UNIX Systemverwaltung

Kristian Köhntopp

29. Juli 1996

Kapitel 1

Dateisysteme

1.1 Dateien aus der Sicht des Benutzers

Um die Ressource „Plattenplatz“ unter mehreren konkurrierenden Benutzern aufzuteilen, Zugriffsschutz zu gewährleisten und dem Benutzer zu helfen, seine Daten zu strukturieren, stellt UNIX wie andere Betriebssysteme auch *Dateisysteme* zur Verfügung.

1.1.1 Dateitypen und –attribute

Unter UNIX ist eine Datei etwas, das unter einem Namen im Dateisystem ansprechbar ist. UNIX kennt verschiedene *Typen* von Dateien. Eine Übersicht aller UNIX–Dateitypen findet sich in Abbildung 1.1.

Zeich.	Typ	Zweck
–	file	normale Datei
b	block special	Gerätedatei
c	character special	Gerätedatei
d	directory	Verzeichnis
l	symlink	Querverweis (eigentlich nur FFS)
p	pipeline	benannte Pipeline
s	socket	Netzwerkverbindung

Abbildung 1.1: Dateitypen in einem UNIX–Dateisystem

Bevor ein Prozeß auf die Daten in einer Datei zugreifen kann, muß er sie mit dem Systemaufruf `open()` öffnen. UNIX überprüft hierbei die Zugriffsrechte an der Datei und gibt dem aufrufenden Prozeß im Erfolgsfall einen *file descriptor* zurück, der für alle weiteren Operationen auf dieser Datei notwendig ist. Mit dem Systemaufruf `close()` gibt der Prozeß die Rechte an einer Datei wieder auf, die Datei wird geschlossen.

Für jede geöffnete Datei unterhält UNIX einen *Dateizeiger*, der die Position in der Datei bezeichnet, die als nächstes beschrieben oder gelesen wird. Mit dem Systemaufruf `lseek()` kann ein Prozeß die Position des Dateizeigers relativ zum Anfang oder Ende der Datei bzw. zur aktuellen Position des Dateizeigers neu definieren. Die Systemaufrufe `read()` und `write()` erlauben schließlich das Lesen und Schreiben von Daten.

Eine normale Datei dient der Speicherung von Daten. Sie enthält eine unstrukturierte, geordnete Folge von Bytes. In UNIX kennt das Betriebssystem selbst keine Satzstruktur in

Dateien, sondern es ist Aufgabe des Anwendungsprozesses, sich gegebenenfalls eine solche Struktur in der Datei zu definieren.

Dateien werden in UNIX in Verzeichnissen zusammengefaßt. Verzeichnisse sind selbst auch Dateien, die jedoch den Typ `d` haben und bestimmten Einschränkungen unterliegen: Sie haben eine bestimmte, vom Dateisystem abhängige Satzstruktur. Verzeichnisse können in UNIX nicht zum Schreiben geöffnet werden. Statt dessen kann man mit den Systemaufrufen `creat()`, `link()`, `symlink` und `unlink()` Datensätze in einem Verzeichnis erzeugen und löschen.

Auch die verschiedenen Geräte des Rechners erscheinen im Dateisystem als Dateien. Per Konvention stehen sie normalerweise im Verzeichnis `/dev`, aber das ist lediglich eine Vereinbarung und in keiner Weise zwingend. UNIX unterscheidet zwischen *block special devices*, die mit dem Kennbuchstaben `b` markiert werden, und *character special devices*, die mit `c` markiert sind. Der Unterschied zwischen beiden Gerätetypen besteht in der Verwaltung des *buffer cache*, eines internen Zwischenspeichers zur Beschleunigung von Zugriffen: Zugriffe auf block special devices werden im Cache gepuffert, Zugriffe auf character special devices werden nicht gepuffert.

Lese– und Schreiboperationen auf Gerätedateien bewirken nicht, daß die Daten irgendwo abgespeichert werden, sondern versetzen irgendein Gerät in Aktion. So verursachen beispielsweise Schreibzugriffe auf `/dev/lp`, der Gerätedatei für einen Drucker, daß die geschriebenen Daten an den Drucker gesendet werden. Lesezugriffe auf `/dev/tty1` dagegen bewirken, daß Daten von der Tastatur des Terminals `tty1` gelesen werden.

In UNIX ist es möglich, Querverweise auf andere Dateien anzulegen. Eine solches Querverweisdatei (*symbolic link* vom Typ `l` speichert selbst keine Daten, sondern zeigt auf eine andere Datei. Jeder Zugriff auf das Link wird vom Betriebssystem auf die eigentliche Datei umgelenkt.

Die Dateitypen `p` (*pipeline*) und `s` (*socket*) schließlich bezeichnen Dateien, die der Kommunikation von Prozessen dienen: Jeweils ein Prozeß versucht aus einer solchen Datei zu lesen, ein oder mehrere Prozesse können hineinschreiben. Während eine Pipeline auf Prozesse auf demselben Rechner beschränkt ist, stellt ein Socket eine Netzwerkverbindung dar.

```
-rw-r--r--   1 kris     other      1169372 Oct 31 18:07 jargon-3.0.txt
brw-r-----   1 root     operator   3,    0 Oct 22 04:38 hd0a
crw-r-----   1 root     operator  15,    0 Oct 22 04:38 rhd0a
drwxr-xr-x   7 kris     other         1024 Dec 17 15:25 Texte
lrwxrwxrwx   1 root     wheel           20 Dec 27 18:24 tex ->
                                                        /NextLibrary/TeX/tex
prw-r--r--   1 kris     other            0 Jan  4 14:06 chat
srwxrwxrwx   1 kris     other            0 Jan  4 12:58 .TeXview_Socket
```

Abbildung 1.2: Beispiele für Dateien verschiedener Typen

Außer den Informationen, die *in* einer Datei gespeichert sind, muß UNIX auch noch Informationen *über* die Datei speichern. Dazu gehören außer den Informationen, wo auf einer Festplatte denn nun konkret die Daten stehenm, auch die Attribute, die bei der Ausgabe des Befehls `ls -lsi` ausgegeben werden:

```
-rw-r--r-- 1 kris other 1169372 Oct 31 18:07 jargon-3.0.txt
```

type Der Typ der Datei wird als einzelner Buchstabe angezeigt. In Abbildung 1.1 ist eine Übersicht der möglichen Dateitypen zu sehen. Der Typ einer Datei wird beim Anlegen der Datei festgelegt und kann nicht verändert werden.

permissions Die Zugriffsrechte der Datei werden in den 12 Bit `sstrwxrwxrwx` abgespeichert. Da die ersten drei Bit der Zugriffsrechte nur sehr selten gesetzt sind, werden sie normalerweise nicht mitangezeigt. Wenn sie gesetzt sind, werden sie bei der Ausgabe an Stelle der `x`–Rechte ausgegeben. Die Zugriffsrechte können mit dem Kommando `chmod` (das vom gleichnamigen Systemaufruf Gebrauch macht) verändert werden.

link count In UNIX kann eine Datei mehr als einen Namen haben. Der Link Count gibt die Anzahl der Namen einer Datei an. Er kann mit dem Kommando `ln` erhöht und dem Kommando `rm` erniedrigt werden. Die Befehle verwenden die Systemaufrufe `link()` und `unlink()`.

owner Für den Dateieigentümer gelten die in der ersten `rwx`–Gruppe festgelegten Zugriffsrechte. Standardmäßig ist der Eigentümer einer Datei derjenige Benutzer, der die Datei angelegt hat. Dies kann jedoch auf einigen UNIX–Systemen mit dem Befehl `chown` (der den `chown()`–Systemaufruf verwendet) geändert werden.

group Für Benutzer, die in der gleichen Gruppe sind, der die Datei angehört, gelten die in der zweiten `rwx`–Gruppe festgelegten Zugriffsrechte. In System V gehört eine Datei standardmäßig der Gruppe an, in der der anlegende Benutzer gerade ist. Dieser Ansatz funktioniert in BSD–UNIX nicht, wo ein Benutzer in mehreren Gruppen zugleich sein kann. Hier erbt eine neue Datei die Gruppenzugehörigkeit des Verzeichnisses, in dem sie angelegt wird. Sie kann mit dem Befehl `chgrp` (und dem Systemaufruf `chown()`) geändert werden.

size Bei normalen Dateien und Verzeichnissen gibt diese Information die Länge der Datei in Byte an.

device numbers Bei Gerätedateien wird an Stelle der Längenangabe ein Zahlenpaar, die *major* und *minor device number*, ausgegeben. Die Major Number legt den Gerätetreiber fest, der für dieses Gerät zuständig ist. Die Minor Number wählt spezielle Unterfunktionen des Gerätetreibers aus (z.B. legt sie bei einem Diskettentreiber fest, welches Diskettenlaufwerk angesprochen wird). Ausschlaggebend bei einem Gerät sind nicht der Gerätename oder daß die Datei im Verzeichnis `/dev` steht, sondern allein die Kombination von Major und Minor Device Number.

modification time Die Modifikationszeit gibt die Zeit des letzten Schreibzugriffes auf die Daten der Datei an. Intern speichert UNIX die Zeit als Sekunden seit Beginn des Jahres 1970 und in GMT ab. Für die Ausgabe wird dies in ein besser lesbares Format in der lokalen Zeitzone umgewandelt. Die *mtime* der Datei kann von den Systemaufrufen `write()`, `utimes()` und `mknod()` gesetzt werden.

access time UNIX speichert noch zwei weitere Zeiten zu jeder Datei, die jedoch normalerweise von `ls` nicht mitangezeigt werden. Mit bestimmten Optionen kann man den Befehl jedoch veranlassen, statt der *mtime* eine andere Zeitangabe anzuzeigen. Die Zugriffszeit gibt den Zeitpunkt des letzten Lesezugriffes auf eine Datei an. Sie wird von den Systemaufrufen `read()`, `write()`, `utimes()` und `mknod()` gesetzt. Aus Effizienzgründen wird die Zugriffszeit an Verzeichnissen nicht gesetzt, wenn ein Verzeichnis durchsucht wird, obwohl man dies erwarten könnte.

change time Die Veränderungszeit gibt das Datum der letzten Statusänderung der Datei an. Sie wird immer dann gesetzt, wenn die Informationen *über* die Datei sich ändern. In einigen Texten wird die *ctime* **fälschlicherweise** als die *creation time* der Datei bezeichnet. Das ist nicht richtig: UNIX speichert das Datum der Erzeugung einer Datei nicht. Statt dessen wird die *ctime* von den Systemaufrufen `chmod()`, `chown()`, `link()`, `mknod()`, `rename()`, `unlink()`, `utimes()` und `mknod()` neu gesetzt.

name Der Name einer Datei ist schließlich der Bezeichner, unter dem das Betriebssystem die Daten der Datei für einen Prozeß verfügbar macht. Zugriffe auf Dateien sind in UNIX ausschließlich über den Dateinamen und den `open()`–Systemaufruf möglich. Der Name einer Datei kann mit dem Kommando `mv` (und dem Systemaufruf `rename()`) geändert werden.

1.1.2 Eine typische Dateisystemstruktur

In UNIX befindet sich jede Datei in irgendeinem Verzeichnis. Da Verzeichnisse selbst auch wieder Dateien sind, können sie ebenfalls in Verzeichnissen enthalten sein. Durch diese Verschachtelungsmöglichkeit ergibt sich die typische, baumartige Verzeichnisstruktur eines UNIX–Systems. Im Gegensatz zu MS-DOS hat UNIX nur einen Dateibaum. Mit dem Kommando `mount` werden die verschiedenen Platten und Laufwerke des Systems in leere Verzeichnisse des Dateibaums eingehängt. Bei dem Wechsel in ein gemountetes Verzeichnis wechselt ein Benutzer so automatisch auch die Platte, auf der er arbeitet. Mit dem Kommando `umount` werden Platten wieder beim System abgemeldet.

Der Ausgangspunkt des UNIX–Dateibaumes ist das Wurzelverzeichnis (*root directory*. Es wird mit dem Namen `/` bezeichnet. Wie in der Informatik bei Baumstrukturen üblich, wird die Wurzel des Dateibaumes in Diagrammen normalerweise oben oder links eingezeichnet.

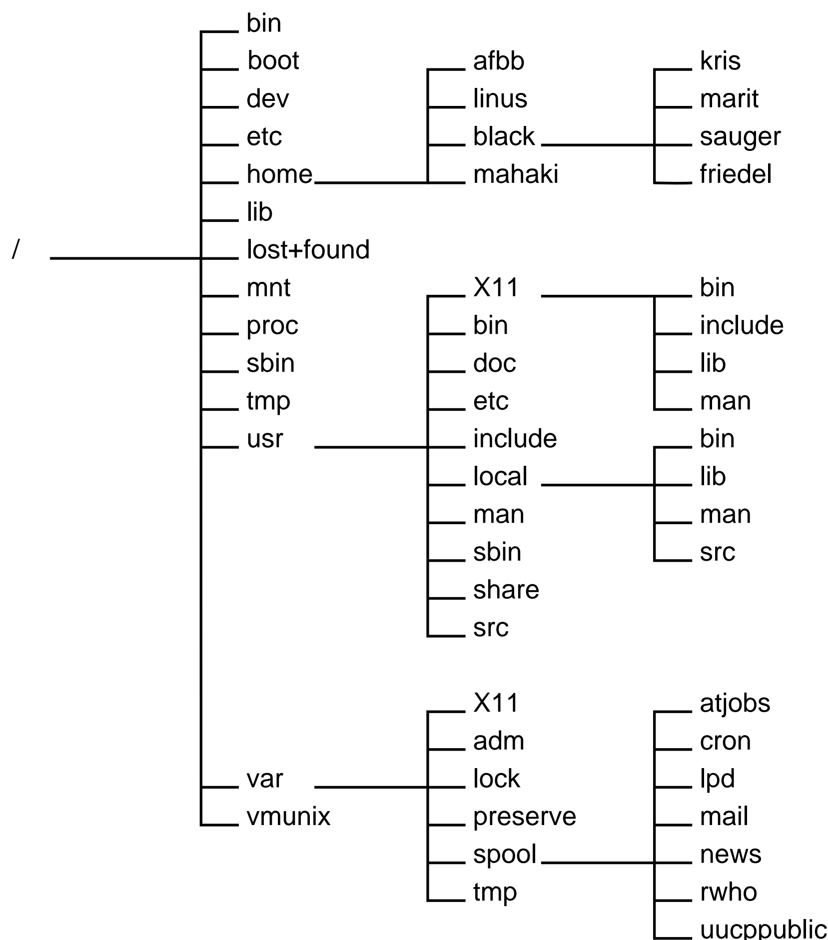


Abbildung 1.3: Ein UNIX–Dateibaum

Abbildung 1.3 zeigt den Aufbau des Dateibaums bei einem modernen UNIX. Unterhalb des Hauptverzeichnisses befinden sich außer dem Betriebssystem selbst (oft heißt die Betriebssystemdatei `/unix` oder `/vmunix`) weitere Verzeichnisse.

/bin Im Verzeichnis befinden sich ausführbare Programme für jeden Benutzer. Hier befinden sich die allernotwendigsten Systembefehle zum Einloggen, zum Arbeiten mit Dateien, die Shells und andere grundlegende Befehle.

/boot Das Verzeichnis enthält die zum Booten des Systems notwendigen Befehle und Konfigurationsdateien. Diese sind oftmals systemspezifisch und unterscheiden sich von UNIX zu UNIX sehr stark.

/dev Das Verzeichnis enthält traditionell alle Gerätedateien des Systems. Für jedes angeschlossene Gerät findet sich hier eine block special oder character special Datei, mit der man es ansprechen kann.

/etc Im Verzeichnis befinden sich Konfigurationsdateien und –Datenbanken, die vom Systemverwalter angepaßt werden können. Bei einigen UNIX–Versionen sind hier auch ausführbare Befehle für den Systemverwalter hineingerutscht.

/home Das Verzeichnis enthält entweder direkt die Homeverzeichnisse der Benutzer oder für jeden Rechner eines Netzwerkes ein Unterverzeichnis. Platten mit den Home–Verzeichnissen anderer UNIX–Systeme können dann im passenden Unterverzeichnis eingeblendet werden.

/lib Das Verzeichnis enthält die Bibliotheken, die zum Übersetzen und Linken von ausführbaren Maschinenprogrammen benötigt werden. Bei einigen UNIX–Versionen sind hier auch ausführbare Programme hineingerutscht, die nicht direkt vom Benutzer angewendet werden sollen, sondern von UNIX–Befehlen im Hintergrund aufgerufen werden.

/lost+found Wenn nach einen Systemabsturz die Struktur des Dateisystems beschädigt worden ist, versucht UNIX, das Dateisystem mit dem Kommando `fsck` (*file system check*) zu reparieren. Dabei kann es vorkommen, daß verlorene Dateien wiedergefunden werden. Diese Dateien werden von `fsck` im Verzeichnis /lost+found abgelegt, wo der Systemverwalter sich dann um sie kümmern kann.

/mnt Dieses Verzeichnis ist normalerweise leer und dient als Anmeldepunkt für Dateisysteme, die nur vorübergehend verfügbar sind.

/proc Neuere UNIX–Versionen sind in der Lage, die Prozeßliste des Rechners als Pseudodateien einzublenden. Ein solches Prozeßdateisystem erleichtert die Entwicklung von systemunabhängigen Werkzeugen zur Ausgabe von Systemstatusinformationen.

/sbin Das Verzeichnis enthält eine Reihe von Kommandos, die für die Konfiguration des Systems notwendig sind. Ihre Ausführung ist meistens dem Systemverwalter vorbehalten.

/tmp In diesem Verzeichnis, das für alle Benutzer frei beschreibbar ist, können Zwischendateien angelegt werden.

/usr Das `/usr`–Verzeichnis liegt bei vielen Konfigurationen auf einer gesonderten Festplatte. Es enthält eine Reihe von Unterverzeichnissen, die weitere Systembefehle, Zusatzdateien und Subsysteme enthalten.

/var Bei älteren UNIX–Systemen sind unterhalb von `/usr` statische und veränderliche Dateien sowie exportierbare Dateien und lokale Konfigurationsdateien bunt gemischt.

Bei modernen UNIX–Systemen hat man versucht, die Konfigurationsdateien konzentriert in das `/etc`–Verzeichnis auszulagern. Veränderliche Dateien hat man nach `/var` verlegt. Dadurch wird ermöglicht, den `/usr`–Dateibaum in ein Netzwerk zu exportieren und zwischen mehreren UNIX–Systemen zu teilen.

Das wichtigste Unterverzeichnis von `/var` ist `/var/spool`, in dem der Druckerspooler, das Mailsystem und das Newssystem sowie andere Hintergrundprogramme ihre Zwischendateien ablegen.

1.2 Dateien aus der Sicht des Betriebssystems

Während Dateisysteme sich aus der Sicht des Benutzers hauptsächlich durch die Anordnung der Dateien, Dateinamenskonventionen und Dateibearbeitungsmöglichkeiten auszeichnen, kommt es aus der Sicht des Betriebssystems hauptsächlich darauf an, wie die Daten auf einer Platte verwaltet werden.

1.2.1 Hardware

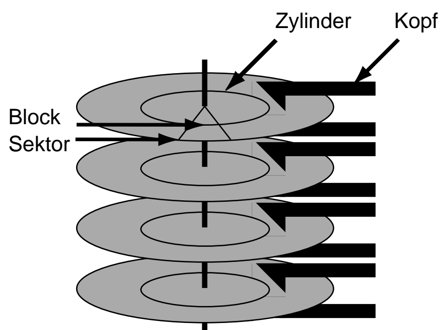


Abbildung 1.4: Aufbau einer Festplatte

Festplatten für PCs und Workstations bestehen aus einer oder mehreren Metallplatten, die beidseitig mit hochfein polierten Metalloxiden beschichtet sind. Ein Kamm von beweglichen Schreib/Leseköpfen greift seitlich in den rotierenden Plattenstapel hinein und kann so konzentrische Kreise von Magnetisierungsmustern abtasten oder erzeugen.

Dementsprechend ist das Koordinatensystem einer Festplatte aufgebaut, dem man folgen muß, wenn man Daten auf der Platte lokalisieren möchte. Abbildung 1.4 illustriert dies: Der Kamm der Schreib/Leseköpfe erzeugt konzentrische Kreise auf der Oberfläche des Mediums. Denkt man sich den Rest der Platte weg, erscheinen diese Kreise als ein Zylinder im Raum.

Dieser Zylinder besteht aus mehreren Spuren, von denen jede von jeweils einem Schreib–/Lesekopf erzeugt wird. Eine solche kreisförmige Spur wird dann in Kreissegmente, die Sektoren unterteilt. Durch die Angabe von Zylinder, Kopf und Sektor kann man so genau einen Abschnitt einer Festplatte, den Block, adressieren. Der Block stellt die kleinste adressierbare Einheit einer Festplatte dar, d.h. zur Änderung eines einzelnen Bits muß immer ein ganzer Block gelesen und wieder geschrieben werden.

Aus den Informationen über die Geometrie einer Festplatte kann man die Kapazität der Platte berechnen. Ist außerdem noch die Rotationsgeschwindigkeit des Plattenstapels bekannt, kann man zusätzlich die maximale Datenrate der Festplatte ausrechnen, indem man bestimmt, wie viele Blöcke pro Sekunde an einem Kopf vorbei rotieren. Diese Datenrate

Zylinder pro Platte	617	2048
Spuren pro Zylinder	4	15
Sektoren pro Spur	17	37
Bytes pro Sektor	512	512
Gesamtkapazität	etwa 20 MB	etwa 550 MB
Byte pro Spur	8704	21504
Umdrehungen pro Minute	3600	4680
Maximale Datenrate	510 KB/sec	1638 KB/sec

Abbildung 1.5: Geometrie einer älteren und einer modernen Platte

stellt eine theoretische Obergrenze für den Datenfluß von der Plattenoberfläche zum Kopf dar und kann nicht überschritten werden.

1.2.2 Partitionen

Für einige Zwecke kann es nützlich oder notwendig sein, statt einer großen Festplatte mehrere kleinere Platten zu haben. So möchte man eventuell auf einem Rechner mehr als ein Betriebssystem installieren und bei Bedarf zwischen den verschiedenen Installationen hin– und her wechseln können. Auch wenn man nur ein einzelnes Betriebssystem verwendet, kann es nützlich sein, eine Festplatte aufzuteilen, etwa um die Datensicherung zu vereinfachen oder verschiedene Dateisystemtypen oder verschieden wichtige Daten voneinander zu isolieren.

Die meisten Betriebssysteme stellen dazu den Mechanismus der *Partitionierung* von Festplatten zur Verfügung. Eine *Partition* ist dabei ein zusammenhängender Abschnitt von Zylindern einer Platte. So könnte man sich die Platte aus Abbildung 1.4 in zwei Partitionen unterteilt vorstellen. Die erste Partition würde auf den äußeren Zylindern beginnen und sich bis zum markierten Zylinder erstrecken, die zweite würde dort beginnen und die gesamten inneren Zylinder der Platte umfassen.

Partitionen einer Festplatte dürfen sich niemals überlappen. Sie stoßen im Normalfall nahtlos aneinander. Normalerweise wird man eine Festplatte vollständig partitionieren, d.h. keinen Platz ungenutzt liegen lassen.

Eine Partition einer Festplatte wirkt auf ein Betriebssystem dann wie eine einzelne, kleine Festplatte. Auf einer Partition kann in der Regel genau ein Dateisystem eingerichtet werden[1] bzw. ein Betriebssystem installiert werden. Dadurch ist es beispielsweise möglich, auf einem PC mit einer Festplatte in einer Partition MS–DOS und in einer anderen Partition eine UNIX–Version zu installieren. Mit einem speziellen Ladeprogramm kann man beim Einschalten des Rechners angeben, von welcher Partition man booten möchte.

Partitionen werden im PC–Bereich in der Regel mit dem Programm `fdisk` erzeugt, das mit dem jeweiligen Betriebssystem mitgeliefert wird. Es hat sich als notwendig herausgestellt, die Partitionen des jeweiligen Betriebssystems mit seinem eigenen `fdisk`–Programm zu erstellen. Falls ein Rechner wahlweise mit MS–DOS oder OS/2 und UNIX betrieben werden soll, ist es eine gute Idee, die Partitionen für MS–DOS und OS/2 anzulegen, bevor man die UNIX–Version auf einem PC installiert. Zur Installation eines PC mit MS–DOS und Linux wird man also zunächst eine Partition für MS–DOS mit dem `fdisk`–Programm von MS–DOS anlegen und MS–DOS installieren. Danach wird man Linux booten und mit dem `fdisk`–Programm von Linux die Linux–Partitionen anlegen und Linux installieren.

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

Bei PCs wird die Partitionierung einer Festplatte am Anfang der Platte in einer *Partitionstabelle* festgehalten. Ursprünglich konnte diese Tabelle genau vier Einträge enthalten, d.h. eine physikalische Festplatte konnte in maximal vier verschiedene logische Laufwerke, die sogenannten primären Partitionen, unterteilt werden. Durch die Einführung größerer Festplatten und durch die Verfügbarkeit von mehr unterschiedlichen Betriebssystemen war man später gezwungen, dieses Verfahren zu erweitern. Man muß eine primäre Partition opfern, um sich eine sogenannte erweiterte Partition zu definieren. Diese läßt sich dann in beliebig viele Partitionen unterteilen. Leider lassen sich jedoch viele Betriebssysteme nur aus einer primären Partition booten.

1.2.3 Fragmentierung

Dadurch, daß Festplattenplatz nur in Einheiten zu jeweils einem Block zur Verfügung steht, geht natürlich Platz verloren. Eine Datei, die nur ein einzelnes Byte belegt, wird trotzdem die Mindestgröße von einem Plattenblock belegen und damit bei einer Blockgröße von 512 Byte 511 Byte verschwenden. Wenn man annimmt, daß Dateigrößen (modulo Blockgröße) gleichmäßig verteilt sind, wird im Mittel pro Datei jeweils ein halber Block verschwendet. Bei angenommenen 20000 Dateien auf einer Platte und bei einer Blockgröße von 512 Byte werden etwa 5 MB mehr verbraucht. Faßt man mehrere physikalische Plattenblöcke zu logischen Blöcken zusammen, steigt der Platzverlust durch diese sogenannte *interne Fragmentierung* stark an. Im Beispiel 1.6 wird eine Festplatte von 768 MB Größe mit 20000 Dateien zugrunde gelegt. Man sieht leicht, daß die Blockgröße einer Festplatte für eine optimale Platzausnutzung möglichst klein sein sollte.

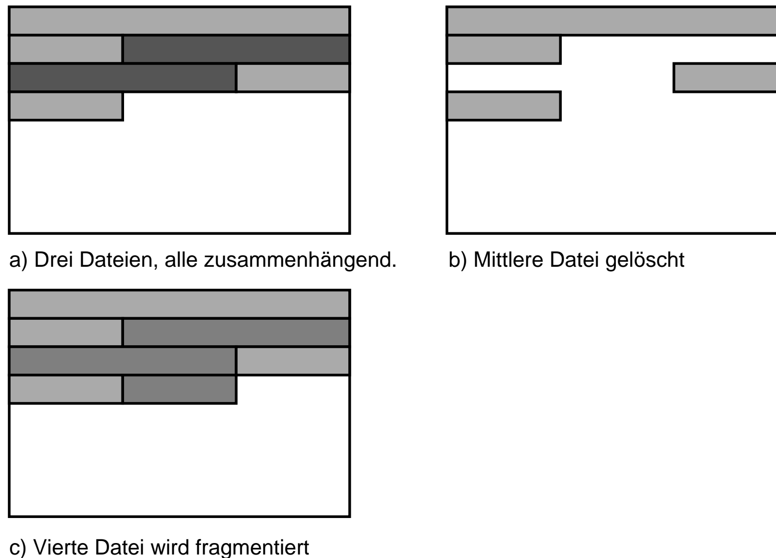
logische Blockgröße	verlorene KB	% Verlust
512	5000	0.00
1024	10000	1.00
2048	20000	2.00
4096	40000	5.00
8192	80000	10.00

Abbildung 1.6: Plattenplatzverlust durch interne Fragmentierung

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

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

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



a) Drei Dateien, alle zusammenhängend.
b) Mittlere Datei gelöscht
c) Vierte Datei wird fragmentiert

Abbildung 1.7: Externe Fragmentierung

1.3 Das klassische System V–Dateisystem

1.3.1 Dateien

Im ursprünglichen UNIX Dateisystem wurde mit einer logischen Blockgröße von 512 Byte gearbeitet. In heutigen System V–Dateisystemen wird mit Blöcken von 1024 Byte operiert, aber die grundlegenden Datenstrukturen sind noch immer gleich. Bei der Einrichtung eines Dateisystems wird im ersten Block ein sogenannter *Superblock* installiert. Er verwaltet Informationen, die das Dateisystem beschreiben. Dazu gehört die Angabe der Größe des Dateisystems, die Anzahl der freien Blöcke und so weiter.

Der folgende, vordere Teil der Festplatte für die *I–Nodetabelle* reserviert. Diese Tabelle stellt ein Array von *I–Node*–Datenstrukturen dar. Eine einzelne I–Node speichert dabei alle Verwaltungsinformationen über eine Datei mit Ausnahme des Dateinamens. Der Name I–Node habe dabei keine besondere Bedeutung. Er kommt von dem Wort *Index–Node* und soll andeuten, daß man auch die I–Nodes mit einem Index wie auf eine Tabelle zugreifen kann.

Für jede Datei existiert genau eine I–Node, aber andersherum muß für jede Datei, die angelegt werden soll, auch genau eine I–Node vorhanden sein. Da die Größe der I–Nodetabelle beim Anlegen des Dateisystems festgelegt wird und auch später nicht mehr geändert werden kann, ohne das Dateisystem dabei zu zerstören, muß ein Systemverwalter beim Anlegen von Dateisystem schon eine ausreichende Anzahl von I–Nodes miteinplanen. Zum Glück ist eine I–Node eine relativ kompakte Datenstruktur. Im klassischen System V Dateisystem belegt eine I–Node nur 64 Byte, so daß jeweils 16 I–Nodes in einen Datenblock passen. Normalerweise wird man pro 4 Kilobyte Plattenplatz jeweils eine I–Node reservieren, d.h. man wird mit einer mittleren Dateilänge von 4 KB rechnen. Der Verwaltungsaufwand beträgt dann genau $4096/64 = 1/64 = 1.6\%$ des Plattenplatzes.

Wenn entweder der verfügbare Plattenplatz oder die I–Nodes erschöpft sind, meldet das Betriebssystem sich mit der Fehlermeldung `disk full`. Unter UNIX kann also der bizarre Fall eintreten, daß auf einer Platte noch viele Megabytes an Platz verfügbar wären, aber nicht genutzt werden können, weil der Systemverwalter nicht genügend I–Nodes für das betroffene Dateisystem vorgesehen hat. In einem solchen Fall hilft nur das Sichern des

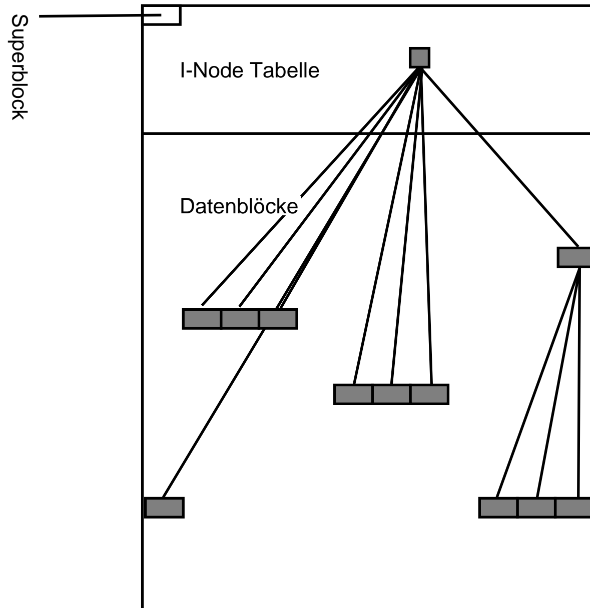


Abbildung 1.8: Layout eines System V–Dateisystems

Platteninhaltes, Neuanlegen des Dateisystems mit mehr I–Nodes und Zurückspielen der Datensicherung.

Eine I–Node, wie sie in Abbildung 1.9 dargestellt ist, enthält bis auf den Dateinamen alle Informationen *über* eine Datei: Dateityp, Zugriffsrechte, Anzahl der Dateinamen[2], numerische ID des Eigentümers und der Gruppe der Datei, Größe der Datei in Byte sowie die drei Zeitangaben an der Datei sind hier gespeichert. Im zweiten Teil der I–Node vermerkt das System, wo auf der Platte die Daten *in* der Datei gespeichert sind.

Kleine Dateien sind häufiger als große Dateien. Die Nummern der ersten zehn Datenblöcke sind direkt in der I–Node selbst vermerkt. Auf diese Weise kann nach dem Lesen der I–Node einer Datei direkt auf die Daten einer kleinen Datei zugegriffen werden, ohne noch weitere Blöcke laden zu müssen. Für große Dateien ist ein solches Verfahren natürlich nicht praktikabel, denn es würde die Datenstruktur der I–Node aufblähen. Wächst eine Datei über die Größe von 10 Datenblöcken hinaus, besorgt sich das Betriebssystem einen zusätzlichen freien Datenblock, in dem es die Nummern weiterer 256 Datenblöcke hinterlegt. Dieser Datenblock enthält also selber keine Daten, sondern die Blocknummern von Datenblöcken. Er wird *single indirect block* genannt.

Mit zehn *direct blocks* und einem *indirect block* kann man Dateien von bis zu 266 Datenblöcken Größe verwalten. Wächst die Datei noch weiter, so wird ein *double indirect block* belegt. Dieser Block speichert die Blocknummern von bis zu 256 indirect Blocks, die wiederum jeweils die Nummern von 256 Datenblöcken enthalten können. Auf diese Weise kann man Dateien von bis zu 65802 Blöcken Größe verwalten. Für noch größere Dateien muß ein *triple indirect block* beschafft werden, der die Nummern von 256 doppelt indirekten Blöcken speichert, die wiederum jeweils auf 256 einfach indirekte Blöcke zeigen, von denen jeder wieder auf 256 Datenblöcke zeigen kann. Insgesamt kann man so mehr als 16 Millionen Datenblöcke verwalten. In der Praxis ist die maximale Größe einer Datei aber schon durch das 4 Byte breite Größenfeld in der I–Node (und beim `lseek()` Systemaufruf) auf maximal 4 Gigabyte begrenzt. Für einige Datenbankanwendungen ist dies eine echte Einschränkung. Hersteller von UNIX–Großrechnern, die derart riesige Da-

[2] In UNIX kann eine Datei mehr als einen Namen haben.

Feld	Größe	Summe
Dateiattribute		
Dateityp	0.5	0.5
Permissions	1.5	2
Link Count	2	4
Owner	2	6
Group	2	8
Size (Byte)	4	12
Access Time	4	16
Modification Time	4	20
Change Time	4	24
Datenblöcke		
10 Datenblöcke	30	54
1 Single Indirect	3	57
1 Double Indirect	3	60
1 Triple Indirect	3	63

Abbildung 1.9: Daten in einer I–Node

teien verwalten müssen, haben dafür dann aber auch Abhilfe geschaffem. Abbildung 1.10 zeigt die baumartig verästelte Struktur, die sich aus der Verzeigerung der verschiedenen Indirektionsebenen ergibt.

Wenn ein Prozeß eine Datei zum Lesen oder Schreiben öffnet, assoziiert das Betriebssystem einen *Dateizeiger* in die Datei mit diesem Prozeß. Jeder Lese– und Schreibzugriff bewegt den Dateizeiger in der Datei um die entsprechend übertragene Anzahl von Bytes nach vorne[3]. Das Betriebssystem kann aus der Position des Dateizeigers direkt den Datenblock bestimmen, der von der Operation betroffen ist: Die Blocknummer ergibt sich, indem man die Position des Dateizeigers durch die Blockgröße dividiert. Aus der Blocknummer kann man dann direkt ablesen, ob es sich um einen direkt oder einfach bzw. mehrfach indirekt erreichbaren Datenblock handelt.

Dadurch ist es möglich, eine Datei nicht zusammenhängend zu beschreiben. Das folgende Programm schreibt z.B. ein Integer am Anfang einer Datei und ein Byte an die Byteposition 8192:

```
#include <sys/file.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/types.h>
#include <unistd.h>

main()
{
        int fd;
        int wert = 17;

        fd = open("sparsefile", O_WRONLY | O_CREAT, 0666);

        lseek(fd, 0, SEEK_SET);
        write(fd, &wert, sizeof(wert));
```

[3]Mit dem Systemaufruf `lseek()` kann der Dateizeiger ausdrücklich an eine gewünschte Position bewegt werden. Diese Position kann auch jenseits des bisherigen Dateiendes liegen.

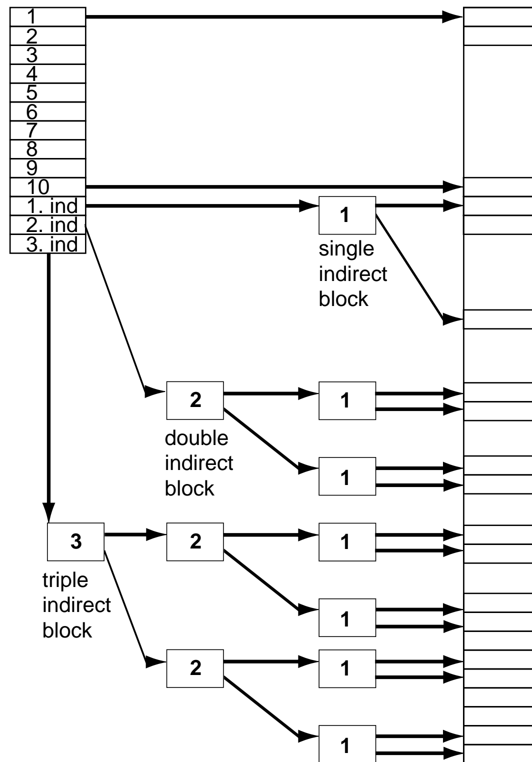


Abbildung 1.10: I–Node und Datenblöcke

```
        lseek(fd, 8192, SEEK_SET);
        write(fd, &wert, sizeof(wert));

        close(fd);
}
```

Eine auf solche Weise erzeugte Datei belegt tatsächlich nur die beschriebenen Blöcke. Die Datei im Beispiel belegt also nur 2 Datenblöcke auf der Festplatte: Den direct Block 0, um das Byte an der Position 0 zu speichern, und den direct Block 8, um das Byte an der Position 8192 zu speichern. Die Blocknummern der direct Blocks 1 bis 7 und 9 sowie die Nummern aller indirect Blocks in der I–Node der Datei sind auf den Wert 0 gesetzt. Liest man diese Datei jetzt sequentiell durch, werden für die nicht vorhandenen Blöcke Nullbytes zurückgegeben.

Eine Datei, die nicht zusammenhängend beschrieben worden ist und so weniger Plattenblöcke belegt als man ihrer Länge nach erwarten würde, heißt *sparse file* (*dünn besetzte Datei*). Die Coredumps von fehlerhaften Programmen sind in einigen UNIX–Versionen solche Dateien, shared Libraries in Linux und die Datenbanken einiger `libdbm`–Bibliotheken sind ebenfalls dünn besetzt.

Dünn besetzte Dateien machen Probleme beim Sichern auf Band oder beim Kopieren: Normale Datensicherungsprogramme treffen keine Vorkehrungen zum Erkennen und Erhalten von dünn besetzten Dateien, sondern lesen Dateien naiv von vorne nach hinten durch. Dadurch, daß beim Lesen eines sparse Files Nullbytes gelesen werden, wird nicht zwischen Folgen von tatsächlich vorhandenen Nullbytes und Lücken in der Datei unterschieden. Die Kopie bzw. die Sicherung der Datei wird durchgehend beschrieben und belegt so wirklich so viele Blöcke, wie sie ihrer Längenangabe belegen müßte. Das kann natürlich dazu führen,

daß auf der Zielplatte oder dem Sicherungsband nicht genügend Platz vorhanden ist, um die jetzt aufgeblähte Datei zu speichern.

Einige Programme, namentlich der `GNU-tar`–Befehl von der FSF, treffen auf Wunsch besondere Vorkehrungen zum Erkennen und Behandeln von dünn besetzten Dateien. Wenn man also ein sparse File nicht mit `mv` umbenennen kann, etwa weil die Umbenennung über die Grenze eines Dateisystems hinaus führen würde, sollte man eine Kombination von zwei `GNU-tar`–Kommandos verwenden. Wird die Datei auf diese Weise vorschoben, kann man die Struktur erhalten.

```
$ gtar -cSf - ./sparsefile | ( cd ziel; gtar xSf -)
```

Abbildung 1.11: Verschieben einer dünn besetzten Datei mit `GNU-tar`

1.3.2 Verzeichnisse

Bisher haben wir uns nur mit I–Nodes und Dateien beschäftigt, aber diese Dateien haben noch keinen Namen und sind auch noch nicht in der baumförmigen Verzeichnisstruktur angeordnet, die man von UNIX kennt.

In UNIX ist ein Verzeichnis eine Datei von einem bestimmten Typ (Typ `d` für *directory*) und besitzt einen festgelegten Aufbau. Dadurch, daß ein Verzeichnis eine Datei ist, hat jedes Verzeichnis eine I–Node und Datenblöcke wie jede andere Datei auch. Ein Verzeichnis ist jedoch eine besondere Datei, an deren Aufbau das Betriebssystem einige Anforderungen stellt: Im klassischen Dateisystem von System V besteht ein Verzeichnis aus einer Folge von Datensätzen zu jeweils 16 Byte. Die ersten zwei Byte enthalten die I-Nodenummer einer Datei, die folgenden 14 Byte speichern den Namen der Datei. Ist der Dateiname kürzer als 14 Zeichen, wird er mit Nullbytes aufgefüllt. Der erste Datensatz eines Verzeichnisses ist immer der Name `.` (Punkt), der die I-Nodenummer des Verzeichnisses selbst enthält. Der zweite Eintrag ist immer der Name `..` (Punkt Punkt), der die I-Nodenummer des übergeordneten Verzeichnisses speichert.

17	. (Punkt)
18	.. (Punkt Punkt)
47	.profile
118	sparsefile

Abbildung 1.12: Verzeichnis im System V Dateisystem

Dadurch, daß alle Informationen über die Datei in der I–Node gespeichert sind, ist es möglich, mehr als einen Verzeichniseintrag für eine Datei zu haben, ohne Inkonsistenzen befürchten zu müssen. Weitere Verzeichniseinträge für eine Datei erzeugt man mit dem Systemaufruf `link()`, der den alten und einen neuen Namen für die Datei mitgeteilt bekommen muß. Der Aufruf schlägt die I–Nodenummer der Datei über den alten Namen nach und erzeugt einen Eintrag für den neuen Namen einer Datei mit derselben I–Nodenummer. Beide Verzeichniseinträge verweisen auf dieselbe I–Node, daher haben beide Dateien bis auf den Namen dieselben Attribute. Beide Namenseinträge sind gleichberechtigt, d.h. es ist

nachträglich nicht feststellbar, welcher Name der Datei der originale Name war. Mit dem Systemaufruf `unlink()` können Namen einer Datei entfernt werden. Sobald eine Datei keine Namen mehr hat, also ihr Link Count–Feld in der I–Node auf den Wert 0 sinkt, wird der Speicherplatz für die Datei wieder freigegeben.

Beim Öffnen einer Datei muß das Betriebssystem eine Verzeichnisdatei öffnen und durchlesen. Wenn es den gesuchten Namen gefunden hat, kann es die I–Nodenummer der gewünschten Datei ablesen und mit Hilfe dieser auf die Datei zugreifen. Tatsächlich werden UNIX aber keine Dateinamen übergeben, sondern Pfadnamen, die sich aus mehreren Dateinamenskomponenten zusammensetzen. In UNIX übernimmt eine betriebssysteminterne, zentrale Funktion, `namei()`, die Umwandlung von Pfadnamen in I–Nodenummern.

Pfadnamen sind entweder *absolute Pfadnamen*, wenn sie mit einem Slash beginnen, oder *relative Pfadnamen*. Absolute Pfadnamen werden relativ zur Wurzel des Dateisystems ausgewertet, relative Pfadnamen relativ zum aktuellen Verzeichnis. Die I–Nodenummer der Dateisystemwurzel und des aktuellen Verzeichnisses sind dabei in der Prozeßstruktur desjenigen Prozesses gespeichert, der den Pfadnamen an das Betriebssystem übergeben hat. Auf diese Weise kann jeder Prozeß ein eigenes aktuelles Verzeichnis, aber auch ein eigenes Wurzelverzeichnis haben.

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

Auf diese Weise können beliebig lange Pfade aufgelöst werden. Abbildung 1.13 zeigt, wie die I–Nodes dabei auf die Datenblöcke eines Verzeichnisses zeigen, während die Datenblöcke des Verzeichnisses wiederum auf I–Nodes zurückverweisen. Die sich so ergebende rekursive Struktur springt zwischen dem I-Nodebereich und dem Datenbereich einer Platte hin und her. Wie jede endliche, rekursive Struktur muß auch diese in einem Punkt verankert sein. Dieser Ankerpunkt ist die I–Nodenummer des Wurzelverzeichnisses oder des aktuellen Verzeichnisses in der Prozeßstruktur.

Beim System V–Dateisystem liegen die I–Nodes konzentriert am Anfang einer Festplatte und belegen in der Regel weniger als 2 Prozent des gesamten Plattenplatzes. Die Datenblöcke von Verzeichnissen können irgendwo auf dem Rest der Platte liegen. Deswegen sind beim Auflösen von Pfadnamen unter Umständen mehrere Suchoperationen und umfangreiche Kopfbewegungen notwendig. Ein Aufruf von `namei()` zur Auflösung eines Pfadnamens ist also eine vergleichsweise teure Operation.

Für eine effiziente Abwicklung von `namei()` ist daher der Plattencache von UNIX ganz entscheidend. Datenblöcke und I–Nodes, die kürzlich erst im Zugriff waren, bleiben im RAM des Rechners zwischengespeichert und müssen bei erneuter Anforderung nicht noch einmal von der Festplatte gelesen werden. Dadurch wird insbesondere das Öffnen von Dateien sehr stark beschleunigt[4].

[4] Dieses Problem ist nicht auf UNIX beschränkt, sondern tritt in dieser Form auch bei anderen Betriebssystemen auf. Unter DOS hat man sich vor der Einführung von `SMARTDRV` mit dem Namenscache `FASTOPEN` beholfen, der eine Tabelle von kompletten Pfadnamen und Plattenblöcken gespeichert hat. Wenn auf einen Pfadnamen wiederholt zurückgegriffen wurde, war es dadurch nicht mehr notwendig, einen kompletten Durchlauf durch die Verzeichnisse zu machen. Stattdessen stand das Resultat dieser Dateisuche bereits abrufbereit im Cache. Der Ansatz von UNIX mit einem universellen Plattencache ist allerdings genereller und dann bei DOS auch übernommen worden.

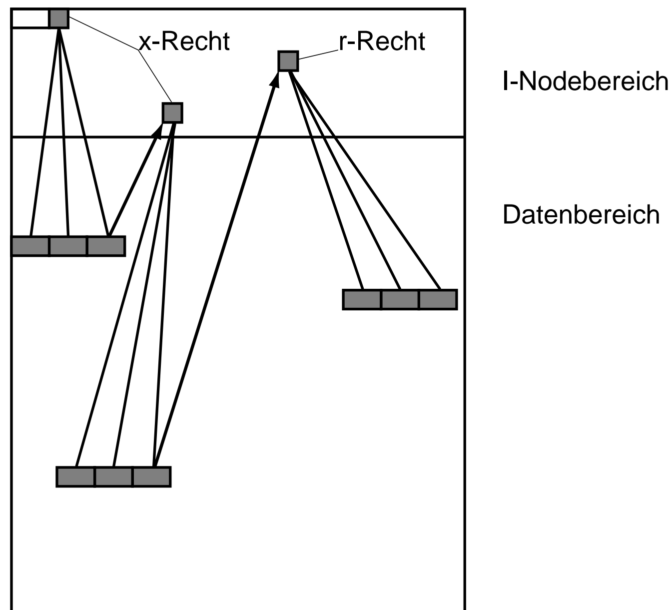


Abbildung 1.13: Auflösung eines absoluten Pfadnamens

1.3.3 Zugriffsrechte

Betrachtet man die interne Struktur von UNIX, stellt man fest, daß das betriebssystemintern ausschließlich mit den I–Nodenummern von Dateien arbeitet. Alle Betriebssystemaufrufe verwenden jedoch durchgehend Pfadnamen. Der Funktion `namei()` kommt bei der Umwandlung von Pfadnamen in I–Nodenummern die tragende Rolle zu. Sie ist auch ein idealer Ansatzpunkt um Zugriffsrechte zu kontrollieren.

In UNIX werden Zugriffsrechte auf Dateien nur beim Öffnen der Datei kontrolliert. Die Auswertung der Rechte erfolgt dabei in zwei Phasen. Wenn ein Programm einen Systemaufruf wie

```
open("/tmp/sparsefile", O_RDONLY);
```

ausführt, muß zunächst der Dateiname in eine I–Nodenummer umgewandelt werden. `open()` überläßt diese Aufgabe selbstverständlich `namei()`. Um den Namen zu wandeln, muß `namei()` zweimal eine I–Nodenummer aus einem Verzeichnis entnehmen: Einmal muß die I–Nodenummer von `tmp` aus dem Verzeichnis `/` ausgelesen werden und einmal muß die I-Nodenummer von `sparsefile` im Verzeichnis `tmp` gesucht werden. Dies entspricht jeweils den aufwärts zeigenden Pfeilen in Abbildung 1.13. `namei()` erlaubt diese Aufwärtsbewegung jedoch nur, wenn der aufrufende Prozeß ein x–Recht an den enthaltenden Verzeichnissen hat, andernfalls wird die Ausführung des Systemaufrufes mit der Fehlermeldung `permission denied` sofort abgebrochen.

Durch das Entziehen von x–Rechten an einem Verzeichnis kann man also pauschal den Zugriff auf alle Dateien in darunterliegenden Verzeichnissen sperren, unabhängig von den Rechten der einzelnen Dateien.

Falls der Aufrufer im Beispiel oben die beiden benötigten x–Rechte hat, gelingt der `namei()`–Aufruf und `open()` kennt jetzt die I–Nodenummer der gewünschten Datei. Es wird die Datei aber nur dann öffnen, wenn der Aufrufer die im Aufruf gewünschten Rechte (hier: r–Recht) an der Datei hält. Wenn dies auch der Fall ist, gelingt der Systemaufruf und gibt einen gültigen Filedescriptor zurück. UNIX garantiert mit dem Filedescriptor

den angeforderten Zugriff auf die Datei. Auch wenn ein anderer Prozeß versucht, die Datei zu löschen, oder ein neues Laufwerk über das enthaltende Verzeichnis gemountet wird, hat dies auf den Prozeß mit der bereits geöffneten Datei keine Auswirkung. Erst mit einem `close()`–Systemaufruf gibt ein Prozeß seine Rechte an einer Datei wieder an das System zurück.

Auch das w–Recht an Verzeichnissen hat eine besondere Bedeutung. In UNIX ist es nicht möglich, ein Verzeichnis zum Schreiben zu öffnen. Wäre dem nicht so, könnte man ja mit einem passenden Programm willkürlich I–Nodenummern in ein Verzeichnis eintragen und so den Zugriffsschutz durch die x–Rechte umgehen. Verzeichnisse können in UNIX nur mit wenigen ausgewählten Operationen manipuliert werden.

creat() Der `creat()`–Systemaufruf (bzw. seine in `open()` integrierte Variante) erzeugt eine neue, gewöhnliche Datei. Er bringt den Link Count einer Datei also von Null auf Eins und erzeugt den ersten Verzeichniseintrag der Datei.

mkdir() Der Aufruf `mkdir()` leistet ähnliches für das Anlegen von Verzeichnissen. Er legt jedoch gleichzeitig die Einträge für `.` und `..` im neuen Verzeichnis an. In alten Versionen von UNIX war `mkdir()` nicht als Systemaufruf realisiert und wurde durch `mknod()` simuliert.

mknod() Der Aufruf `mknod()` dient zum Anlegen von neuen Dateien beliebigen Typs. Er kann auch Gerätedateien erzeugen und ist deswegen nur für den Systemverwalter freigegeben.

symlink() Mit `symlink()` erzeugt man eine Querverweisdatei, ein *symbolic link*. Ein Zugriff auf eine solche Datei vom Typ `l` wird auf die Datei, auf die der Querverweis zeigt, umgelenkt. In einem normalen System V–Dateisystem sind Querverweise dieser Art nicht möglich; sie sind erst mit dem BSD FFS eingeführt worden. Einige UNIX–Hersteller haben ihr altes Dateisystem jedoch mit Symbolic Links nachgerüstet.

link() Mit `link()` versieht man eine Datei mit einem weiteren, gleichberechtigten Verzeichniseintrag. Er bringt den Link Count einer Datei also von einem Wert größer Null auf den nächstgrößeren Wert.

unlink() Schließlich kann man mit `unlink()` den Link Counter einer Datei wieder erniedrigen. Erreicht der Zähler den Wert 0, so wird der Platz, der zu einer Datei gehört, wieder freigegeben.

Zugriffsrechte an Verzeichnissen werden also etwas anders interpretiert als Zugriffsrechte an Dateien. Bei gewöhnlichen Dateien haben die Rechte `rwx` folgende Bedeutung:

r–Recht Das r–Recht an einer Datei gibt einem Prozeß das Recht, die Datei zum Lesen zu öffnen. In Systemaufrufen ausgedrückt bedeutet dies, daß ein `open(..., O_RDONLY)` auf die Datei ausgeführt werden kann.

w–Recht Das w–Recht an einer Datei gibt einem Prozeß das Recht, die Datei zum Schreiben zu öffnen und die Länge der Datei zu verändern. In Systemaufrufen: `open(..., O_WRONLY)` wird freigegeben.

x–Recht Das x–Recht an einer Datei gibt einem Prozeß das Recht, eine Datei auszuführen, d.h. den Systemaufruf `exec(...)` auf die Datei anzuwenden.

Bei Verzeichnissen ist die Bedeutung dieser Rechte etwas anders:

r–Recht Das r–Recht erlaubt das Öffnen und Durchlesen des Verzeichnisses zur Gewinnung einer Namensliste. Dabei gelangt ein Prozeß an die Namen und die I–Nodenummern aller Dateien in einem Verzeichnis. Letztere nützen ihm aber nichts, weil UNIX bei Systemaufrufen keine I–Nodenummern, sondern nur Namen akzeptiert.

w–Recht Das w–Recht erlaubt das Erzeugen und Vernichten von Verzeichniseinträgen in einem Verzeichnis mit den genannten Systemaufrufen. Dabei ist insbesondere beim Löschen von Dateien uninteressant, wer Eigentümer der zu löschenden Datei ist. Diese Eigenschaft von UNIX ist gelegentlich unerwünscht und kann durch Setzen des t–Rechtes (siehe unten) an einem Verzeichnis abgeschaltet werden.

x–Recht Das x–Recht schließlich befähigt die `namei()`–Routine, auf die Dateien in einem Verzeichnis zuzugreifen. Man kann dieses Recht auch als *Durchgangsrecht* an einem Verzeichnis bezeichnen.

Jede Datei in UNIX hat nun drei komplette Sätze von `rwx`–Rechten zur Verfügung. Welches dieser Tripel auf einen Systemaufruf angewendet wird, hängt von den Eigentumsrechten an der Datei und der Identität des Prozesses ab, der den Systemaufruf tätigt. UNIX vergleicht die effektive UID des Prozesses mit der UID der Datei. Falls beide übereinstimmen, gelten die Zugriffsrechte im ersten Tripel einer Datei. Falls die effektive UID eines Prozesses nicht mit der an der Datei übereinstimmt, werden die effektive GID und die GID der Datei verglichen. Bei Übereinstimmung gelten die Rechte des zweiten Tripels. In allen anderen Fällen gelten die Rechte des dritten Tripels.

Dabei kann es vorkommen, daß der Eigentümer einer Datei weniger Rechte an der Datei hält als der Rest der Welt. Entscheidend ist ausschließlich die UID bzw. GID der Datei.

An Dateien und Verzeichnissen können noch drei weitere Rechte gesetzt sein, die nur selten benutzt werden. Sie werden mit den Buchstaben `sst` bezeichnet und werden beim `ls`-Befehl an Stelle der x–Rechte einer Datei angezeigt, wenn sie vorhanden sind.

```
-rwsr-xr-x 1 root wheel 9668 Aug 18 16:32 /bin/passwd
```

Abbildung 1.14: Datei mit gesetztem `s`–Recht

Das erste s–Recht an einer Datei wird auch als *set user ID bit (SUID)* bezeichnet. Das zweite s–Recht trägt den Namen *set group ID bit (SGID)/.* Wenn das SUID–Bit an einer ausführbaren Datei gesetzt ist, bewirkt dies, daß die effektive UID eines Prozesses beim Start dieses Programmes durch die UID des Programmes überschrieben wird. Dadurch ist es möglich, bestimmten Programmen mehr Rechte zu verleihen, als der aufrufende Benutzer normalerweise hat.

Normalerweise laufen Prozesse mit den Rechten ihres Erzeugers. Wenn ein Benutzer etwa einen Editor startet, um die Paßwort–Datei `/etc/passwd` zu bearbeiten, wird dies in der Regel nicht funktionieren, denn ein Benutzer hat nicht die Rechte, um die Paßwort–Datei zu beschreiben. Trotzdem muß dieser Benutzer die Möglichkeit haben, die Paßwort–Datei kontrolliert zu verändern, etwa um sein Paßwort mit dem Befehl `/bin/passwd` neu zu setzen.

Deswegen ist am `passwd`–Kommando das SUID–Bit gesetzt, wie in Abbildung 1.14 zu sehen ist. Startet ein Benutzer den Befehl, wird seine effektive UID im `passwd`–Prozeß mit der Eigentümer–UID des `passwd`–Kommandos überschrieben. In diesem Fall bedeutet dies, daß der Befehl mit Systemverwalterrechten abläuft, weil der Eigentümer des Kommandos `root` ist. Ein Prozeß mit Systemverwalterrechten darf die Paßwort–Datei jedoch verändern, also dort ein neues Paßwort setzen.

Ein gesetztes SGID–Bit funktioniert analog, nur daß statt der effektiven UID die effektive GID ersetzt wird.

Das t–Recht an Dateien und Verzeichnissen hat unterschiedliche Bedeutungen, die zudem noch von der genauen UNIX–Variante abhängen, die eingesetzt wird. Es hat als eine Art Universalkennzeichen gedient, wenn eine Datei hervorgehoben werden mußte. Verbreitet sind die Verwendung als Löschschutz an Verzeichnissen und als Markierung an ausführbaren Dateien.

Wenn an einem Verzeichnis ein t–Recht gesetzt ist, schränkt dies den `unlink()`–Systemaufruf in seinen Möglichkeiten ein. Dateien können dann nur noch von ihrem Eigentümer oder dem Systemverwalter gelöscht werden. Dies ist besonders bei öffentlichen Verzeichnissen interessant, denn es verhindert, daß ein Benutzer anderen Benutzern Daten zerstören kann. Daher sind an den öffentlich benutzbaren Verzeichnissen `/tmp` und `/var/tmp` standardmäßig die Rechte `rwxrwxrwt` gesetzt.

1.3.4 Umgang mit Dateisystemen

Die Namenskonventionen von Gerätedateien für Diskettenstation und Festplatten unterscheiden sich von UNIX–Version zu UNIX–Version geringfügig. Kapitel 4 der Manual–Pages enthält Hinweise auf die lokalen Namenskonvetionen und Eigenschaften der beiden Gerätetypen. Die folgenden Erläuterungen zu Gerätedateien und Befehlen orientieren sich an Linux, sind aber leicht abgewandelt auch auf anderen UNIX–Versionen anwendbar.

/dev/fd*

In Linux stehen die Dateien `/dev/fd*` für Diskettenlaufwerke. Außer zum Formatieren von Disketten wird man nur mit den Geräten `/dev/fd0` und `/dev/fd1` arbeiten müssen. Diese Geräte erkennen das verwendete Diskettenformat automatisch und stellen den Diskettentreiber auf die benötigte Datenrate um.

Zum Formatieren von Disketten muß man jedoch ausdrücklich die gewünschte Diskettensorte angeben, denn zu diesem Zeitpunkt ist noch keine Datenstruktur auf der Diskette vorhanden, an der sich der Diskettentreiber orientieren könnte. Die Namenskonvention in Linux ist so, daß nach dem Namen `fd` die Nummer des gewünschten Laufwerks `0` oder `1` kommt. Danach folgt ein Buchstabe, wobei Großbuchstaben 3.5 Zoll Laufwerke und Kleinbuchstaben 5.25 Zoll Laufwerke bezeichnen. Ist der Buchstabe ein `d` oder `D`, wird mit doppelter Dichte geschrieben (DD–Disketten), ist er ein `h` oder `H`, wird mit hoher Dichte geschrieben (HD–Disketten). Schließlich kommt eine Zahl, die die Anzahl der Blöcke auf der Diskette angibt.

/dev/hd*

Die Dateien `/dev/hda*` und `/dev/hdb*` bezeichnen beiden möglichen AT–Bus Festplatten, die in einen PC eingebaut werden können. Dabei repräsentiert `/dev/hda` die gesamte erste Festplatte inclusive der Partitionstabelle und die durchnummerierten Geräte `/dev/hda1` usw. stehen für die einzelnen Partitionen der Platte. Analog wird mit der zweiten Platte `/dev/hdb` verfahren.

/dev/sd*

An einen SCSI–Bus können bis zu sieben Festplatten angeschlossen werden, die in Linux unter den Namen `/dev/sd*` zur Verfügung gestellt werden. Das Verfahren zum Ansprechen der einzelnen Partitionen ist entsprechend den Konventionen bei AT–Bus Festplatten.

fdformat

Um ein Dateisystem anlegen zu können, muß auf dem Medium bereits eine Blockstruktur vorhanden sein. Bei Festplatten ist dies in der Regel ab Werk der Fall, während Disketten erst formatiert werden müssen. Dafür steht in UNIX der Befehl `fdformat` zur Verfügung. Ausschließlich der Systemverwalter kann in Linux Disketten formatieren. Die Syntax ist einfach: `fdformat /dev/fd0H1440` formatiert etwa eine 1.44 MB Diskette in Laufwerk A.

mkfs

Nach dem Formatieren enthält eine Diskette eine Reihe von leeren Blöcken ohne Struktur. Bereits jetzt kann sie zusammen mit dem `tar`–Kommando zur Datenspeicherung verwendet werden. Um die Diskette oder auch eine Festplatte als Dateisystem nutzen zu können, muß erst einmal eine Dateisystemstruktur erzeugt werden. Das Kommando `mkfs` legt eine solche Struktur an, das bedeutet, es erzeugt einen Superblock, eine I–Node Tabelle und ein leeres Hauptverzeichnis.

Linux kennt mehrere verschiedene Typen von Dateisystemen und dementsprechend auch verschiedene Versionen des `mkfs`–Kommandos. Der Befehl `mkfs` erzeugt ein Minix–kompatibles Dateisystem, `mke2fs` erzeugt das leistungsfähigere Extended II Dateisystem. Die Syntax ist in beiden Fällen gleich:

```
$ mkfs [-i bytes per inode] /dev/<device> [ blocks ]
```

Als Pflichtparameter muß die Gerätedatei angegeben werden, auf der das Dateisystem eingerichtet werden soll. Bei Festplatten kann `mkfs` die Größe des Dateisystems aus der Partitionstabelle selbst bestimmen, bei Disketten muß sie angegeben werden. Optional kann man noch die mittlere erwartete Dateigröße angeben, d.h. man bestimmt für jeweils wieviele Bytes Plattenplatz eine I–Node reserviert werden soll.

mount

Ein so vorbereitetes Dateisystem ist fertig für die Benutzung und kann beim Betriebssystem angemeldet werden. Dies geschieht durch den Befehl `mount`:

```
$ mount -t <typ> /dev/<device> <directory>
```

Mit dem Kommando wird das Dateisystem auf dem Gerät */dev/device* vom Typ *typ* als Verzeichnis *directory* in den Dateibaum eingeblendet. Das betreffende Verzeichnis muß bereits existieren und sollte leer sein. Falls das Verzeichnis nicht leer ist, werden die bereits im Verzeichnis vorhandenen Dateien durch den Mount überlagert und sind nicht mehr erreichbar, bis das Dateisystem wieder abgemeldet wird. Alle Zugriffe auf ein gemountetes Verzeichnis werden abgefangen und auf das zugehörige Gerät umgeleitet.

Eine Diskette mit einem angemeldeten Dateisystem darf unter keinen Umständen aus dem Laufwerk genommen werden, solange das Dateisystem noch angemeldet ist. Dadurch, daß UNIX Schreibzugriffe auf Geräte puffert, kann es sein, daß Daten noch im internen Cache liegen und nicht auf die Diskette zurückgeschrieben worden sind. Wenn die Diskette jetzt unangekündigt aus dem Laufwerk genommen wird, kann es zu Datenverlusten oder sogar zur Zerstörung des Dateisystems auf der Diskette kommen.

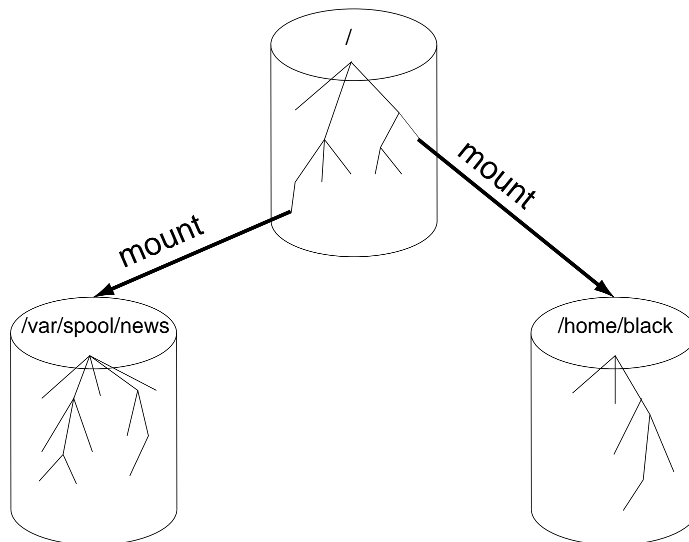


Abbildung 1.15: Dateisystem mit mehreren gemounteten Platten

umount

Stattdessen muß ein Dateisystem vor der Entfernung des Datenträgers beim System abgemeldet werden. Dies geschieht mit dem Befehl `umount`. Die Syntax ist

```
$ umount <dateisystem>
```

Dabei kann als *Dateisystem* entweder der Name der Gerätedatei oder der Name des Verzeichnisses, in dem die betreffende Platte gemountet ist, angegeben werden. Durch das Abmelden des Dateisystems wird sichergestellt, daß sich im Plattencache von UNIX keine Datenblöcke mehr befinden, deren Inhalt nicht auf die Platte zurückgeschrieben worden ist.

Dateisysteme, auf denen ein Prozeß noch Dateien geöffnet hat, lassen sich nicht abmelden. Stattdessen meldet `umount` in diesem Fall den Fehler `device busy`. Einige UNIX–Versionen haben einen speziellen Befehl, um anzuzeigen, welche Prozesse auf einem Dateisystem noch Dateien geöffnet haben.

Durch das Abmelden des Dateisystems wird das Dateisystem *abgeschlossen*. In seinem Superblock wird ein Kennzeichen gesetzt, das dieses Dateisystem als sauber abgemeldet und in einem konsistenten Zustand befindlich kennzeichnet. Dateisysteme, die dieses Kennzeichen nicht haben, müssen vor dem mounten mit `fsck` überprüft und korrigiert werden. Sauber abgemeldete Dateisysteme brauchen nicht überprüft zu werden, was den Bootprozeß stark beschleunigt.

Die Kommandos `mount` und `umount` haben noch weitere Optionen, die speziell bei der An– und Abmeldung von Netzwerkplatten und anderen Spezialdateisystemen zum Tragen kommen. Diese Optionen unterscheiden sich auch geringfügig von System zu System, sodaß in jedem Fall ein Blick in das Handbuch des jeweiligen Systems notwendig ist. Dort findet man in der Manualpage zum `mount`–Befehl in der Regel auch eine Übersicht der bekannten Dateisysteme und ihrer Typbezeichnungen.

/etc/fstab

Typischerweise wird man beim Systemstart viele Dateisysteme immer wieder an dieselben Stellen mounten wollen. UNIX kennt dazu die Konfigurationsdatei `/etc/fstab` in der

die ständig präsenten Dateisysteme mit jeweils einer Zeile aufgeführt werden. Der Aufbau dieser Zeile ist folgendermaßen:

```
device    dir    type    options    frequency    passno
```

Beim Starten des Systems wird der `mount`–Befehl mit der Option `-a -t nonfs` gestartet und meldet zunächst alle lokalen Festplatten *device* in ihrem jeweiligen Verzeichnis *dir* an. Später, nach dem Start des Netzwerkes, werden dann mit der Option `-a -t nfs` die Netzwerklaufwerke nachgeholt. Der Eintrag *type* gibt dabei den Typ des Dateisystem an, der beim Anmelden verwendet werden soll, während *options* eine Liste Optionen ist, die für ein Dateisystems dieses Typs gültig sind und die beim Anmelden dieses Dateisystem gesetzt werden müssen. In der List werden die einzelnen Optionsworte mit Kommata voneinander getrennt, Leerzeichen sind nicht zugelassen.

Das Feld *frequency* gibt dabei die Sicherungshäufigkeit des Dateisystems in Tagen an und wird vom Dateisicherungsbefehl `dump` verwendet. Wenn beim Starten des Systems ein automatischer Dateisystemcheck notwendig werden sollte, gibt das Feld *passno* an, in welcher Reihenfolge die Dateisysteme überprüft werden. Dabei ist es zulassig, mehreren Dateisysteme die gleiche Nummer zu geben. Diese werden dann parallel überprüft. Man sollte darauf achten, nur Dateisystemen auf unterschiedlichen physikalischen Platten die gleiche Durchlaufnummer zu geben, weil es sonst zu Konkurrenz zweier `fsck`–Prozesse um eine Plattenmechanik kommen kann. Das würde in tausenden überflüssiger Kopfbewegungen und damit in einer extremen Verlangsamung des Überprüfungsprozesses resultieren.

ncheck

Wie man im Beispiel 1.13 sehen kann, ist es relativ leicht, zu einem gegebenen Dateinamen eine I–Nodenummer zu bestimmen. Der umgekehrte Weg, nämlich alle Namen zu einer gegebenen Nummer zu finden, ist ungleich schwieriger. Auf jeden Fall muß außer der I–Nodenummer der gesuchten Datei auch das Gerät bekannt sein, auf dem die Datei liegt, denn eine I–Nodenummer ist nur für ein einzelnes Dateisystem eindeutig. So hat jedes Dateisystem zum Beispiel eine I–Node mit der Nummer 2, die in der Regel das Hauptverzeichnis dieses Dateisystems repräsentiert.

Ein Programm, das die Namen zu einer I–Nodenummer findet, müßte alle Verzeichnisse eines Dateisystems durchlesen und nach der gewünschten Nummer durchsuchen. Aus dem Pfad zum Verzeichnis und dem bei der gesuchten Nummer gefundenen Namen kann dann ein Pfadname der Datei gebildet werden. Je nachdem, wieviele I–Nodes und wieviele Verzeichnisse das Dateisystem hat, kann dieser Prozeß unterschiedlich lange dauern. Da für den Zugriff auf die Gerätedatei eines Dateisystems Systemverwalterrechte notwendig sind, kann auch nur der Systemverwalter diese Umkehrung durchführen.

In UNIX existiert der Befehl `ncheck`, der diese Aufgabe wahrnimmt. Er entdeckt außerdem Dateibäume, die nicht mehr über einen Pfad mit dem Hauptverzeichnis verbunden sind und Endlosschleifen von Verzeichnissen, die sich selbst enthalten.

fsck

Ein allgemeineres Programm zum Überprüfen von Dateisystemen ist der *file system checker* `fsck`. Er führt wesentlich weitergehende Tests durch, als es `ncheck` tun kann und hat auch begrenzte Reparaturfähigkeiten.

Normalerweise wird `fsck` automatisch beim Systemstart aufgerufen und überprüft alle in der `/etc/fstab` aufgelisteten Dateisysteme. Diejenigen Dateisysteme, die dabei als sauber abgemeldet gekennzeichnet sind, werden als fehlerfrei angesehen und nicht weiter

getestet. Alle anderen Dateisysteme werden vollständig durchgelesen und einer ganzen Reihe von Konsistenzprüfungen unterworfen:

- Blöcke, die sowohl in einer Datei vermerkt sind, als auch als frei gekennzeichnet sind, werden erkannt und als belegt gekennzeichnet.
- Blöcke, die von mehr als einer Datei belegt werden, werden erkannt und gemeldet[5]. Sie müssen gelöscht werden.
- Blocknummern, die in einer I–Node vorkommen, aber zu hoch oder zu niedrig für dieses Dateisystem sind, werden erkannt. Dateien, die solche Blöcke enthalten, müssen gelöscht werden.
- Die Größenangabe aller Dateien in der I–Node wird mit der höchsten belegten Blocknummer verglichen. Die Größenangabe der Datei muß unter Umständen korrigiert werden.
- Fehlerhafte I–Nodes werden erkannt und gemeldet. Die betroffenen Verzeichniseinträge müssen gelöscht werden.
- Belegte I–Nodes, die in keinem Verzeichnis auftauchen, werden gefunden und im Verzeichnis `lost+found` mit ihrer I–Nodenummer als Name eingetragen[6]. Dort kann der Systemverwalter die Dateien durchsehen und weiter entscheiden, was mit ihnen geschehen soll.
- Datenblöcke, die von keiner Datei belegt sind, aber nicht als frei gekennzeichnet sind, werden gefunden und wieder dem Pool der verfügbaren Blöcke des Dateisystems zugeführt.
- Verzeichniseinträge, die ungültige oder unbelegte I–Nodenummern enthalten, müssen gelöscht werden.
- Die Anzahl der Namen einer Datei wird bestimmt und mit dem Link Count–Feld in der I–Node selber verglichen. Stimmen sie nicht überein, wird der Link Count in der I–Node korrigiert.
- Die verschiedenen Informationen über das Dateisystem im Superblock werden auf Stand gebracht.

Beim Systemstart werden zwar alle Unstimmigkeiten im Dateisystem erkannt, aber nur diejenigen behoben, die ohne Datenverlust korrigiert werden können. Ist das Dateisystem so stark beschädigt, daß es nur unter Datenverlusten repariert werden kann, kann der normale Systemstart nicht durchgeführt werden. Stattdessen meldet sich UNIX im Single User Modus und verlangt nach einem Systemverwalter, der den `fsck` von Hand startet und die verschiedenen Löschungen, die zur Reparatur des Dateisystem notwendig sind, bestätigt und protokolliert. Die fehlenden Dateien müssen dann von einem Backup ersetzt werden.

Natürlich ist es witzlos, `fsck` auf Dateisysteme anzuwenden, die gerade angemeldet sind und auf denen Benutzer während der Prüfung veränderungen durchführen. Zudem verändert `fsck` den Inhalt des Dateisystems selbst, ohne dem Betriebssystem diese Änderungen mitzuteilen. Daher muß verhindert werden, daß das Betriebssystem während einer Überprüfung auf das betroffene Dateisystem zugreifen kann. Deswegen Dateisysteme sollten vor einem Check immer abgemeldet werden. Beim root–Dateisystem `/` gestaltet sich dies natürlich schwierig. Die meisten `fsck`–Programme haben entweder eine spezielle Option, die zum

[5] Der DOS–Befehl CHKDSK leistet ähnliches, wenn er Crosslinks erkennt und meldet

[6] Der DOS–Befehl CHKDSK leistet etwas ähnliches, wenn er *verlorene Ketten* in Dateien umwandelt.

Testen des root–Dateisystems gesetzt werden muß oder sind so programmiert, daß die Maschine nach dem Checken des root–Dateisystem einen Neustart durchführt, ohne die Caches des root-Dateisystems zurückzuschreiben[7].

1.4 Das verbesserte BSD Fast Filing System

1.4.1 Probleme des System V–Dateisystems

Im Jahre 1984 stellten die Entwickler des BSD UNIX–Systems ein neues UNIX–Dateisystem vor. Diese Neuentwicklung, das sogenannte BSD *Fast Filing System (FFS)*, behebt eine ganze Reihe von Schwächen des klassischen UNIX–Dateisystems und stellt einige neue Features zur Verfügung, die im originalen Dateisystem nicht vorhanden waren.

Ein Mangel, der mit den wachsenden Festplattengrößen zutage trat, ist der, daß für die Darstellung von I–Nodenummern nur 16 Bit breite Worte zur Verfügung stehen. Diese Einschränkung begrenzt die Anzahl der Dateien in einem Dateisystem effektiv auf 65536 Stück. Auf großen Festplatten mit einer geringen mittleren Dateigröße wird dieses Limit leicht überschritten. Eine Erweiterung des Datentype I–Nodenummer auf 32 Bit machte eine Überarbeitung des Verzeichniskonzepts und der anderen Datenstrukturen eines Dateisystem notwendig. Bei dieser Gelegenheit hat man zugleich auch die anderen Felder einer I–Node auf 32 Bit erweitert. Auf diese Weise wird der Bereich der zur Verfügung stehenden UIDs und GIDs stark vergrößert, was dem Einsatz des Dateisystems im Netz sehr entgegenkommt: Bei Einsatz von NFS sollte eine UID netzwerkweit einheitlich sein. Weiterhin erlaubte eine neue Verzeichnisstruktur lange Dateinamen von bis zu 255 Zeichen Länge. Durch die Änderungen am Dateisystem sind I–Nodes beim FFS 128 Byte groß.

Der eigentliche Grund jedoch, der das BSD Entwicklerteam zur Neugestaltung des Dateisystems motivierte, war die Beobachtung, daß der Plattendurchsatz eines typischen UNIX–Systems stark hinter dem berechneten Maximaldurchsatz der Festplatten zurückbleibt. Das ursprüngliche Dateisystem arbeitete mit einer Blockgröße von 512 Byte. Schon durch eine Vergrößerung der Blockgröße auf ein Kilobyte konnte die Transfergeschwindigkeit mehr als Verdoppelt werden: Zum einen konnten jetzt größere Blöcke von der Platte in den Speicher bewegt werden, wodurch der Verwaltungsaufwand des Betriebssystems geringer wurde. Zum anderen waren durch die größeren Blöcke bei gleichbleibender mittlerer Dateigröße im Schnitt weniger Zugriffe auf indirekte Blöcke notwendig.

Damit nicht genug: Durch Messungen konnte bestätigt werden, daß der Plattendurchsatz eines typischen UNIX–Systems nach wenigen Wochen der Benutzung auf ca. ein Fünftel der ursprünglichen Geschwindigkeit sinkt. Der Grund dafür liegt in der Organisation der freien Blöcke beim herkömmlichen Dateisystem. Das alte Dateisystem speichert freie Blöcke in einer Liste, der sogenannten *free list* ab. Diese Liste ist in Form eines LIFO–Speichers organisiert: Freiwerdende Blöcke werden vorne in die Liste eingefügt, benötigte Blöcke werden ebenfalls von vorne her der Liste entnommen. Wenn in einem System häufig Dateien erzeugt und gelöscht werden, ist die Free List des System schon bald gut durchmischt. Das bedeutet, daß die Datenblöcke neu angelegter Dateien über die ganze Platte verstreut liegen. Eine zusammenhängende Speicherung von Dateien um die Suchzeiten auf der Platte zu minimieren ist nicht möglich, denn um in der Free List zusammenhängende Datenblöcke zu finden, müßte jedesmal die gesamte Free List von vorne her durchsucht werden.

Beim alten Dateisystem sind die I–Nodes eines Dateisystem in einer Tabelle geballt am Anfang der Platte abgelegt. Das macht beim Zugriff auf die Datei weite Kopfbewegungen über die Plattenoberfläche notwendig. Schöner wäre es, würden die I–Nodes in der Nähe der Daten, die sie referenzieren, angelegt werden können.

[7] ... denn das würde die Änderungen, die `fsck` durchgeführt hat, möglicherweise wieder überschreiben.

- Beschränkte Zahl an I–Nodes
- Dateinamen nur 14 Zeichen lang
- kleine Blockgröße oder hohe interne Fragmentierung
- hohe externe Fragmentierung wegen Verwendung einer Free List
- I–Node und Daten weit auseinanderliegend
- keine Auswertung von Informationen über die Plattengeometrie durch das Dateisystem
- keine Verweise über Dateisystemgrenzen hinweg
- keine Begrenzung des Plattenplatzverbrauches einzelner Benutzer möglich (keine Quota).

Abbildung 1.16: Schwächen des alten Dateisystems in Stichworten

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

1.4.2 Blockverwaltung

Um den Durchsatz des neuen Dateisystems zu erhöhen, entschieden sich die Entwickler des FFS zunächst dafür, die verwendete die Blockgröße drastisch zu vergrößern. Das neue Dateisystem verwendete zuerst Blöcke von 4 KB, heutige UNIX–Versionen nehmen standardmäßig sogar Blöcke von 8 KB. Anders als im alten Dateisystem werden die I–Nodes nicht in einer Tabelle am Anfang des Dateisystems gesammelt. Stattdessen teilt man die Platte in Gruppen von Zylindern auf. Jede Zylindergruppe enthält eine Kopie des Superblocks, eine *Bitmap*, die die freien Blöcke bezeichnet, einen Anteil an den I–Nodes des Dateisystems und Datenblöcke.

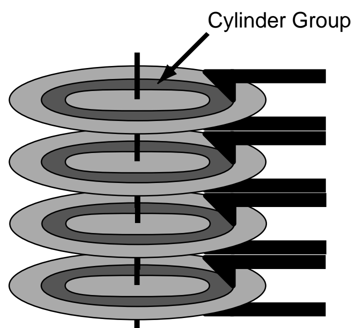


Abbildung 1.17: Eine Zylidergruppe im BSD FFS

Durch die Verwendung einer Bitmap statt einer Free List entfällt das Durchmischen der freien Blöcke. Zusammenhängende freie Bereiche sind gut zu erkennen und mit wenig Aufwand zu finden.

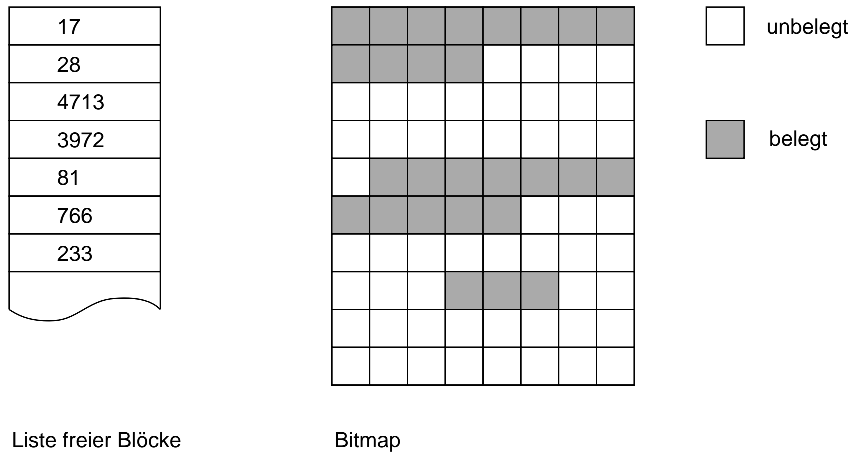


Abbildung 1.18: Verwaltung freier Blöcke

Durch das Konzept der Zylindergruppen verteilen sich die I–Nodes über die gesamte Festplatte, anstatt sich am Anfang einer Platte zu sammeln. Beim Anlegen von Dateien versucht UNIX Datenblöcke in derselben oder einer benachbarten Zylindergruppe zu finden, in der die I–Node für die Datei liegt. Dadurch stehen die Verwaltungsinformationen einer Datei in der Nähe der Daten der Datei und die Schreib/Lesekopfbewegungen, die zum Zugriff auf die Daten notwendig sind, werden minimiert.

Dadurch entsteht jedoch ein Problem mit großen Dateien: Wenn in einer Zylindergruppe eine sehr lange Datei angelegt wird, wird diese alle hier vorhandenen Datenblöcke belegen. In dieser Zylindergruppe sind jedoch überproportional viele I–Nodes frei. Wenn diese jetzt ebenfalls belegt werden, müssen die Datenblöcke zu diesen I–Nodes in benachbarten Zylindergruppen gesucht werden, weil alle lokal vorhandenen Blöcke ja belegt sind. Dadurch wird wiederum die I–Node/Datenblock–Balance in diesen Zylindergruppen durcheinandergebracht. Um diesen Effekt zu vermeiden, wird beim Schreiben von langen Dateien nach jedem geschriebenen Megabyte ein *long seek* erzwungen, in dem die Zylindergruppe gewechselt wird. Damit soll erreicht werden, daß sich die Daten einer sehr großen Datei über mehrere Zylindergruppen verteilen und das Verhältnis zwischen freien I–Nodes und freien Datenblöcken in allen Zylindergruppen etwa gleich bleibt.

Eine andere Idee, die im FFS verwirklicht worden ist, ist das Gruppieren verwandter Daten. So wird zum Beispiel versucht, Dateien eines Verzeichnisses in derselben Zylindergruppe zu plazieren. Auf der anderen Seite darf die Gruppierung von Daten aber nicht übertrieben werden: Würde man dies auf die Spitze treiben, fänden sich alle Dateien in einer Gruppe wieder und man hätte nichts gewonnen. Um zu verhindern, daß die Gruppierungsmaßnahmen zu große Gruppen erzeugen, werden neue Verzeichnisse in Zylindergruppen angelegt, die überdurchschnittlich viele freie I–Nodes und möglichst wenige Verzeichnisse enthalten.

Durch den Übergang auf eine Blockgröße von 8192 Bytes ist es den BSD Entwicklern gelungen, die Transfergeschwindigkeit sogar bei verstreut angeordneten Dateien stark zu erhöhen. Wie wir in Abbildung 1.6 jedoch gesehen haben, steigt der Platzverlust durch interne Fragmentierung stark an. Beim BSD FFS hat man einen Trick angewandt, um dieses Problem in den Griff zu bekommen.

Betroffen von internen Fragmentierung sind immer nur die jeweils letzten Blöcke einer Datei. Im FFS ist es möglich, einzelne, 8 KB große Datenblöcke in zwei 4 KB große Fragmente aufzuteilen. Ein solches Fragment kann wiederum in zwei kleine Fragmente zerlegt werden und so weiter, bis herab zur physikalischen Blockgröße der Festplatte. Wenn bei der Speicherung einer Datei der letzte Datenblock nicht ganz gefüllt werden kann, bricht

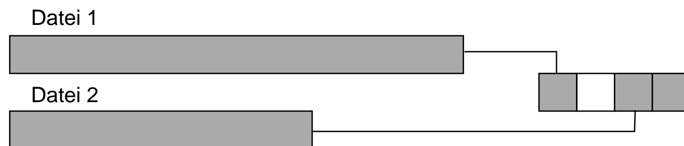


Abbildung 1.19: Fragmente können die Dateienden von mehreren Dateien enthalten

das Dateisystem einen Datenblock in Fragmente auf und speichert das Dateiende in einem so kleinen Fragment wie möglich. Der Rest des Datenblocks steht zur Speicherung weiterer Dateienden anderer Dateien zur Verfügung. Auf diese Weise gelingt es, den Platzverlust durch interne Fragmentierung bei einem FFS mit 8 KB großen Blöcken und 512 Byte großen Fragmenten in derselben Größenordnung zu halten, wie bei einem alten Dateisystem mit 512 Byte Blockgröße.

1.4.3 Neue Features

Lange Dateinamen

Durch eine Änderung des internen Aufbaus von Verzeichnissen ist es im FFS auch möglich geworden, extrem lange Dateinamen zu verwenden. Das BSD FFS läßt Dateinamen von bis zu 255 Zeichen Länge zu.

Softlinks

Ein weiteres neues Feature des Dateisystems ist die Möglichkeit, symbolische Links zu legen. Ein *Symlink* oder *Softlink* ist eine Datei vom Typ `l`. Sie enthält als Inhalt den Namen der Datei, auf die sie zeigt als Text. Jeder Versuch, auf das Link zuzugreifen, wird durch das Betriebssystem auf die Datei umgelenkt, auf die das Link zeigt.

Im Gegensatz zu Hardlinks belegen Softlinks eine I–Node und haben deswegen eigene Permissions und einen eigenen Eigentümer (die aber keine Rolle spielen). Sie können Dateisystemgrenzen überspannen und auch auf Verzeichnisse verweisen. Bei einem Symlink sind der originale Name der Datei und das Link nicht gleichberechtigt, sondern zu jedem Zeitpunkt voneinander zu unterscheiden.

Während Hardlinks mit dem Systemaufruf `link()` erzeugt werden, werden Symlinks mit dem Systemaufruf `symlink()` erstellt. UNIX stellt für die Erzeugung beider Linktypen den Befehl `ln` zur Verfügung, der standardmäßig ein Hardlink erzeugt. Mit der Option `-s` wird stattdessen ein Symlink gelegt. Beide Arten von Link werden mit dem Kommando `rm` wieder entfernt.

Quotas

Das für den Systemverwalter auffälligste neue Feature des BSD FFS gegenüber dem alten Dateisystem ist die Quota–Verwaltung. Quotas sind Begrenzungen der Anzahl der belegten Blöcke oder der Anzahl Files, die ein Systemverwalter für einzelne Benutzer oder ganze Benutzergruppen vergeben kann.

Der Systemverwalter kann das Quotasystem für jedes Dateisystem getrennt aktivieren. Er muß dann für jedes Dateisystem getrennt festlegen, wieviele Datenblöcke und wieviele I–Nodes jeder Benutzer auf diesem Dateisystem in Anspruch nehmen kann. Auch für Benutzergruppen können diese Obergrenzen festgelegt werden.

Das Quotasystem verwaltet pro Benutzer oder Benutzergruppe zwei Limits, die *soft limit* und *hard limit* genannt werden. Das Softlimit ist die Grenze an Plattenressourcen, die ein Benutzer nicht überschreiten *soll*. Das Hardlimit ist eine Grenze, die ein Benutzer nicht überschreiten *kann*. Jeder Versuch eines Benutzers, mehr Datenblöcke oder I–Nodes zu belegen, als dem ihm in seinem Hardlimit zugestanden werden, wird vom Betriebssystem mit einem Schreibfehler quittiert.

Das Softlimit ist üblicherweise niedriger als das Hardlimit. Es soll auf die Werte gesetzt werden, die der Systemverwalter dem Benutzer für seine normale Arbeit zubilligen möchte. Überschreitet der Benutzer das Softlimit, erhält er beim nächsten Login eine Warnung mit der Bitte, soviele Dateien zu löschen, bis er wieder innerhalb seiner Quota ist. Das System toleriert Überschreitungen des Softlimits für eine gewisse Anzahl von Tagen, die sogenannte *grace period*. Kürzt ein Benutzer innerhalb dieser Zeit seinen Datenbestand nicht derart, daß er wieder in der Quota liegt, behandelt das System das Softlimit fortan wie ein Hardlimit, d.h. der Benutzer bekommt Schreibfehler gemeldet.

Während das Softlimit im allgemeinen einen Wert definiert, den ein Benutzer zwischen zwei Loginsessions nicht überschreiten soll, definiert das Hardlimit einen Wert, der während einer Session nicht überschritten wird. Das Hardlimit soll demnach auf einen Wert gesetzt werden, den ein Benutzer auf keinen Fall überschreiten können soll.

Auf keinen Fall sollte der Systemverwalter Quotas auf `/tmp`–Verzeichnisse legen. Falls ein Benutzer sein Limit überschreitet, während er seine Daten aus einer Anwendung heraus sichern möchte, scheitert das Abspeichern. Er muß dann ein Verzeichnis haben, in dem er seine Daten in jedem Fall vorübergehend sichern kann.

Da Quotas in einem UNIX–System dateisystemweise festgelegt werden, ist es sinnvoll, die Homeverzeichnisse der Benutzer in einem gesonderten Dateisystem zu halten[8]. In den allermeisten Fällen wird es genügen, lediglich auf diesem Dateisystem Quotas einzurichten.

Die Aufgabe eines Systemverwalters ist es, seine Benutzer bei der reibungslosen Abwicklung ihrer Arbeiten am Rechner zu unterstützen. Um seinen Job effizient zu erledigen, muß er mit seinen Benutzern zusammenarbeiten und nicht gegen sie. Die Einrichtung von Beschränkungen soll nicht dazu dienen, einen Benutzer in seiner Arbeit zu behindern, sondern sie soll ausschließen, daß ein Benutzer durch übermäßige Benutzung von Systemressourcen die Arbeit anderer behindert. Es ist daher sinnvoll, den zur Verfügung stehenden Plattenplatz großzügig aufzuteilen und erst im Falle einer Plattenplatzverknappung die Quotas nach Maß heraufzusetzen. Überadministration verursacht dem Systemverwalter hier unnötige Arbeit durch die Verwaltung umfangreicher Quotatabellen.

Installation von Quotas

Das Quotasystem wird aktiviert[9], indem das Dateisystem mit den Optionen `usrquota` oder `grpquota` gemountet wird. Das kann geschehen, indem die Optionen beim `mount`–Kommando nach dem Schalter `-o` mit angegeben werden oder indem sie in der Optionen–Spalte in der Konfigurationsdatei `/etc/fstab` (siehe Seite 20) dauerhaft eingetragen werden.

Das Betriebssystem führt in zwei Dateien `quota.user` und `quota-group` über den Plattenplatzverbrauch der einzelnen Benutzer bzw. Benutzergruppen Buch. Die Dateien werden, wenn nicht anders angegeben, im Hauptverzeichnis des jeweiligen Dateisystems angelegt. Einige UNIX–Versionen erlauben es, die Quota–Dateien an anderer Stelle im

[8] Es existieren noch weitere Gründe, Homeverzeichnisse auf einem gesonderten Dateisystem zu halten: Die Datensicherung wird wesentlich einfacher und die Datensicherheit ist größer. Systemupdates werden vereinfacht.

[9] Bei Linux muß das Quota-Subsystem unter Umständen erst noch im Kern installiert werden. Slackware 1.1.1 enthält ein Package `quota`, daß alle benötigten Dateien dafür enthält. Es wird in `/usr/src/quota...` installiert. In diesem Verzeichnis befindet sich auf eine Datei HOWTO, die die Aktivierung des Quota–Systems beschreibt

```
# mount -t ext2 -o usrquota,grpquota /dev/sdb4 \
        /home/mahaki/vol3

# mount -t ext2 -o usrquota="/usr/adm/vol2.quota" \
>       /dev/sdb3 /home/mahaki/vol2

# cat /etc/fstab | grep quota
/dev/sdb3    /home/mahaki/vol2  ext2   \
                       defaults,usrquota,grpquota
/dev/sdb4    /home/mahaki/vol1  ext2   \
                       defaults,usrquota,grpquota
```

Abbildung 1.20: Aktivierung der Quotas

NAME

quotacheck — Erstellen der Quota–Dateien.

SYNTAX

`quotacheck [-avug]`
`quotacheck [-vug dateisystem]`

OPTIONS

-a (*all*) Erstelle Quota–Dateien für alle Dateisysteme in `/etc/fstab`

-v (*verbose*) Erzeuge Meldungen über das Fortschreiten der Erstellung.

-u (*usrquota*) Erstelle nur `quota.user`.

-g (*grpquota*) Erstelle nur `quota.group`.

Abbildung 1.21: `quotacheck` Kommando

Dateisystem anzulegen. Die Position der Quotadateien wird dann mit der `usrquota` bzw. `grpquota`–Option angegeben. Dabei ist zu beachten, daß die maximale Länge einer Zeile in der `/etc/fstab` begrenzt ist. Die Pfadnamen dürfen also nicht zu lang werden.

Bevor die Quota für ein Dateisystem zum ersten Mal aktiviert wird, müssen die Quotadateien angelegt werden. Dies geschieht mit dem Befehl `quotacheck`, der alle Dateien eines Dateisystems durchscannt und Länge und Eigentümer aller Dateien feststellt und vermerkt. Es empfiehlt sich, `quotacheck` nach Aktivierung der Quota einmal manuell aufzurufen und zusätzlich **nach** dem `fsck`–Aufruf und dem Mounten **aller** lokalen Dateisysteme in die Bootscripte des Systems einzutragen.

Um das Quotasystem scharf zu machen, muß es nach der Konfiguration mit dem Kommando `quotaon` tatsächlich aktiviert werden. Das Betriebssystem beginnt dann tatsächlich Fehler zu melden, wenn die Quotas eines Benutzers oder einer Gruppe überschritten werden. Syntax und Optionen sind genau wie beim `quotacheck`–Befehl. Mit dem Kommando `quotaoff` wird das Quotasystem wieder entschärft. Das Betriebssystem loggt jetzt zwar immer noch Veränderungen im Platzverbrauch mit, aber meldet bei Überschreitungen der Quota keine Fehler. Die Syntax und die Optionen sind auch hier dieselben.

Quota festlegen und überprüfen

NAME

quota — Plattenplatzverbrauch und Limit anzeigen.

SYNTAX

`quota [-vug | q]`

OPTIONS

-v (*verbose*) Zeige alle vorhandenen Informationen über die eigene Quota an.

-u (*usrquota*) Zeige die eigene Quota an.

-g (*grpquota*) Zeige die Quota der eigenen Usergruppe an.

-q (*quiet*) Zeige nur dann eine Warnung an, wenn eine Quota überschritten ist.

Abbildung 1.22: `quota` Kommando

NAME

edquota — Quota für einen Benutzer bearbeiten.

SYNTAX

```
edquota [-p prototype user] [-ug] name
edquota [-t] [-ug] name
```

OPTIONS

-p proto Übernimmt die Definitionen für den User *name* ohne Bearbeitung vom Prototyp–Benutzer *proto*. Dadurch ist das Anlegen vieler Benutzerquotas im Batch möglich.

-u (*usrquota*) Bearbeite die Userquota des Benutzers *name*. Dies ist der Default.

-g (*grpquota*) Bearbeite die Groupquota des Gruppe *name*.

-t (*time*) Bearbeite die *grace period* für dieses Dateisystem. Diese Zeit legt fest, wielange ein Benutzer seine Softlimits überschreiten darf, bevor das Softlimit für ihn zu einem Hardlimit wird.

Abbildung 1.23: `edquota` Kommando

Mit dem Kommando `quota` kann sich ein Benutzer anzeigen lassen, ob und wieweit er seine Quota ausgeschöpft hat. Es empfiehlt sich, am Ende von `/etc/profile` bzw. `/etc/cshrc` einen Aufruf der Art `quota -q` einzufügen. Das Kommando benachrichtigt einen Benutzer beim Login automatisch, falls er seine Quota überschritten hat. Es ist günstig, diese Meldung unauffällig zu halten (d.h. nicht `quota -v` zu verwenden), da diese Meldung bei jedem Login erscheint.

Der Systemverwalter legt mit dem Kommando `edquota` die Quota für einen Benutzer fest. Das Kommando startet den in der Umgebungsvariable `EDITOR` festgelegten Defaulteditor mit einer ASCII–Repräsentation der Quotadaten eines Benutzers. Im Editor kann diese Darstellung bearbeitet und abgespeichert werden. `edquota` liest die modifizierte Datei ein und wandelt sie in das interne Format des Betriebssystems zurück.

Mit der Option `-p` startet `edquota` keinen Editor, sondern übernimmt die Quotainformationen für den neuen Benutzer von einem Prototyp-Benutzer. Auf diese Weise kann eine ganze Reihe von Benutzern mit derselben Quota installiert werden, ohne daß man für jeden dieser Benutzer durch den Editor gehen muß. Stattdessen kann man mit dem Kommando `edquota -p olduser newuser` die Quotas von `olduser` für einen neuen Benutzer

übernehmen. Zusammen mit einer `for`–Schleife über alle Benutzer einer Gruppe kann man so z.B. die Quota für alle Benutzer auf einen einheitlichen Wert setzen.

```
# edquota -u kris
Quotas for user kris:
/dev/sda2: blocks in use: 105, limits (soft = 10000, hard = 20000)
           inodes in use:  12, limits (soft = 10000, hard = 20000)

# quota kris
Disk quotas for user kris (uid 1001):
     Filesystem blocks quota limit grace files quota limit grace
      /dev/sda2 105    10000 20000       12    10000 20000

# repquota -v -a
*** Report for user quotas on /dev/sda2 (/home/mahaki/vol1)
                     Block limits                File limits
User         used    soft    hard  grace    used  soft  hard  grace
kris    --    105   10000   20000             12 10000 20000
```

Abbildung 1.24: Das Quotasystem in der Anwendung

Mit dem Kommando `repquota` schließlich kann der Systemverwalter sich jederzeit eine Übersicht über den Plattenplatzverbrauch seiner Benutzer anzeigen lassen. Benutzer, die ihr Limit überschritten haben, werden in der zweiten Spalte der Ausgabe mit Pluszeichen markiert. Abbildung 1.24 zeigt das Quotasystem in der Anwendung.

1.5 Dateisysteme mit hoher Verfügbarkeit

IBM hat für die Rechner der RS/6000 Serie ein eigenes Dateisystem entwickelt, das besondere Sicherheitsmerkmale aufweist. Das AIX Journaled File System[10] ist ein Dateisystem mit *hoher Verfügbarkeit*, denn es hat Eigenschaften, die es besonders ausfallsicher machen können. Es ist außerdem ein Dateisystem mit erweiterten Sicherheitsmerkmalen, denn die Zugriffsrechte auf Dateien können wesentlich individueller festgelegt werden, als bei einem gewöhnlichen UNIX–Dateisystem.

- Das Dateisystem ist *transaktionsorientiert*. Alle Veränderungen am Dateisystem sind atomar, d.h. auch wenn der Rechner durch einen Systemcrash oder einen Stromausfall mitten in einer Dateioperation stehenbleibt, kommt es niemals zu Inkonsistenzen wie doppelt belegten Blöcken, Verzeichniseinträgen ohne zugehörige I–Node usw.

- Das Dateisystem arbeitet mit *disk striping*. Dadurch kann sich ein Dateisystem über mehr als eine physikalische Platte erstrecken und im laufenden Betrieb repartitioniert werden.

- Das Dateisystem arbeitet mit *disk mirroring*. Dadurch kann mehr als eine physikalische Kopie der Daten gehalten werden oder Daten können im laufenden Betrieb von einer Platte auf eine andere migriert werden. Datenverlusten durch Ausfall von Hardware kann so vorgebeugt werden. Zusammen mit geeigneter Hardware ist es möglich, defekte Platte und Controller bei laufendem Rechner auszutauschen.

[10]Auch andere Hersteller (etwa Hewlett Packard) bieten für ihre Workstations inzwischen Dateisysteme mit ähnlichen Merkmalen an.

- Das Dateisystem arbeitet mit *access control lists*. Dadurch kann an einer Datei prinzipiell für jeden Benutzer und für jede Benutzergruppe getrennt bestimmt werden, ob die Datei gelesen, beschrieben oder ausgeführt werden darf.

1.5.1 Disk Striping

In AIX werden eine oder mehrere physikalische Festplatten in einer Gruppe, der sogenannten *volume group (VG)* zusammengefaßt. AIX behandelt eine solche Gruppe von Festplatten als eine Einheit, die nach Bedarf in Dateisysteme aufgeteilt werden kann. Anders als bei normalen UNIX–Dateisystemen muß ein Dateisystem dabei nicht aus zusammenhängendem Plattenplatz bestehen.

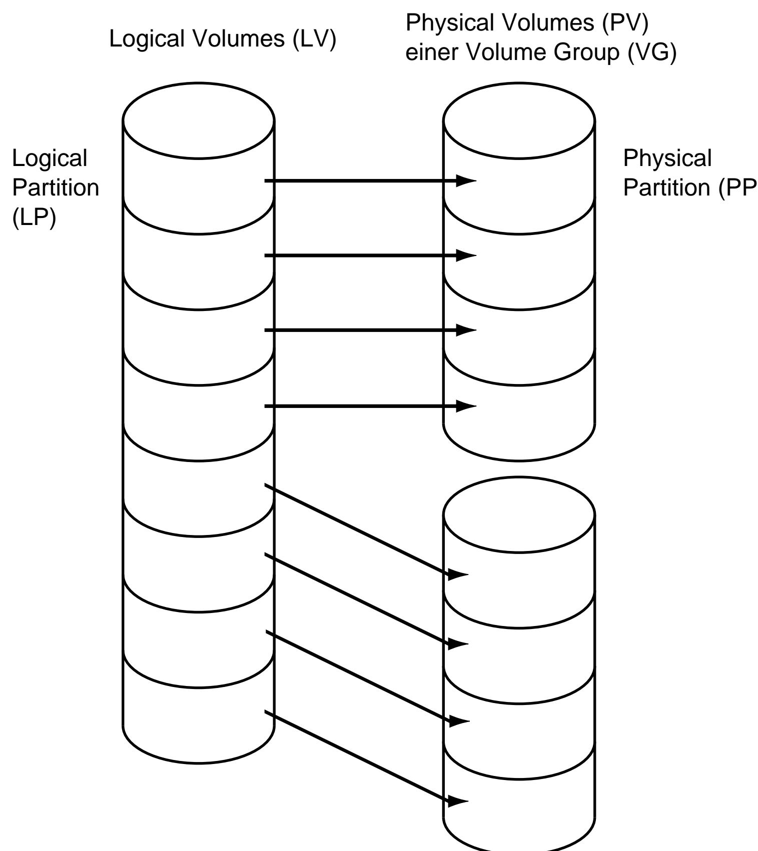


Abbildung 1.25: Physikalische und logische Partitionen bei AIX

Stattdessen wird der Platz in einer Volume Group in gleichgroße Streifen von 1, 2, 4 (Standardwert) oder 8 Megabyte Größe unterteilt. AIX bezeichnet einen solchen Streifen als eine *physical partition (PP)*. Ihm kommt in etwa dieselbe Rolle in Bezug auf Dateisysteme zu wie einer Speicherseite in Bezug auf die Speicherverwaltung: Die Physical Partition dient als Baustein, aus dem der Systemverwalter nach Bedarf Dateisysteme zusammensetzen kann.

Dazu definiert er sich ein Dateisystem, das *logical volume (LV)*, dessen Größe ein ganzzahliges Vielfaches der physikalischen Partitionsgröße sein muß. Genau wie die physikalischen Festplatten einer Volume Group ist auch das logische Laufwerk in Partitionen unterteilt. Ihre Größe muß den Partitionsgrößen der physikalischen Partitionen entsprechen.

Über eine Tabelle wird jetzt jeder logischen Partition ein physikalisches Äquivalent zugeordnet. Dabei kann sich das logische Laufwerk über mehr als eine physikalische Festplatte erstrecken, solange diese Festplatten einer Volume Group angehören. Die Zuordnung kann, wie in der Abbildung, linear sein, muß es aber nicht.

AIX ist in der Lage, ein logisches Laufwerk nachträglich zu vergrößern. Dazu müssen in der betreffenden Volume Group lediglich physikalische Partitionen vorhanden sein, die noch keinem logischen Laufwerk zugewiesen sind. Dieses Anstückeln kann dabei im laufenden Betrieb und mit aktiven Benutzern auf dem betroffenen Dateisystem geschehen.

Die Befehle, die zum Managment von AIX Festplatten und Dateisystemen dienen, unterscheiden sich stark von denen eines herkömmlichen UNIX–Systems. Sie sind in der AIX Dokumentation erläutert und sollen hier nicht weiter erklärt werden.

1.5.2 Disk Mirroring

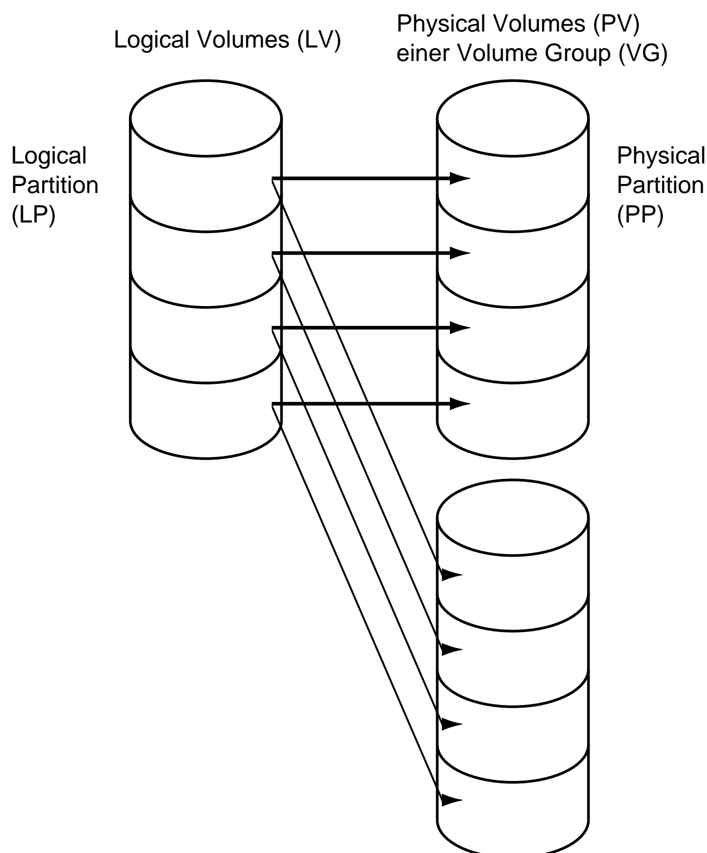


Abbildung 1.26: Disk Mirroring

Damit Daten auf einem logischem Volume gespeichert werden können, ist es notwendig, jeder logischen Partition mindestens eine physikalische Partition zuzuweisen. AIX erlaubt es dem Systemverwalter jedoch, allen logischen Partitionen eines logischen Laufwerks bis zu drei physikalische Partitionen zuzuweisen. AIX hält auf diese Weise mehrere Kopien der Daten auf wichtigen Dateisystemen. Bei Änderungen an den Daten werden automatisch alle Kopien der logischen Partition aktualisiert.

Wenn man die Lage der Kopien auf den physikalischen Platten richtig auswählt, kann man so die Verfügbarkeit der Daten deutlich steigern. Entscheidend ist, daß bei Ausfall

einer Festplatte nicht alle Kopien der Daten beschädigt werden, sondern mindestens eine physikalische Partition für jede logische Partition erhalten bleibt. Die Kopien sollten also auf jeweils unterschiedlichen physikalischen Platten angelegt werden, wie in Abbildung 1.26 gezeigt. Falls eine Platte beschädigt wird, steht eine Kopie der Daten auf der anderen Platte immer noch zur Verfügung. Nach dem Austausch der defekten Platte legt das Betriebssystem dann eine neue Sicherheitskopie der Daten auf dem neuen Laufwerk an.

Falls jede der beiden physikalischen Platten aus der Abbildung außerdem noch an einem eigenen Plattencontroller hängen würde und eine eigene Stromversorgung hätte, wäre man auch gegen das Versagen dieser Teile geschützt.

Wenn die Daten einer gespiegelten logischen Partition verändert werden, müssen diese Änderungen auf jeder physikalischen Partition vorgenommen werden, die der logischen Partition zugeordnet worden ist. Dabei kann man zwei verschiedene Strategien verfolgen:

- Beim *parallelen Update* werden alle Kopien der Partition zugleich geändert. Schreibzugriffe auf eine so gespiegelte Platte sind in etwa gleichschnell wie Zugriffe auf eine ungespiegelte Platte, falls die gespiegelten Platten jeweils an einen eigenen Controller angeschlossen sind[11].

  Der Nachteil dieser Methode besteht darin, daß es Momente geben kann, in denen von einem bestimmten Datensatz keine konsistene Kopie existiert, da ja alle Kopien der Daten gleichzeitig bearbeitet werden. Fällt in diesem Moment das System aus, geht der momentan in der Bearbeitung befindliche Datensatz möglicherweise verloren.

- Beim *sequentiellen Update* vermeidet man dieses Problem, indem man die Schreibzugriffe auf die einzelnen Kopien einer Partition nacheinander ausführt. Ein einzelner Zugriff dauert auf diese Weise jedoch je nach Anzahl der Kopien entsprechend länger als beim parallelen Update.

1.5.3 Access Control Lists (ACL)

Das AIX Dateisystem unterstützt eine Verfeinerung von Zugriffsrechten, die *access control lists (ACLs)* genannt wird[12]. Mit Hilfe von ACLs ist es möglich, Zugriffsrechte auf Dateien auch für andere Benutzer als den Dateieigentümer und die Dateigruppe genau anzugeben.

ACLs definieren keine neuen Zugriffsrechte an Dateien: Um auf eine Datei zuzugreifen, muß ein Benutzer noch immer dieselben Kombinationen von r–, w– und x–Rechten haben, wie ohne ACLs. Aber mit einer ACL kann man für beliebige Benutzer oder Benutzergruppen beliebige rwx–Tripel an einer Datei definieren. Beim Zugriff auf die Datei gilt dann dieses Tripel anstelle des herkömmlichen User–, Group– oder World–Tripels.

Zum Zugriff auf ACLs stellt AIX die folgenden Befehle bereit:

aclget Der Befehl dient zum Auslesen der Access Control List einer Datei. Die ACL wird in eine Textdarstellung umgewandelt, die mit einem normalen Editor modifiziert werden könnte.

aclput Der Befehl liest eine Textdarstellung einer ACL ein und wandelt sie in die betriebssysteminterne Form um.

acledit Der Befehl ist eine Kombination von `aclget` und `aclput`. Er erzeugt eine Textdarstellung der ACL einer Datei, speichert diese in einer Zwischendatei und startet den in der Variablen `$EDITOR` benannten Editor zur Bearbeitung dieser

[11] Sind keine separaten Controller vorhanden, wird der Zugriff in jedem Fall langsamer, weil ja entsprechend mehr Daten zu bewegen sind.

[12] ACLs sind auch auf anderen UNIX–Systemen als eine herstellerspezifische Erweiterung verfügbar.

Datei. Nach Beendigung des Editorprogrammes wird die Textdarstellung der ACL eingelesen und aktiviert-

Die Textdarstellung einer ACL ist in Abbildung 1.27 zu sehen. Sie gliedert sich in drei Abschnitte, die jeweils mit den Überschriften *attributes*, *base permissions* und *extended permissions* eingeleitet werden. Die beiden ersten Abschnitte stellen dabei die normalen UNIX–Rechte einer Datei dar, der dritte Abschnitt enthält die erweiterten Zugriffsrechte des ACL–Mechanismus.

Als Dateiattribute bezeichnet AIX dabei die beiden s–Bits und das t–Bit, die als `SETUID`, `SETGID` und `SVTX` bezeichnet werden. Um eines oder mehrere dieser Bits zu setzen, sind die entsprechenden Namen (ggf. durch Komma getrennt) in der `attributes`–Zeile einzutragen.

```
$ aclget klausur4.tex
attributes:
base permissions:
        owner(kris):    rw-
        group(other):   r--
        others:         ---
extended permissions:
        enabled
        specify         rw- u:kub
        deny            rwx g:kurs
        permit          -w- u:alf, g:lehrer
```

Abbildung 1.27: AIX Access Control List

Im folgenden Abschnitt werden die normalen rwx–Rechte nach Eigentümer, Gruppe und Rest der Welt getrennt aufgeschlüsselt. Sie können durch Editieren der rwx–Abschnitte der entsprechenden Zeilen modifiziert werden. Der Name des Dateieigentümers bzw. der Gruppe können nicht geändert werden. Dazu ist ein ausdrücklicher `chown` oder `chgrp`–Befehl notwendig.

Der letzte Abschnitt gibt schließlich die erweiterten Zugriffsrechte an. Die erste Zeile dieses Abschnittes ist entweder `enabled` oder `disabled`. Die auf dieses Schlüsselwort folgenden Zeilen haben nur dann eine Bedeutung, wenn es auf `enabled` steht.

Mit den erweiterten Zugriffsrechten kann man die Rechte für eine einen einzelnen Benutzer oder eine Gruppe entweder *definieren* (Schlüsselwort `specify`), *vergrößern* (Schlüsselwort `permit`) oder *verkleinern* (Schlüsselwort `deny`).

Wenn man für einen bestimmten Benutzer Zugriffsrechte definiert, dann gilt für ihn genau das in der `specify`–Zeile angegebene rwx–Tripel, unabhängig von den Base Permissions der Datei. Im Beispiel hat der Benutzer `kub` genau die Zugriffsrechte `rw-` an der Datei, egal welche Base Permissions die Datei hat.

Erhöht man die Zugriffsrechte eines Benutzers, dann gelten für ihn das in den Base Permissions angegebene rwx–Tripel, dem die Rechte aus den erweiterten Zugriffsrechten hinzugefügt worden sind. Im Beispiel hat der Benutzer `alf`, wenn er in der Gruppe `lehrer` ist, die Zugriffsrechte aus den Base Permissions, aber zusätzlich in jedem Fall Schreibrecht.

Genauso kann man einzelnen Benutzern oder Benutzergruppen Zugriffsrechte verweigern. Im Beispiel hat die Gruppe `kurs` in jedem Fall keine Rechte an der Datei.

Eine Zeile in einer ACL heißt *access control entry (ACE)*. Ein ACE kann Angaben zu einem Benutzer oder zu einer Gruppe enthalten. Benutzer werden in der Form `u:name`

angegeben, Gruppen werden in der Form `g:name` notiert. Wird mehr als ein Benutzer oder eine Gruppe in einem ACE genannt, dann müssen alle Eigenschaften erfüllt sein, damit der ACE gültig wird. Der `permit`–Eintrag aus dem Beispiel gilt also nur für den Benutzer `alf`, solange er auch in der Gruppe `lehrer` ist. Er gilt nicht für andere Benutzer der Gruppe `lehrer`. Wollte man den Zugriff für `alf` **oder** die Gruppe `lehrer` erlauben, müßte man zwei Einträge machen:

```
        permit          -w- u:alf
        permit          -w- g:lehrer
```

Zugriffsrechte aus ACLs werden immer in der am meisten einschränkenden Weise miteinander kombiniert. Würde man einer Gruppe in einem ACE den Zugriff auf eine Datei erlauben, einem einzelnen Benutzer in einem anderen ACE den Zugriff aber wieder verbieten, so hätte der betreffende Benutzer keinen Zugriff auf die Datei. Im Beispiel: Würde man `alf` aus der Gruppe `lehrer` den Schreibzugriff auf eine Datei geben, der Gruppe `lehrer` aber wieder entziehen, hätte `alf` keinen Schreibzugriff auf die Datei:

```
attributes:
base permissions:
        owner(kris):    rw-
        group(other):   r--
        others:         ---
extended permissions:
        enabled
        permit          -w- g:lehrer
        deny            -w- u:alf
```

Damit ein Benutzer ein Recht an einer Datei hat, muß ein Eintrag existieren, der ihm das entsprechende Recht gibt und es darf kein Eintrag existieren, der es ihm ausdrücklich wieder nimmt.

1.6 Datensicherung

Ein wichtiger Punkt beim Betrieb eines Mehrbenutzersystems ist die Datensicherung. Verlust oder Beschädigung von Dateien oder gar ganzen Dateisystemen sind hier besonders kostspielig, weil bei einem Ausfall nicht nur die Daten einer Person, sondern unter Umständen eines ganzes Teams oder einer ganzen Abteilung verloren gehen. Rechnet man die Zeitstunden zur Rekonstruktion oder Neuerstellung der verlorenen Daten ab – oftmals viele Mannjahre –, dann kann ein einfacher Festplattenausfall leicht gigantische Kosten erzeugen. Gegen solche Verluste schützt nur ein gut ausgearbeitetes und vor allen Dingen auch in die Tat umgesetztes Backupkonzept.

Zur Sicherung von Daten stehen eine ganze Reihe von Medien zur Verfügung:

Disketten Diskettenlaufwerke findet man praktisch überall. Jedoch ist selbst mit Datenkompression die Kapazität einer Diskette in der Regel viel zu klein, um damit eine sinnvolle Datensicherung durchführen zu können. Disketten sind bestenfalls geeignet, wenige wichtige Systemkonfigurationsdateien zu sichern.

QIC Tapes Viertelzollkassetten (*quarter inch cartridges*) für SCSI–Bandlaufwerke werden an den meisten UNIX–Workstations zur Datensicherung und zum Datentausch eingesetzt. Wegen ihrer großen Zuverlässigkeit und der leichten Handhabung sind sie sehr verbreitet. Häufige Standardkapazitäten sind 150 und 525 Megabyte, es gibt jedoch auch Bandlaufwerke, die 1 oder 2 Gigabyte aufzeichnen.

NAME

mt – Programm zur Wartung von Magnetbändern

SYNTAX

`mt -f` *tapename kommando count*

OPTIONS

eof, weof Schreiben von *count* Endemarken an der aktuellen Bandposition.

fsf, bsf Vorspulen (Zurückspulen) um *count* Endemarken von der aktuellen Bandposition.

rewind Zurückspulen des Bandes.

offline Band zurückspulen und offline nehmen. Das Band kann danach entnommen werden.

status Statusinformation über das Bandgerät anzeigen.

Abbildung 1.28: `mt`–Kommando

DAT und Exabyte Bänder Noch höhere Kapazitäten haben Bandlaufwerke mit rotierenden Köpfen. Die Datensicherung erfolgt auf speziellen Audio– oder Videokassetten mit 4 oder 8 mm Breite. Auf ein Band passen in der Regel 2, 8 oder mehr Gigabyte Daten. Genau wie Videorecorder sind solche Bandlaufwerke jedoch relativ empfindlich, was Spurlage und Temperatur angeht.

WORM und optische Speicher Optische Speichermedien sind im allgemeinen langsam und nicht oder nur einmal zu beschreiben, haben jedoch sehr große Kapazitäten. Diese Eigenschaft macht sie für die Zwecke der Datensicherung unbrauchbar. Auf der anderen Seite existieren Anwendungen, in denen es erwünscht ist, ein sicheres, nicht fälschbares Logbuch der Veränderungen eines Datenbestandes zu haben. Hier sind derartige *write once, read many* Speicher die ideale Lösung.

externe Fest– und Wechselplatten In einigen Fällen kann es sinnvoll sein, Datenbestände auf eine externe Fest– oder Wechselplatte zu schreiben. Insbesondere wenn ein übersichtlicher Datenbestand regelmäßig komplett gesichert werden muß, kann dies die ideale Lösung sein. Im Normalfall liegen die Kosten pro Megabyte bei diesen Medien jedoch deutlich über denen von Bandgeräten.

In den allermeisten Fällen wird ein Bandlaufwerk für Viertelzollbänder oder DAT–Laufwerk das Datensicherungsmedium der Wahl sein. Dieses Gerät ist in den meisten Fällen unter dem Namen `/dev/rmt*` , `/dev/rst*` oder `/dev/tape` im System anzusprechen. Meistens existiert auch noch ein zugehöriges Laufwerk unter dem Namen `/dev/nrmt*`, `/dev/nrst0` oder `/dev/ntape`.

Zur Wartung von Bändern, d.h. zum Umspulen des Bandes, zum Überspringen von Aufzeichnungen und zum Schreiben von Bandmarken dient der `mt`–Befehl. Seine genaue Syntax in Abbildung 1.28 beschrieben. Die meisten Optionen des Kommandos arbeiten relativ zur aktuellen Bandposition. Sein Einsatz ist deswegen vor allen Dingen im Zusammenhang mit den nicht zurückspulenden Bandlaufwerken sinnvoll.

So kann man durch `rewind`– und `fsf`–Kommandos zum Beispiel mehr als ein Archiv auf ein Band schreiben. Ein Beispiel dafür wird in Abbildung 1.30 nach der Erklärung des `tar`–Kommandos gegeben.

1.6.1 Backup mit tar

Das Kommando `tar` (der Name kommt von *tape archive*) dient zum Archivieren von Dateien. Die Sicherung kann auf einem Bandgerät oder einer Diskette erfolgen. Es ist aber auch möglich, einen Satz von Dateien in einer gewöhnlichen Datei zu archivieren. Sowohl Einpacken als auch Auspacken eines Archives erfolgen mit demselben Kommando, wobei die Datenrichtung dabei durch die erste Option des Kommandos bestimmt wird. Die wichtigsten dieser Optionen sind:

-c *create* Anlegen eines neuen Archives. Die Dateien werden an den Anfang des Archives geschrieben, ein möglicherweise vorhandener Archivinhalt wird überschrieben.

-t *table of contents* Die Namen der benannten Dateien werden angezeigt, sofern sie im Archiv vorhanden sind. Wird ein Verzeichnis angegeben, wird sein Inhalt und alle seine Unterverzeichnisse ausgepackt. Der Default ist, den gesamten Archivinhalt anzuzeigen.

-x *extract* Auspacken eines Archives. Die benannten Dateien werden aus dem Archiv extrahiert. Auch hier wird rekursiv abgestiegen.

Es muß genau eine dieser Optionen angegeben werden, damit das Kommando weiß, was der Benutzer mit dem Archiv machen möchte. Die folgenden Optionen (wieder nur eine Auswahl) können dann angegeben werden, um das weitere Verhalten von `tar` zu bestimmen.

-v *verbose operation* Normalerweise arbeitet `tar`, ohne weitere Ausgaben zu erzeugen. Durch Angabe der Option `-v` weist man den Befehl an, beim Schreiben und Lesen von Archiven den Namen jeder Datei zu drucken, bevor sie gesichert wird. Beim Anzeigen eines Inhaltsverzeichnisses bewirkt die Option die Anzeige einer Langform des Verzeichnisses.

-f *file* Das nächste Argument nach der Option ist der Name eines Devices oder einer Datei. Das Archiv wird unter diesem Namen gelesen oder geschrieben. Der Dateiname `-` steht dabei je nach Datenrichtung für die Standardeingabe bzw. Standardausgabe.

-w *wait* Wird die Option `-t` angegeben, gibt `tar` jeden Dateinamen aus und wartet auf eine Benutzereingabe. Wenn diese Eingabe ein `y` ist, wird die entsprechende Datei ein– oder ausgepackt, in allen anderen Fällen tut `tar` nichts.

Um also das eigene `$HOME`–Verzeichnis auf das Band `/dev/rst0` zu sichern, muß das Kommando `tar -cvf /dev/rst0 $HOME` angegeben werden. `tar` gibt wegen der `-v`–Option während der Sicherung die Namen der gesicherten Dateien aus, sodaß man den Backupvorgang am Bildschirm verfolgen kann. Der Inhalt des Bandarchivs kann dann mit `tar -tvf /dev/rst0` eingesehen werden und mit `tar -xvf /dev/rst0` komplett wieder eingelesen werden.

Dabei ist zu beachten, daß `tar` absolute Pfadnamen auch als absolute Pfade im Archiv speichert. Das kann beim Wiedereinspielen des Archivs auf einem anderen System sehr ärgerlich sein, wenn das Verzeichnis dort nicht existiert oder nur auf einer Platte existiert, die nicht genügend freien Platz enthält. `tar` sieht keine Möglichkeit vor, absolute Pfadnamen beim Wiedereinspielen umzubenennen oder das Problem sonstwie zu korrigieren. Daher sollte man schon beim Erzeugen des Archivs darauf achten, relative Pfadnamen zu erzeugen. Im Beispiel oben sollte besser das Kommando `cd /; tar -cvf /dev/rst0 ./$HOME` zum Erzeugen des Archivs verwendet werden.

Ein tar–Archiv muß nicht auf einer Gerätedatei erzeugt werden. Stattdessen kann es auch sinnvoll sein, einen Satz von vielen Dateien und Verzeichnissen (etwa ein Softwarepaket) in einer Archivdatei

Das Datenformat von tar ist über Systemgrenzen hinweg kompatibel. UNIX–Software wird deswegen oft in Form eines Tape–Archives verteilt, das gelegentlich noch mit einem der beiden Packprogramme compress oder gzip gepackt worden ist. Eine solche Archivdatei hat per Konvention die Endung .t oder .tar, wenn sie gepackt ist, also .t.Z bei compress oder .t.gz bei gzip[13]. Praktisch alle Public–Domain–Softwarepakete für UNIX werden als gepacktes Tapearchiv verteilt.

```
$ tar cf bigtar.t Anleitung
$ ls -l bigtar.t
-rw-r--r--  1 kris other  9127430 Feb  8 19:10 bigtar.t
$ gzip -9 bigtar.t
gzip: /dev/sda2 -- disk full

$ rm bigtar.t
$ tar -cf - ../Anleitung | gzip -9 > bigtar.t.gz
$ ls -l bigtar.t.gz
-rw-r--r--  1 kris other   534789 Feb  8 19:11 bigtar.t.gz

$ gzcat bigtar.t.gz | ( cd /zielverzeichnis; tar -xf -)

$ ( cd /quellverzeichnis; tar cf - .) |
> ( cd /zielverzeichnis;  tar xvf -)
```

Abbildung 1.29: Arbeiten mit komprimierten Archiven ohne Zwischendatei

Solche Archive können sehr groß werden. Packt man sie mit zwei getrennten gzip– und tar–Befehlen ein oder aus, kann es sein, daß man wegen der benötigten Zwischendateien Platzprobleme bekommt. Abbildung 1.29 zeigt, wie man das Problem umgehen kann. Im ersten Abschnitt wird das Archiv bigtar.t erzeugt. Beim Packen der Datei wird für kurze Zeit Platz für die gepackte und die ungepackte Datei benötigt, der aber nicht vorhanden ist. Im zweiten Abschnitt wird das Archiv erzeugt und auf die Standardausgabe geschrieben. Die Standardausgabe ist über eine Pipeline die Standardeingabe einem Kompressionsprogrammes, hier gzip. Die komprimierte Ausgabe von gzip wird schließlich in eine Datei gelenkt. Um das Archiv auszupacken, geht man genau umgekehrt vor: Der gzcat–Befehl, ein Synonym für gzip -dc, entpackt eine Datei und gibt das Resultat auf der Standardausgabe aus. Diese bildet über eine Pipeline die Eingabe eines tar -xf -–Kommandos. Als Besonderheit wurde das Kommando hier in einer Subshell (in runden Klammern, siehe **??**) gestartet, deren aktuelles Verzeichnis verlegt worden ist. Da das Archiv relative Pfadnamen enthält, wird es an anderer Stelle im Dateibaum, nämlich unterhalb von /zielverzeichnis ausgepackt. Das kann man benutzen, um ganze Dateibäume zu verschieben, wie im vierten Abschnitt gezeigt wird.

Beim Schreiben auf Band kann man durch die Kombination von mt und tar erreichen, daß mehr als ein Archiv auf dem Band Platz findet. In Abbildung 1.30 werden zwei Verzeichnisse mit zwei getrennten tar–Kommandos auf dem Band aufgezeichnet. Nach dem Ende jedes der beiden Einzelbackups wird jeweils eine Filemarke auf das Band geschrieben. Die Backups stehen dadurch jeweils als einzelne Dateien auf dem Band und können mit dem mt–Kommando übersprungen werden.

[13] Man bezeichnet solche Dateien manchmal auch als *geteert und gefedert*.

```
$ # Band zurueckspulen
$ mt -f /dev/nrst0 rewind
$ # Zwei Archive aufzeichnen
$ tar -cf /dev/nrst0 ../Afbb
$ tar -cf /dev/nrst0 ../Anleitung
$ # Zurueckspulen und das erste Archiv ueberspringen
$ mt -f /dev/nrst0 rewind
$ mt -f /dev/nrst0 fsf
$ # Bandinhalt kontrollieren
$ tar -tvf /dev/nrst0
rwxr-xr-x102/20      0 Oct 12 17:32 1993 ../Anleitung/
rw-r--r--102/20   8253 Oct 12 08:47 1993 ../Anleitung/anl.tex
[ ... geloescht ... ]
```

Abbildung 1.30: Mehrere `tar`–Archive auf einem Band

1.6.2 Backup mit cpio

Ein anderes Backup-Programm, das UNIX serienmäßig mitliefert, ist `cpio`, der Name steht für *copy in/out*. Das Programm leistet grundsätzlich dasgleiche wie `tar`, ist aber in den Kommandozeilenoptionen und im Archivformat inkompatibel. Welches Program man letztendlich verwendet, um seine Daten zu sichern, ist vermutlich Geschmacks– und Gewohnheitssache.

Die Grundidee von `cpio` ist die eines Konverters: Das Programm nimmt auf der Standardeingabe eine Liste von Dateinamen, wie sie etwa der `find`–Befehl generiert. Es liest diese Dateien ein und erzeugt auf der Standardausgabe ein Archiv dieser Dateien. Dieses Archiv kann auf Band beschrieben werden oder anderweitig bearbeitet werden.

Wie auch bei `tar` dient ein Kommando gleichzeitig als Lese– und Schreibprogramm für die von ihm erzeugten Archive. Die Datenrichtung wird dabei wieder durch eine Hauptoption festgelegt. Auch `cpio` kann zum Verschieben von ganzen Dateibäumen eingesetzt werden und stellt dazu eine besondere Option bereit:

-i *copy in* Auf der Standardeingabe wird ein Archiv gelesen und in Dateien zerlegt.

-o *copy out* Das Kommando liest eine Liste von Dateinamen auf der Standardeingabe und erzeugt aus diesen Dateien ein Archiv, welches auf der Standardausgabe ausgegeben wird.

-p *pass through* Kombiniert die beiden oben erwähnten Arbeitsgänge in einem: Auf der Standardeingabe wird eine Liste von Dateinamen gelesen, die in anzugebendes Zielverzeichnis verschoben werden.

Diese Hauptoptionen können wieder durch eine Liste von Zusatzoptionen ergänzt werden. Wieder soll nur eine Auswahl der wichtigsten Optionen gezeigt werden:

-c *compatibility* Die Option bewirkt, daß die Archivheader in ASCII und damit unabhängig von der Word–Order der Maschine geschrieben werden. Solche Archive sind von einer Maschine auf eine andere portabel. Die Option sollte eigentlich immer mit angegeben werden.

-t *show table of contents* (Nur bei *copy in*). Es wird ein Inhaltsverzeichnis des Archivs angezeigt, aber es werden keine Dateien ausgepackt.

-v *verbose* Das Inhaltsverzeichnis wird in einer Langform angezeigt bzw. `cpio` druckt den jeweils in Bearbeitung befindlichen Dateinamen aus.

-d *create directories* (Nur bei `-i` und `-p`) Verzeichnisse werden angelegt, wie sie benötigt werden.

-l *link when possible* (Nur bei `-p`) Wenn immer möglich werden die Zieldateien nicht als Kopien, sondern als Hardlinks auf die Originale erstellt.

-m *retain modification time* (Nur bei `-i` und `-p`) Die Modifikationsdaten der Dateien werden auf die Werte der Originaldateien und nicht auf das Datum des Zurückspielens gesetzt.

-u *copy unconditionally* (Nur bei `-i` und `-p`) Normalerweise überschreibt `cpio` keine Dateien, die neuer sind als Dateien gleichen Namens im Archiv. Dadurch wird verhindert, daß beim Wiedereinspielen des Archivs alte Versionen von Dateien gelesen werden. Mit dieser Option wird dieser Mechanismus deaktiviert, d.h. es werden vorhandene Dateien unbedingt durch den Inhalt des Archivs überschrieben.

Datenrichtung	Kommando
Schreibend	`tar cvf - .` `find . -depth -print | cpio -ocv`
Lesend	`cat archiv.t | tar xvf -` `cat archiv.cpio | cpio -icudvm`
Verzeichnis	`cat archiv.t | tar tvf -` `cat archiv.cpio | cpio -ictv`

Abbildung 1.31: Vergleich von `cpio` und `tar`

Der Trick bei der Verwendung von `cpio` (und auch `tar`) liegt darin, eine Liste der Dateien zu generieren, die zu sichern sind. Das ist im Werkzeugkasten–Konzept von UNIX natürlich nicht Aufgabe des Datensicherungsprogrammes. Dieses hat nur die Aufgabe, diese Liste von Dateien irgendwie zu bearbeiten.

Eine Möglichkeit, eine solche Liste von Dateien zu generieren, ist der `find`–Befehl. Er bietet eine Vielzahl von Optionen, nach denen man Dateien durchsehen und zum Backup auswählen kann. Abbildung 1.32 zeigt, wie ein Script zur Erzeugung eines `cpio`–Archives aussehen kann.

Das Script erwartet eine Konfigurationsdatei, die eine Liste von zu sichernden Verzeichnissen, jeweils ein Verzeichnisname pro Zeile, enthält. Die Liste kann Kommentarzeilen, die mit einem Doppelkreuz beginnen müssen, enthalten. In den Backticks wird diese Datei ausgegeben und alle Zeilen, die mit einem Doppelkreuz beginnen, werden ausgefiltert: Die `-v`–Option von `egrep` liefert alle Zeilen, auf die das Suchmustern nicht paßt, d.h. alle Zeilen, die kein Doppelkreuz am Zeilenanfang haben.

Durch den Wechsel in das Hauptverzeichnis und den Zugriff auf `./$i` im `find` werden aus eventuell vorhandenen absoluten Pfadnamen relative Pfade. In der `for`–Schleife werden alle Worte in der Variablen `$VERZEICHNISSE` durchgegangen. Das in der Schleife enthaltene `find` druckt rekursiv alle Namen von Dateien und Unterverzeichnissen des angegebenen Verzeichnisses aus. Dabei macht es nichts, wenn ein Wort in der Variablen kein Verzeichnis, sondern eine einzelne Datei bezeichnet: `find <datei> -depth -print` druckt dann den Namen dieser Datei unverändert aus.

Die Option `-depth` bewirkt dabei, daß die Namen von Verzeichnissen erst ausgegeben werden, wenn alle Dateien, die in ihnen enthalten sind, ausgegeben worden sind. Auf diese Weise wird ein Verzeichnis erst nach den Dateien gesichert, die es enthält. Das ist beim

```
#! /bin/sh --
# Konfigurationsdatei
CONFIG=/usr/adm/backuplist

# Liste der zu sichernden Verzeichnisse einlesen
VERZEICHNISSE="`cat $CONFIG | egrep -v '^#'`"

# Aus dieser Liste eine Liste der Dateinamen
# erzeugen, dabei relative Pfade generieren
# und als Eingabe in cpio verwenden.
cd /
for i in $VERZEICHNISSE
do
        find ./$i -depth -print
done |
cpio -ocv > /dev/rst0
```

Abbildung 1.32: Erzeugung eines Archivs mit `find` und `cpio`

Wiedereinspielen wichtig: Wenn ein Verzeichnis mit den Rechten 000 erzeugt wird, bevor alle Dateien vorhanden sind, die es enthalten soll, gibt es beim Einlesen Probleme mit den Zugriffsrechte. Dadurch, daß das Verzeichnis selbst nach allen seinen Dateien gesichert wird, ist sichergestellt, daß dieses Problem nicht auftreten kann.

Die Schleife generiert eine Liste von Dateinamen der zu sichernden Dateien, die in dieser Form als Eingabe für `cpio` dienen kann. Hier wird dann aus diesen Namen ein Archiv generiert, das dann auf Band geschrieben wird.

Mögliche Erweiterungen für das Script bestehen in weiteren Optionen für den `find`–Befehl. So besteht möglicherweise kein Bedarf, Dateien mit bestimmten Namen oder Endungen zu sichern[14]. Möglicherweise möchte man nicht alle Daten sichern, sondern nur Dateien finden, die sich in den letzten paar Tagen verändert haben. Derlei Filter sind mit einigen zusätzlichen Optionen zu `find` oder durch ein `egrep` zwischen der Schleife und dem `cpio` schnell programmiert. Auch die Ausgabe von `cpio` kann noch weiter bearbeitet werden: Wenn man etwa ein komprimiertes Backup machen möchte, kann man hier noch ein `gzip` oder `compress` einsetzen[15].

1.6.3 Weitere Kommandos

Zum geblockten Lesen und Schreiben von Bändern und Disketten existiert in UNIX das Kommando `dd`. Ähnlich wie `cat` kopiert `dd` standardmäßig seine Standardeingabe auf die Standardausgabe. Im Gegensatz zu `cat` können dabei jedoch Eingabe– und Ausgabeblockgrößen vorgegeben werden sowie verschiedene Konvertierungen angegeben werden. Die Syntax von von `dd`–Optionen unterscheidet sich von der Syntax regulärer UNIX–Befehle. Optionen haben bei `dd` die Form `name=wert`.

[14] Dateien mit den Namen `core` und den Endungen `.bak`, `.o` oder Tilde zum Beispiel.

[15] Ein Backup mit Datenkompression kann viel Platz sparen und sogar schneller sein, als ein Backup ohne Kompression. Bei Datenkompression beziehen sich aber Daten auf dem Band andere Daten, die davor auf dem Band stehen. Das bedeutet, daß Lesefehler auf einem Band nicht einfach übersprungen werden können. Stattdessen wird durch einen Lesefehler der gesamte Rest des Bandes unlesbar.

Wenn der Wert eine Zahl ist, kann sie als dezimale Zahl angegeben werden oder ihr kann ein Buchstabe als Multiplikator nachgestellt werden. Der Buchstabe `k` steht dabei für den Faktor 1024, der Buchstabe `b` steht für den Faktor 512 und der Buchstabe `w` steht für den Faktor 2. Die Angabe 1024k steht also für 1024 Kilobyte, die Angabe 5b für 5 Blöcke a 512 Byte und die Angabe 17w für 17 Worte a 2 Byte. Außerdem können Produkte als Paare von Zahlen mit einem `x` in der Mitte angegeben werden. Eine Angabe der Form `17x13` bezeichnet also 221 Bytes.

`dd` versteht die Optionen

if=*datei* *input file* Der Name der Eingabedatei. Wird der Parameter weggelassen, liest `dd` von der Standardeingabe.

of=*datei* *output file* Der Name der Ausgabedatei. Wird der Parameter weggelassen, schreibt `dd` auf die Standardausgabe.

ibs=*n* *input block size* Die Größe der Blöcke, die von der Eingabe gelesen werden. Wird der Parameter weggelassen, nimmt das Programm den Standardwert von 512 Byte.

obs=*n* *output block size* Die Größe der Blöcke, die auf die Ausgabe geschrieben werden. Wird der Parameter weggelassen, schreibt das Programm den Standardwert von 512 Byte.

cbs=*n* *conversion buffer size* Die Größe eines internen Konvertierungspuffers. Der Puffer wird nur bei bestimmten Konvertierungen benötigt.

bs=*n* *block size* Setzt gleichzeitig den Ausgabe– und Eingabepuffer auf die gleiche Größe. Wenn außerdem keine Umwandlung mit dem Parameter `conv=` angegeben wurde, spart sich `dd` das Kopieren der Daten vom Eingabe– in den Ausgabepuffer, sondern arbeitet mit einem Puffer.

skip=*n* Überspringt *n* Eingaberecords, bevor das Kopieren der Eingabe auf die Ausgabe einsetzt.

seek=*n* Steuert den *n*ten Record der Eingabe an, bevor das Kopieren der Eingabe auf die Ausgabe einsetzt. `seek` und `skip` haben letztendlich die gleiche Wirkung, aber einige Bandgeräte können bei einem `seek` wesentlich schneller positionieren, als sie Daten lesen können.

count=*n* Kopiert maximal *n* Blöcke von der Eingabe auf die Ausgabe.

conv=*key,...* Konvertiere die Daten beim Kopieren von der Eingabe auf die Ausgabe. Die Konvertierung wird dabei durch die angegeben Schlüsselworte *key* festgelegt. Werden mehrere Schlüsselworte angegeben, sind diese durch Kommata ohne Spaces zu trennen. Mögliche Konvertierungen sind:

- **ascii** Konvertiert EBCDIC nach ASCII.
- **ebcdic** Konvertiert ASCII nach EBCDIC.
- **ibm** Konvertiert ASCII nach IBM-EBCDIC.
- **lcase** Konvertiert Buchstaben in Kleinbuchstaben.
- **ucase** Konvertiert Buchstaben in Großbuchstaben.
- **swab** Vertauscht jeweils ein Paar von Bytes. Diese Konvertierung erlaubt das Einlesen von Archiven, die auf einer Maschine mit anderer Byteordnung geschrieben wurden.
- **noerror** Ignoriert Lesefehler.

```
$ dd if=/dev/fd0 of=disk-image bs=18k count=80
[ Diskette wechseln ]
$ dd if=disk-image of=/dev/rfd0 bs=18k count=80

$ tar cf - . | dd of=/dev/rst0 bs=256k

$ cat singlechar
#! /bin/sh --
stty raw -echo
zeichen=`dd count=1 bs=1 2>/dev/null`
stty -raw echo
echo Sie haben $zeichen eingegeben.
$ singlechar
Sie haben m eingegeben.
```

Abbildung 1.33: Beispiel für Anwendungen von dd.

unblock Konvertiert Zeilen mit fester Länge und Leerzeichen als Füllzeichen in normale Zeilen variabler Länge und mit einem Newline am Ende. Die feste Zeilenlänge wird durch die Angabe einer `cbs`–Größe bestimmt.

block Konvertiert normale Zeilen in Zeilen fester Länge. Die Zeilen werden am Ende mit Leerzeichen aufgefüllt. Die feste Zeilenlänge wird auch hier durch die Angabe einer `cbs`–Größe bestimmt.

Einige Beispielanwendungen von `dd` finden sich in Abbildung 1.33. Das erste Beispiel zeigt, wie sich der Inhalt einer 1.44 MB Diskette als 80 Blöcke zu jeweils 18 KB lesen und in einer Datei abspeichern läßt. Nach dem Wechseln der Diskette wird das Disk-Image auf eine andere Diskette zurück geschrieben. Im zweiten Beispiel wird mit dem `tar`–Befehl ein Archiv auf der Standardausgabe erzeugt und mit dem `dd`–Kommando in Blöcken zu 256 KB auf ein Bandgerät geschrieben. Durch experimentieren mit der Blockgröße kann man bei einigen Bandgeräten den Durchsatz deutlich steigern.

Das dritte Beispiel ist etwas komplizierter. Der `dd`–Befehl liest einen einzigen Block von nur einem Byte Länge. Die Ausgabediagnostik des Kommandos wird durch die Umlenkung der Standardfehlerausgabe verworfen. Damit der Terminaltreiber das Zeichen sofort und ohne Druck auf die Returntaste bereitstellt, wird er mit einem `stty`–Befehl zuvor in den Raw–Mode versetzt. Durch das Abschalten des Echos wird zugleich die Ausgabe des eingegebenen Zeichens an den Benutzer unterdrückt. `dd` liest also ein Zeichen vom Benutzer, ohne auf das Drücken der Returntaste zu warten. Das Zeichen wird auf der Standardausgabe ausgegeben und wegen der Backticks also der Variablen `$zeichen` zugewiesen. Dort kann es weiter bearbeitet werden.

Diese Art, ein einzelnes Zeichen ohne Returntaste einzulesen ist bemerkenswert ineffizient. Immerhin müssen drei Programme geladen werden (stty, dd und wieder stty), um ein einzelnes Zeichen zu lesen, aber sie demonstriert, daß es allein mit UNIX Bordmitteln möglich ist, das Problem zu lösen.

`dd` kann verwendet werden, um den Anfang eines unbekannten Bandes oder einer unbekannten Diskette anzusehen und den Typ des Archivs zu bestimmen. Archive, die von `cpio` ohne Einsatz der Option `-c` geschrieben wurden, beginnen mit der oktalen Zahlenfolge 070707. In Abbildung 1.34 ist im ersten Beispiel der Anfang eines solchen Archivs in Oktal, Hexadezimal und ASCII zu sehen. Wird alternativ die Option `-c` mit angegeben, schreibt cpio den Archivheader als ASCII-String. Das zweite Beispiel zeigt, wie der Archivheader 070707 jetzt also ASCII–String auftaucht. `tar`–Archive beginnen dagegen

```
[ cpio ohne -c ]
$ dd if=/dev/rst0 | od -oxa | head -3
0000000   070707  003010  024326  100644  000146  000024  000001  153760
            71c7    0608    28d6    81a4    0066    0014    0001    d7f0
           q   G ack  bs   (   V soh   $ nul   f nul dc4 nul soh   W   p

[ cpio mit -c ]
$ dd if=/dev/rst0 | od -xa | head -4
0000000     3037    3037    3037    3030    3330    3130    3032    3433
           0   7   0   7   0   7   0   0   3   0   1   0   0   2   4   3
0000020     3236    3130    3036    3434    3030    3031    3436    3030
           2   6   1   0   0   6   4   4   0   0   0   1   4   6   0   0

[ tar ]
$ dd if=/dev/rst0 | od -xa | head -4
0000000     3031    5f53    6865    6c6c    2e61    7578    0000    0000
           0   1   _   S   h   e   l   l   .   a   u   x nul nul nul nul
0000020     0000    0000    0000    0000    0000    0000    0000    0000
         nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul

[ MS-DOS Diskette auf NeXT erstellt ]
$ dd if=/dev/fd0 | od -a | head -5
0000000    i nul nul   N   E   X   T  sp  sp  sp  sp nul stx soh soh nul
0000020  stx   ` nul   @  vt   p  ht nul dc2 nul stx nul nul nul nul nul
0000040  nul nul nul nul nul nul nul nul nul nul nul   N   A   M   E   N
0000060    L   O   S   E   D   I nul nul nul nul nul nul nul nul nul nul
0000100  nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul nul
```

Abbildung 1.34: Katalogisierung unbekannter Band– und Disketteninhalte

ohne spezielle Kennung direkt mit einem Dateinamen, wie im dritten Beispiel zu sehen ist. Das vierte Beispiel zeigt den Kopf einer Diskette im MS-DOS Format. Ab Byteposition 3 steht eine Kennung des Formatierprogrammes (die Beispieldiskette wurde auf einer NeXTstation formatiert), weiter hinten kann man den Namen der Diskette sehen.

```
$ hostname
mahaki
$ tar cvf - . | rsh black -l backup dd of=/dev/rst0
```

Abbildung 1.35: Backup im Netzwerk

`dd` wird auch oft in Zusammenhang mit den noch zu besprechenden Netzwerkkommandos von UNIX eingesetzt, um auf dem Bandlaufwerk eines anderen Rechners ein Band einzulesen oder zu schreiben. In Abbildung 1.35 werden Kommandos auf dem Rechner `mahaki` eingegeben. Dort erzeugt `tar` ein Archiv des aktuellen Verzeichnis auf der Standardausgabe. Das Kommando `rsh black -l backup` nimmt einen Befehl als Parameter und startet diesen auf dem angebenen Rechner `black` und dem angegebenen Loginnamen `backup`. Die Standardausgabe des Kommandos `tar` auf mahaki wird also über eine Pipeline durch das Netz an das Kommando `dd` gesendet, das auf dem Rechner `black` läuft. Dort schreibt `dd` auf das Bandlaufwerk `/dev/rst0`, das an `black` angeschlossen ist.

1.6.4 Geplantes Backup und Automatisierung

Das Anfertigen von Backups ist eine sehr langweilige und häufig auftretende Tätigkeit. Deswegen ist es wichtig, Backups einmal zu planen und das eigentliche Herstellen der

Sicherheitskopien zu möglichst weitgehend zu automatisieren, um sicherzustellen, daß sie auch gemacht werden.

Bei der Planung von Backups definiert man sich Sätze von Dateien und Verzeichnissen, die zusammen auf ein Band gesichert werden sollen. Einen solchen Backupsatz nennt man ein *volume*. Für jedes Volume muß festgelegt werden, in welchem *Zyklus* bzw. mit welcher *Frequenz* es gesichert werden muß. Die Frequenz, mit der ein Volume gesichert werden muß, hängt von der Lebensdauer der Dateien ab, die sich darin befinden. Ein Band mit dem Betriebssystem wird immer dann gesichert werden müssen, wenn die Systemkonfiguration geändert worden ist, ein Band mit Homeverzeichnissen von Benutzern wird dagegen mindestens einmal zum Wochenabschluß gesichert werden. Andere Daten können so kurzlebig sein, daß sie täglich gesichert werden müssen. Ob Daten gesichert werden müssen oder nicht, hängt von ihrer Wichtigkeit ab: Dateien in `/tmp` werden sicher nicht gesichert werden müssen.

Aus der Definition von Volumes und den Sicherungsfrequenzen ergibt sich ein Backup–Plan (*backup schedule*). Er gibt an, an welchem Tag welche Volumes zur Sicherung anstehen. Das Abarbeiten des Backup–Plans sollte automatisch geschehen: Ein Shellscript sollte den Operator informieren, welche Bänder zur Sicherung wo einzulegen sind und wie sie zu markieren sind.

Beim Anfertigen des Plans sollte man berücksichtigen, daß in sich stimmige Sicherungen von Daten auf einem Dateisystem nur dann möglich sind, wenn während der Zeit der Sicherung keine Arbeiten auf dem Dateisystem durchgeführt werden. Das bedeutet streng genommen, daß das zu sichernde Dateisystem mit `umount` abgemeldet und an einer Stelle mit `mount` angemeldet worden ist, zu der nur der Systemverwalter Zugang hat. In der Praxis wird dies kaum gemacht, weil es bedeutet, daß das System für die Zeit der Datensicherung nicht zur Verfügung steht und das ist etwas, das in den meisten Umgebungen nicht toleriert wird. Daten zu sichern, die online sind, bedeutet jedoch immer, in Kauf zu nehmen, daß Teile der Sicherung nicht konsistent sind.

Um nicht von der Lesbarkeit eines einzelnen Bandes abhängig zu sein, hat man mehrere, meistens drei *Generationen* eines Volumes. Diese werden als *Großvater*, *Vater* und *Sohn* bezeichnet. Das Band eines Volumes, das am heutigen Tag gesichert wird, ist das Sohnband. Selbst wenn die Maschine bei der Sicherung der Daten abstürzen würde und dabei sowohl die Daten als auch das Band zerstört, hätte man immer noch das Vater– und Großvaterband als Sicherungskopie. Bei der nächsten sicherung eines Volumes wird das bisherige Großvaterband (also das älteste Backup dieses Volumes) als Sohnband verwendet. Das bisherige Vaterband wird zum Großvaterband und das bisherige Sohnband wird Vaterband.

```
home 1
root 1
data 1
home 2
root 2
data 2
home 3
root 3
data 3
```

Abbildung 1.36: Konfigurationsdatei `schedule` für das Backupscript

Abbildung 1.37 zeigt ein Shellscript, das eine solche Datensicherung machen kann. Das Script verwendet eine Konfigurationsdatei `schedule`, die die Namen und Nummern von Volumes enthält. In einem Verzeichnis `lists` stehen Definitionsdateien für die Volumes.

Die Definitionsdatei `home` hätte also den Namen `/usr/adm/backup/lists/home` und würde in jeder Zeile den Namen eines Verzeichnisses oder einer Datei enthalten, die zu diesem Volume gehört.

Das Volume, das am heutigen Tag zu sichern ist, wird durch den ersten Eintrag in dieser Liste bestimmt. Er wird mit der `head`–Anweisung isoliert und in der Variablen `$TODAY` gespeichert. Die `tail` und `echo`–Konstruktion in den folgenden Zeilen schneidet den ersten Listeneintrag ab und stellt ihn an das Ende der Liste. Die Liste rotiert so Eintrag um Eintrag nach vorne.

Aus dem Inhalt von `$TODAY` werden Name des Volumes und die Bandnummer bestimmt. Falls der Name des zu sichernden Volumes `none` ist, wird kein Backup angefertigt. Der Aufrufer des Scriptes wird über den Namen und die Nummer des zu sichernden Bandes informiert und gebeten, das richtige Band einzulegen. Das Script pausiert und wird mit Druck auf die Returntaste fortgesetzt. Die folgenden Anweisungen entsprechen im Prinzip denen aus 1.32.

Das Script ist insofern primitiv, als das es davon ausgeht, daß immer alle Daten eines Volumes zu sichern sind, d.h. das Script macht immer sogenannte *Vollbackups*. Durch einsetzen von `-mtime`–Optionen in die `find`–Anweisung kann man es leicht so abändern, daß nur Daten gesichert werden, die sich seit dem letzten Backup geändert haben. Man hätte dann ein Script, daß ein *inkrementelles Backup* macht. In diesem Fall würde sich aber die Verwaltung des Backup–Planes und der Bänder verkomplizieren. Für ein richtiges inkrementelles Backup wären also weitergehende Änderungen am Script notwendig.

Das Script setzt außerdem voraus, daß ein Volume immer vollständig auf ein Band paßt. Um größere Volumes zu sichern, wäre eine Logik notwendig, die den Operator bei Bedarf zum Bandwechsel auffordert. Einige Versionen von `tar` und `cpio` haben Optionen, die es erlauben, die Länge des Bandes anzugeben und den Bandwechsel handhaben. Volumes, die sich über mehrere Bänder erstrecken, können jedoch auf keinen Fall mehr ohne den Eingriff eines Operators gesichert werden. Das steht dem Konzept des automatischen Backups jedoch im Wege. Am schönsten wäre es, könnte man am Ende eines Arbeitstages das richtige Band einlegen und die Sicherung würde automatisch über Nacht stattfinden.

```
#! /bin/sh

# Verzeichnis mit Definitionen der Volumes
LISTENDIR=/usr/adm/backup/lists

# Liste der zu sichernden Volumes
SCHEDULE=/usr/adm/backup/schedule

# Bestimme das heute zu sichernde Volume
TODAY="`head -1 $SCHEDULE`"

# und stelle das heute zu sichernde Volume
# an das Ende der Liste
tail +2 $SCHEDULE > $SCHEDULE.new
echo $TODAY >> $SCHEDULE.new
mv $SCHEDULE.new $SCHEDULE

# Bestimme Name und Bandnummer
# des heutigen Backups
set $TODAY
BACKVOL=$1
BANDNR=$2

# Ist heute ein Backup notwendig?
if [ "$BACKVOL" = "none" ]
then
        echo "Heute ist kein Backup notwendig."
        exit 0
fi

# Operator informieren
echo "Bitte legen Sie das Band mit dem Namen $BACKVOL"
echo "und der Nummer $BANDNR ein."
read q

# Liste der zu sichernden Verzeichnisse
BACKLIST="`cat $LISTENDIR/$BACKVOL | egrep -v '^#'`"

# Liste der zu sichernden Dateien
cd /
for i in $BACKLIST
do
        find ./$i -depth -print
done |
egrep -v "(core)|(.*\.o)|(.*\.bak)|(.*~)" |
cpio -oc |
dd of=/dev/rst0 bs=256k

echo "Backup beendet."
```

Abbildung 1.37: Script zur Datensicherung

Inhaltsverzeichnis

1 Dateisysteme **1**

1.1 Dateien aus der Sicht des Benutzers 1
1.1.1 Dateitypen und –attribute 1
1.1.2 Eine typische Dateisystemstruktur 4
1.2 Dateien aus der Sicht des Betriebssystems 6
1.2.1 Hardware 6
1.2.2 Partitionen 7
1.2.3 Fragmentierung 8
1.3 Das klassische System V–Dateisystem 9
1.3.1 Dateien 9
1.3.2 Verzeichnisse 13
1.3.3 Zugriffsrechte 15
1.3.4 Umgang mit Dateisystemen 18
1.4 Das verbesserte BSD Fast Filing System 23
1.4.1 Probleme des System V–Dateisystems 23
1.4.2 Blockverwaltung 24
1.4.3 Neue Features 26
1.5 Dateisysteme mit hoher Verfügbarkeit 30
1.5.1 Disk Striping 31
1.5.2 Disk Mirroring 32
1.5.3 Access Control Lists (ACL) 33
1.6 Datensicherung 35
1.6.1 Backup mit tar 37
1.6.2 Backup mit cpio 39
1.6.3 Weitere Kommandos 41
1.6.4 Geplantes Backup und Automatisierung 44

Abbildungsverzeichnis

1.1 Dateitypen in einem UNIX–Dateisystem 1
1.2 Beispiele für Dateien verschiedener Typen 2
1.3 Ein UNIX–Dateibaum 4
1.4 Aufbau einer Festplatte 6
1.5 Geometrie einer älteren und einer modernen Platte 7
1.6 Plattenplatzverlust durch interne Fragmentierung 8
1.7 Externe Fragmentierung 9
1.8 Layout eines System V–Dateisystems 10
1.9 Daten in einer I–Node 11
1.10 I–Node und Datenblöcke 12
1.11 Verschieben einer dünn besetzten Datei mit GNU-tar 13
1.12 Verzeichnis im System V Dateisystem 13
1.13 Auflösung eines absoluten Pfadnamens 15
1.14 Datei mit gesetztem s–Recht 17
1.15 Dateisystem mit mehreren gemounteten Platten 20
1.16 Schwächen des alten Dateisystems in Stichworten 24
1.17 Eine Zylidergruppe im BSD FFS 24
1.18 Verwaltung freier Blöcke 25
1.19 Fragmente können die Dateienden von mehreren Dateien enthalten 26
1.20 Aktivierung der Quotas 28
1.21 quotacheck Kommando 28
1.22 quota Kommando 29
1.23 edquota Kommando 29
1.24 Das Quotasystem in der Anwendung 30
1.25 Physikalische und logische Partitionen bei AIX 31
1.26 Disk Mirroring 32
1.27 AIX Access Control List 34
1.28 mt–Kommando 36
1.29 Arbeiten mit komprimierten Archiven ohne Zwischendatei 38
1.30 Mehrere tar–Archive auf einem Band 39
1.31 Vergleich von cpio und tar 40
1.32 Erzeugung eines Archivs mit find und cpio 41
1.33 Beispiel für Anwendungen von dd. 43
1.34 Katalogisierung unbekannter Band– und Disketteninhalte 44
1.35 Backup im Netzwerk 44
1.36 Konfigurationsdatei schedule für das Backupscript 45
1.37 Script zur Datensicherung 47