die Methode invalidateEntries freigegeben werden. Der DirEntryCache sorgt dafür, dass die Inodes der von ihm verwalteten Einträge nicht aus dem InodeCache entfernt werden, indem er den Nutzungszähler entsprechend erhöht und bei Einträgen, die gelöscht werden, die Inodes freigibt.

Im Gegensatz zu Unix bzw. Linux wird der DirEntryCache ausschließlich im virtuellen Dateisystem verwendet; Dateisystemimplementierungen kennen die Zuordnung Pfad-Inode nicht. Unter Unix werden mit dem DirEntryCache Einträge verwaltet, die eine Inode mit einem bestimmten Namen (nicht Pfad!) verknüpfen und einen Verweis auf die Inode des übergeordneten Verzeichnisses enthalten. Dadurch ist der Unix-DirEntryCache in weitaus größerem Maße mit den anderen Caches und den Dateisystem-Implementierungen verwoben als unsere Lösung, die einfach und modular realisiert werden konnte.

3.4.5 Zugriff auf eine Inode

Im Folgenden soll an zwei Szenarien die Verwendung der Inodes und der Caches und die Interaktion der Klassen des virtuellen Dateisystems mit denen einer konkreten Dateisystemimplementierung verdeutlicht werden. Als Beispiel soll die Operation create(String pfad) der Klasse FS dienen, die eine neue Datei unter dem angegebenen Pfad erzeugt.

Zuerst wird die Inode des Verzeichnisses ermittelt, in dem die Datei angelegt werden soll. Falls es sich nicht um das aktuelle Verzeichnis handelt, wird die Methode `lookup` aufgerufen, die zunächst im DirEntryCache nach einem Eintrag mit dem angegebenen Pfad sucht. Konnte die Inode nicht ermittelt werden, läuft die Methode ausgehend vom Wurzelverzeichnis („`/`“) durch rekursive Aufrufe den Pfad entlang und löst dabei symbolische Links auf. Der Algorithmus, der verwendet wird, um die dem Namen zugeordnete Inode zu erhalten, wird im Folgenden ausführlich erläutert.

Jede Inode besitzt die Methode `lookup`, die — falls es sich um ein Verzeichnis handelt — als ersten Schritt den Namen auf „`.`“ und „`..`“ überprüft und bei Übereinstimmung die Inode selbst bzw. die übergeordnete (`parent`) zurückliefert. Als Nächstes wird der Name in der Liste der Namen der überlagerten Inodes gesucht und bei einem Treffer die aufgesetzte Inode zurückgegeben. Wurde der Name bis jetzt nicht gefunden, wird die Methode `getInode` aufgerufen, die in der Klasse `Inode` des Pakets `fs` als *abstract* definiert ist. Die Implementation dieser Routine befindet sich in der Klasse `DirInode` im Paket `javafs`; wir wechseln also vom virtuellen Dateisystem zum *JavaFS* (s. Abbildung 3.4).

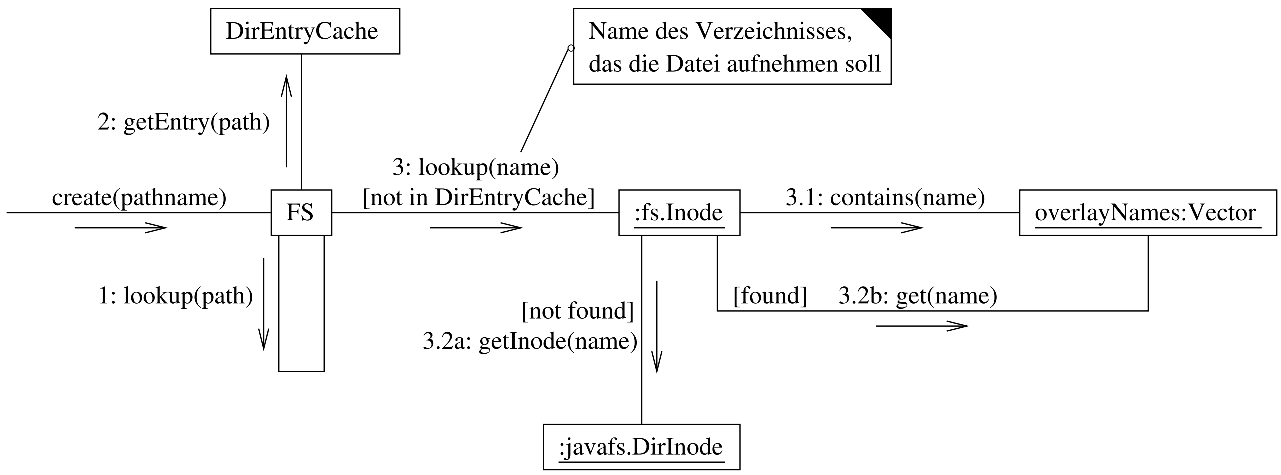


Abbildung 3.4: Ablaufdiagramm für den Befehl `create` (a)

Innerhalb dieser Methode werden über `findDirEntry` die Verzeichniseinträge nach dem Namen des gewünschten Verzeichnisses durchsucht. Wird der Eintrag nicht gefunden, wird eine `InodeNotFoundException` geworfen. Im Erfolgsfall besitzen wir die Nummer der Inode (gespeichert im Verzeichniseintrag). Damit kann die Klasse `InodeCache` abgefragt werden.

Wurde die Inode vorher schon mal angesprochen, befindet sie sich wahrscheinlich noch im Cache und die Arbeit der Methode `getInode` ist erledigt. Falls nicht, wird von der `Super`-Klasse ein `InodeData`-Objekt angefordert, aus dem der Typ der Inode ausgelesen wird. Die Methode `Super.getInodeData` rechnet dazu die Inodenummer in die Nummer der Blockgruppe, die Nummer des Blocks der Inodetabelle dieser Gruppe und den Offset innerhalb dieses Blocks um. Je nach Typ wird ein neues `FileInode`-, `DirInode`- oder `SymlinkInode`-Objekt erzeugt. Die Inode wird dem Cache hinzugefügt (`InodeCache.addInode`) und zurückgegeben (s. Abb. 3.5).

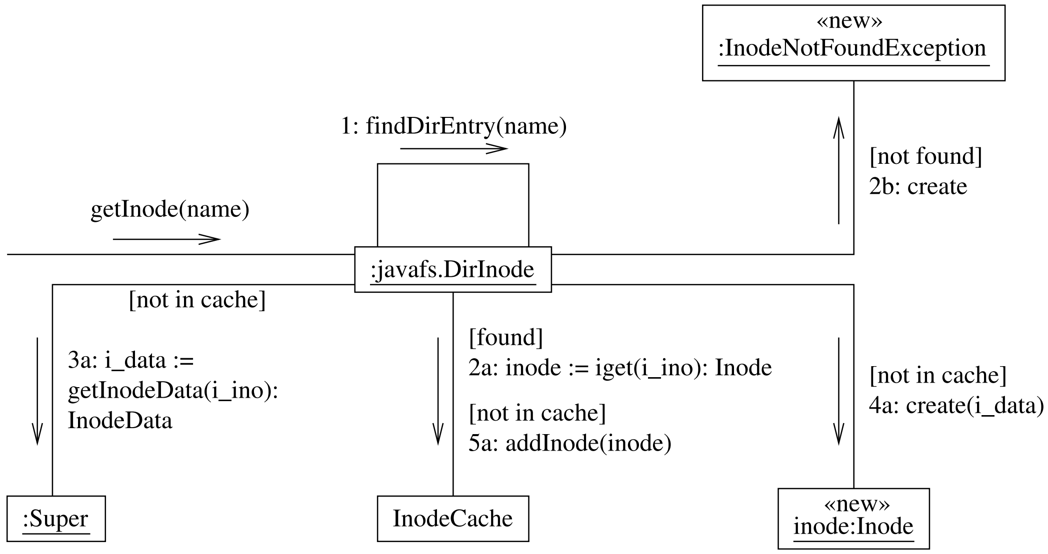


Abbildung 3.5: Ablaufdiagramm für den Befehl `create` (b)

3.4.6 Erzeugen einer Inode

Nachdem die Inode des Verzeichnisses ermittelt wurde, die die Datei aufnehmen soll, wird an dieser Inode die Methode `create(String name)` mit dem Dateinamen (ohne Pfad) aufgerufen (Pfade existieren nur innerhalb der `FS`-Klasse). Diese Methode ist in der Klasse `fs.Inode` ebenfalls als *abstract* definiert und findet sich wiederum in `fs.javafs.DirInode`.

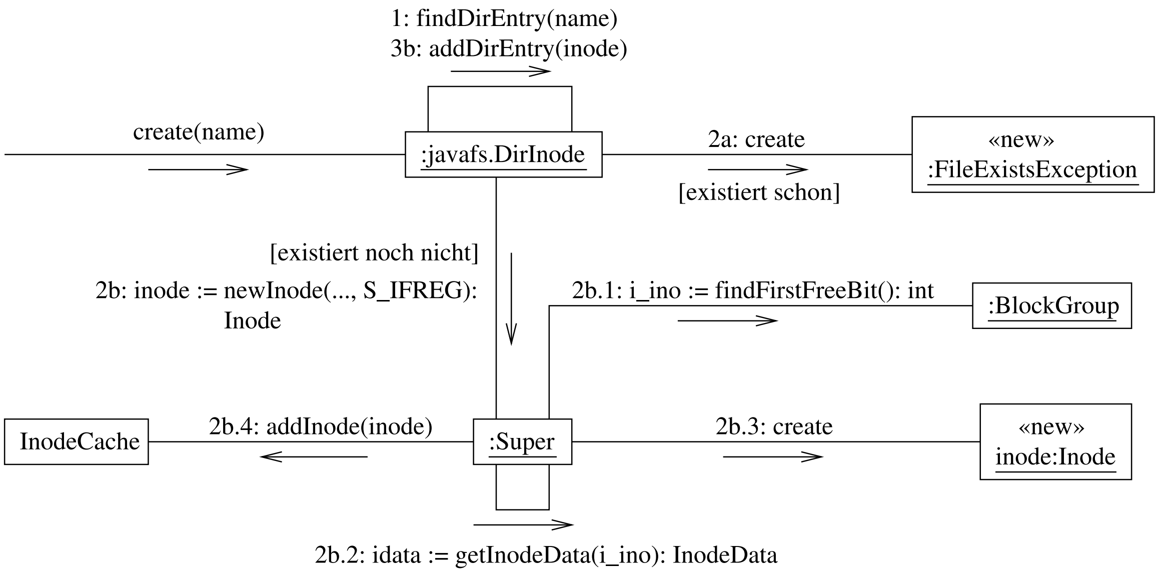


Abbildung 3.6: Ablaufdiagramm für den Befehl `create` (c)

Dort wird zunächst überprüft, ob ein Eintrag mit diesem Namen nicht schon existiert (Methode `findDirEntry(name)`) und gegebenenfalls eine `FileExistsException` ausgelöst. Die Inode wird mittels `Super.newInode(Inode dir, int mode)` (mit `mode = S_IFREG`) neu erzeugt. Dazu sucht die Super-Klasse nach einer freien Inode innerhalb der Blockgruppe des Verzeichnisses und wenn dort keine frei ist, auch in den anderen Blockgruppen. Das entsprechende Bit in der Inodebitmap der Gruppe wird als belegt markiert und der Zähler für die freien Inodes (`free_inodes_count`) erniedrigt. Die Methode `getInodeData` liefert die Inodestruktur, die anschließend initialisiert wird. Schließlich wird ein Inodeobjekt je nach Typ (`Dir`-, `Symlink`- oder `FileInode`) erzeugt und dem Cache hinzugefügt (`addInode`). Jetzt muss die Methode `create` lediglich mittels `addDirEntry(name, inode.i_ino)` einen neuen Verzeichniseintrag anlegen und dem Verzeichnis hinzufügen (s. Abbildung 3.6). Der letzte Schritt besteht darin, den Verzeichniseintrag in den DirEntryCache zu übernehmen (Methode `DirEntryCache.addEntry`, in der Abbildung nicht gezeigt).

3.5 Das Dateisystem *JavaFS*

3.5.1 Struktur

Das Layout von *JavaFS* innerhalb einer Partition zeigt Abbildung 3.7.

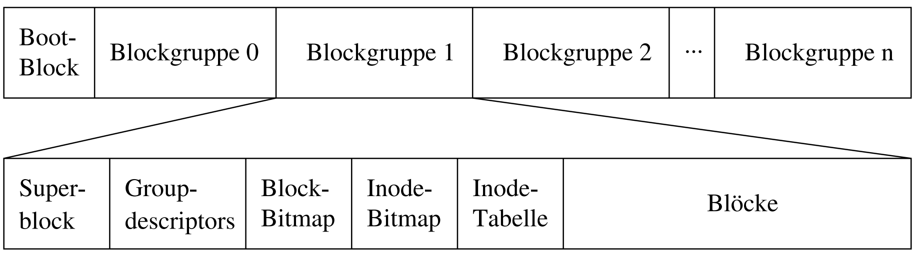


Abbildung 3.7: Layout des *JavaFS*

Das Dateisystem ist aufgeteilt in Blockgruppen. Jede Blockgruppe besitzt eine feste Anzahl von Inodes und Blöcken. Falls Blöcke angefordert werden (z.B. wenn in eine Datei geschrieben wird), werden diese — so weit es möglich ist — innerhalb einer Blockgruppe reserviert; dadurch wird Fragmentierung von Dateien weitestgehend verhindert. Eine neue Inode wird — falls es sich nicht um ein Verzeichnis handelt — innerhalb der gleichen Blockgruppe angelegt, die auch die Inode des übergeordneten Verzeichnisses enthält. Neue Verzeichnisse werden in der Blockgruppe angelegt, die am meisten freien Platz bietet, um das Dateisystem möglichst gleichmäßig zu füllen.

Der Vorteil von Blockgruppen im Gegensatz zu linear aufgebauten Dateisystemen (wie *s5fs*) liegt also darin, dass sich zusammengehörende Blöcke und Inodes in einem Verzeichnis besser gruppieren lassen. Verzeichnisse werden gleichmäßig über den zur Verfügung stehenden Platz verteilt und die Wahrscheinlichkeit verringert, den Inhalt einer Datei zu fragmentieren.

Ein Block umfasst einen oder mehrere Festplattensektoren; erlaubt sind die Größen 1024, 2048 und 4096 Byte. Zur Verwaltung der freien Blöcke und Inodes dienen Bitmaps, die die Größe eines Blocks haben, also bei einem Bit pro Inode bzw. Block zwischen 8192 und 32768 Einträge enthalten können. Dadurch wird die Größe einer Blockgruppe festgelegt. In der Klasse `Bitmap` existieren Funktionen, um einzelne Bits zu setzen und zu löschen und nach freien Bits zu suchen. Die Inodebitmap verwaltet allerdings weitaus weniger Inodes als die Zahl der Bits zulassen würde, da der Datenbereich einer Inode normalerweise mehr als einen Block umfasst und die Anzahl der Blöcke pro Blockgruppe bereits beschränkt ist. Wie bei der Beschreibung des Dateisystems *Ext2* bereits dargestellt wurde, kann durch die Verwendung einer Bitmap Fragmentierung von Dateien verhindert werden. Wird ein freier Block angefordert, versucht das Dateisystem ein freies Byte in der Bitmap zu finden, so dass nachfolgende Anfragen besser gruppiert werden können. Bitmaps lassen sich außerdem einfacher handhaben als Listen.

Die Inodes schließlich sind in den Inodetabellen gespeichert; die Inodebitmap gibt an, welche Einträge gültig sind und welche nicht. Eine Inode belegt 128 Byte, pro Block sind also zwischen 8 und 32 Inodes untergebracht. Eine Liste von Blöcken, die innerhalb der Inode platziert ist, verweist (direkt oder indirekt über weitere Listen) auf die Blöcke, die den *Datenbereich* einer Inode ausmachen. Darunter zu verstehen ist bei einem Verzeichnis die Liste der Verzeichniseinträge, bei einer Datei der Dateiinhalt.

Zu Beginn jeder Gruppe ist ein Bereich für die *group descriptors* reserviert. Diese Strukturen enthalten die Blocknummern der Blockbitmap, Inodebitmap und Inodetabelle dieser Gruppe, die Anzahl der freien Blöcke und Inodes und die Zahl der Verzeichnisse, die die Gruppe enthält.

Der Superblock beinhaltet allgemeine, für das gesamte Dateisystem relevante Daten, wie die Gesamtzahl Inodes und Blöcke, die Anzahl der freien Inodes und Blöcke, Zeitstempel, die Anzahl der Blöcke und Inodes pro Blockgruppe und die Größe der Blöcke und Inodes.

Der Superblock sowie die Strukturen der Gruppen sind in jeder Blockgruppe als Kopien vorhanden. Nur die Daten aus der ersten Blockgruppe werden verwendet, die Kopien dienen als Sicherung im Falle eines Festplattenfehlers innerhalb der ersten Blockgruppe.

3.5.2 Integration in das virtuelle Dateisystem

Das Dateisystem *JavaFS* besitzt ebenfalls eine Klasse `Inode`, die von `fs.Inode` erbt und als Basisklasse für spezialisierte Inode-Klassen dient — `DirInode` für Verzeichnisse, `FileInode` für Dateien und `SymlinkInode` für symbolische Links. In der Verzeichnis-Klasse `DirInode` sind nur die Funktionen implementiert, die auf Verzeichnissen anwendbar sind: `mkdir`, `rmdir`, `create`, `unlink`, `getInode` und `readdirNames`. Die Klasse `FileInode` stellt die Funktionen `read` und `write` für den Zugriff auf den Dateiinhalt zur Verfügung. Für symbolische Links zuständig ist schließlich die Klasse `SymlinkInode`, die die Methoden `setSymlink` und `getSymlink` aufweist, die einen Verweis auf eine andere Datei setzen bzw. auflösen. Methoden, die für die jeweilige Inode-Unterklasse keinen Sinn machen, werfen Ausnahmen, die auf den falschen Typ hinweisen: `NoDirectoryInodeException`, `NoFileInodeException` und `NoSymlinkInodeException`.

Außerdem ist eine `FileSystem`-Klasse vorhanden, die die Methoden des gleichnamigen Interface im Paket `fs` implementiert.

Die Klasse `Inode` bietet die Funktionalität, die von allen abgeleiteten Klassen benötigt wird; es existieren Methoden, die die Inodestruktur mit den direkten und indirekten Verweisen auf die Datenblöcke einlesen. Der Datenbereich wird auf eine linear durchnummerierte Folge von Blöcken abgebildet, die angefordert (`getblk`) und eingelesen (`bread`) werden können. Die Vererbungshierarchie der Inode-Klassen des *JavaFS* soll anhand von Abbildung 3.8 verdeutlicht werden.

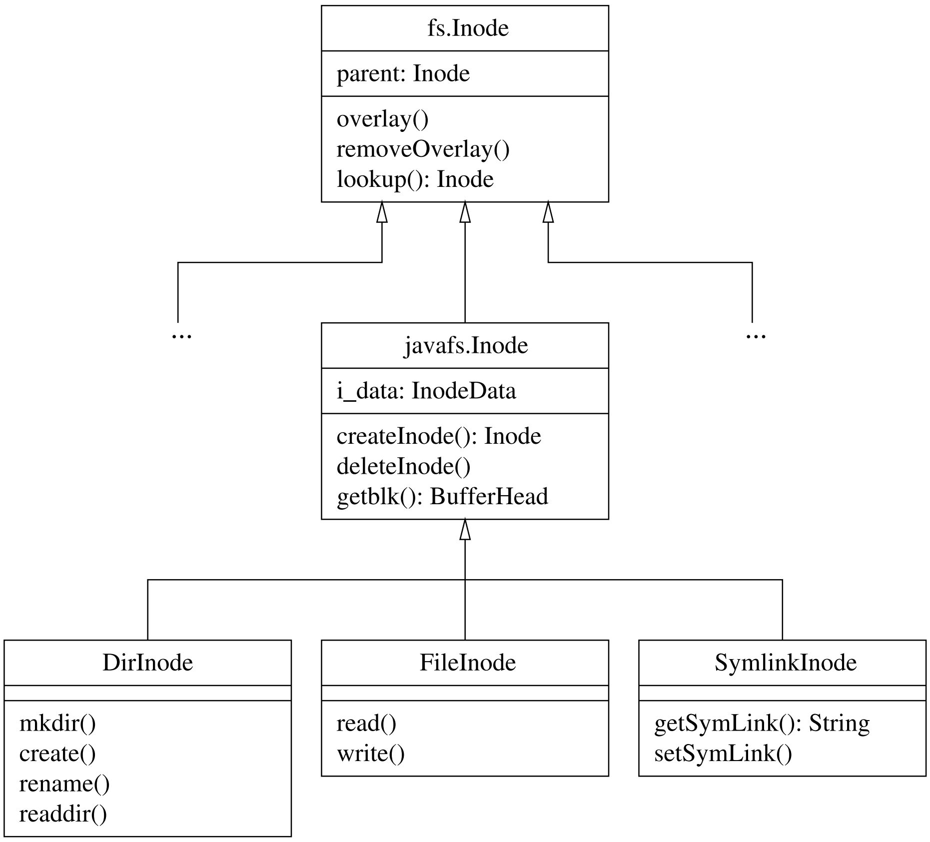


Abbildung 3.8: `Inode` und Unterklassen

3.6 Das Festplattenlayout

3.6.1 Der Zugriff auf Blockinhalte

Um den Inhalt von Blöcken zu lesen und zu ändern, dienen die Objekte der `BufferHead`-Klasse, die einen Buffer in Form eines `Memory`-Objekts in der Größe eines Blocks enthalten. Jedem Block kann ein `BufferHead`-Objekt zugewiesen werden, wenn auf den Inhalt zugegriffen werden soll; diese Objekte werden durch die Blocknummer und Partitionskennung eindeutig identifiziert (wie eben auch die Blöcke im Dateisystem). Innerhalb des Festplattentreibers und des Dateisystems werden Blockinhalte ausschließlich durch `BufferHeads` repräsentiert; diese Objekte werden in den Klassen des Pakets `fs` für den Informationsaustausch verwendet und stellen ein wichtiges Konzept des Dateisystems dar. Die `Memory`-Objekte als Datenpuffer stellen ein effektives Mittel für Lese- und Schreiboperationen dar; der Konstruktor eines `BufferHeads` erzeugt ein `Memory`-Objekt in der gewünschten Größe. Der Festplattentreiber kann die Adresse des Speicherbereichs, der durch ein `Memory`-Objekt dargestellt wird, direkt für die Datenübertragung verwenden (s. `inputData` und `outputData` der `Ports`-Klasse, die über die Prozessorbefehle `inl` bzw. `outl` realisiert sind). Für die Verwaltung der `BufferHeads` dient die Klasse `BufferCache`, die Verweise auf diese Objekte in einer Hashtabelle speichert und so schnellen Zugriff auf Blockinhalte ermöglicht.

Die Funktionen `read` und `write` dienen als Schnittstelle zum Anwender und erwarten als Argument ein byte-Array, das den Puffer für die zu lesenden bzw. schreibenden Daten darstellt. In der Klasse `Inode` wird der Inhalt dieser byte-Arrays in die `Memory`-Objekte der `BufferHeads` (bzw. umgekehrt) kopiert, diese dienen ja zum Datenaustausch innerhalb der Betriebssystem-Ebene.

Der Inhalt der `BufferHeads` wird nicht interpretiert, wenn es sich um den Inhalt einer Datei handelt. Diese Datenblöcke bilden die Mehrheit der Blöcke einer Partition. Ein Dateisystem besitzt natürlich auch eine bestimmte Menge von Blöcken mit Verwaltungsdaten (Superblöcke, Inodes, Bitmaps) deren Inhalt für das Dateisystem von Bedeutung ist. Diese Daten sind in den Blöcken ähnlich wie eine C-Struktur angeordnet, je nach Offset innerhalb des Blocks muss der dort gespeicherte Wert anders interpretiert werden, wobei nur bestimmte Offsets gültig sind und Daten unterschiedliche Längen haben können (je nach Typ, z.B. Integer, Byte, String). Um den Inhalt dieser `BufferHeads` ansprechen zu können, wurde die Klasse `BufferHeadAccess` entwickelt, die Paare von Zugriffsfunktionen (zum Lesen und Schreiben) bereitstellt, die jeweils den Offset und den Datentyp bestimmen[1] (s. Abbildung 3.9).

Als Beispiel soll die Methode `i_size` der Klasse `InodeData` dienen, die die Größe der Inode (Datei, Verzeichnis) im Dateisystem liefert bzw. setzt. Diese Methode liest bzw. schreibt an der Position 4 im `Memory`-Objekt 2 Byte, da es sich um einen `short`-Wert handelt:

```
short i_size();
void  i_size(short value);
```

Alle Zugriffsfunktionen einer Abstraktion sind in einer eigenen Klasse gruppiert, die die Endung „`Data`“ erhält, z.B `InodeData`. Diese Klasse wird mit dem `BufferHead`, der die Daten enthält, und einem Offset innerhalb dieses Buffers initialisiert, z.B.:

```
InodeData i_data = new InodeData(bh, 0);
```

Alle „`Data`“-Klassen sind von der Klasse `BufferHeadAccess` (im Paket `fs`) abgeleitet, die neben den Zugriffsfunktionen auch Methoden zum Löschen der Datenstruktur und zur Neuinitialisierung bietet. Zugriffsfunktionen existieren für folgende Datentypen: `byte`, `short`, `int`, `long`,

[1] Genau diese Funktionalität weist das Java-API mit den Klassen `DataInputStream` und `DataOutputStream` im Paket `java.io` auf, oder jede andere Klasse, die das Interface `DataInput` bzw. `DataOutput` implementiert. Im Unterschied zu `BufferHeadAccess` geschieht der Zugriff allerdings auf byte-Arrays.

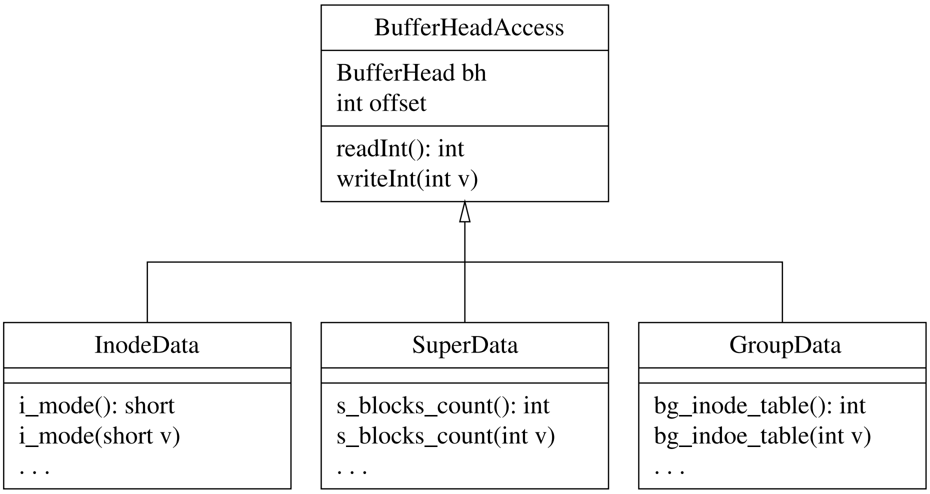


Abbildung 3.9: `BufferHeadAccess` und Unterklassen

`String` und `Time`, mit dem sich ein Java-Zeitstempel (ein 64 Bit-Wert mit den Millisekunden seit 1.1.1970) in einen C-Zeitstempel (ein 32 Bit-Wert mit den Sekunden seit 1.1.1970) transformieren lässt. Ein Java-`String` wird nicht im Unicode-, sondern (wie in C) im 8 Bit-„ISO 8859_1“-Format gesichert.

Es existieren folgende Data-Klassen:

Im Paket `devices.ide`: `DriveIdData`, `CmosData`, `MBRData` und `PartitionData`

Im Paket `fs.javafs`: `InodeData`, `InodeBlockData`, `DirEntryData`, `SuperBlockData`, `SymlinkBlockData` und `GroupData`

3.6.2 Die Inode

Die Inode ist eines der Basiskonzepte des Systems; jede Datei und jedes Verzeichnis im Dateisystem wird durch genau eine Inode repräsentiert. Die Schnittstelle zwischen dem virtuellem und einem konkreten Dateisystem wird durch die Inode-Klasse realisiert. Wichtige Daten einer Inode werden — wie beim Superblock — in einer `Data`-Klasse gesichert (`InodeData`). Objekte dieser Klasse bilden die Inodetabelle; jede Struktur umfasst 128 Byte. Pro Blockgruppe existiert eine Bitmap, die die belegten und freien Einträge verwaltet.

Eine Inode enthält unter anderem folgende Informationen:

Typ Hier ist zum einen der Typ kodiert, den diese Inode beschreibt: Datei, Verzeichnis oder symbolischer Link. Zum anderen werden hier die Zugriffsrechte gespeichert.

Informationen über den Besitzer Benutzerkennung (`user identifier`) und Gruppenkennung (`group identifier`) des Besitzers dieser Datei oder dieses Verzeichnisses. Dem Dateisystem wird dadurch ermöglicht, die Zugriffe auf die Inode zu kontrollieren.

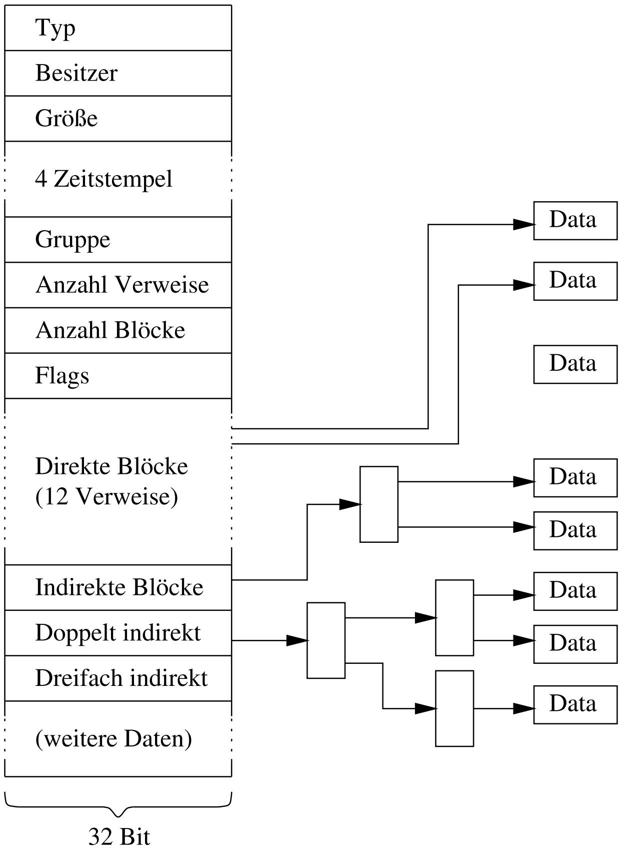


Abbildung 3.10: Die Inodestruktur

Größe Die Größe der Datei in Byte und als Zahl der belegten Blöcke im Dateisystem.

Zeitstempel Der Zeitpunkt des Erzeugens und Löschens der Inode, des letzten Zugriffs und der letzten Änderung.

Verweise Die Zahl der Verweise auf diese Inode; bei einem Verzeichnis die Zahl der Unterverzeichnisse. Die Inode wird gelöscht, falls diese Variable den Wert 0 annimmt.

Datenbereich Eine Liste der Nummern der Blöcke, die die Daten der Inode enthalten. Die ersten zwölf sind direkte Verweise auf Datenblöcke und die letzten drei einfach, doppelt und dreifach indirekte Verweise auf weitere Blöcke mit Blocklisten. Der doppelt indirekte Zeiger zum Beispiel verweist auf einen Block mit Nummern von Blöcken, die die Nummern von Datenblöcken enthalten. Die direkten Verweise (die ersten 12 Einträge) werden natürlich schneller aufgelöst als die indirekten, da hier nicht noch Blöcke nachgeladen werden müssen (s. Abbildung 3.10).

Falls es sich um einen symbolischen Link handelt, enthält der Datenbereich (der erste Datenblock) den Pfad der Datei, auf den sich der Verweis bezieht. Ist dieser Pfad kleiner als 60 Zeichen, wird er in der Blockliste innerhalb der Inodestruktur untergebracht. In diesem Fall besitzt die Inode keinen Datenbereich.

Bei einem Verzeichnis wird der Datenbereich von den Verzeichniseinträgen gebildet, die sich aus Name und Inodenummern zusammensetzen. Die „Größe“ eines Verzeichnisses gibt also Auskunft über den von den Verzeichniseinträgen belegten Platz. Im Gegensatz zur Datei kann auf den Datenbereich allerdings nicht zugegriffen werden; dafür bietet die Klasse `DirInode` die Methoden `readdirNames`, `addDirEntry`, `findDirEntry` und `removeDirEntry`.

3.6.3 Der Superblock

Der Superblock verwaltet grundlegende Daten des Dateisystems; er stellt eine der wichtigsten Datenstrukturen dar. Normalerweise wird nur der Superblock in der ersten Blockgruppe gelesen, wenn das Dateisystem angemeldet wird (`mount`-Operation), aber jede Blockgruppe enthält zur Sicherheit eine Kopie des Superblocks.

Der Superblock beinhaltet folgende Informationen:

Dateisystem-Kennung Dieser Wert wird zur Identifizierung des Dateisystems verwendet. Da das *JavaFS* kompatibel zum Dateisystem *Ext2* gehalten wurde, ist die „magic number“ 0xef53.

Revisionsnummer Die Revisionsnummer dient wie die Dateisystem-Kennung zur Identifizierung des Dateisystems. Darüberhinaus ist eine Unterscheidung verschiedener Versionen des Dateisystems mit unterschiedlichen Eigenschaften möglich. Das Feld wird auf 0 gesetzt, was der aktuellen *Ext2*-Implementation entspricht.

Blockgruppen-Nummer Die Nummer der Blockgruppe, die diese Kopie des Superblocks enthält

Blockgröße Die Größe eines Blocks für dieses Dateisystems (0 bedeutet 1024). Mit der Formel $2^{(\text{Wert}+10)}$ lässt sich die Größe in Byte berechnen.

Blöcke pro Gruppe Anzahl der Blöcke pro Blockgruppe. Wird beim Erzeugen des Dateisystems festgelegt.

Freie Blöcke Anzahl der freien Blöcke im Dateisystem

Freie Inodes Anzahl der freien Inodes im Dateisystem

Inodegröße Die Größe einer Inode in Byte (normalerweise 128)

Erste Inode Das ist die Nummer der ersten Inode im Dateisystem, die nicht für die interne Verwaltung verwendet wird (normalerweise 11). Die ersten 10 Inodes sind reserviert, z.B. ist dem Wurzelverzeichnis („ / “) die Nummer 2 zugeordnet.

Um das Layout kompatibel zum Dateisystem *Ext2* zu halten, enthält der Superblock noch weitere Felder, die aber im *JavaFS* nicht verwendet werden.

3.6.4 Die Blockgruppe

Jede Blockgruppe besitzt eine Struktur, in der wichtige Daten der Gruppe gesichert werden (*group descriptor*). Wie der Superblock sind diese Strukturen in allen Blockgruppen als Kopie vorhanden.

Folgende Informationen werden von einer Blockgruppe verwaltet:

Blockbitmap Die Blockbitmap ist eine Folge von Bits, die Auskunft über die belegten und freien Blöcke innerhalb einer Blockgruppe gibt (so bedeutet der Bitwert 1: Block belegt, 0: Block frei). Auf diese Bitmap wird zugegriffen, wenn ein Block angefordert wird oder wenn Blöcke freigegeben werden. Dieses Feld sichert die Nummer des Blocks, der die Bitmap enthält.

Inodebitmap Die Inode Bitmap entspricht der Block Bitmap und zeigt die freien und belegten Inodes innerhalb der Inodetabelle der Blockgruppe.

Inodetabelle Das Feld enthält die Nummer des ersten Blocks der Inodetabelle für diese Blockgruppe. Die Inodetabelle umfasst unter Umständen mehrere Blöcke und beinhaltet die Strukturen der Inodes dieser Gruppe (s. Kapitel 3.6.2).

Freie Blöcke Anzahl der freien Blöcke in der Blockgruppe

Freie Inodes Anzahl der freien Inodes in der Blockgruppe

Verzeichnisse Anzahl der Verzeichnisse in der Blockgruppe

Die Informationen aller Gruppen sind in einer Tabelle angeordnet, die u.U. mehrere Blöcke umfasst (je nach Anzahl der Gruppen). Jede Gruppe enthält die gesamte Tabelle, sie folgt der Kopie des Superblocks. Allerdings wird nur die erste Kopie (in der Gruppe 0) verwendet; die anderen Kopien dienen als Sicherung, falls die erste Blockgruppe Fehler enthält oder nicht mehr lesbar ist.

3.6.5 Verzeichnisse

Verzeichnisse sind spezielle Dateien, die verwendet werden, um eine hierarchische Struktur in das Dateisystem zu bringen; beim Benutzer wird der Eindruck erzeugt, dass Verzeichnisse Dateien und weitere Verzeichnisse *enthalten*. Als Datenbereich, der bei einer Datei von ihrem Inhalt gebildet wird, besitzt ein Verzeichnis eine Liste von Verzeichniseinträgen (*directory entries*), wobei jeder Eintrag folgende Informationen enthält:

Inode Dieser Wert ist die Nummer der Inode, auf die der Eintrag verweist. Diese Zahl wird in die Nummer der Blockgruppe und den Index innerhalb der Inodetabelle der Blockgruppe umgerechnet.

Länge des Eintrags die Länge des Eintrags (kann u.U. größer sein als der tatsächlich belegte Platz, s. unten)

Länge des Namens die Länge des Datei- bzw. Verzeichnisnamens, der der Inode zugeordnet ist

Name der Name des Verzeichniseintrags (Datei oder Verzeichnis)

Die ersten zwei Einträge sind für jedes Verzeichnis immer „ . “ (aktuelles Verzeichnis) und „ .. “ (übergeordnetes Verzeichnis). Die Verzeichniseinträge werden so verwaltet, dass beim Löschen eines Eintrags der vorhergehende entsprechend vergrößert wird. Beim Hinzufügen eines Eintrags wird zuerst ein Eintrag gesucht, der groß genug ist, um sich selbst und den neuen aufzunehmen, erst dann wird die Liste der Einträge erweitert. Es werden also zwei Längen unterschieden: die tatsächliche Länge des Eintrags (entspricht der Länge des Namens + 8) und die Länge, die der Eintrag in dem Block einnimmt (entspricht beim einzigen Eintrag im Block der Größe des Blocks). Die Länge der Einträge wird so gewählt, dass die Liste immer an einer Blockgrenze endet. Die „Größe“ eines Verzeichnisses wir durch die Länge dieser Liste bestimmt. Jeder Eintrag muss eine Länge aufweisen, die durch 4 teilbar ist, um das Layout kompatibel zum Dateisystem *Ext2* zu halten.

An dieser Stelle erst geschieht die Zuordnung Name — Inode; im Dateisystem existieren u.U. viele Dateien, die erst durch die Verzeichnisse in eine (hierarchische) Struktur gebracht werden und durch Namen adressierbar sind (innerhalb des Dateisystems werden Dateien lediglich durch die Nummer ihrer Inode identifiziert). Die Zuordnung eines Namens dient dazu, die Schnittstelle zum Benutzer einfacher und anschaulicher zu gestalten, für das Layout auf der Festplatte ist sie ohne Bedeutung. Die Kette vom Wurzelverzeichnis bis zu dem Verzeichnis, das eine bestimmte Datei enthält, zusammen mit dem Namen, der der Datei in diesem Verzeichnis zugeordnet ist, wird als Pfad(name) bezeichnet.

3.6.6 Überblick über die Abhängigkeiten

Abbildung 3.11 zeigt die Klassen des Dateisystems und ihre Beziehungen zueinander. In dieser Übersicht werden die Klassen des Treibers und die `Exception`-Klassen nicht aufgeführt, da sie nicht weiter zum Verständnis beitragen. Neben `GroupData` (Informationen zur Blockgruppe) besitzen die Klassen `Inode`, `Super`, `SymlinkInode` und `DirEntry` jeweils eine `Data`-Klasse, auf deren Darstellung der Einfachheit halber ebenfalls verzichtet wurde.

Die beiden Caches erzeugen und verwalten jeweils die Inodes und BufferHeads. Eine neue Inode bzw. ein neuer BufferHead wird allerdings über `Super.newInode` bzw. `Super.newBlock` angefordert. Die Klasse `Super` greift dazu auf `Bitmap`-Objekte zurück, um die gewünschte Inode bzw. den optimalen Block zu finden und zu reservieren. Ein `BufferHead`-Objekt wird von `Super` nicht erzeugt, sondern vom BufferCache über die Methode `getblk` angefordert. Im Gegensatz dazu werden `Inode`-Objekte in der Super-Klasse erzeugt und dem InodeCache über die Methode `addInode` hinzugefügt.

3.7 Kompatibilität zum Dateisystem *Ext2* von Linux

Das Festplatten-Layout der Java-Implementierung ist zu der des Dateisystems *Ext2* kompatibel, d.h. die Felder im Superblock haben die gleiche Position und Bedeutung. Allerdings werden einige Felder des Superblocks und der Inode-Strukturen im *JavaFS* nicht verwendet (z.B. wird die UUID-Kennung im Superblock einfach auf 0 gesetzt). *JavaFS* unterstützt nur die Inodetypen *Datei*, *Verzeichnis* und *symbolischer Link*, *special files* wie Block- oder Character Devices werden ignoriert.

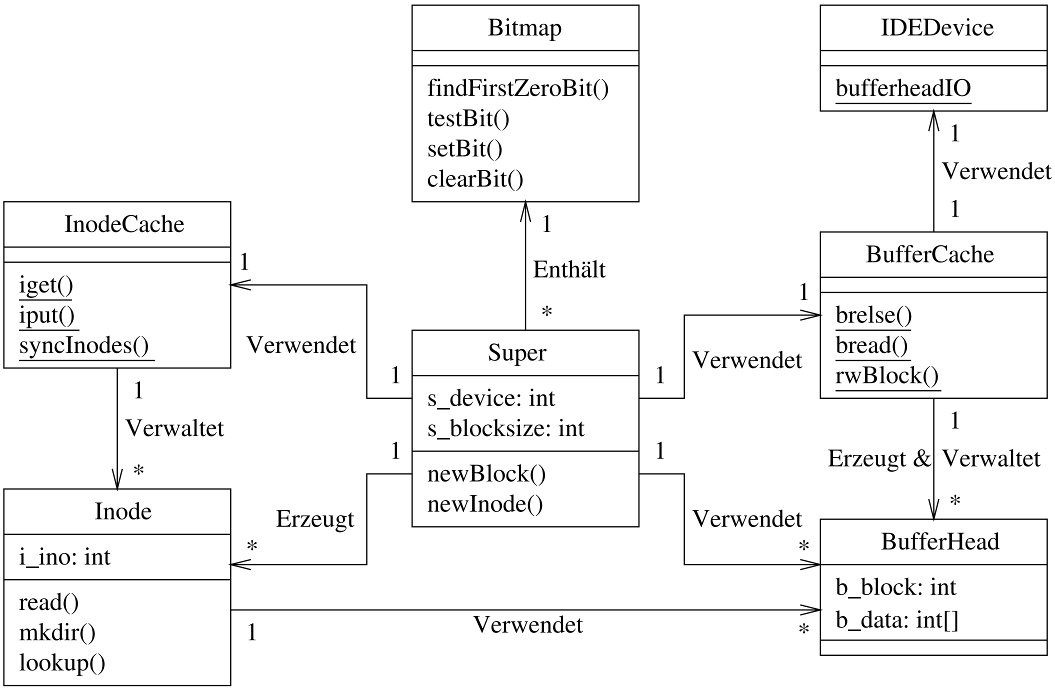


Abbildung 3.11: Überblick über die Abhängigkeiten des Dateisystems *JavaFS*

Ein Problem tritt auf, wenn der Wertebereich einer `byte`, `short` oder `int`-Variable in Java erreicht wird. Da diese Datentypen vorzeichenbehaftet sind, kann es sein, dass bei Berechnungen negative Zahlen auftreten. Die unter Linux verwendeten `unsigned` Datentypen (z.B. `unsigned int`), die vorzeichenlose 8, 16 oder 32 Bit-Zahlen repräsentieren, existieren in Java nicht (wenn man vom Typ `char` absieht, der aber ausschließlich für Unicode-Zeichen verwendet wird). Von diesem Problem betroffen ist als einziges das Feld in der Inodestruktur, das die Größe der Inode angibt. Da aber der Wertebereich einer Java-`int`-Variable über 2 Milliarden reicht, treten Schwierigkeiten erst bei extrem großen Dateien auf. Die Zähler der (freien) Blöcke und Inodes im Superblock und der verwendeten Blöcke in der Inodestruktur sind davon nicht betroffen, da Blöcke im IDE-Standard mit nur 28 Bit adressiert werden (s. Kapitel 2.5).

3.8 Zusammenfassung

Es wurden die Eigenschaften und Konzepte eines Dateisystems, wie sie aus der Entwicklung der UNIX-Systeme hervorgegangen sind, aufgezeigt. Anhand verschiedener UNIX-Dateisysteme wurde ein kurzer Überblick über zugrunde liegende Designmuster gegeben. Schließlich wurde ausführlich das im Zuge dieser Arbeit entwickelte Dateisystem erläutert, wobei auf Gemeinsamkeiten und Unterschiede zum *Ext2* von Linux eingegangen wurde.

Kapitel 4

Die Schnittstelle zwischen Java und anderen Sprachen

Um dem Java-Betriebssystem *metaXaOS* die Ansteuerung der Hardware zu ermöglichen, wurde auf das *Java Native Interface* zurückgegriffen, das eine Schnittstelle zwischen nicht-Java-Programmcode und Java definiert. In den folgenden Kapiteln wird kurz auf die Programmierung dieser Schnittstelle eingegangen.

4.1 Java Native Interface

Direkte Zugriffe auf die Hardware sind in Java nicht möglich, so dass diese Funktionalität in C bzw. Assembler implementiert werden muss (wie auch bei traditionellen Betriebssystemen). Die Verwendung maschinennaher Programmiersprachen lässt sich allerdings (neben der virtuellen Maschine) auf eine nur kleine Menge von Situationen beschränken, wie der Zugriff auf Register bzw. Speicheradressen, die Prozessumschaltung und die Speicherverwaltung.

Die Schnittstelle zwischen einer C-Funktion und einer Java-Klasse stellt das *Java Native Interface* (JNI) von Javasoft dar. JNI ist seit der Version 1.1 Bestandteil von Java und bietet die Möglichkeit, innerhalb von Java nicht-Java-Programmcode auszuführen (*native*-Funktionen). Programme, die in Assembler oder C (oder einer anderen Sprache) verfasst sind, können unter Verwendung des JNI

- Java Objekte erzeugen und Informationen über Java Objekte erhalten (auch Arrays)
- Java Methoden aufrufen
- Ausnahmen erzeugen und verarbeiten
- Klassen laden und Informationen über Klassen erhalten
- Typüberprüfung zur Laufzeit durchführen

4.2 Der Aufruf einer *native*-Funktion

Soll in einem Java-Objekt eine *native*-Funktion angesprochen werden, muss eine neue Java-Methode mit dem Schlüsselwort `native` definiert werden, z.B. existiert für den Zugriff auf die Register eines Gerätes die Funktion

```
public class Ports {
    ...
    public static native byte inb(int addr);
    ...
}
```

Eine so definierte Methode kann wie jede andere Java-Methode verwendet werden (insofern eine entsprechende C-Funktion gefunden werden kann). Ob es sich um eine Implementierung in Java oder einer anderen Programmiersprache handelt, ist von außen nicht sichtbar.

4.3 Die Implementierung einer *native*-Funktion

Eine *native*-Funktion, die als Java-Methode aufgerufen werden soll, muss folgender Namenskonvention gehorchen: Der Funktionsname beginnt immer mit „`Java`“, gefolgt von den Namen der Pakete, dem Klassennamen (der Klasse, die die Definition der *native*-Methode enthält), und schließlich dem Namen der Java-Methode. Als Trennzeichen dient der Unterstrich. Falls die Funktion überladen ist, wird die Signatur der Funktion an den Methodennamen angehängt. Um bei unserem Beispiel zu bleiben:

```
jint Java_metaxa_os_Ports_inb(JNIEnv *env, jobject this,
    jshort port) { return (jbyte)inb(port); }
```

Dieser Vorgang lässt sich auch automatisieren, indem das Programm `javah` auf die `.class`-Datei angewandt wird: `javah -jni Ports`. `javah` liest die Klassendatei und erzeugt für jede *native*-Methode einen Funktionsprototyp in einer C oder C++-Headerdatei.

Die ersten beiden Argumente der Funktion sind bei allen *native*-Funktionen vorhanden (die Java Methode `inb` besitzt ja nur ein Argument: `jshort port`). Das erste ist ein Zeiger auf die JNIEnv-Struktur, über die auf die JNI-Funktionen und damit auf die virtuelle Maschine zugegriffen werden kann. Das zweite Argument ist für nicht-statische Methoden eine Referenz auf das Objekt, das die *native*-Methode aufgerufen hat, also vergleichbar mit dem `this`-Zeiger in Java. Für statische Methoden (wie `inb`) ist es eine Referenz auf das `Class`-Objekt, in dem die Methode definiert ist.

4.3.1 Der Zugriff auf JNI-Funktionen über das JNIEnv-Argument

Um innerhalb einer *native*-C/C++-Routine auf Java Methoden und Felder zugreifen zu können, existiert eine Vielzahl von JNI-Funktionen, die über den Zeiger `JNIEnv *env` erreicht werden. Die Menge der Funktionen (insgesamt über 200) lässt sich in mehrere Gruppen aufteilen:

- Abfrage von Versionsinformationen
- Klassen- und Objekt-Operationen
- Verwaltung globaler und lokaler Referenzen auf Java-Objekte
- Zugriff auf Felder (auch statische)
- Ausführen von Methoden (auch statische)
- String- und Array-Operationen
- Erzeugen und Verarbeiten von Ausnahmen

Das Argument `env` zeigt auf die Struktur `struct JNINativeInterface`, die Zeiger auf diese Funktionen enthält. Diese Struktur stellt demnach nur ein Interface zu den eigentlichen Funktionen dar, die Implementierung kann jede JVM individuell gestalten.

Um ein Feld oder eine Methode aufzurufen, wird der entsprechende `Identifier` benötigt. Es existieren JNI-Funktionen, die unter Verwendung des Klassen-Objekts, des Namens und der Signatur[1] des Feldes bzw. der Methode den `Identifier` ermitteln. Durch diese Vorgehensweise wird erreicht, dass das interne Layout eines Java-Objektes nicht bekannt sein muss; jede JVM kann ihr eigenes Format verwenden. Das benötigte Klassenobjekt muss ebenfalls von der virtuellen Maschine angefordert werden.

Als Beispiel soll die Klassenmethode `Thread.sleep` dienen, die einen Prozess für kurze Zeit schlafen legt:

```
jclass thread_class = (*env)->FindClass(env, "java/lang/Thread");
...
jmethodID mid =
  (*env)->GetStaticMethodID(env, thread_class, "sleep", "(I)V");
...
(*env)->CallStaticVoidMethod(env, thread_class, mid, duration);
```

Vor dem Aufruf einer `native`-Funktion werden lokale Referenzen der verwendeten Objekte erzeugt und danach wieder gelöscht, damit der Garbage Collector während der Abarbeitung der Funktion keine beteiligten Objekte einsammelt (er wird aber nicht unterbrochen).

[1]Um eine Methode in Java zu kennzeichnen, wird folgende Signatur verwendet:

(Typen der Argumente) Typ des Returnwerts

Falls kein Wert zurückgegeben wird, muss anstelle eines Returnwerts `V` (für `void`) stehen. Für die primitiven Typen existieren einbuchstabige Signaturen: `Z` für `boolean`, `B` für `byte`, `C` für `char`, `S` für `short`, `I` für `int`, `J` für `long`, `F` für `float` und `D` für `double`. Eine Klasse wird durch „LKlasse`;`“ gekennzeichnet, wobei der komplette Klassenpfad mit angegeben werden muss (z.B. `Ljava/lang/String;`). Arrays werden durch `[` eingeleitet; ein `int`-Array wird also als „`[int`“ kodiert.

4.3.2 Ausnahmen (Exceptions)

Es existieren einige JNI-Funktionen, die die Realisierung und Verarbeitung von Ausnahmen ermöglichen. Mittels `ThrowNew()` bzw. `Throw()` lassen sich Ausnahmen erzeugen und „werfen“ bzw. eine existierende Ausnahme erneut aktivieren. `ExceptionOccurred()` überprüft, ob eine Ausnahme besteht. Diese lässt sich durch `ExceptionClear()` löschen oder kann mittels `ExceptionDescribe()` ausgegeben werden (mit Stacktrace).

Der Programmierer muss (über `ExceptionOccurred()`) selbst dafür sorgen, dass eine Ausnahme erkannt wird, es existiert kein Äquivalent zum `try`-Ausdruck.

Das oben beschriebene Programmfragment soll dazu etwas erweitert werden:

```
jclass thread_class = (*env)->FindClass(env, "java/lang/Thread");
if ((*env)->ExceptionOccurred(env) != NULL) {
    (*env)->ExceptionDescribe();
    return;
}
```

4.4 Zusammenfassung

Anhand konkreter Beispiele wurde in die Programmierung des *Java Native Interface* als Schnittstelle zwischen Java und anderen Programmiersprachen eingeführt. Das JNI erweitert die Einsatzmöglichkeiten von Java; ohne eine derartige standardisierte Schnittstelle würde sich die Entwicklung eines Betriebssystems in Java wesentlich schwieriger gestalten.

Kapitel 5

Abschließende Betrachtungen

Es wird kurz auf den Einsatz von Linux als Testumgebung eingegangen und die verwendeten Programme erläutert. Schließlich folgt ein kurzer Ausblick auf die Zukunft des Betriebssystems und die Erweiterungsmöglichkeiten.

5.1 Linux als Test- und Vergleichssystem

Linux ist eine frei kopierbare Unix-Variante für PCs, die im Heimcomputerbereich immer mehr an Bedeutung gewinnt. Das Betriebssystem wird ständig weiterentwickelt und kann inzwischen, was Funktionsumfang, Sicherheit und Geräteunterstützung angeht, mit modernen kommerziellen Produkten mithalten. Das Standard-Dateisystem von Linux ist *Ext2* (s. Kapitel 3.2.3). Verwendet wurde ein Minimal-Linux-System auf Diskette mit den wichtigsten Programmen.

Zum Testen der Dateisystem-Implementierung (sowohl auf Programmfehler wie auch auf Kompatibilität) wurden unter anderem die „Ext2 Filesystem Utilities" von Theodore Ts'o in der Version 1.18 verwendet ([Ts99]). Es folgt eine kurze Beschreibung der verwendeten Programme aus diesem Paket:

dumpe2fs liefert Informationen zum Dateisystem, die im Superblock und in den Gruppen enthalten sind. Der Aufruf erfolgt mit Angabe der Partition, die das Dateisystem enthält, z.B. `dumpe2fs /dev/hda7`.

tune2fs bietet die Möglichkeit, Einstellungen des Dateisystems, die im Superblock verwaltet werden, zu ändern.

e2fsck überprüft ein *Ext2*-Dateisystem. Mit diesem Werkzeug war es möglich, die Kompatibilität der Java-Implementierung zum Dateisystem *Ext2* zu testen. Der Aufruf erfolgt mit Angabe der Partition, die das Dateisystem enthält, z.B. `e2fsck -f /dev/hda7`. Die Option `-f` bewirkt, dass ein als „clean" markiertes Dateisystem trotzdem überprüft wird.

mke2fs erzeugt ein neues *Ext2*-Dateisystem. Auch hier muss beim Aufruf die Partition angegeben werden, auf der das Dateisystem installiert werden soll. Es existiert eine Vielzahl von Optionen für die Konfiguration des Dateisystems (z.B. `-b block-size` zum Festlegen der Blockgröße).

Zum Testen wurde ein Dateisystem mit *JavaFS* angelegt und unter Linux angemeldet (`mount -t ext2 ...`). Sowohl unter *metaXaOS* als auch Linux wurden Dateien und Verzeichnisse erzeugt und unter dem jeweils anderen Betriebssystem der Zugriff darauf getestet.

5.2 Zusammenfassung und Ausblick

Sicherlich müssen im Vergleich zu traditionellen Betriebssystemen erhebliche Performanceeinbußen in Kauf genommen werden. Allerdings bietet der Einsatz eines Just-in-Time Compilers und die Ausarbeitung der Speicherverwaltung noch einigen Spielraum für Geschwindigkeitsoptimierungen. Natürlich sind Fehler in der Implementierung nicht auszuschließen, da trotz umfangreicher Tests nicht alle denkbaren Szenarien durchgespielt werden konnten. Nichtsdestotrotz ist das Dateisystem komplett und voll funktionsfähig. Offene Schnittstellen zum Festplattentreiber und zum virtuellen Dateisystem gewährleisten die Erweiterbarkeit des Systems. Die Integration in das Java-API (s. Anhang) ermöglicht den Einsatz von Java-Software, die nicht speziell für dieses System entwickelt wurde.

Für die Zukunft sind weitere Datenträgertypen (SCSI-, ATAPI-Geräte), sowie verschiedene Dateisystem-Implementationen (z.B. ein Log-orientiertes Dateisystem) denkbar. Sobald ein Just-in-Time Compiler zur Verfügung steht, kann die Performance in Relation zu einem in C bzw. Assembler verfassten System gemessen werden. Dazu bietet sich wiederum Linux mit der *Ext2*-Implementierung als Vergleichssystem an.

Anhang A

Die Integration in das Java-API

Um auch über das Java-API (*A*pplication *P*rogrammer *I*nterface) die Funktionalität des Dateisystems zu erhalten, wurden die Klassen im Paket `java.io` nachprogrammiert. Zugriffe auf das Dateisystem erfolgen über die Klasse `FS`. Dabei wurde Wert darauf gelegt, die Methoden des JDK (*J*ava *D*evelopment *K*it) in der Version 1.1 vollständig nachzubilden, um Kompatibilität zu bereits existierender Java-Software zu erreichen. Betroffen sind die Klassen `File`, `FileDescriptor`, `FileInputStream`, `FileOutputstream` und `RandomAccessFile`; alle weiteren Klassen im Paket `java.io` setzen auf diesen Klassen auf und bedürfen keiner Anpassung.

Im Folgenden werden die neuen Klassen kurz beschrieben; es wird auf Besonderheiten bei der Implementierung eingegangen, indem interessante Teile des Quelltextes vorgestellt werden. Die `java.io`-Klassen dienen gewissermaßen als Beispiel für eine Implementation unter Verwendung der Dateisystem-Klassen.

A.1 Klasse `java.io.File`

Diese Klasse erfüllt im Wesentlichen zwei Aufgaben: sie dient der Verwaltung eines Pfadnamens und bietet grundlegende Funktionen wie: Ermitteln, ob eine Datei existiert, eine Datei löschen oder ein Verzeichnis anlegen. Die Abstraktion der *Datei* wird in Java also durch einen Pfadnamen realisiert. Die Klasse bietet keinerlei Funktionalität für die Ein-/Ausgabe von Daten, dazu dienen `FileInputStream`, `FileOutputStream` und `RandomAccessFile`.

Die Implementierung ist denkbar einfach: Die Methoden `mkdir`, `renameTo` und `delete` rufen die entsprechenden Methoden der Klasse `FS` auf. Bei `isFile`, `isDirectory`, `canRead`, `canWrite`, `lastModified`, `length` und `list` (den Inhalt eines Verzeichnisses ausgeben) wird zuerst über `FS.lookup` die Inode der Datei ermittelt und an sie die meist gleichlautenden Nachrichten geschickt; `exists` ruft ebenfalls `FS.lookup` auf. Die Implementierung der Methoden zum Zugriff auf den Pfadnamen (`getParent`, `getAbsolutePath`, `getName` usw.) entspricht weitestgehend der Implementierung der `FS`-Methoden und wurde im folgenden Quelltextfragment weggelassen.

```
package java.io;
```

```
import java.io.IOException;
import java.io.FilenameFilter;
import java.util.Vector;
import metaxa.os.fs.*;

public final class File implements Serializable {
    // name = pathName + separator + fileName
    private String name;
    private String pathName;
    private String fileName;

    public static final String separator = "/";
    public static final char separatorChar = '/';
    public static final String pathSeparator = "/";
    public static final char pathSeparatorChar = '/';

    public File(String path) throws NullPointerException {
        if (path == null)
            throw new NullPointerException();

        int n = path.lastIndexOf(separatorChar);
        if (n < 0) {
            fileName = path;
          pathName = FS.getCwdPath();
        } else {
            fileName = path.substring(n + 1);
            if (n == 0) {
                pathName = "/";
            } else {
                pathName = path.substring(0, n);
            }
        }
        name = path;
        inode = null;
    }

    // ... weitere Konstruktoren

    public boolean mkdir() {
        try {
            FS.mkdir(fileName);
        } catch (FSException e) {
            return false;
```

```
        }
        return true;
    }

    public String[] list() {
        try {
            Inode inode = FS.lookup(name);
            if (inode.isDirectory() == false) {
                inode.releaseInode();
                return new String[0]; // null
            }
            Vector dirList = inode.readdirNames();
            dirList.removeElement(".");
            dirList.removeElement("..");
            String[] sl = new String[dirList.size()];
            dirList.copyInto(sl);
            inode.releaseInode();
            return sl;
        } catch (FSException e) {
            return new String[0]; // null
        }
    }

    public boolean delete() {
        Inode inode = null;
        try {
            Inode inode = FS.lookup(name);
            if (inode.isDirectory())
                FS.rmdir(name);
            else
                FS.unlink(name);
        } catch (FSException e) {
            return false;
        } finally {
            if (inode != null)
                inode.releaseInode();
        }
        return true;
    }

    // ... weitere Methoden
}
```

A.2 Klasse `java.io.FileDescriptor`

Die Klasse `FileDescriptor` dient dazu, Dateien eindeutig zu identifizieren, wird aber normalerweise nicht verwendet (intern werden Pfadnamen zur eindeutigen Kennzeichnung gebraucht). Außerdem werden drei spezielle Dateibezeichner statisch erzeugt, die den Zugriff auf die Console ermöglichen: `FileDescriptor.in` ermöglicht Tastatureingaben, `FileDescriptor.out` und `FileDescriptor.err` dienen zur Textausgabe auf der Console. Wird ein `FileOutputStream`-Objekt mit den Dateibezeichner `out` oder `err` initialisiert, führen `write`-Befehle dazu, dass die übergebenen Daten (normalerweise Text) mittels `System.out.write` auf der Console ausgegeben werden.

A.3 Klasse `java.io.FileInputStream`

Diese Klasse bietet Funktionen zum sequentiellen Auslesen einer Datei.

Wird ein `FileInputStream`-Objekt mit dem Dateibezeichner der Standardeingabe initialisiert (`FileDescriptor.in`), so wird bei einer Leseoperation auf eine Tastatureingabe des Benutzers gewartet. Beim Zugriff auf eine Datei wird die Methode `read()` der entsprechenden Inode ausgeführt. Es wird ein Zeiger auf die aktuelle Position innerhalb der Datei verwaltet, der bei jeder Leseoperation um die Anzahl gelesener Bytes weitergesetzt wird. Ein Aufruf der Methode `skip(long)` setzt diesen Zeiger weiter, ohne Daten zu lesen. Mittels `mark()` lässt sich die aktuelle Position markieren und später über `reset()` wieder anspringen.

Die Implementierung entspricht weitgehend der von `FileOutputStream` (hier wird allerdings die Datei nicht angelegt, falls sie noch nicht existiert), weshalb hier auf die Darstellung des Quelltextes vollständig verzichtet wird.

A.4 Klasse `java.io.FileOutputStream`

Diese Klasse bietet Funktionen zum sequentiellen Schreiben in eine Datei.

Ein `FileOutputStream`-Objekt, dass mit dem Dateibezeichner der Standardausgabe oder der Fehlerausgabe (`FileDescriptor.out` bzw. `FileDescriptor.err`) initialisiert wurde, setzt Schreiboperationen in Ausgaben auf die Console um. Beim Schreiben in eine Datei wird die Methode `write()` der zur Datei gehörenden Inode ausgeführt; auch hier wird ein Zeiger auf die aktuelle Position innerhalb der Datei mitgeführt. Die Methode `skip(long)` setzt diesen Zeiger weiter, ohne Daten zu schreiben. Der `mark()/reset()`-Mechanismus der Klasse `FileOutputStream` ist hier ebenfalls implementiert. Bei der Initialisierung eines Objekts der `FileOutputStream`-Klasse wird die gewünschte Datei, falls sie noch nicht existiert, neu angelegt.

```
package java.io;

import metaxa.os.fs.*;
```

```
public class FileOutputStream extends OutputStream {
    private FileDescriptor fd;
    private String name;
    private long filePosition;
    private Inode inode;

    // ... Methoden finalize(), close() und getFD()

    public void write(byte[] b, int off, int len)
        throws IOException {
        if (fd.equals(FileDescriptor.out)) {
            System.out.write(b, off, len);
            return;
        } else if (fd.equals(FileDescriptor.err)) {
            System.err.write(b, off, len);
            System.err.flush();
            return;
        } else if (fd.equals(FileDescriptor.in))
            throw new IOException();

        if (inode == null)
            throw new IOException();

        try {
            retval = inode.write(b, filePosition+off, len);
            filePosition += retval;
        } catch (FSException e) {
            throw new IOException();
        }
    }

    public void write(byte[] b) throws IOException {
        write(b, 0, b.length);
    }

    public void write(int b) throws IOException {
        byte[] buf = new byte[1];
        buf[0] = (byte) (b & 0xFF);
        write(buf);
    }

    public FileOutputStream(String name, boolean append)
        throws IOException {
```

```
        this.name = name;
        this.filePosition = 0;

        if (name != null) {
            try {
                inode = FS.lookup(name);
                if (append)
                    filePosition = inode.getLength();
            } catch (InodeNotFoundException e) {
                try {
                    FS.create(name);
                    inode = FS.lookup(name);
                } catch (FSException e) {
                    inode = null;
                }
            } catch (FSException e) {
                inode = null;
            }
        }

        this.fd = new FileDescriptor(inode);
    }

    // ... weitere Konstruktoren
}
```

A.5 Klasse `java.io.RandomAccessFile`

Diese Klasse bietet wahlfreien Zugriff auf den Inhalt einer Datei. Beim Erzeugen eines Objekts dieser Klasse muss der Zugriffsmodus angegeben werden: entweder „`r`“ für Lesen oder „`rw`“ für Lesen und Schreiben. Es werden die Interfaces `DataInput` und `DataOutput` implementiert, über die die primitiven Datentypen auf einen Stream geschrieben bzw. von einem Stream gelesen werden können. Um diese Funktionalität zu erreichen, wird ein `FileInputStream`- und ein `FileOutputStream`-Objekt erzeugt, die einem `DataInputStream`- bzw. `DataOutputStream`-Objekt übergeben werden. Der wahlfreie Zugriff wird durch die Methode `seek(long)` ermöglicht, der die gewünschte Position innerhalb der Datei übergeben werden muss. Diese Methode ist letztendlich mittels der `reset()`-Methoden aus `FileInputStream` und `FileOutputStream` implementiert.

```
package java.io;

import metaxa.os.fs.*;
```

```
public class RandomAccessFile implements DataInput, DataOutput {
    private FileDescriptor fd = new FileDescriptor();
    private long filePosition;
    private Inode inode;

    private DataInputStream din;
    private DataOutputStream dout;

    public final FileDescriptor getFD() throws IOException {
        return fd;
    }

    public synchronized void close() throws IOException {
        if (inode == null)
            throw new IOException();
        dout.close();
        din.close();
        fd = null;
    }

    public long getFilePointer() throws IOException {
        return filePosition;
    }

    public void seek(long pos) throws IOException {
        filePosition = pos;
        din.reset();
        dout.reset();
        din.skip(pos);
        dout.skip(pos);
    }

    public long length() throws IOException {
        if (inode == null)
            throw new IOException();
        return inode.length();
    }

    public synchronized int skipBytes(int n) throws IOException {
        if (inode == null)
            throw new IOException();

        if (filePosition + n > inode.length())
```

```
            throw new EOFException();

        filePosition += n;
        din.skip(n);
        dout.skip(n);
        return n;
    }

    public final int read() throws IOException {
        return din.read();
    }

    // ... weitere read-Methoden (s. DataInputStream)

    public final void readFully(byte b[]) throws IOException {
        readFully(b, 0, b.length);
    }

    public final void readFully(byte b[], int off, int len)
        throws IOException {
        int retval;

        if (inode == null)
            throw new IOException();

        if (filePosition + len > inode.length())
            throw new EOFException();

        try {
            retval = inode.read(b, filePosition+off, len);
            filePosition += retval;
        } catch (FSException e) {
            throw new IOException();
        }

        return retval;
    }

    // ... weitere write-Methoden (s. DataOutputStream)

    public RandomAccessFile(String path, String mode)
        throws IOException {
        if (!mode.equals("r") && !mode.equals("rw"))
```

```
            throw new IllegalArgumentException();

        try {
            try {
                inode = FS.lookup(path);
            } catch(InodeNotFoundException e) {
                if (mode.equals("r")) {
                    throw new IOException("could not create file
                                           with mode \"r\"");
                }
                Inode pi = FS.lookup(FS.getPathName(path));
                inode = pi.create(FS.getFileName(path));
            }

            din = new DataInputStream(new FileInputStream(path));

            if (mode.equals("rw"))
                dout = new DataOutputStream(new
                           FileOutputStream(path));
            else
                dout = null;
        } catch(Exception e) {
            e.printStackTrace();
            throw new IOException(e.getMessage());
        }
    }

    public RandomAccessFile(File file, String mode)
        throws IOException {
        this(file.getPath(), mode);
    }
}
```

Anhang B

Dokumentation der wichtigsten Klassen

Die folgenden Seiten enthalten die Dokumentationen zu den Klassen des Festplattentreibers und des virtuellen Dateisystems, die entweder die Funktion einer Schnittstelle haben oder wichtige Methoden für die Integration einer konkreten Dateisystem-Implementation aufweisen. Die Dokumentationen wurden mit *javadoc* und dem Doclet *TexDoclet* der Firma *C2 Technologies* im LaTeX2e-Format generiert.

Folgende Aufstellung gibt einen Überblick über die dokumentierten Klassen und ihre Funktion:

- **Schnittstelle zwischen IDE-Festplattentreiber und Dateisystem**

  Die Klasse `devices.ide.IDEDevice` bietet Funktionen für die Datenübertragung zwischen Hauptspeicher und Festplattencontroller und gibt Auskunft über vorhandene Partitionen und Festplatten sowie deren Kapazität.

- **Schnittstelle zwischen virtuellem Dateisystem und konkreter Implementierung**

  Über die Klasse `fs.Inode` und das Interface `fs.FileSystem` kann auf die Methoden einer Dateisystem-Implementierung wie das *JavaFS* zugegriffen werden; jedes Dateisystem stellt eine eigene `Inode`-Klasse bereit, die von `fs.Inode` abgeleitet wird, und implementiert `fs.FileSystem`.

  Die Klassen `fs.BufferCache` und `fs.InodeCache` dienen der Verwaltung und Zwischenspeicherung der wichtigsten Strukturen des Dateisystems (`BufferHead` und `Inode`) und sollten von einer Dateisystem-Implementierung verwendet werden.

- **Schnittstelle zwischen Dateisystem und Applikationen**

  Die gesamte Funktionalität des Dateisystems ist über die Klasse `fs.FS` erreichbar.

Klasse IDEDevice

Diese Klasse bietet Funktionen für die Datenübertragung zwischen Hauptspeicher und Festplattencontroller und gibt Auskunft über vorhandene Partitionen und Festplatten und deren Kapazität. Sie stellt einen Monitor für den Zugriff auf die IDE-Controller dar.

Declaration

public class IDEDevice
extends java.lang.Object

Fields

- public static final int MAX_DRIVES
  - max. Anzahl Laufwerke pro Controller
- public static boolean dmaSupported
  - `true`, falls mit dem Motherboard-Chipsatz Busmaster-DMA möglich ist
- public static boolean ultraDmaSupported
  - `true`, falls vom Motherboard-Chipsatz Ultra-DMA unterstützt wird

Methods

- *bufferheadIO*
  `public void` **bufferheadIO** `( metaxa.os.fs.BufferHead` **bh**`, int` **sector**`, boolean` **read**`, boolean` **synchronous** `)`
  - **Usage**
    * Bewirkt das Lesen bzw. Schreiben von Sektoren einer Partition. Diese Methode setzt die Sektornummern innerhalb einer Partition auf Sektornummern innerhalb der Festplatte um und ruft `Drive.bufferheadIO` für den Datentransfer auf. Es wird außerdem überprüft, ob die Sektornummer im gültigen Bereich liegt.
  - **Parameters**
    * `bh` - der `BufferHead`, der die Blocknummer, die Partitionskennung, den Puffer und die Anzahl zu übertragender Byte für die I/O-Operation enthält
    * `sector` - der Sektor, ab dem gelesen werden soll (relativ zum Partitionsanfang)
    * `read` - Falls `true`, wird von der Partition gelesen, ansonsten wird geschrieben.
    * `synchronous` - Falls `true`, wird auf das Ende der I/O-Operation gewartet (synchrones Lesen bzw. Schreiben).

- *delay*
  `public static void` **delay** `( int` **ticks** `)`
  - **Usage**
    - Wartet die angegebene Anzahl an Ticks (10 ms).
  - **Parameters**
    - `ticks` - Dauer in 10 ms-Einheiten

- *deviceToName*
  `public static String` **deviceToName** `( int` **device** `)`
  - **Usage**
    - Gibt den Namen der Partition mit der angegebenen Kennung zurück.
  - **Parameters**
    - `device` - die Kennung der Partition
  - **Returns** - der Namen der Partition mit dieser Kennung („hda“, „hdb“, „hdc“ oder „hdd“ + Nummer der Partition)

- *getCapacity*
  `public int` **getCapacity** `( int` **device** `)`
  - **Usage**
    - Liefert die Kapazität (die Anzahl der Sektoren) der angegebenen Partition zurück (Partition 0 umfasst die gesamte Festplatte).
  - **Parameters**
    - `device` - die Kennung der Partition
  - **Returns** - die Anzahl der Sektoren der Partition (-1 falls die angegebene Partition nicht existiert)

- *instance*
  `public static IDEDevice` **instance** `( )`
  - **Usage**
    - Liefert die Instanz des Treibers zurück (es existiert nur ein IDE-Festplattentreiber-Objekt).
  - **Returns** - das Festplattentreiber-Objekt

- *isPresent*
  `public boolean` **isPresent** `( int` **device** `)`
  - **Usage**
    - Ermittelt das Vorhandensein der angegebenen Partition.
  - **Parameters**

    * `device` - die Kennung der Partition
  - **Returns** - `true`, falls die Partition vorhanden ist, ansonsten `false`

- *nameToDevice*
  `public static int` **nameToDevice**`( java.lang.String` **name** `)`
  - **Usage**
    * Gibt die Kennung der Partition mit dem angegebenen Namen zurück.
  - **Parameters**
    * `device` - der Name der Partition (muss entweder „hda“, „hdb“, „hdc“ oder „hdd“ sein, gefolgt von einer Nummer)
  - **Returns** - die Kennung der Partition mit diesem Namen

Klasse **Inode**

Diese Klasse stellt mit überwiegend als `abstract` deklarierten Methoden die Basisklasse für die Inode-Klassen der Dateisysteme dar. Sie enthält außerdem die notwendigen Felder und Methoden für die Verwaltung im InodeCache. Die abstrakten Methoden sind: `i_nlinks`, `deleteInode`, `writeInode`, `isFile`, `isDirectory`, `isWritable`, `isReadable`, `readdirNames`, `getInode`, `mkdir`, `rmdir`, `create`, `unlink`, `read`, `write`, `available` und `getLength`. Die Schnittstelle zwischen virtuellem Dateisystem und den konkreten Dateisystem-Implementierungen wird durch diese Klasse realisiert.

Declaration

public abstract class Inode
extends java.lang.Object

Fields

- public InodeHashKey i_hashkey
  - verwendet vom InodeCache
- public Vector i_list
  - verwendet vom InodeCache
- public int i_ino
  - verwendet vom InodeCache
- public int i_count
  - verwendet vom InodeCache
- public boolean i_ishashed
  - verwendet vom InodeCache
- public int i_device
  - verwendet vom InodeCache
- public static final int S_IFMT
  - Maske für den Typ der Inode (Datei, Verzeichnis oder symbolischer Link)
- public static final int S_IFLNK
  - Inode repräsentiert einen symbolischen Link
- public static final int S_IFREG

  - Inode repräsentiert eine reguläre Datei

- public static final int S_IFDIR
  - Inode repräsentiert ein Verzeichnis

- public static final int S_ISUID
  - Dateien mit „Set UID"-Bit erhalten die effektive User-ID bei der Ausführung, bei Verzeichnissen hat „Set UID" keinen Effekt.

- public static final int S_ISGID
  - Dateien, die in einem Verzeichnis mit gesetztem „Set GID"-Bit erzeugt werden, erhalten die Gruppen-ID dieses Verzeichnisses und ebenfalls das GID-Bit. Dateien mit „Set GID"-Bit erhalten die effektive Gruppen-ID bei der Ausführung.

- public static final int S_ISVTX
  - Bei einem Verzeichnis bedeutet das „sticky"-Bit, dass darin enthaltene Dateien nur von ihrem Besitzer, vom Besitzer des Verzeichnisses und von Root umbenannt oder gelöscht werden dürfen.

- public static final int S_IRWXU
  - die Zugriffsrechte für den Besitzer der Inode

- public static final int S_IRUSR
  - Datei: der Besitzer darf auf den Dateiinhalt lesend zugreifen; Verzeichnis: der Besitzer darf die Verzeichniseinträge auslesen.

- public static final int S_IWUSR
  - Datei: der Besitzer darf auf den Dateiinhalt schreibend zugreifen; Verzeichnis: der Besitzer darf Dateien im Verzeichnis erzeugen und löschen.

- public static final int S_IXUSR
  - Datei: der Besitzer darf die Datei ausführen; Verzeichnis: der Besitzer darf das Verzeichnis nach einem Dateinamen durchsuchen.

- public static final int S_IRWXG
  - die Zugriffsrechte für die der Inode zugeordnete Gruppe

- public static final int S_IRGRP
  - Datei: die Gruppe darf auf den Dateiinhalt lesend zugreifen; Verzeichnis: die Gruppe darf die Verzeichniseinträge auslesen.

- public static final int S_IWGRP

  - Datei: die Gruppe darf auf den Dateiinhalt schreibend zugreifen; Verzeichnis: die Gruppe Dateien im Verzeichnis erzeugen und löschen.

- public static final int S_IXGRP
  - Datei: die Gruppe darf die Datei ausführen; Verzeichnis: die Gruppe darf das Verzeichnis nach einem Dateinamen durchsuchen.
- public static final int S_IRWXO
  - die Zugriffsrechte für alle anderen
- public static final int S_IROTH
  - Datei: Alle anderen dürfen auf den Dateiinhalt lesend zugreifen; Verzeichnis: Alle anderen dürfen die Verzeichniseinträge auslesen.
- public static final int S_IWOTH
  - Datei: Alle anderen dürfen auf den Dateiinhalt schreibend zugreifen; Verzeichnis: Alle anderen dürfen Dateien im Verzeichnis erzeugen und löschen.
- public static final int S_IXOTH
  - Datei: Alle anderen dürfen die Datei ausführen; Verzeichnis: Alle anderen dürfen das Verzeichnis nach einem Dateinamen durchsuchen.
- public static final int S_IRWXUGO
  - die Zugriffsrechte der Inode (Besitzer, Gruppe, Andere)
- public static final int S_IALLUGO
  - alle Bits, die Zugriffsrechte beschreiben
- public static final int S_IRUGO
  - Datei: Lesezugriff; Verzeichnis: Auslesen der Verzeichniseinträge.
- public static final int S_IWUGO
  - Datei: Schreibzugriff; Verzeichnis: Dateien erzeugen und löschen.
- public static final int S_IXUGO
  - Datei: Datei darf ausgeführt werden; Verzeichnis: Suche nach Dateinamen.

Constructors

- *Inode*
  `protected` **Inode**`( )`
- *Inode*
  `protected` **Inode**`( metaxa.os.fs.Inode` **parent** `)`

Methods

- *available*
  `public abstract long` **available**`( )`
  - **Usage**
    * Liefert den freien Platz (in Byte) auf der Partition, die die Inode enthält, zurück.

- *create*
  `public abstract Inode` **create**`( java.lang.String` **name**`, int` **mode** `)`
  - **Usage**
    * Erzeugt einen neuen Verzeichniseintrag (eine neue Datei) mit dem angegebenen Namen innerhalb des durch diese Inode dargestellten Verzeichnisses.
  - **Parameters**
    * `name` - der Name der neuen Datei
    * `mode` - die Zugriffsrechte der neuen Datei
  - **Returns** - die Inode der neu angelegten Datei
  - **Exceptions**
    * `metaxa.os.fs.FileExistsException` - falls eine Datei oder ein Verzeichnis mit diesem Namen bereits existiert
    * `metaxa.os.fs.InodeIOException` - falls ein Fehler bei der Ein-/Ausgabe auftritt
    * `metaxa.os.fs.NoDirectoryInodeException` - falls es sich nicht um ein Verzeichnis handelt
    * `metaxa.os.fs.PermissionException` - falls die Zugriffsrechte des Verzeichnisses die Operation nicht erlauben oder das Dateisystem als nur lesbar angemeldet wurde

---

- *deleteInode*
  `public abstract void` **deleteInode**`( )`
  - **Usage**
    * Löscht die Inode (notwendig bei den Kommandos `unlink` und `rmdir`).

- *finalize*
  `public void` **finalize**`( )`
  - **Usage**
    * Stellt sicher, dass das Inodeobjekt im Cache freigegeben wird, bevor es der Garbage Collector löscht.

- *getInode*
  `public abstract Inode` **getInode**`( java.lang.String` **name** `)`
  - **Usage**
    * Liefert die Inode, die dem Verzeichniseintrag mit dem angegebenen Namen zugeordnet ist (sofern vorhanden).
  - **Parameters**
    * `name` - der Name des Verzeichniseintrags, dessen Inode ermittelt werden soll
  - **Returns** - die dem Verzeichniseintrag zugeordnete Inode
  - **Exceptions**
    * `metaxa.os.fs.InodeIOException` - falls ein Fehler bei der Ein-/Ausgabe auftritt
    * `metaxa.os.fs.InodeNotFoundException` - falls die gewünschte Inode nicht gefunden werden kann
    * `metaxa.os.fs.NoDirectoryInodeException` - falls es sich nicht um ein Verzeichnis handelt
    * `metaxa.os.fs.PermissionException` - falls die Zugriffsrechte des Verzeichnisses die Operation nicht erlauben

---

- *getLength*
  `public abstract long` **getLength**`( )`
  - **Usage**
    * Liefert die Größe des von der Inode belegten Bereichs auf der Partition zurück.

- *getParent*
  `public Inode` **getParent**`( )`
  - **Usage**
    * Liefert die übergeordnete Inode (`parent`-Verzeichnis) zurück.
  - **Returns** - die übergeordnete Inode

- *getSymlink*
  `public abstract String` **getSymlink**`( )`
  - **Usage**
    * Liefert den Pfad der Inode, auf die der symbolische Link verweist.
  - **Exceptions**
    * `metaxa.os.fs.InodeIOException` - falls bei der Ein-/Ausgabe ein Fehler auftritt
    * `metaxa.os.fs.NoSymlinkInodeException` - falls das Objekt keinen symbolischen Link darstellt
    * `metaxa.os.fs.NotSupportedException` - falls das Dateisystem diese Operation nicht unterstützt

    * `metaxa.os.fs.PermissionException` - falls die Zugriffsrechte des symbolischen Links die Operation nicht erlauben

---

- *i_nlinks*
  `public abstract int` **i_nlinks**`( )`
  - **Usage**
    * Liefert die Anzahl der Verweise auf die Inode zurück. Bei einem Verzeichnis ist die Anzahl der Verweise gleich der Anzahl an Verzeichnissen, die darin enthalten sind (also mindestens 2 für „ . “ und „ .. “). Weiterhin erhöht jeder symbolische Link auf die Inode die Zahl der Verweise.

- *isDirectory*
  `public abstract boolean` **isDirectory**`( )`
  - **Usage**
    * Testet, ob die Inode ein Verzeichnis repräsentiert.
  - **Returns** - `true`, falls es sich um ein Verzeichnis handelt

---

- *isDirty*
  `public boolean` **isDirty**`( )`
  - **Usage**
    * Ermittelt, ob die Inode noch geschrieben werden muss („dirty“).
  - **Returns** - `true`, falls die Inode als „dirty“ markiert ist

---

- *isFile*
  `public abstract boolean` **isFile**`( )`
  - **Usage**
    * Testet, ob die Inode eine reguläre Datei repräsentiert.
  - **Returns** - `true`, falls es sich um eine Datei handelt

- *isLocked*
  `public boolean` **isLocked**`( )`
  - **Usage**
    * Ermittelt, ob auf die Inode zugegriffen werden kann.
  - **Returns** - `true`, falls die Inode gesperrt ist

- *isOverlayed*
  `public synchronized boolean` **isOverlayed**`( java.lang.String` **name** `)`

- **Usage**
  - Überprüft, ob der Eintrag mit dem angegebenen Namen von einem anderen Dateisystem überlagert wird.
- **Parameters**
  - `name` - der Name des Eintrags, dessen Zustand ermittelt werden soll
- **Exceptions**
  - `metaxa.os.fs.NoDirectoryInodeException` - falls es sich nicht um ein Verzeichnis handelt

- *isReadable*
  `public abstract boolean` **isReadable**`( )`
  - **Usage**
    - Testet, ob der Inhalt der Inode gelesen werden kann.
  - **Returns** - `true`, falls es sich um eine Datei handelt und der Inhalt dieser Datei gelesen werden darf

---

- *isSymlink*
  `public abstract boolean` **isSymlink**`( )`
  - **Usage**
    - Testet, ob die Inode einen symbolischen Link repräsentiert.
  - **Returns** - `true`, falls es sich um einen symbolischen Link handelt

---

- *isWritable*
  `public abstract boolean` **isWritable**`( )`
  - **Usage**
    - Testet, ob die Inode geschrieben werden kann.
  - **Returns** - `true`, falls es sich um eine Datei handelt und in diese Datei geschrieben werden darf

---

- *lastModified*
  `public abstract long` **lastModified**`( )`
  - **Usage**
    - Liefert den Zeitpunkt der letzten Änderung zurück.
  - **Returns** - den Zeitstempel der letzten Änderung

- *lookup*
  `public Inode` **lookup**`( java.lang.String` **name** `)`
  - **Usage**

    * Liefert die Inode, die dem Verzeichniseintrag mit dem angegebenen Namen zugeordnet ist, sofern vorhanden. Zuerst wird der Name mit „ . “ und „ .. “ verglichen und bei Übereinstimmung die Inode selbst (`this`) bzw. die übergeordnete Inode (`parent`) zurückgeliefert. Als Nächstes wird überprüft, ob die Inode verdeckt wurde und eventuell die überdeckende Inode zurückgegeben. Schließlich muss der Eintrag im Dateisystem gesucht werden, was über die abstrakte Funktion `getInode` geschieht.
  - **Parameters**
    * `name` - der Name des Verzeichniseintrags, dessen Inode ermittelt werden soll
  - **Returns** - die dem Verzeichniseintrag zugeordnete Inode
  - **Exceptions**
    * `metaxa.os.fs.InodeIOException` - falls ein Fehler bei der Ein-/Ausgabe auftritt
    * `metaxa.os.fs.InodeNotFoundException` - falls die gewünschte Inode nicht gefunden werden kann
    * `metaxa.os.fs.NoDirectoryInodeException` - falls es sich nicht um ein Verzeichnis handelt
    * `metaxa.os.fs.PermissionException` - falls die Zugriffsrechte einer Pfadkomponente die Operation nicht erlauben

---

- *mkdir*
  `public abstract Inode` **mkdir**`( java.lang.String` **name**`, int` **mode** `)`

  - **Usage**
    * Erzeugt ein neues Verzeichnis mit dem angegebenen Namen innerhalb des durch diese Inode dargestellten Verzeichnisses, sofern noch kein Eintrag mit diesem Namen vorhanden ist.
  - **Parameters**
    * `name` - der Name des neuen Verzeichnisses
    * `mode` - die Zugriffsrechte des neuen Verzeichnisses
  - **Returns** - die Inode des neu angelegten Verzeichnisses
  - **Exceptions**
    * `metaxa.os.fs.FileExistsException` - falls eine Datei oder ein Verzeichnis mit diesem Namen bereits existiert
    * `metaxa.os.fs.InodeIOException` - falls ein Fehler bei der Ein-/Ausgabe auftritt
    * `metaxa.os.fs.NoDirectoryInodeException` - falls es sich nicht um ein Verzeichnis handelt
    * `metaxa.os.fs.PermissionException` - falls die Zugriffsrechte des Verzeichnisses die Operation nicht erlauben oder das Dateisystem als nur lesbar angemeldet wurde

---

- *overlay*
  `public synchronized void` **overlay**`(` `metaxa.os.fs.Inode` **newChild,** `java.lang.String` **name** `)`
  - **Usage**
    * Überlagert die Inode des angegebenen Verzeichniseintrags mit der Inode eines anderen Dateisystems („mount"-Mechanismus). Die Verdeckung der ursprünglichen Inode findet nicht innerhalb des Dateisystems statt (an der physikalischen Struktur auf der Festplatte ändert sich nichts), sie wird durch die Methoden dieser Klasse und der `FS`-Klasse realisiert und bietet dadurch die Möglichkeit, mehrere Dateisysteme in einen Verzeichnisbaum „einzuhängen". Jedesmal, wenn über die Methode `lookup` (weiter unten) auf die Inode mit dem Namen `name` zugegriffen wird, wird die Inode `newChild` zurückgeliefert.
  - **Parameters**
    * `newChild` - die Inode, mit der der ursprüngliche Eintrag überlagert werden soll
    * `name` - der Name des Eintrags, der überlagert werden soll
  - **Exceptions**
    * `metaxa.os.fs.InodeIOException` - falls ein Fehler bei der Ein-/Ausgabe auftritt
    * `metaxa.os.fs.InodeNotFoundException` - falls die zu überlagernde Inode nicht gefunden werden kann
    * `metaxa.os.fs.NoDirectoryInodeException` - falls es sich nicht um ein Verzeichnis handelt
    * `metaxa.os.fs.PermissionException` - falls die Zugriffsrechte des Verzeichnisses die Operation nicht erlauben
  - **See Also**
    * `metaxa.os.fs.Inode.lookup(String name)`

- *putInode*
  `public abstract void` **putInode**`( )`
  - **Usage**
    * Gibt die Ressourcen, die von der Inode verwendet werden, frei (wird aufgerufen, wenn die Inode aus dem Cache entfernt wird). Die Inode auf der Partition wird nicht gelöscht, nur ihr Objekt im Speicher.

- *read*
  `public abstract int` **read**`( byte []` **b,** `int` **off,** `int` **len** `)`
  - **Usage**
    * Liest den Inhalt der Datei.
  - **Parameters**

    * `b` - der Puffer, der die zu lesenden Daten aufnehmen soll
    * `off` - die Position innerhalb der Datei, ab der gelesen werden soll
    * `len` - die Anzahl zu lesender Byte
  - **Exceptions**
    * `metaxa.os.fs.InodeIOException` - falls bei der Ein-/Ausgabe ein Fehler auftritt
    * `metaxa.os.fs.NoFileInodeException` - falls es sich nicht um eine Datei handelt (Lesen eines Verzeichnisses ist nicht erlaubt)
    * `metaxa.os.fs.PermissionException` - falls die Zugriffsrechte der Datei die Operation nicht erlauben

---

- *readdirNames*
  `public abstract Vector` **readdirNames** `( )`
  - **Usage**
    * Liest den Inhalt des durch diese Inode repräsentierten Verzeichnisses aus (auch die Einträge „ . “ und „ .. “).
  - **Returns** - ein `Vector`-Objekt, das die Namen der Dateien und Verzeichnisse enthält
  - **Exceptions**
    * `metaxa.os.fs.NoDirectoryInodeException` - falls es sich nicht um ein Verzeichnis handelt

- *releaseInode*
  `public void` **releaseInode** `( )`
  - **Usage**
    * Gibt die Inode frei. Dem Cache wird mitgeteilt, dass die Inode nicht mehr verwendet wird.

- *removeAllOverlays*
  `public synchronized void` **removeAllOverlays** `( )`
  - **Usage**
    * Entfernt alle vorhandenen Überlagerungen.
  - **Exceptions**
    * `metaxa.os.fs.NoDirectoryInodeException` - falls es sich nicht um ein Verzeichnis handelt

- *removeOverlay*
  `public synchronized void` **removeOverlay** `( metaxa.os.fs.Inode` **child** `)`
  - **Usage**
    * Entfernt die Überlagerung durch die angegebene Inode.

– **Parameters**
  * `child` - die überlagernde Inode, die entfernt werden soll
– **Exceptions**
  * `metaxa.os.fs.InodeNotFoundException` - falls die überlagernde Inode nicht gefunden werden kann
  * `metaxa.os.fs.NoDirectoryInodeException` - falls es sich nicht um ein Verzeichnis handelt

- *rename*
  `public abstract void` **rename** `( java.lang.String` **oldname**, `metaxa.os.fs.Inode` **new_dir**, `java.lang.String` **newname** `)`

  – **Usage**
    * Verschiebt die Inode des angegebenen Verzeichniseintrags bzw. ändert deren Namen.
  – **Parameters**
    * `oldname` - der Name des Verzeichniseintrags, der verschoben bzw. umbenannt werden soll
    * `new_dir` - die Inode des Verzeichnisses, das den zu verschiebenden Verzeichniseintrag aufnehmen soll
    * `newname` - der neue Name des Verzeichniseintrags
  – **Exceptions**
    * `metaxa.os.fs.InodeIOException` - falls bei der Ein-/Ausgabe ein Fehler auftritt
    * `metaxa.os.fs.InodeNotFoundException` - falls die zu verschiebende Inode nicht gefunden werden kann
    * `metaxa.os.fs.NoDirectoryInodeException` - falls es sich nicht um ein Verzeichnis handelt
    * `metaxa.os.fs.PermissionException` - falls es sich bei `new_dir` um ein anderes Dateisystem handelt, die Zugriffsrechte des Verzeichnisses die Operation nicht erlauben oder das Dateisystem als nur lesbar angemeldet wurde

---

- *rmdir*
  `public abstract void` **rmdir** `( java.lang.String` **name** `)`

  – **Usage**
    * Entfernt das Verzeichnis mit dem angegebenen Namen aus dem durch diese Inode dargestellten Verzeichnis. Die Inode, die mit diesem Eintrag verknüpft war, wird freigegeben.
  – **Parameters**
    * `name` - der Name des zu löschenden Verzeichnisses
  – **Exceptions**
    * `metaxa.os.fs.DirNotEmptyException` - falls das zu löschende Verzeichnis noch Dateien/Verzeichnisse enthält

  * `metaxa.os.fs.InodeIOException` - falls ein Fehler bei der Ein-/Ausgabe auftritt
  * `metaxa.os.fs.InodeNotFoundException` - falls die zu löschende Inode nicht gefunden werden kann
  * `metaxa.os.fs.NoDirectoryInodeException` - falls es sich nicht um ein Verzeichnis handelt
  * `metaxa.os.fs.PermissionException` - falls die Zugriffsrechte des Verzeichnisses die Operation nicht erlauben oder das Dateisystem als nur lesbar angemeldet wurde

---

- *setDirty*
  `public void` **setDirty** `( boolean` **value** `)`
  - **Usage**
    * Legt fest, ob die Inode noch geschrieben werden muss oder ob die Version auf der Partition mit dem Speicherobjekt übereinstimmt.
  - **Parameters**
    * `der` - neue Zustand der Inode (`true` für „dirty“)

- *setLocked*
  `public void` **setLocked** `( boolean` **value** `)`
  - **Usage**
    * Sperrt die Inode bzw. gibt die Sperre frei.
  - **Parameters**
    * `der` - neue Zustand der Inode (`true` für „gesperrt“)

- *setParent*
  `public void` **setParent** `( metaxa.os.fs.Inode` **parent** `)`
  - **Usage**
    * Setzt die übergeordnete Inode (`parent`-Verzeichnis).
  - **Parameters**
    * `parent` - die übergeordnete Inode

- *sleep*
  `public void` **sleep** `( )`
  - **Usage**
    * Blockiert solange, bis die Inode durch einen Aufruf von `wakeUp` aktiviert wird. Damit kann auf die Freigabe der Sperre der Inode gewartet werden.

- *symlink*
  `public abstract void` **symlink**`(` `java.lang.String` **symname**`,` `java.lang.String` **newname** `)`

  - **Usage**
    * Erzeugt einen „symbolischen Link“, einen Verweis auf einen Verzeichniseintrag. Von außen ist zwischen Verweis und ursprünglichen Eintrag kein Unterschied festzustellen.
  - **Parameters**
    * `symname` - der Pfad, auf den ein Verweis angelegt werden soll (der Eintrag muss nicht existieren)
    * `newname` - der Name des Verweises
  - **Exceptions**
    * `metaxa.os.fs.FileExistsException` - falls bereits ein Eintrag mit angegebenen Namen (`newname`) existiert
    * `metaxa.os.fs.InodeIOException` - falls bei der Ein-/Ausgabe ein Fehler auftritt
    * `metaxa.os.fs.NoDirectoryInodeException` - falls es sich nicht um ein Verzeichnis handelt
    * `metaxa.os.fs.NotSupportedException` - falls das Dateisystem diese Operation nicht unterstützt
    * `metaxa.os.fs.PermissionException` - falls die Zugriffsrechte des Verzeichnisses die Operation nicht erlauben oder das Dateisystem als nur lesbar angemeldet wurde

---

- *unlink*
  `public abstract void` **unlink**`(` `java.lang.String` **name** `)`

  - **Usage**
    * Entfernt die Datei mit dem angegebenen Namen aus dem durch diese Inode dargestellten Verzeichnis. Die zu dem Eintrag gehörende Inode wird freigegeben.
  - **Parameters**
    * `name` - der Name der zu löschenden Datei
  - **Exceptions**
    * `metaxa.os.fs.InodeIOException` - falls ein Fehler bei der Ein-/Ausgabe auftritt
    * `metaxa.os.fs.InodeNotFoundException` - falls die zu löschende Inode nicht gefunden werden kann
    * `metaxa.os.fs.NoDirectoryInodeException` - falls es sich nicht um ein Verzeichnis handelt
    * `metaxa.os.fs.NoFileInodeException` - falls der zu löschende Eintrag existiert, aber keine Datei ist

    * `metaxa.os.fs.PermissionException` - falls die Zugriffsrechte des Verzeichnisses die Operation nicht erlauben oder das Dateisystem als nur lesbar angemeldet wurde

---

- *wakeUp*
  `public synchronized void` **wakeUp**`( )`
  - **Usage**
    * Aktiviert eine Inode, die mittels `sleep` auf die Freigabe der Sperre wartet.

- *write*
  `public abstract int` **write**`( byte [] `**b**`, int `**off**`, int `**len**` )`
  - **Usage**
    * Schreibt Daten in die Datei. Die Datei wird u.U. vergrößert.
  - **Parameters**
    * `b` - der Puffer, der die zu schreibenden Daten enthält
    * `off` - die Position innerhalb der Datei, ab der geschrieben werden soll
    * `len` - die Anzahl zu schreibender Byte
  - **Exceptions**
    * `metaxa.os.fs.InodeIOException` - falls bei der Ein-/Ausgabe ein Fehler auftritt
    * `metaxa.os.fs.NoFileInodeException` - falls es sich nicht um eine Datei handelt (Lesen eines Verzeichnisses ist nicht erlaubt)
    * `metaxa.os.fs.PermissionException` - falls die Zugriffsrechte der Datei die Operation nicht erlauben oder das Dateisystem als nur lesbar angemeldet wurde

---

- *writeInode*
  `public abstract void` **writeInode**`( )`
  - **Usage**
    * Schreibt Änderungen, die an der Inode vorgenommen wurden, auf die Partition.

Interface FileSystem

Dieses Interface stellt Operationen zur Verfügung, die das Dateisystem als Ganzes betreffen. Eine konkrete Dateisystem-Implementierung stellt dem virtuellen Dateisystem damit Funktionen zur Verfügung, um ein Dateisystem zu an- und abzumelden, zu erzeugen und zu reparieren.

Declaration

public interface FileSystem

Methods

- *build*
  `public void` **build**`(` `java.lang.String` **name**`,` `int` **blocksize**`,` `int` **device** `)`
  - **Usage**
    * Erzeugt ein neues Dateisystem auf der angegebenen Partition (die Partition wird überschrieben).
  - **Parameters**
    * `name` - der Name des neuen Dateisystems
    * `blocksize` - die Blockgröße innerhalb des Dateisystems (Vielfache von 512)
    * `device` - die Kennung der Partition, auf der das Dateisystem installiert werden soll

---

- *check*
  `public void` **check**`(` `int` **device** `)`
  - **Usage**
    * Überprüft das Dateisystem auf der angegebenen Partition und repariert es gegebenenfalls, um es in einen konsistenten Zustand zu bringen.
  - **Parameters**
    * `device` - die Kennung der Partition, die überprüft werden soll

---

- *device*
  `public int` **device**`(` `)`
  - **Usage**
    * Gibt die Kennung der Partition zurück, die das Dateisystem enthält.
  - **Returns** - die Partitionskennung

- *init*
  `public void` **init**`(` `int` **device**`,` `boolean` **read_only** `)`

- **Usage**
  - Initialisiert das Dateisystem (wird vom `mount`-Mechanismus der Klasse `FS` verwendet). Auf der angegebenen Partition muß sich das entsprechende Dateisystem befinden.
- **Parameters**
  - `device` - die Kennung der Partition, die das Dateisystem enthält
  - `read_only` - falls `true`, wird das Dateisystem als nur lesbar gekennzeichnet

---

- *name*
  `public String` **name**`( )`
  - **Usage**
    - Liefert den Namen des Dateisystems zurück.
  - **Returns** - Zeichenkette mit dem Dateisystem-Namen

- *release*
  `public void` **release**`( )`
  - **Usage**
    - Gibt vom Dateisystem belegte Strukturen wieder frei und schreibt Blöcke, die als „dirty" markiert sind, auf die entsprechende Partition.

Klasse **BufferHead**

Objekte der Klasse BufferHead werden verwendet, um Daten zwischen Festplattentreiber und Dateisystem und innerhalb des Dateisystems zu übertragen. Sie besitzen ein Memory-Objekt (`b_data`) in der vom Dateisystem verwendeten Blockgröße (1024, 2048 oder 4096 Byte). Unterschieden werden BufferHeads (innerhalb des BufferCaches) anhand von Partition/Festplatte (`b_device`), Blocknummer (`b_block`) und Größe (`b_data.length`).

Declaration

public class BufferHead
extends java.lang.Object

Fields

- public BufferHeadHashKey b_hashkey
  - vom BufferCache verwendet
- public long b_flushtime
  - vom BufferCache verwendet
- public int b_count
  - vom BufferCache verwendet
- public int b_list
  - vom BufferCache verwendet
- public Memory b_data
  - das Speicherobjekt mit dem Inhalt des Blocks
- public int b_device
  - die Kennung der Partition
- public int b_block
  - die Blocknummer innerhalb des Dateisystems
- public int b_size
  - die Blockgröße (1024, 2048 oder 4096 Byte)

Constructors

- *BufferHead*
  `public` **BufferHead** `( int` **device**`, int` **block**`, int` **size** `)`

Methods

- *clear*
  `public synchronized void` **clear** `( )`
  - **Usage**
    * Löscht den gesamten Inhalt des BufferHeads. Der Inhalt wird auf 0 gesetzt.

- *clear*
  `public synchronized void` **clear** `( int` **from**`, int` **to** `)`
  - **Usage**
    * Löscht einen Bereich des BufferHeadinhalts. Der Bereich wird auf 0 gesetzt.
  - **Parameters**
    * `from` - das erste zu löschende Byte des Inhalts
    * `to` - das erste Byte des Bereichs, das nicht mehr zu gelöscht werden soll

---

- *dirty*
  `public boolean` **dirty** `( )`
  - **Usage**
    * Gibt an, ob der Inhalt des BufferHeads geändert und noch nicht auf die Festplatte geschrieben wurde.
  - **Returns** - `true`, falls der Inhalt des BufferHeads nicht mit dem auf der Festplatte übereinstimmt

---

- *endIo*
  `public synchronized void` **endIo** `( boolean` **error**`, boolean` **synchronous** `)`
  - **Usage**
    * Beendet den Lese- bzw. Schreibevorgang. Diese Methode wird vom Treiber aufgerufen.
  - **Parameters**
    * `error` - zeigt an, ob bei der Operation ein Fehler aufgetreten ist (`true`)
    * `synchronous` - falls `true`, wird auf das Ende der Operation (mittels `sleep`) gewartet

---

- *locked*
  `public boolean` **locked**`( )`
  - **Usage**
    * Liefert den Zustand der Sperre zurück.
  - **Parameters**
    * `value` - `true`, falls der BufferHead gesperrt ist

- *markDirty*
  `public void` **markDirty**`( boolean` **value** `)`
  - **Usage**
    * Markiert den BufferHead als „dirty“, d.h. sein Inhalt hat sich geändert und stimmt nicht mehr mit seinem Abbild auf der Festplatte überein (dem Inhalt des entsprechenden Blocks); er muss noch geschrieben werden.
  - **Parameters**
    * `value` - falls `true`, muss der Inhalt noch geschrieben werden

---

- *markLocked*
  `public void` **markLocked**`( boolean` **value** `)`
  - **Usage**
    * Sperrt den BufferHead bzw. gibt die Sperre des BufferHeads frei. Für die Dauer der Sperre können keine anderen Threads darauf zugreifen. Um auf das Ende der Sperre zu warten, dient die Methode `waitOn`; bei einer Freigabe der Sperre lassen sich andere Threads/Prozesse mittels `notifyAll` wieder aktivieren.
  - **Parameters**
    * `value` - der neue Zustand der Sperre

- *markUptodate*
  `public void` **markUptodate**`( boolean` **value** `)`
  - **Usage**
    * Legt den Zustand des BufferHead-Inhalts fest. „Uptodate“ bedeutet, dass die Leseoperation erfolgreich war und der Inhalt gültig ist.
  - **Parameters**
    * `value` - der neue Zustand des Inhalts: `false` bedeutet, dass bei der Operation ein Fehler aufgetreten und der Inhalt nicht mehr gültig ist

---

- *uptodate*
  `public boolean` **uptodate**`( )`

- **Usage**
  - Liefert den Zustand des BufferHead-Inhalts zurück.
- **Returns** - `false` bedeutet, dass bei der Operation ein Fehler aufgetreten und der Inhalt nicht mehr gültig ist

---

- *waitOn*
  `public void` **waitOn**`( )`
  - **Usage**
    - Wartet, bis die Sperre des BufferHeads aufgehoben wird.

Klasse **BufferCache**

Der BufferCache dient zur Zwischenspeicherung von `BufferHead`-Objekten. Die Klasse stellt Methoden bereit, um `BufferHeads` zu lesen bzw. auf Festplatte zu schreiben. Um einen schnellen Zugriff auf die `BufferHeads` zu gewährleisten, wird eine Hashtabelle (`java.util.Hashtable`) zur Verwaltung der Objekte verwendet. Nicht benutzte `BufferHeads` werden in einer Least-Recently-Used-Liste gespeichert.

Declaration

public class BufferCache
extends java.lang.Object

Fields

- public static final boolean READ
  - Sektor lesen
- public static final boolean WRITE
  - Sektor schreiben

Methods

- *bforget*
  `public synchronized void` **bforget**`( metaxa.os.fs.BufferHead` **bh** `)`
  - **Usage**
    * Hängt den `BufferHead` ans Ende der LRU-Liste.
  - **Parameters**
    * `bh` - der `BufferHead`, der freigegeben („vergessen“) werden soll

---

- *bread*
  `public synchronized BufferHead` **bread**`( int` **device**`, int` **block**`, int` **size** `)`
  - **Usage**
    * Liest den angegebenen Block von Festplatte. Der entsprechende `BufferHead` wird mittels `getblk` angefordert und der Inhalt des Blocks vom Festplattentreiber gelesen.
  - **Parameters**
    * `device` - die Kennung der Partition, die den Block enthält

    * `block` - die Nummer des Blocks, der gelesen werden soll
    * `size` - die Größe des gewünschten Blocks (normalerweise Vielfache von 1024)
  - **Returns** - der `BufferHead` mit dem Inhalt des angegebenen Blocks

---

- *brelse*
  `public synchronized void` **brelse**`( metaxa.os.fs.BufferHead` **bh** `)`
  - **Usage**
    * Gibt den `BufferHead` frei. Dem Cache wird dadurch mitgeteilt, dass der `BufferHead` nicht mehr verwendet wird und in die LRU-Liste eingefügt werden kann. Falls er als „dirty“ markiert ist, wird der Zeitpunkt festgesetzt, zur dem er frühestens geschrieben werden darf (um weitere Zugriffe auf den Puffer ohne die Unterbrechung eines langsamen Festplattenzugriffs zu ermöglichen). Diese Zeitspanne beträgt 30 Sekunden. Ist der Zugriffszähler des `BufferHeads` (`b_count`) gleich 0, wird er, falls sein Inhalt gültig ist, ans Ende der LRU-Liste gehängt, ansonsten an den Anfang.
  - **Parameters**
    * `bh` - der `BufferHead`, der freigegeben werden soll

- *findBuffer*
  `public synchronized BufferHead` **findBuffer**`( int` **device**`, int` **block**`, int` **size** `)`
  - **Usage**
    * Liefert den `BufferHead` für den angegebenen Block zurück. Im Gegensatz zu `getblk` wird nur versucht, den gewünschten `BufferHead` in der Hashtabelle zu finden.
  - **Parameters**
    * `device` - die Kennung der Partition, die den Block enthält
    * `block` - die Nummer des Blocks, dessen `BufferHead` gefunden bzw. erzeugt werden soll
    * `size` - die Größe des gewünschten Blocks (normalerweise Vielfache von 1024)
  - **Returns** - der `BufferHead` des angegebenen Blocks

- *flush*
  `public synchronized void` **flush**`( )`
  - **Usage**
    * Alle `BufferHeads`, die als „dirty“ markiert sind, d.h. noch geschrieben werden müssen, werden mittels `rwBlock` auf die Festplatte zurückgeschrieben, sofern sie nicht gesperrt sind bzw. noch nicht alt genug sind.

- *getblk*
  `public synchronized BufferHead` **getblk**`( int` **device,** `int` **block,** `int` **size** `)`

  – **Usage**
    * Liefert den `BufferHead` für den angegebenen Block zurück. Es wird zuerst versucht, den gewünschten `BufferHead` in der Hashtabelle zu finden. Falls er noch nicht im Cache verzeichnet ist, wird der erste Block aus der Liste mit den freien Blöcken entfernt, initialisiert und in die Hashtabelle eingefügt. Falls die Liste leer ist, also keine freien `BufferHeads` zur Verfügung stehen, wird versucht, den Cache zu vergrössern. Dies ist aber nur bis zu einer bestimmten Grenze möglich. Falls auch diese Grenze erreicht ist, wird gewartet, bis ein `BufferHead` frei wird.

  – **Parameters**
    * `device` - die Kennung der Partition, die den Block enthält
    * `block` - die Nummer des Blocks, dessen `BufferHead` gefunden bzw. erzeugt werden soll
    * `size` - die Größe des gewünschten Blocks (normalerweise Vielfache von 1024)

  – **Returns** - der `BufferHead` des angegebenen Blocks

- *instance*
  `public static BufferCache` **instance**`( )`

  – **Usage**
    * Liefert die Instanz des Caches zurück (es existiert nur ein BufferCache-Objekt).

  – **Returns** - das BufferCache-Objekt

- *rwBlock*
  `public synchronized void` **rwBlock**`( boolean` **rw,** `metaxa.os.fs.BufferHead` **bh,** `boolean` **synchronous** `)`

  – **Usage**
    * Liest den Inhalt des `BufferHeads` von bzw. schreibt in auf die Festplatte (unter Verwendung des Festplattentreibers). Hier geschieht die Umsetzung von Blocknummern und variablen Blockgrößen in Sektornummern mit einheitlicher Größe (512 Byte), wie sie vom Festplattentreiber verwendet werden (die Sektornummer bezeichnet den Sektor relativ zum Anfang der Partition).

  – **Parameters**
    * `rw` - falls `true`, wird der Block gelesen, sonst geschrieben
    * `bh` - der `BufferHead`, dessen Inhalt gelesen bzw. geschrieben werden soll
    * `synchronous` - falls `true`, wird auf das Ende der Operation gewartet

---

- *showBuffers*
  `public synchronized void` **showBuffers**`( )`

- **Usage**
  - Gibt eine Statistik der `BufferHeads` im Cache und der LRU-Liste aus. Die Zahl der gesperrten und als „dirty“ markierten `BufferHeads` wird ermittelt.

- *stopDemon*
  `public synchronized void` **stopDemon**`( )`
  - **Usage**
    - Beendet den Thread, der in regelmässigen Abstanden die `BufferHeads` des Caches auf die Festplatte schreibt („flush“).

- *syncDevice*
  `public synchronized void` **syncDevice**`( int` **device** `)`
  - **Usage**
    - Schreibt die `BufferHeads` des angegebenen Dateisystems (Partition), die als „dirty“ markiert sind, auf die angegebene Partition.
  - **Parameters**
    - `device` - die Partition mit dem Dateisystem, das in einen konsistenten Zustand gebracht werden soll

Klasse **InodeCache**

Diese Klasse dient als Verwaltungsstelle und Zwischenspeicher für Inodes, um häufige Festplattenzugriffe beim Lesen und Schreiben von Inodes zu verhindern.

Declaration

public class InodeCache
extends java.lang.Object

Methods

- *addInode*
  `public synchronized void` **addInode**`( metaxa.os.fs.Inode` **inode** `)`
  - **Usage**
    * Fügt dem Cache eine Inode hinzu.
  - **Parameters**
    * `inode` - die Inode, die in den Cache aufgenommen werden soll

---

- *iget*
  `public synchronized Inode` **iget**`( int` **device**`, int` **ino** `)`
  - **Usage**
    * Liefert die Inode mit der angegebenen Nummer aus dem Cache. Falls die Inode im Cache nicht vorhanden ist, wird `null` zurückgegeben.
  - **Parameters**
    * `device` - die Kennung der Partition und damit des Dateisystems, zu dem die Inode gehört
    * `ino` - die Nummer der Inode innerhalb ihres Dateisystems
  - **Returns** - die gewünschte Inode, falls im Cache vorhanden, ansonsten `null`

- *instance*
  `public static InodeCache` **instance**`( )`
  - **Usage**
    * Liefert die Instanz des Caches zurück (es existiert nur ein InodeCache-Objekt).
  - **Returns** - das InodeCache-Objekt

- *invalidateInodes*
  `public synchronized boolean` **invalidateInodes**`( int` **device** `)`
  - **Usage**

  * Entfernt die Inodes des durch die angegebene Partition bezeichneten Dateisystems aus dem Cache. Vorher sollte `syncInodes` aufgerufen werden, damit Änderungen nicht verloren gehen.
- **Parameters**
  * `die` - Partition, die das Dateisystem mit den nicht mehr benötigten Inodes enthält

---

- *iput*
`public synchronized void` **iput** `( metaxa.os.fs.Inode` **inode** `)`
  - **Usage**
    * Teilt dem Cache mit, das die angegebene Inode nicht mehr verwendet wird. Der Nutzungszähler der Inode wird erniedrigt. Falls auf die Inode nicht mehr verwiesen wird (`inode.i_nlinks`), wird die Inode gelöscht.
  - **Parameters**
    * `inode` - die freizugebende Inode

- *markInodeDirty*
`public synchronized void` **markInodeDirty** `( metaxa.os.fs.Inode` **inode** `)`
  - **Usage**
    * Setzt den Status der angegebenen Inode auf „dirty“, d.h. ihre Verwaltungsdaten (Metadaten) wurden verändert und müssen noch geschrieben werden.
  - **Parameters**
    * `inode` - die Inode, die als „dirty“ markiert werden soll

---

- *showInodes*
`public synchronized void` **showInodes** `( )`
  - **Usage**
    * Gibt eine Statistik der Inodes im Cache aus.

- *syncInodes*
`public synchronized void` **syncInodes** `( int` **device** `)`
  - **Usage**
    * Schreibt die Verwaltungsdaten aller Inodes, die als „dirty“ markiert sind und dem Dateisystem angehören, das sich auf der angegebenen Partition befindet, auf die Festplatte.
  - **Parameters**
    * `device` - die Partition, die das Dateisystem mit den zu schreibenden Inodes enthält

---

- *writeInodeNow*
  `public synchronized void` **writeInodeNow** `( metaxa.os.fs.Inode` **inode** `)`
  - **Usage**
    * Die Inode wird auf Festplatte geschrieben, sofern sie als „dirty“ markiert ist.
  - **Parameters**
    * `inode` - die Inode, die geschrieben werden soll

Klasse **FS**

Diese Klasse stellt die Schnittstelle des Dateisystems zur Anwendungsebene dar. Sie enthält Methoden, die grundlegende Dateisystemoperationen ausführen (ähnlich den Systemcalls in einem Unix-Dateisystem). Dazu zählen: `cd`, `mkdir`, `rmdir`, `create`, `unlink`, `rename`, `symlink`, `read` und `write`.

Der Unterschied zu den gleichlautenden Befehlen der Klasse `Inode` ist, dass hier Dateisystem-weite Pfade verwendet werden können und nicht Einträge innerhalb einer Verzeichnis-Inode angesprochen werden. Hier geschieht also die Umsetzung von Befehlen auf Pfaden in Methodenaufrufen von Inode-Objekten. Dabei wird ein aktuelles Verzeichnis mitgeführt. Soll z.B. der Befehl `mkdir /usr/local/bin` ausgeführt werden, wird in der Inode des Wurzelverzeichnisses (die der Klasse immer bekannt ist) nach dem Eintrag „usr“ gesucht, dessen Inode geladen, in dieser Inode nach dem Eintrag „local“ gesucht und in der zugehörigen Inode die Methode `mkdir(bin)` aufgerufen. Es werden zwei Inodes permanent gespeichert: die des Wurzelverzeichnisses und die des aktuellen Verzeichnisses zusammen mit ihrem Pfad. Es wird im Folgenden nur auf die Methoden eingegangen, die nicht auch in der `Inode`-Klasse vorhanden sind, da diese im Wesentlichen auf eine Zerlegung des Pfades und Aufruf der entsprechenden Methoden in der Inode des Dateinamens bestehen.

Declaration

public class FS
extends java.lang.Object

Methods

- *available*
  `public long` **available**`( )`
  - **Usage**
    * Liefert den freien Platz innerhalb des Dateisystems.
  - **Returns** - der freie Platz in Byte

- *cd*
  `public void` **cd**`( java.lang.String` **path** `)`
  - **Usage**
    * Setzt das aktuelle Verzeichnis auf den angegebenen Pfad (sofern gültig).
  - **Parameters**
    * `path` - der Pfad, der als aktuelles Verzeichnis verwendet werden soll

---

- *cleanUp*
  `public void` **cleanUp**`( )`

– **Usage**
  * Gibt belegte Ressourcen frei und beendet Threads des Dateisystem. Diese Methode ist für das „Herunterfahren“ des Systems gedacht; nach `cleanUp` sollte nicht mehr auf das Dateisystem zugegriffen werden.

• *create*
`public void` **create**`( java.lang.String` **path**`, int` **mode** `)`

– **Usage**
  * Erzeugt eine neue Datei mit dem angegebenen Pfadnamen.

– **Parameters**
  * `path` - der Pfad der neuen Datei
  * `mode` - die Zugriffsrechte der neuen Datei

– **Exceptions**
  * `metaxa.os.fs.FileExistsException` - falls eine Datei/ein Verzeichnis mit diesem Pfad bereits existiert
  * `metaxa.os.fs.InodeIOException` - falls ein Fehler bei der Ein-/Ausgabe auftritt
  * `metaxa.os.fs.InodeNotFoundException` - falls das Verzeichnis, das die neue Datei aufnehmen soll, nicht gefunden werden kann
  * `metaxa.os.fs.NoDirectoryInodeException` - falls es sich bei dem Pfad zur Datei nicht um ein Verzeichnis handelt

---

• *getCwdInode*
`public Inode` **getCwdInode**`( )`

– **Usage**
  * Liefert die Inode, die dem aktuellen Verzeichnis zugeordnet ist.

– **Returns** - die Inode des aktuellen Verzeichnisses

• *getCwdPath*
`public String` **getCwdPath**`( )`

– **Usage**
  * Liefert den Pfad des aktuellen Verzeichnisses („current working directory“).

– **Returns** - der absolute Pfad des aktuellen Verzeichnisses

---

• *getFileName*
`public String` **getFileName**`( java.lang.String` **path** `)`

– **Usage**
  * Liefert den Datei- bzw. Verzeichnisnamen auf den der angegebene Pfad zeigt. Falls der Pfadname keine Pfadkomponenten enthält, wird er unverändert zurückgegeben.

- **Parameters**
  * `path` - der Pfadname, der zerlegt werden soll
- **Returns** - der Datei- oder Verzeichnisname, auf den der Pfad zeigt

---

- *getPathName*
  `public String` **getPathName**`( java.lang.String` **path** `)`
  - **Usage**
    * Liefert das übergeordnete Verzeichnis zum angegebenen Pfadnamen. Falls der Pfadname keine Pfadkomponenten enthält, wird der Pfad des aktuellen Verzeichnisses zurückgegeben.
  - **Parameters**
    * `path` - der Pfadname, dessen übergeordnetes Verzeichnis zurückgegeben werden soll
  - **Returns** - der Pfad des übergeordneten Verzeichnisses

- *instance*
  `public static FS` **instance**`( )`
  - **Usage**
    * Liefert die Instanz der FS-Klasse zurück (es existiert nur ein FS-Objekt).
  - **Returns** - das FS-Objekt

- *lookup*
  `public Inode` **lookup**`( java.lang.String` **path** `)`
  - **Usage**
    * Liefert die Inode zum angegebenen Pfad zurück. Repräsentiert die Inode einen symbolischen Link, wird dieser verfolgt und die Inode, auf die sich der Verweis bezieht, zurückgegeben.
  - **Parameters**
    * `path` - der Pfad, dessen zugeordnete Inode zurückgegeben werden soll
  - **Returns** - die Inode des durch den angegebenen Pfad bezeichneten Verzeichniseintrags
  - **Exceptions**
    * `metaxa.os.fs.InodeIOException` - falls ein Fehler bei der Ein-/Ausgabe auftritt
    * `metaxa.os.fs.InodeNotFoundException` - falls der Verzeichniseintrag (und damit die Inode) nicht gefunden werden kann
    * `metaxa.os.fs.NoDirectoryInodeException` - falls es sich bei einem Teil des Pfades, auf den ein „ / “ folgt, nicht um ein Verzeichnis handelt

---

- *mkdir*
  `public void` **mkdir**`( java.lang.String` **path**`, int` **mode** `)`

- **Usage**
  * Erzeugt ein neues Verzeichnis mit dem angegebenen Pfadnamen.
- **Parameters**
  * `path` - der Pfad des neuen Verzeichnisses
  * `mode` - die Zugriffsrechte des neuen Verzeichnisses
- **Exceptions**
  * `metaxa.os.fs.FileExistsException` - falls eine Datei/ein Verzeichnis mit diesem Pfad bereits existiert.
  * `metaxa.os.fs.InodeIOException` - falls ein Fehler bei der Ein-/Ausgabe auftritt
  * `metaxa.os.fs.InodeNotFoundException` - falls das Verzeichnis, das das neue Verzeichnis aufnehmen soll, nicht gefunden werden kann
  * `metaxa.os.fs.NoDirectoryInodeException` - falls es sich bei dem Pfad zum Verzeichnis nicht um ein Verzeichnis handelt

---

- *mount*
  `public void` **mount** `( metaxa.os.fs.FileSystem` **filesystem**`, int` **device**`, java.lang.String` **path**`, boolean` **read_only** `)`

  - **Usage**
    * Hängt ein Dateisystem in den Verzeichnisbaum ein, indem eine Inode durch die Wurzelinode des Dateisystems überlagert wird. Die Wurzelinode ist unter dem angegebenen Namen ansprechbar.
  - **Parameters**
    * `filesystem` - das Dateisystem, das angemeldet werden soll
    * `device` - die Partition, die das Dateisystem enthält
    * `path` - der Pfad, unter dem das einzuhängende Dateisystem erreichbar sein soll. Die Inode, die durch diesen Pfad repräsentiert wird, wird überlagert.
    * `read_only` - falls `true`, kann auf die Partition nur lesend zugegriffen werden
  - **Exceptions**
    * `metaxa.os.fs.InodeIOException` - falls ein Fehler bei der Ein-/Ausgabe auftritt
    * `metaxa.os.fs.InodeNotFoundException` - falls der Verzeichniseintrag (und damit die Inode) nicht gefunden werden kann
    * `metaxa.os.fs.NoDirectoryInodeException` - falls es sich bei dem zu überlagernden Verzeichniseintrag nicht um ein Verzeichnis handelt

---

- *mountRoot*
  `public void` **mountRoot** `( metaxa.os.fs.FileSystem` **filesystem**`, int` **device**`, boolean` **read_only** `)`

  - **Usage**

    * Initialisiert das angegebene Dateisystem als Root-Dateisystem. Setzt das Wurzelverzeichnis und das aktuelle Verzeichnis auf die Root-Inode des Dateisystems, den Pfad des aktuellen Verzeichnisses auf „ / “.
  - **Parameters**
    * `filesystem` - ein FileSystem-Objekt, das das Root-Dateisystem repräsentiert
    * `device` - die Partition, die das Root-Dateisystem enthält und auf die zugegriffen werden soll
    * `read_only` - falls `true`, kann auf die Partition nur lesend zugegriffen werden

---

- *read*
  `public int` **read** `( java.lang.String` **path**`, byte [] `**b**`, int `**off**`, int` **len** `)`
  - **Usage**
    * Liest den Inhalt der Datei mit dem angegebenen Pfadnamen.
  - **Parameters**
    * `path` - der Pfadname der zu lesenden Datei
    * `b` - das Byte-Array, das den Dateiinhalt aufnehmen soll
    * `off` - der Offset innerhalb der Datei
    * `len` - die Anzahl zu lesender Byte
  - **Returns** - die Anzahl tatsächlich gelesener Byte
  - **Exceptions**
    * `metaxa.os.fs.InodeIOException` - falls ein Fehler bei der Ein-/Ausgabe auftritt
    * `metaxa.os.fs.NoFileInodeException` - falls es sich bei dem angegebenen Pfad nicht um eine Datei handelt

---

- *rename*
  `public void` **rename** `( java.lang.String` **path**`, java.lang.String` **pathneu** `)`
  - **Usage**
    * Verschiebt die Datei bzw. das Verzeichnis an einen anderen Ort im Verzeichnisbaum bzw. ändert den Namen.
  - **Parameters**
    * `path` - der Pfad des Verzeichniseintrags, der verschoben bzw. umbenannt werden soll
    * `pathneu` - der Zielpfad
  - **Exceptions**
    * `metaxa.os.fs.InodeIOException` - falls ein Fehler bei der Ein-/Ausgabe auftritt
    * `metaxa.os.fs.InodeNotFoundException` - falls der Verzeichniseintrag (und damit die Inode) nicht gefunden werden kann

    - * `metaxa.os.fs.NoDirectoryInodeException` - falls es sich beim Pfad zur Datei/zum Verzeichnis nicht um ein Verzeichnis handelt
    - * `metaxa.os.fs.PermissionException` - falls es sich um ein Rootverzeichnis oder einen Mountpunkt handelt

---

- *rmdir*
  `public void` **rmdir** `( java.lang.String` **path** `)`
  - **Usage**
    - * Entfernt das Verzeichnis mit dem angegebenen Pfadnamen.
  - **Parameters**
    - * `path` - der Pfad des zu löschenden Verzeichnisses
  - **Exceptions**
    - * `metaxa.os.fs.DirNotEmptyException` - falls das zu löschende Verzeichnis nicht leer ist
    - * `metaxa.os.fs.InodeIOException` - falls ein Fehler bei der Ein-/Ausgabe auftritt
    - * `metaxa.os.fs.InodeNotFoundException` - falls das Verzeichnis, das das zu löschende Verzeichnis enthalten soll, nicht gefunden werden kann
    - * `metaxa.os.fs.NoDirectoryInodeException` - falls es sich bei dem Pfad zum Verzeichnis nicht um ein Verzeichnis handelt

---

- *symlink*
  `public void` **symlink** `( java.lang.String` **path**`, java.lang.String` **pathneu** `)`
  - **Usage**
    - * Legt einen symbolischen Link an, der auf den angegebenen Pfad verweist.
  - **Parameters**
    - * `path` - der Verzeichniseintrag, auf den verwiesen werden soll
    - * `pathneu` - der Pfad des anzulegenden symbolischen Links
  - **Exceptions**
    - * `metaxa.os.fs.FileExistsException` - falls der Pfad des symbolischen Links bereits existiert
    - * `metaxa.os.fs.InodeIOException` - falls ein Fehler bei der Ein-/Ausgabe auftritt
    - * `metaxa.os.fs.InodeNotFoundException` - falls der Verzeichniseintrag (und damit die Inode) nicht gefunden werden kann
    - * `metaxa.os.fs.NoDirectoryInodeException` - falls es sich bei dem Pfad zur Datei/zum Verzeichnis nicht um ein Verzeichnis handelt

---

- *unlink*
  `public void` **unlink** `( java.lang.String` **path** `)`

- **Usage**
  - Entfernt die Datei mit dem angegebenen Pfadnamen.
- **Parameters**
  - `path` - der Pfad der zu löschenden Datei
- **Exceptions**
  - `metaxa.os.fs.InodeIOException` - falls ein Fehler bei der Ein-/Ausgabe auftritt
  - `metaxa.os.fs.InodeNotFoundException` - falls das Verzeichnis, das die zu löschende Datei enthalten soll, nicht gefunden werden kann
  - `metaxa.os.fs.NoDirectoryInodeException` - falls es sich bei dem Pfad zur Datei nicht um ein Verzeichnis handelt

---

- *unmount*
  `public void` **unmount**`( metaxa.os.fs.FileSystem` **filesystem** `)`

  - **Usage**
    - Entfernt ein Dateisystem aus dem Verzeichnisbaum. Die Inode, die durch die Wurzelinode des Dateisystems überlagert war, wird dadurch wieder zugänglich.
  - **Parameters**
    - `filesystem` - das Dateisystem, das abgemeldet werden soll
  - **Exceptions**
    - `metaxa.os.fs.InodeNotFoundException` - falls der Verzeichniseintrag (und damit die Inode) nicht gefunden werden kann
    - `metaxa.os.fs.NoDirectoryInodeException` - falls es sich bei dem überlagerten Verzeichniseintrag nicht um ein Verzeichnis handelt

---

- *write*
  `public int` **write**`( java.lang.String` **path**`, byte [] ` **b**`, int` **off**`, int` **len** `)`

  - **Usage**
    - Schreibt den Inhalt des Byte-Arrays in die Datei mit dem angegebenen Pfadnamen.
  - **Parameters**
    - `path` - der Pfadname der Datei, deren Inhalt geändert werden soll
    - `b` - das Byte-Array, das die zu schreibenden Daten enthält
    - `off` - der Offset innerhalb der Datei
    - `len` - die Anzahl zu schreibender Byte
  - **Returns** - die Anzahl tatsächlich geschriebener Byte
  - **Exceptions**
    - `metaxa.os.fs.InodeIOException` - falls ein Fehler bei der Ein-/Ausgabe auftritt
    - `metaxa.os.fs.NoFileInodeException` - falls es sich bei dem angegebenen Pfad nicht um eine Datei handelt

* `metaxa.os.fs.PermissionException` - falls das Dateisystem als nur lesbar angemeldet wurde

Literaturverzeichnis

[And90] Prabhat K. Andleigh: *UNIX system architecture*, Prentice Hall, 1990

[Bac86] Maurice J. Bach: *The Design of the UNIX Operation System*, Prentice Hall, 1986

[BBD99] Michael Beck and Harald Böhme and Mirko Dziadzka: *Linux — Kernelprogrammierung. Algorithmen und Strukturen der Version 2.2*, Addison-Wesley, 1999

[Boo94] Grady Booch: *Object-Oriented Analysis and Design with Applications*, Addison-Wesley, 1994

[CTT94] Rémy Card and Stephen Tweedie and Theodore Ts'o: *Design and Implementation of the Second Extended Filesystem*, Proceedings of the First Dutch International Symposium on Linux, 1994, `http://web.mit.edu/tytso/www/linux/ext2intro.html`

[dH00] Pieter den Haan: *Yet Another Enhanced IDE/Fast-ATA/ATA-2 FAQ*, 2000, `pieterh@sci.kun.nl`, news: `comp.sys.ibm.pc.hardware.storage`, `comp.sys.ibm.pc.hardware.misc`

[Eck98] Bruce Eckel: *Thinking in Java*, Prentice Hall, 1998

[FRG] Flux Research Group: *OSKit*, University of Utah, `http://www.cs.utah.edu/flux/oskit`

[JOS] *JOS — A Free Java[tm] Based Operating System*, `http://jos.org/`

[Kle99a] Jürgen Kleinöder: *Folien zur Vorlesung „Ausgewählte Kapitel der praktischen Betriebsprogrammierung“*, Universität Erlangen-Nürnberg, IMMD IV, 1999

[Kle99b] Jürgen Kleinöder: *Folien zur Vorlesung „Object-oriented Concepts in Distributed Systems“*, Universität Erlangen-Nürnberg, IMMD IV, 1999

[Lan96] Hale Landis: *How it Works Series — Partition Tables*, 1996, `landis@sugs.tware.com`

[Lar97] Craig Larman: *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, Prentice Hall, 1997

[Mes98] Hans-Peter Messmer: *PC-Hardwarebuch: Aufbau, Funktionsweise, Programmierung*, Addison-Wesley, 1998

[Pho91] Phoenix Technologies Ltd.: *System BIOS for IBM PCs, compatibles, and EISA computers: the complete guide to ROM-based system software*, 1991

[Rus99] David A. Rusling: *The Linux Kernel*, The Linux Documentation Project, 1999, `http://www.linuxdoc.org/LDP/tlk/tlk.html`

[SG94] Abraham Silberschatz and Peter B. Galvin: *Operating System Concepts*, 4. Auflage, Addison-Wesley, 1994

[Sla98] Terry Slade: *HDTech*, Micro Firmware Technical Support, 1998, `http://www.firmware.com/support/bios/hdtech.htm`

[Ste93] W. Richard Stevens: *Advanced Programming in the UNIX Environment*, Addison-Wesley, 1993

[Suna] Sun, IBM: *JavaOS[tm] for Business[tm]/Consumers[tm]*, `http://www.sun.com/javaos`

[Sunb] Sun: *Java White Papers: Java Language Overview*, http://java.sun.com/docs/overviews/java/java-overview-1.html

[Tan90a] Andrew S. Tanenbaum: *Betriebssysteme — Entwurf & Realisierung, Teil 1: Lehrbuch*, Hanser, 1990

[Tan90b] Andrew S. Tanenbaum: *Betriebssysteme — Entwurf & Realisierung, Teil 2: MINIX-Leitfaden und kommentierter Programmtext*, Hanser, 1990

[TCT97a] Technical Committee T13 of the Accredited Standards Committee NCITS: *Information Technology — AT Attachment with Packet Interface Extension (ATA/ATAPI-4)*, Working Draft T13 1153D, 1997, `ftp://fission.dt.wdc.com/pub/standards`

[TCT97b] Technical Committee T13 of the Accredited Standards Committee NCITS: *Information Technology — AT Attachment-3 Interface (ATA-3)*, Working Draft X3T13 2008D, 1997, `ftp://fission.dt.wdc.com/pub/standards`, 1997

[Ts99] Theodore Ts'o: *E2fsprogs: Ext2 Filesystem Utilities*, 1999, `http://web.mit.edu/tytso/www/linux/e2fsprogs.html`

[Vah96] Uresh Vahalia: *UNIX Internals — The New Frontiers*, Prentice Hall, 1996

[vdL99] Peter van der Linden: *Just Java 1.2*, Sun Microsystems, Inc, 1999

[Wai87] Mitchell Waite: *UNIX papers for UNIX developers and power users*, The Waite Group, 1987