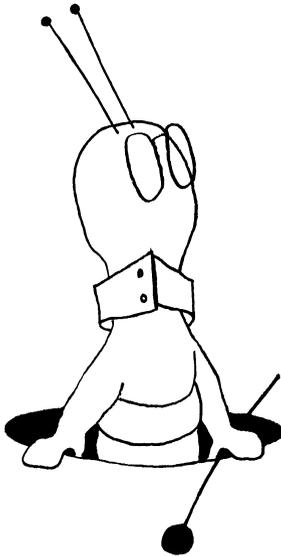


3 Konzepte und Begriffe

Dieses Kapitel geht über die Einführung von Kapitel 2 hinaus und erläutert für die wichtigen Bereiche Konzepte und Nomenklatur des Linux-Systems. Es versucht, Informationen, die in der Linux-Dokumentation teilweise nur verstreut zu finden sind, kompakt und zusammenhängend darzustellen. Dabei lassen sich Wiederholungen nicht ganz vermeiden, weil zuweilen die gleiche Information beim Kommando und in der Übersicht auftritt.

Wenn hier von *Linux* die Rede ist, so treffen die Beschreibungen mit wenigen Ausnahmen ebenso auf Unix zu, gleichen sich doch gerade die Basiskonzepte sehr stark. Just dies macht für viele Unternehmen Linux dort attraktiv, wo sie mit Linux-Systemen ihre teureren proprietären Systeme ablösen oder ergänzen können. Unix-Kenner kommen sehr einfach mit Linux zurecht und Linux-Kenner ebenso mit den gängigen Unix-Varianten wie HP/UX, IBM AIX oder Sun Solaris. Auch für die anderen freien Unix-Derivate wie etwa OpenBSD, NetBSD und FreeBSD gilt dies.

Einige Begriffe werden verwendet, wie sie sonst im deutschen IT-Wortschatz nicht vorkommen. So wird z. B. das Wort *mounted* mit *montiert* oder *eingehängt* übertragen. Die Autoren sind mit einigen solcher Übersetzungen selbst nicht zufrieden, wollten jedoch nicht auf umständliche Umschreibungen wie *in den Systemdateibaum eingehängt* für *mounted* ausweichen.



Auf Grund von Differenzen in der Implementierung der einzelnen Linux-Distributionen sowie von unterschiedlichen Versionsständen und von bestimmten lokalen Einstellungen kann es zu geringfügigen Abweichungen zwischen der in diesem Buch gegebenen Erklärungen und Optionen und der Realisierung im System des Lesers kommen.

An einigen Stellen wurden bei der Beschreibung Vereinfachungen vorgenommen, soweit sie der Verständlichkeit dienen. Es wird dann durch Bemerkungen wie *in der Regel ist ...* darauf hingewiesen.

3.1 Benutzer und Benutzerumgebung

3.1.1 Der Zugang des Benutzers zum System

Zum Arbeiten an einem Linux-System muss sich der Benutzer bei diesem zunächst explizit anmelden. Dies geschieht mit Hilfe des **login**-Verfahrens unter Angabe eines Namens (für das System der *Benutzername* bzw. die so genannte *Benutzeridentifikation*) und eines Passwortes.¹ Der Login-Name wird auch als *Account* bezeichnet.

Der Name – der nicht mit einem Großbuchstaben beginnen sollte – muss dem System zuvor einmal durch einen privilegierten Benutzer (dem Systemverwalter als *Super-User*) bekannt gemacht werden. Hierzu wird der Name des Benutzers (maximal acht Buchstaben) zusammen mit der *Benutzernummer*, der *Gruppennummer* des Benutzers sowie dem *Standardverzeichnis des Benutzers* in die Passwortdatei (*/etc/passwd*) eingetragen.

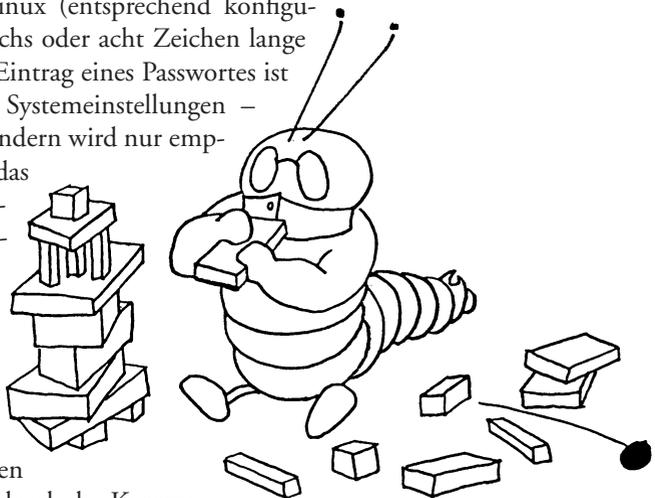
Zu diesen Einträgen gehört auch das Programm, das nach dem Anmelden automatisch gestartet wird. Nahezu immer ist dies eine Shell, die dann – abhängig von der Konfiguration – oft zusätzlich eine grafische Oberfläche hochfährt. Meldet sich der Benutzer danach zum ersten Mal an, so besitzt er möglicherweise noch kein Passwort. Er kann dies dann dem System durch den Aufruf des Kommandos **passwd** mitteilen. Das Passwort wurde früher unter Unix in codierter Form in die Passwortdatei eingetragen. Inzwischen steht es unter Unix und Linux nicht mehr in */etc/passwd*, sondern – besser geschützt – in */etc/shadow*.

Beim Passwort versucht Linux (entsprechend konfiguriert), sichere, d.h. minimal sechs oder acht Zeichen lange

Passwörter zu erzwingen. Der Eintrag eines Passwortes ist – abhängig von bestimmten Systemeinstellungen – nicht unbedingt notwendig, sondern wird nur empfohlen. Einstellungen können das sofortige Setzen eines Passwortes oder die Änderung des vorgelegten Passwortes beim ersten Anmelden erzwingen. Der Benutzer muss dann dazu nicht explizit das **passwd**-Kommando aufrufen. Ein Ändern des Passwortes ist nur durch den Benutzer selbst oder durch den

Super-User möglich (ebenfalls durch das Kommando

passwd). Vergisst ein Benutzer sein Passwort, so muss der Super-User ihm vorübergehend ein neues geben; das alte Passwort lässt sich nicht und durch keinen Benutzer abfragen oder rekonstruieren.



1. Statt eines Account-Namens und eines Passwortes gibt es inzwischen auch Verfahren, bei denen man seine Chipkarte einsteckt und eine PIN eingibt. Dies ist bisher aber nur über Erweiterungen möglich und noch selten im Einsatz.

Die Anforderungen an das Passwort hinsichtlich Komplexität (Länge, Groß-/Kleinschreibung und Sonderzeichen) und Gültigkeitsdauer kann der Systemadministrator festlegen. Bereits die Standardbelegung gewährleistet aber bei den meisten aktuellen Linux-Systemen einen akzeptablen Schutz.

Das System selbst unterscheidet drei Arten von Benutzern:

- ▶ den normalen Benutzer
- ▶ den Super-User
- ▶ spezielle Administrations- und Prozessbenutzer

Der **Super-User** zeichnet sich dadurch aus, dass für ihn die normalen Schutzmechanismen, z. B. bezüglich der Zugriffsrechte auf Dateien, nicht und nur eingeschränkt gelten. Er ist in der Lage, praktisch alle Dateien und Verzeichnisse zu lesen, zu modifizieren, zu löschen und deren Attribute zu ändern. Der Name des Super-Users ist **root**. Er ist (Abweichungen sind möglich) auch als Besitzer des Verzeichnisses an der Systemwurzel (*root directory*) der meisten Geräteknoten im Verzeichnis */dev* sowie der Verzeichnisse */bin*, */etc*, */usr* und */tmp* eingetragen. Daneben ist er der Besitzer der Passwortdateien */etc/passwd* und */etc/shadow* und als einziger berechtigt, darin Änderungen vorzunehmen.

Darüber hinaus muss er als Besitzer aller Programme eingetragen sein, die kontrollierte Modifikationen an seinen wichtigen Systemverwaltungsdateien vornehmen. In diesen Programmen ist dann das *Set-UID-Bit* an Stelle des *x*-Rechtes gesetzt (siehe hierzu Abschnitt 3.3.1 auf Seite 160).

Aus Sicherheitsgründen sollte der Super-User stets ein Passwort besitzen! Auch Benutzer des Systems, die als Systemverwalter fungieren, sollten den Super-User-Status nur dann benutzen, wenn es für Verwaltungsarbeiten notwendig ist.

Mit UNIX System V.4.2 wurden abgestufte bzw. individuell vergebbare Rechte für die Aufgaben der Systemverwaltung eingeführt. Dieses Konzept hat auch Linux übernommen. Es gestattet, gewisse administrative Aufgaben – etwa das Einrichten eines weiteren Benutzers – auch durch spezielle, andere Benutzer ausführen zu lassen, ohne dass diese das Root-Recht benötigen.

3.1.2 Benutzernummer, Gruppennummer

Sein *Super-User-Privileg* erhält der Super-User lediglich durch die Benutzernummer 0 im Eintrag der Datei */etc/passwd*.

Von dieser privilegierten Benutzernummer 0 abgesehen, gibt es keine weitere harte Festlegung bezüglich Benutzernummern und Benutzerprivilegien. Außer dem Super-User sind damit alle Benutzer gleichberechtigt. An vielen Installationen existieren zwar Konventionen bezüglich der Vergabe von Benutzernummern; ¹diese haben jedoch vom Systemkern aus gesehen keine Bedeutung.

Die *Benutzernummer*, auch als UID (*User Identification*) bezeichnet, sollte eindeutig sein, d. h. nur einmal vorhanden. Sie stellt die eigentliche, systeminterne Identifikation eines Benutzers dar.

1. Systemnahe Arbeiten z. B. unter den Benutzernummern 1–99, normale Benutzer ab 500.

Neben der Benutzernummer besitzen alle Benutzer und Dateien eine **Gruppennummer**. Diese erlaubt es, mehrere Benutzer zu einer Gruppe mit gesondert vergebbareren Zugriffsrechten zusammenzufassen. Die Gruppennummer wird unter Linux auch mit **GID** (*Group Identification*) abgekürzt.

Neuere Unix- und aktuelle Linux-Systeme halten die Benutzerpasswörter nicht mehr in der – allgemein lesbaren – Passwort-Datei */etc/passwd*, sondern benutzen dazu eine Schattendatei (*/etc/shadow*), die nur durch den Benutzer *root* lesbar ist. Hier steht dann in */etc/passwd* statt des verschlüsselten Benutzerpasswortes lediglich ein ›x‹. Einträge und Änderungen in der Schattendatei könnten zwar vom Systemverwalter prinzipiell auch über einen Editor vorgenommen werden, sollten aber aus Konsistenzgründen nur über spezielle Administrationskommandos durchgeführt werden. Damit nicht für jede Rechteüberprüfung auf die Dateien zugegriffen werden muss, baut Linux eine Hilfsdatenbank hierzu auf, die zusätzlich im System Speicher gepuffert gehalten wird.

Linux bietet einen Passwort-Alterungsmechanismus, der es erlaubt, dass der Benutzer nach einer vorgebbaren Benutzungsdauer sein Passwort ändern muss.¹ Er wird dazu zuvor darauf hingewiesen.² Dieser Mechanismus ermöglicht in sicherheitsrelevanten Umgebungen die Passwortsicherheit durch einen ständigen Wechsel zu erhöhen. Zugleich verursacht dies jedoch einen erheblich höheren Verwaltungsaufwand, da nach dem Ablauf des Passwortes nur noch der Systemverwalter den Benutzerzugang freigeben kann. Fraglich ist auch, ob durch ständige erzwungene Passwort-Änderungen die Sicherheit tatsächlich erhöht wird. Es führt oft nur dazu, dass der Benutzer sich sein aktuelles Passwort nicht merken kann und es sich daher schriftlich notiert – eine Todsünde in sicherheitsrelevanten Umgebungen.

Analog zur Datei */etc/passwd*, in der für jeden Benutzer ein Passwort eingetragen werden kann, gibt es eine Datei */etc/group*, in der die Benutzergruppen mit ihren Mitgliedern und Passwörtern verzeichnet sind. In dieser Datei wird nach dem Gruppennamen gesucht, wenn dieser vom System benötigt wird (z. B. bei `ls -g`). Systemintern ist der Benutzer immer nur unter seiner Benutzer- und Gruppennummer bekannt. Unter Linux kann ein Benutzer zugleich Mitglied mehrerer Gruppen sein und damit die Zugriffsrechte aller Gruppen haben, in denen er als Mitglied eingetragen ist. Wie zu */etc/passwd* kann es auch zu */etc/group* eine Schattendatei */etc/gshadow* geben, in der die wirklichen Gruppenpasswörter verschlüsselt hinterlegt sind. Diese Gruppen-Schattendatei ist aber nur vorhanden, wenn Gruppenpasswörter vergeben wurden, was häufig nicht der Fall ist.

Den Benutzern der gleichen Gruppe können gleiche Zugriffsrechte bezüglich des Zugriffs auf gemeinsame Dateien erteilt werden.³ Er hat jedoch eine Primärgruppe. Dies ist jene, welche (per Gruppennummer) für ihn in */etc/passwd* eingetragen ist und auch jene, die in der Regel angezeigt wird. Wird von einem Programm (z. B. bei `ls -g ...`) statt eines Benutzernamens oder Gruppennamens eine Benutzernummer oder Gruppennummer ausgegeben (beispielsweise nach Einspielen fremder Dateien), so ist dies ein Anzeichen dafür, dass unter dieser Nummer kein Benutzer in der entsprechenden Passwortdatei eingetragen ist.

1. Dies wird durch spezielle Werte in der Schattendatei definiert.

2. Auch hier kann vorgegeben werden, wieviele Tage vor dem Ablauf dies erfolgen soll.

3. Siehe hierzu ›Zugriffsrechte auf eine Datei – der Datei-Modus‹ auf Seite 128.

Zweck der Benutzer- und Gruppennummer ist die Realisierung von Schutzmechanismen für Zugriffsrechte bei Dateien und ähnlichen Objekten. Hier werden entsprechend der Benutzerunterteilung der *Besitzer*, die *Gruppe* und *alle anderen* unterschieden. So vergibt man häufig die gleiche Gruppe(nnummer) an alle Mitarbeiter einer Abteilung – sie können so z. B. alle auf die Informationen (Dateien) der Abteilung zugreifen oder allen Benutzern, die ein bestimmtes Programm ausführen dürfen, wobei man dann dem Programm die Ausführungsrechte so setzt, dass nur die Benutzer der betreffenden Gruppe es ausführen dürfen. Daneben erlauben Benutzer- und Gruppennummer z. B. eine Abrechnung für Systemnutzung und Belegung der Hintergrundspeicher oder der Nutzung von Druckern. Dieses *Abrechnen* wird im Computerjargon als *Accounting* bezeichnet und wird unter *Systempflege* behandelt (siehe Kapitel 9.5).

3.1.3 Dateiverzeichnisse des Benutzers

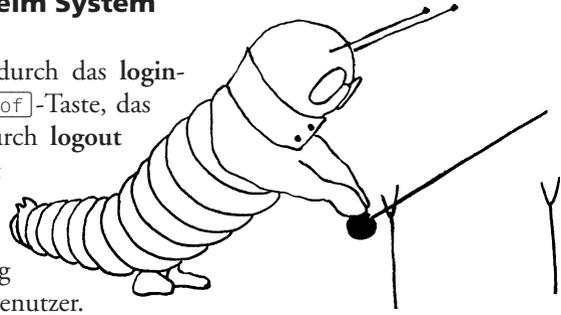
Für jeden Benutzer sollte ein eigenes (Datei-)Verzeichnis (*directory*) vorhanden sein, in welchem der Benutzer frei Dateien anlegen und löschen darf und als dessen Besitzer er eingetragen ist. Dieses Verzeichnis wird in der Regel als ›*Standardverzeichnis nach dem Anmelden* oder kürzer **login-Verzeichnis** (englisch: *login directory*) bezeichnet und im entsprechenden Benutzereintrag in der Datei */etc/passwd* festgelegt. Hierdurch erhält der Benutzer dieses Verzeichnis beim Anmelden (**login**) automatisch als *Standardverzeichnis* (gleichbedeutend mit *aktuellem Verzeichnis* oder *Arbeitsverzeichnis*).

Dieses Verzeichnis wird beim Anmelden auch der Shell-Variablen HOME zugewiesen und wird damit auch zum **Hauptverzeichnis** (englisch: *home directory*). Das Hauptverzeichnis ist das Verzeichnis, das man als *aktuelles Verzeichnis* zugewiesen bekommt, wenn man das **cd**-Kommando ohne einen Parameter aufruft. Es wird in der deutschsprachigen Literatur auch als **Heimatverzeichnis** oder als **HOME-Verzeichnis** bezeichnet. Das HOME-Verzeichnis lässt sich vom Benutzer durch Zuweisung eines neuen Verzeichnisnamens an die Shell-Variable HOME ändern, während das **login-Verzeichnis** nur durch den Super-User in der Passwortdatei bzw. durch entsprechend privilegierte Benutzer über eine Systemverwaltungs-Oberfläche geändert werden kann.

Diese Benutzerverzeichnisse, die als *login directories* eingetragen sind, liegen per Konvention im Verzeichnis *//home* vor. Wird ein neuer Benutzer durch ein entsprechendes Verwaltungswerkzeug eingerichtet (durch **adduser** oder Oberflächen wie den YaST2 unter der Funktion *Sicherheit & Benutzer* → *Benutzer bearbeiten und anlegen* unter **admintool** unter Sun Solaris, **SMIT** unter IBM AIX oder **SAM** unter HP/UX), so werden Zugriffsrechte und Besitzer bereits korrekt gesetzt. Durch Systemverwaltungswerkzeuge wird ein Benutzer meist nicht nur als zulässiger Benutzer eingetragen, sondern für ihn wird auch das Mail-System konfiguriert, und ihm werden die wichtigsten Konfigurationsdateien für seine Standardumgebung (Arbeitsumgebung, grafische Oberfläche, Mail-System) kopiert, so dass er sich meist sofort und produktiv anmelden kann. Während man einzelne Benutzer bequemer durch die erwähnten Werkzeuge mit grafischer Oberfläche anlegt, ist das Anlegen und Löschen zahlreicher Benutzer mit dem Kommando **adduser** wesentlich effizienter.

3.1.4 Das An- und Abmelden beim System

Ein Benutzer meldet sich beim System durch das **login**-Kommando an und durch Eingabe der `eof`-Taste, das **exit**-Kommando oder bei der C-Shell durch **logout** wieder ab. Bei grafischen Oberflächen gibt es in aller Regel einen Logout- oder Abmeldeknopf. Durch ein zweites **login**-Kommando meldet er sich ab und gleichzeitig wieder neu an – eventuell als ein anderer Benutzer.



Anmelden

Wird das System neu gestartet oder hat sich der vorherige Benutzer des Systems ordnungsgemäß abgemeldet, so zeigt der (alphanumerische) Bildschirm die Login-Meldung, die etwa wie folgt aussehen kann (*lapyli* ist hier der Rechnername):

```
Welcome to SuSE Linux 8.1 (i386) - Kernel 2.4.19-4GB (0)
lapyli login
```

Bei einer grafischen Oberfläche – und dies ist inzwischen der Standard für die meisten Linux-Arbeitsplätze – sieht es etwa wie folgt aus:



Abb. 3.1: Beispiel eines Login-Bildschirms bei einer grafischen Oberfläche

Hier sind nun die Benutzeridentifikation und das Passwort einzugeben und jeweils durch `cr` abzuschließen.

Für das Passwort erfolgt – abhängig von den Systemeinstellungen – keinerlei Anzeige am Bildschirm oder die Ausgabe von Sternchen. Aus Sicherheitsgründen erfolgt die Benutzerprüfung immer erst nach korrekter Eingabe beider Felder. Nach der Überprüfung des Benutzernamens und des Passwortes führt das System den Anmeldeprozess durch. Wurde mehrmals hintereinander eine falsche Benutzer-/Passwort-Kombination eingegeben, so sperrt das System die Dialogstation für eine gewisse Zeit. Hiermit sollen

automatisierte Angriffe über ein Durchprobieren verschiedener Passwörter durch einen anderen Rechner erschwert werden.

Der Benutzer bekommt beim Login seine Benutzer-, seine Gruppennummer und sein Hauptverzeichnis (englisch: *home directory*) zugeordnet, für einen zeichenorientierten Bildschirm werden Terminal-Charakteristika gesetzt und anschließend das in der Passwortdatei angegebene Initial-Programm gestartet. Dies ist in der Regel eine Shell (*/bin/bash* oder */bin/tcsh*) oder ein Desktop.

Danach wird die grafische Oberfläche – sofern vorhanden und konfiguriert – mit ersten Anwendungen (Terminal, Uhr, Mail) gestartet und (bei der Bourne- und Korn-Shell sowie der **bash**) die Kommandos der Datei */etc/profile* ausgeführt. Sofern vorhanden, wird eine Systemnachricht aus der Datei */etc/motd* (*message of the day*) am Bildschirm ausgegeben. Liegen Nachrichten (*mail*) für den Benutzer vor, so wird – abhängig vom login-Skript – der Benutzer mit der Meldung *You have mail* informiert.

Für das Starten der grafischen Oberfläche, d.h. des X Window Systems, ersten Anwendungen unter der grafischen Oberfläche und ggf. eines Desktop-Managers,¹ gibt es viele unterschiedliche und in sehr weitem Rahmen konfigurierbare Möglichkeiten und Konventionen. So kann beispielsweise das X Window System unmittelbar nach dem Hochfahren des Systems gestartet werden und dann selbst die Benutzerauthorisierung vornehmen (z. B. mittels **xdm**; siehe Abb. 3.1). Alternativ erhält der Benutzer auch auf einem grafischen Bildschirm zunächst eine zeichenorientierte Login-Aufforderung und startet erst nach seiner erfolgreichen Anmeldung das X Window System² zusammen mit einigen typischen ersten Applikationen.

Bei einer grafischen Oberfläche mit Desktop-Umgebung könnte sich der Bildschirm auch wie in Abb. 3.2 präsentieren.

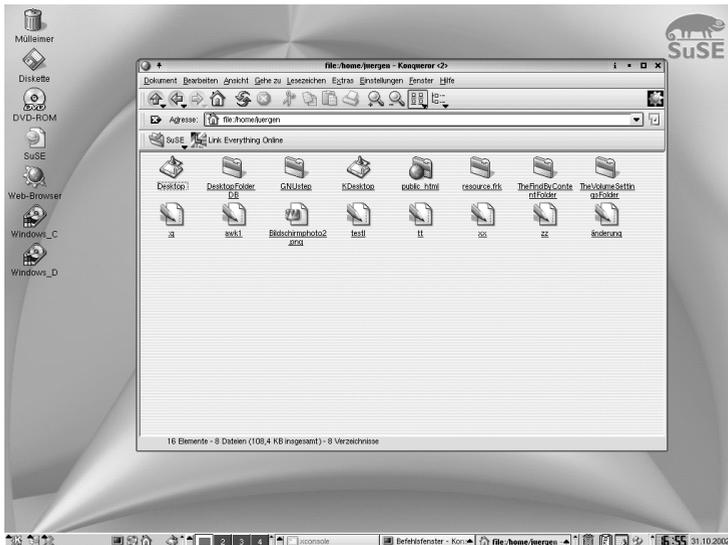


Abb. 3.2: Bildschirm eines Linux-Desktops (hier unter KDE bei SuSE)

1. Siehe ‚Grafische Oberfläche: Desktop-System‘ auf S. 36.
2. Mittels Kommandos wie **xinit** oder **startx**.

Wird die Shell (oder die **bash**) gestartet, so sieht diese im Standardverzeichnis des Benutzers nach, ob eine Datei mit dem Namen *.profile* existiert und führt, falls vorhanden, die darin stehenden Kommandos aus. Bei der C-Shell oder **tcsh** sind dies die Kommandos der Dateien *.login* (nur bei einer *Login-Shell*) und *.cshrc* (von jeder weiteren C-Shell). Wird die Shell über die Terminalemulation **xterm** des X Window Systems gestartet, so sollte darauf geachtet werden, dass diese tatsächlich als so genannte *Login-Shell* gestartet wird. Nur dann durchläuft sie die anfänglichen Konfigurationsdateien. Dies lässt sich auch durch die Option **-ls** (*login shell*) beim Aufruf des **xterm** erreichen.

Mit einem erfolgreichen **login** wird auch das Anmelden mit Benutzername, Dialogstation und Uhrzeit in den Dateien */etc/utmp* (zur Abfrage für das **who**-Kommando) und */etc/wtmp* (für eine Systemabrechnung) eingetragen.

Abmelden

Das Abmelden erfolgt durch die Terminierung der Shell entweder durch die Eingabe des `eof`-Zeichens, durch das Kommando **exit** (bei der Bourne- und Korn-Shell sowie bei der **bash**) oder durch den Befehl **logout** bei der C-Shell. Sie führt dann vor der Terminierung die Kommandos der Datei *.logout* im **login**-Verzeichnis des Benutzers aus.



3.1.5 Die Benutzerumgebung

Wird als Initial-Programm (bzw. als Benutzerschnittstelle) beim **login** eine Shell aufgerufen, so stellt diese eine Benutzerumgebung her. Sie besteht aus dem aktuellen Verzeichnis (*working directory*), dem Hauptverzeichnis (*home directory*), dem Suchpfad für Programme und dem Typ der Dialogstation. Zugleich werden durch Benutzer- und Gruppennummer die Privilegien des Benutzers und seine Zugriffsrechte auf Dateien festgelegt. Entsprechend werden die globalen Shell-Variablen besetzt. Die hier verwendeten Variablen sind teilweise abhängig vom benutzten UNIX-/Linux-System und der eingesetzten Shell. Einige wichtige Vorbelegungen sind z. B. in folgenden Variablen zu finden (man kann sich deren Werte per **echo \$variablenname** ausgeben lassen):

HOME	Hauptverzeichnis, Heimatverzeichnis eines Benutzers
PATH	Suchpfad für Programme (siehe Kapitel 3.1.6)
TERM	Bildschirmtyp
TZ	Zeitzone
LANG	Sprache und Sprachbereich des Benutzers
LC_xxx	verschiedene Variablen für Einstellungen zur Lokalisierung
DISPLAY	Anzeigebildschirm des X Window Systems, normalerweise <i>rechnername:0.0</i> (genauer: Name des X-Servers)
PS1 ... PS4	die verschiedenen Shell-Prompt-Zeichen

Weitere sinnvolle Besetzungen, die in der Regel jedoch vom Systemverwalter oder vom **login**-Prozess vorzugeben werden, wären die Shell-Variablen:¹

LOGNAME	Name des Benutzers beim login
UID	Benutzernummer des aktuellen Benutzers
MAIL	Briefkasten für den Benutzer (Standard unter Linux: <i>/var/mail/benutzer-name</i>)
SHELL	Name des Interpreter-Programms, das aus anderen Programmen heraus gestartet wird, normalerweise unter Linux die bash
HOSTNAME	Rechnername des aktuellen Arbeitsrechners
OSTYPE	Art des Betriebssystems (hier zumeist <code>>linux</code>)

Die Besetzung von **PATH** ist systemabhängig und wird durch das **login**-Programm vorgenommen. Erweiterungen des Suchpfades können vom Systemverwalter in der Datei */etc/profile* oder vom einzelnen Benutzer in der Definitionsdatei seiner Arbeitsumgebung definiert werden.

Weitere vom Benutzer gewünschte Definitionen und Kommandos lassen sich in einer Datei mit dem Namen *.profile* (bei der C-Shell mit dem Namen *.login* und *.cshrc*) im Hauptverzeichnis des Benutzers festlegen. Diese Kommando-prozedur wird von der Login-Shell beim Aufruf automatisch durchlaufen und somit der gewünschte Zustand hergestellt. Zusätzlich Festlegungen sind vom Super-User in */etc/profile* möglich und werden dann als erstes für alle Benutzer (der entsprechenden Shells) durchlaufen.

In *.profile* stehen sinnvollerweise sitzungsbezogene Initialisierungskommandos wie die Angabe eines neuen Standardverzeichnisses (sofern er von dem in der Passwortdatei abweicht), Angaben zum Typ des Terminals und das Setzen von Parametern für die Dialogstation. Hier kann auch die Besetzung von globalen Shell-Variablen (*Environment*-Variablen) vorgenommen werden, deren Werte von einigen Programmen (z.B. **vim**, **more**, **less**), die Standardpfade zu den Informationen für das **man**-Kommando (**MANPATH**), der Suchpfad für das **info**-Kommando (**INFODIR**), **INFOPATH** um **emacs** zu sagen, wo er seine Informationen findet und z.B. im X Window System für Vorbesetzungen benötigt werden. An dieser Stelle ist auch die Definition von speziellen Abkürzungen sinnvoll, welche der Benutzer während seines Dialogs verwenden möchte (siehe hierzu Kapitel 6).

Neben der bisher angeführten Parametrisierung der Benutzerumgebung gibt es eine Reihe weiterer einfacher Verfahren, mit denen ein Benutzer seine Systemumgebung weiter aus- und umbauen kann:

- ▶ Umbenennung von Linux-Kommandos mittels **ln** oder **mv**
- ▶ Verwendung von Abkürzungen mit Shell-Variablen
- ▶ Einführung von Abkürzungen über die Alias-Definition der **bash**
- ▶ Einführung von Abkürzungen über Funktionsdefinition der **bash**
- ▶ Verwendung eigener Kommando-prozeduren
- ▶ Definition weiterer Umgebungsvariablen für häufig benutzte Kommandos und Programme
- ▶ Funktionsmenüs bei grafischen Oberflächen

1. Für eine vollständigere Liste siehe Anhang A.4 auf Seite 854.

Dabei ist es sinnvoll, solche Anpassungen in den so genannten *Profile-Dateien* des jeweiligen Benutzers zu hinterlegen. Die Dateinamen der Profile-Dateien beginnen in aller Regel mit einem Punkt und werden so beim normalen Kommando nicht mit aufgelistet.

Anpassungen, die systemweit als Standardeinstellungen gelten sollen, werden dazu in System-Profile-Dateien hinterlegt. Diese liegen per Konvention im Verzeichnis */etc*. Die entsprechenden Dateien haben im Gegensatz zu den Benutzer-Profile-Dateien keinen führenden Punkt im Dateinamen. Entsprechend gibt es für die Shell bzw. **bash** in */etc* die Datei *profile*, welche vor allen anderen Profile-Dateien beim Shell-Start gelesen wird (so vorhanden). Sie ist aber eher spezifisch für die Linux-/Unix-Distribution und sollte nicht geändert werden, da sie beim nächsten Update eventuell überschrieben wird. Spezifische Anpassungen (für alle Benutzer des betreffenden Rechners) führt man z. B. unter SuSE-Linux deshalb für die Shell in */etc/profile.local* aus.

Die Reihenfolge der ausgewerteten Profile-Dateien beim Shell-Start lautet für die **bash**:¹

Tabelle 3.1: Initialisierungsdateien der Shell (*bash*)

1	<i>/etc/profile.local</i>	Einstellungen für alle Benutzer des lokalen Systems
2	<i>/etc/profile</i>	System- und distributionsweite Einstellungen (nur bei Login-Shell). Setzt sinnvolle Vorbelegungen (z. B. für \$PATH) für Benutzer ohne eigene Profile-Dateien.
3	<i>~/.profile</i> ^a	individuelle Einstellungen für den jeweiligen Benutzer (im Home-Verzeichnis des Benutzers)
4	<i>~/.bash_profile</i>	individuelle Einstellungen für den jeweiligen Benutzer (im Home-Verzeichnis des Benutzers). Sie wird von der bash nur dann gesucht, wenn diese ein Login-Shell ist.
5	<i>~/.bash_login</i>	individuelle Einstellungen für den jeweiligen Benutzer (im Home-Verzeichnis des Benutzers)
6	<i>~/.bashrc</i>	individuelle Einstellungen für den jeweiligen Benutzer (im Home-Verzeichnis des Benutzers). Ist die bash keine Login-Shell, so wird nur diese Datei ausgeführt.
A	<i>/etc/inputrc</i>	steuert – systemweit – das Verhalten der Funktion (Bibliothek) readline , welche auch von der bash verwendet wird.
B	<i>~/.inputrc</i>	benutzerindividuelle Einstellungen für die readline -Funktion

a. Das Tildezeichen *~* steht in der Shell als Abkürzung für das jeweilige Home-Verzeichnis des aktuellen Benutzers bzw. für den Inhalt von \$HOME.

Dabei werden die Dateien ab Position 4 nur gesucht und abgearbeitet, wenn die Dateien der vorhergehenden Positionen nicht gefunden oder gelesen werden können. Später ausgewertete Dateien *überschreiben* dabei Definitionen aus vorhergehenden Dateien.

1. Hier können durch entsprechende Teile in einem ersten Profile-Skript individuelle Änderungen in Distributionen oder lokalen Systemen existieren!

Die Datei `~/.bash_logout` kann Shell-Anweisungen enthalten, die bei der Beendigung einer Login-Shell ausgeführt werden. Hier bringt man in der Regel Aufräumarbeiten unter (z. B. das Löschen bestimmter temporärer Dateien).

Benutzerumgebung in der grafischen Oberfläche

Diese Definitionen gelten für die Arbeit mit der Kommandozeile auf einem zeichenorientierten Bildschirm genauso wie für die Arbeit am Desktop der grafischen Oberfläche.

Dabei ist darauf zu achten, dass die Konfigurationsdateien (`.profile` oder `.cshrc`) für grafische Umgebungen meist wesentlich aufwändiger gestaltet sein müssen. In grafischen Umgebungen werden diese Dateien zu Beginn oder während einer Sitzung meist mehrfach durchlaufen und müssen entsprechend konfiguriert sein:

- ▶ einmal zu Beginn der Sitzung bei der eigentlichen Benutzeranmeldung:
Dabei muss eventuell die grafische Oberfläche hochgefahren werden. In der Konfigurationsdatei sollten dabei keine Ausgaben erfolgen (diese wären nicht sichtbar). Diese erste Shell tritt nach Start der Oberfläche in den Hintergrund und wird nicht interaktiv genutzt.
- ▶ während der Sitzung beim Start einer Terminalemulation (`xterm`):
Dabei darf die grafische Oberfläche natürlich nicht schon wieder hochgefahren werden; andererseits wird die Shell interaktiv genutzt und sollte daher alle nötigen Definitionen enthalten. Im Normalfall laufen auch bei einem Benutzer mehrere solcher Shell-Fenster bzw. Terminalemulationen nebeneinander, die alle beim Aufruf die gleiche Konfigurationsdatei durchlaufen müssen. Die Terminalemulation muss daher jeweils als *login shell* deklariert werden (Aufruf durch `xterm -ls`).

In grafischen Umgebungen muss die Variable `$DISPLAY` korrekt auf die Anzeigestation des jeweiligen Benutzers gesetzt sein. Dies ist normalerweise der Name, den der Rechner im Netz trägt, mit einem angehängten `:0.0`, also beispielsweise `zeus:0.0` bei einem Rechnernamen (nicht Benutzernamen) `zeus`. Geschieht dies nicht, tritt der meist störende Effekt auf, dass Programme des X Window Systems am falschen Bildschirm angezeigt werden.

Abkürzungen mit Shell-Variablen

Benutzt man häufig Dateien mit längeren Pfadnamen ohne sein aktuelles Verzeichnis umsetzen zu wollen (z. B. `/opt/dateien/aktuell/beschreib`), so möchte man diese zumeist abkürzen. Eine Möglichkeit besteht darin, die langen Bezeichner einer Shell-Variablen mit kurzem Namen zuzuweisen. Die Variable wird als *global* deklariert und dann die Variable statt des langen Bezeichners eingesetzt, wie folgendes Beispiel zeigt:

```
$un=/opt/dateien/aktuell/beschreib
$export un
$vi $un
```

Variablenbelegung
Exportieren der Variablen
Variable als Argument statt
eines Dateinamens

In der ersten Zeile wird die Zeichenkette `/opt/dateien/aktuell/beschreib` der Shell-Variablen `un` zugewiesen und diese damit definiert. Mit der zweiten Zeile wird die Variable `un` als global deklariert, so dass sie auch in nun aufgerufenen Shell-Prozeduren gültig ist. In der dritten Zeile wird die Variable im Aufruf des Editors `vi` benutzt. Die Shell substituiert dabei `>$un<` durch den Wert der Variablen `un`. Dies ist hier die zuvor zugewiesene Zeichenkette `/opt/dateien/aktuell/beschreib`. Leider ist der Substitutionsmechanismus der Shell nicht ganz einfach zu verstehen. Es kommt hier für den weniger geübten Benutzer oft zu schwer verständlichen Ergebnissen. Der volle Substitutionsmechanismus ist in Kapitel 6 detailliert erklärt.

Vorbelegungen über Shell-Variablen

In vielen Fällen werden Shell-Variablen zur Vorbelegung wichtiger Steuerparameter von Programmen benutzt. So definiert die Variable `MAIL` z. B. für die Mail-Programme, in welcher Datei der Postkorb liegen soll, während die Häufigkeit, mit welcher der Postkorb auf neue Post zu überprüfen ist, in `MAILCHECK` vorgegeben wird. Das `man`-Kommando, welches die Manualseiten der Linux-Kommandos ausgibt, sucht nach den Kommandobeschreibungen in den Verzeichnissen, die in der Variablen `MANPATH` vorgegeben sind. Diese Variablen sind entweder systemweit vorbelegt oder können durch den Benutzer selbst (normalerweise in der Datei `.profile`) geeignet gesetzt werden.

Funktionsdefinitionen

Die `bash`, Bourne- und Korn-Shell erlauben die Definition von Funktionen wie folgt:

```
funktionsname () kommando
```

oder, falls mehrere Kommandos ausgeführt werden sollen:

```
funktionsname () { kommando_folge ; }
```

In dem Kommando kann mit dem üblichen Parametermechanismus der Shell (`$1`, `$2`, ...) auf die beim Aufruf angegebenen Parameter zugegriffen werden.

Soll z. B. eine Funktion `LL` mit der Aufgabe *Erstelle ein ausführliches Listing* definiert werden, so kann dies mit

```
LL () ls -l $*
```

erfolgen. `LL` kann nun wie ein Shell-internes Kommando aufgerufen werden.

Der Vorteil gegenüber Kommandoprozeduren besteht darin, dass die Definition nur temporär im Speicher existiert und keine Kommandodatei angelegt werden muss. Eine Shell-Funktion ist vergleichbar mit einer Variablen, die Shell-Kommandos enthält. Die Abarbeitung ist damit auch schneller. Mit dem `unset`-Kommando lässt sich eine Funktionsdefinition aufheben. Will man solche Funktionen in jeder Sitzung verwenden, so wird sie in der `.profile`-Datei des Benutzers definiert. Diese Art der Definition ist der Alias-Funktion ähnlich, erlaubt im Gegensatz zur nachfolgend beschriebenen `alias`-Funktion auch rekursive Aufrufe.¹

1. Das heißt, die Funktion darf sich selbst wieder aufrufen.

Alias-Definition der bash und Korn-Shell

Die Alias-Definition der **bash** oder Korn-Shell belegt ähnlich wie die Funktionsdefinition ein Kürzel mit einem längeren Kommando. Dies geschieht wie folgt:

```
alias kürzel=kommando
```

Wird danach *kürzel* als Kommando angegeben, so wird dafür das nachstehende Kommando von der **bash** eingesetzt. Im Kommando dürfen dabei auch Parameter und Optionen vorkommen. Die Parameter beim Aufruf werden wie in Shellprozeduren üblich mittels \$1, \$2, angegeben.

Das nachfolgende Beispiel definiert einen Alias *ll* mit gleicher Funktionalität wie zuvor die LL-Shell-Funktion:

```
alias ll='ls -l $*'
```

Der Aufruf *>ll /etc<* wird dann von der **bash** zu *>ls -l /etc<* expandiert. Das Kommando **alias** (ohne Parameter) zeigt alle existierenden Definitionen an; **unalias ll** ermöglicht, diese Definition wieder aufzuheben. Sollen die Funktionen für den Benutzer in allen Sitzungen zur Verfügung stehen, so kann die Definition in der Datei *.login* erfolgen.

Bei C-Shell oder **tsh** steht statt des Gleichheitszeichen ein Leer- oder Tabulatorzeichen. Hier würde man das entsprechende Alias also wie folgt definieren:

```
alias kürzel kommando
```

Kommandoprozeduren

Werden bestimmte Kommandosequenzen häufiger benutzt, so lohnt es sich, diese in eine Kommandodatei zu schreiben. Hierbei können sehr komplexe Abläufe realisiert und über Parameter beim Aufruf gesteuert werden.

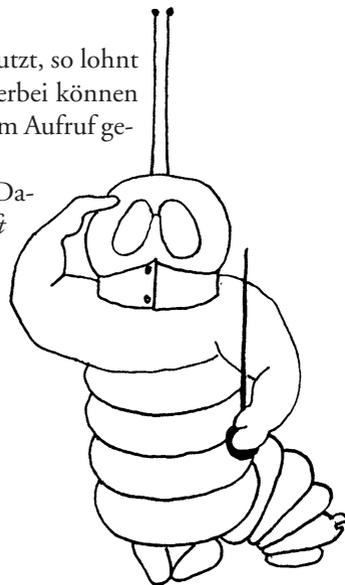
Die nachfolgende Kommandoprozedur steht in der Datei *telefon*. Sie durchsucht die Dateien *privat*, *geschaeft* und *firma*, in denen Namen mit Telefonnummern stehen, nach den im Aufruf angegebenen Namen und gibt die passenden Zeilen aus:

```
for i
do
    for datei in privat geschaeft firma
    do
        grep "$i" $datei
    done
done
```

Der Aufruf erfolgt dann z.B. mit

```
sh telefon Mayer Horten
```

und würde dann die Zeilen der genannten Dateien ausgeben, in denen *Mayer* oder *Horten* vorkommt. Hat man zuvor die Datei *telefon ausführbar* gemacht (mit **chmod a+x**



telefon), so kann das vorangestellte **sh** entfallen, und **telefon** verhält sich wie ein neues Linux-Kommando.

Kommandoprozeduren werden ausführlich im Shell-Kapitel 6.2 beschrieben; Beispiele sind in Kapitel 6.3.12 zu finden.

3.1.6 Der Suchpfad für Programme

Beim Aufruf eines Programms nimmt die Shell das erste Wort auf der Kommandozeile als den Kommando- bzw. Programmnamen und sucht nach einer Programmdatei dieses Namens. Ist der Dateiname des Programms nicht vollständig, d. h. einschließlich Pfadnamen angegeben, so wird beim Suchen der Programmdatei von der Shell nicht nur, wie bei anderen Dateinamen üblich, das aktuelle Verzeichnis dem Kommandonamen vorangestellt, sondern die Shell sucht in bestimmten, ihr vorgebbaren Verzeichnissen nach einer Datei des angegebenen Namens. Dieser Suchpfad ist in einer Variablen der Shell festgelegt. Der Name dieser Variablen lautet **PATH**.¹ In ihr stehen – syntaktisch durch `>:<` getrennt – die zu durchsuchenden Verzeichnisse. Dabei wird in der Reihenfolge der angegebenen Verzeichnisse von links nach rechts gesucht. Eine mögliche Besetzung von **PATH** könnte sein:

```
.:bin:/usr/bin:/home/neuling/bin
```

Der erste Punkt besagt, dass zunächst im aktuellen Verzeichnis gesucht werden soll, dann im Verzeichnis `/bin`, in `/usr/bin` und schließlich in `/home/neuling/bin`. Hat man bei einer solchen Besetzung im eigenen Verzeichnis Programme, welche den gleichen Namen wie Linux-Programme tragen, so werden beim Aufruf die Programme im eigenen Verzeichnis gestartet, da dieses als erstes durchsucht wird. Weitere eigene Programme oder Kommandoprozeduren könnten in einem Unterverzeichnis `bin` im Hauptverzeichnis des Benutzers liegen.

Das Kommando **echo \$PATH** liefert den aktuellen Wert der Shell-Variablen **PATH** (bei der C-Shell in `$path`) zurück. Mit der Anweisung:

```
PATH=wert                bei der bash, Bourne- oder Kornshell
```

kann ihr eine neue Zeichenkette zugewiesen und damit eine neue Folge von Verzeichnissen für die Suche beim Programmaufruf festgelegt werden.

Eine solche Anweisung wird man in der Regel in die Datei mit dem Namen `.profile` im Login-Verzeichnis des Benutzers legen. Dieser Suchpfad ist in der Regel systemweit vorgegeben, kann aber von jedem Benutzer einfach verändert werden.² Dabei sollte man darauf achten, dass der Suchpfad immer nur verlängert, nicht aber explizit neu belegt wird, um nicht die systemweiten Standarddefinitionen zu überschreiben. Um beispielsweise den Standardsuchpfad um das Verzeichnis `bin` im eigenen Home-Verzeichnis zu ergänzen, wäre ein Kommando wie folgt erforderlich.

```
PATH=$PATH:$HOME/bin
```

1. Mehr über Shell-Variablen ist in Kapitel 6 zu finden.

2. Diese Möglichkeit zur Veränderung des Suchpfades kann explizit unterbunden werden.

Hierbei wird die in PATH existierende Belegung in die neue Definition von PATH aufgenommen. Wird PATH weder in */etc/profile* noch in *.profile* gesetzt, so erhält die Variable ihren Wert durch den **login**-Prozess.

3.1.7 Profile-Dateien

Im Login-Verzeichnis des Benutzers befinden sich in der Regel eine Reihe von Dateien, die zu Beginn einer Benutzersitzung (d.h. nach dem Login) von der gestarteten Shell, der Graphikoberfläche sowie von verschiedenen anderen Applikationen zur individuellen Konfiguration der Arbeitsumgebung gelesen werden und das Arbeiten damit wesentlich beeinflussen können. Mit ihnen lässt sich somit in weiten Grenzen die Arbeitsweise am System und in einzelnen Anwendungen steuern. Diese Dateien sind normalerweise verdeckt, d.h. ihr Name beginnt mit einem Punkt und wird daher z. B. beim **ls**-Kommando im Standardfall nicht mit angezeigt und bei Shell-Kommandos nicht bei Namenserweiterungen miterfasst.

Einige Anwendungen (z. B. der KDE-Desktop) haben gleich mehrere Initialisierungsdateien, so dass sie im HOME-Verzeichnis zunächst ein verdecktes (mit `·` beginnendes) Verzeichnis haben, in dem erst die Initialisierungsdateien liegen.

Die Konfigurationsdateien lassen sich grob in vier Kategorien unterteilen:

- ▶ Initialisierungsdateien für die Shells
- ▶ Initialisierungsdateien für die grafischen Oberflächen/Desktops
- ▶ Initialisierungsdateien für individuelle Anwendungen
- ▶ Profile-Dateien, in denen Anwendungen vom Benutzer vorgenommene Einstellungen hinterlegen. Sie dienen als Parameter-Puffer über verschiedene Sitzungen oder Läufe hinweg.

Der Übergang der einzelnen Kategorien ist dabei natürlich fließend, denn auch eine Shell ist eine Anwendung, und die Anwendung **nautilus** (der primäre Dateimanager unter GNOME) ist in Wirklichkeit ein Zwischending zwischen Anwendung und Desktop, was deutlich wird, wenn man ihn unter KDE aufruft).

Daneben ist zwischen privaten (benutzerspezifischen) Einstellungen und systemweit geltenden Einstellungen zu unterscheiden. Während die systemweiten Einstellungen in der Regel in */etc* oder einem Unterverzeichnis davon zu finden sind, liegen die privaten Einstellungsdateien im HOME-Verzeichnis des Benutzers.

Die nachfolgenden Listen sind keineswegs vollständig, sondern dienen nur als Beispiele.

Benutzerspezifische Initialisierungsdateien für Shells

- .bashrc** Konfigurationsdatei der **bash** (siehe Beschreibung auf Seite 104).
- .profile** Wichtigste Konfigurationsdatei der **bash**, Bourne- und Korn-Shell; wird nur von der ersten Shell (Login-Shell) beim Aufruf interpretiert, von weiteren Shells aber nicht mehr automatisch gelesen; sie wird nach der zentralen Datei */etc/profile* und */etc/profile.local* durchlaufen.

- .cshrc** Konfigurationsdatei der C-Shell und **tcsh**; wird von jeder C-Shell – also auch von Kommandoprozeduren in C-Shell-Syntax – nach dem Aufruf gelesen. Zuvor liest die C-Shell noch */etc/csh.login* (wenn es sich um eine Login-Shell handelt) und */etc/cshrc*. Globale Änderungen für das lokale System sollten in */etc/cshrc.local* erfolgen.
- .login** Konfigurationsdatei der C-Shell; sie wird von der ersten C-Shell (*Login-Shell*) beim Aufruf nach der Interpretation von *.cshrc* gelesen.
- .kshrc** Konfigurationsdatei der Korn-Shell; wird von jeder Korn-Shell gelesen. Der Name *.kshrc* ist nicht fest definiert, sondern es wird die Datei verwendet, deren Name in der Variablen **ENV** enthalten ist.

Unter einer grafischen Oberfläche ist die Shell, die ein Benutzer für seine interaktive Arbeit erhält, normalerweise nicht die erste und damit nicht die *Login-Shell*. Es muss daher unter Umständen gesondert dafür Sorge getragen werden, dass die entsprechenden Initialisierungsdateien auch tatsächlich gelesen und ausgewertet werden. Die Terminalemulation **xterm** für die interaktive Arbeit mit der Shell kann hierfür mit dem Parameter **-ls** aufgerufen werden und verhält sich damit wie eine Login-Shell.

Die **Login-Shell** ist die erste Shell, die innerhalb einer Sitzung aufgerufen wird. Bei alphanumerischen Terminals ist es die Shell, welche der **login**-Prozess für die Benutzersitzung startet. Sie führt – abhängig von der Art der Shell – zur Initialisierung Kommandos aus, die von den nachfolgend aufgerufenen Shells der Sitzung nicht mehr durchlaufen werden. Bei der **bash**, Bourne- und Korn-Shell sind dies die Kommandos der Dateien */etc/profile* und *~/.profile*, bei der C-Shell und **tcsh** die Datei *.login*.

Initialisierungsdateien für die grafische Oberfläche

Die grafische Oberfläche ist hochgradig konfigurierbar und befindet sich auch mit der zunehmenden Verbreitung von Desktop-Programmen in einem Umbruchprozess, der noch nicht abgeschlossen und daher ziemlich uneinheitlich ist. Die folgenden Dateinamen sind daher zwar üblich und weit verbreitet, aber nicht auf jedem System in gleicher Weise zu finden:

- .xinitrc** Kommandoprozedur zum Start des X Window Systems (X-Server), des Windows-Managers und erster Applikationen; sie wird vom **xinit**-Kommando verwendet.
- .xsession** Kommandoprozedur zum Start des X Window Systems (X-Server), des Windows-Managers und erster Applikationen; sie wird vom Login-Prozess **xdm** verwendet (*.xinitrc* und *.xsession* können ggf. den gleichen Inhalt haben).
- .Xdefaults** Benutzerindividuelle Detaildefinitionen für Aussehen und Verhalten des X Window Systems – die so genannte *Ressource-Datei*. Das X Window System verwendet eine große Anzahl solcher Ressource-Dateien.

- .kde** Verzeichnis mit weiteren Unterverzeichnissen (z.B. *Autostart* und *share*), welches vom KDE-Desktop verwendet wird
- .gnome-desktop** Verzeichnis mit weiteren Unterverzeichnissen (z.B. *accels* oder *application-info*) sowie der Datei *Gnome*, welche Gnome-Konfigurationsparameter enthält.

Initialisierungsdateien für Applikationen

Nahezu alle größeren Applikationen und auch viele Linux-Kommandos können über derartige Konfigurationsdateien, die typischerweise im HOME-Verzeichnis eines Benutzers liegen und deren Name mit einem Punkt beginnt, an die individuellen Bedürfnisse des Benutzers angepasst werden. Per Konvention tragen diese Dateien, eventuell auch Verzeichnisse, die Buchstaben *rc* am Ende des Namens¹ – aber nicht alle. Beispiele hierfür sind:

- .mailrc** Definitionen für das Mail-System
- .emacs** Definitionen von Grundeinstellungen, Abkürzungen und Tastaturmakros für den Editor **emacs**
- .gnu-emacs** wie **.emacs**, jedoch für die GNU-**emacs**-Version
- .exrc** Definitionen von Grundeinstellungen, Abkürzungen und Tastaturmakros für den Editor **vi**
- .vimrc** Definitionen von Grundeinstellungen, Abkürzungen und Tastaturmakros für den Editor **vim**
- .vimrc** Definitionen von Grundeinstellungen, Abkürzungen und Tastaturmakros für den Editor **vim**
- .rhosts** Festlegung von Rechnern und Benutzernamen, die über Netz Zugriff auf Dateien und Verzeichnisse eines Benutzers haben sollen

Applikationsdateien zum Speichern von Einstellungen (Beispiele)

- .bashhistory** Hier legt die **bash** die History ab, d.h. die *n* letzten vom Benutzer eingegebenen Kommandos, die dann mit dem history-Mechanismus abgerufen werden können.
- .lproptions** Hier legt das Programm **lproptions** (siehe Seite 336) die benutzerspezifischen Optionsvorbelegungen ab.
- .mozilla** Verzeichnis, in dem Einstellungen zum Web-Browser Mozilla liegen
- .nautilus** Verzeichnis, in dem Einstellungen zum GNOME-Dateimanager **nautilus** liegen

1. Es gibt eine Reihe von Interpretationen für dieses *rc* – die einleuchtendste davon besagt, dass *rc* für *run commands* steht.

3.1.8 Grafische Oberflächen – Desktops

Unter Unix gab es zunächst eine ganze Reihe unterschiedlicher grafischer Oberflächen, die – wenn sie eine komplette Arbeitsoberfläche für alle möglichen Funktionen zur Verfügung stellen – auch als *grafischer Desktop* bzw. Schreibtischoberfläche bezeichnet werden. Allmählich setzte sich unter den klassischen Unix-Systemen der CDE-Desktop durch (*Common Desktop Environment*). CDE basiert auf der X11-Motif Bibliothek für grafische Oberflächen. Da Motif einer kostenpflichtigen Lizenz unterliegt, schuf man für das offene Linux andere, freie Bibliotheken und darauf aufbauende Desktops. Dabei haben sich zwei funktional ähnliche, aber von der Technik und Bibliothek her unterschiedliche Lösungen durchgesetzt:

- ▶ KDE (*K-Desktop-Environment*), welches auf der Qt-Bibliothek der Firma Trolltech aufbaut und insbesondere in Deutschland durch den Linux-Distributor SuSE starke Verbreitung gefunden hat.
- ▶ GNOME (*GNU Network Object Model Environment*), welches in der amerikanischen Linux-Gemeinde stärkere Verbreitung hat und ein GNU-Projekt ist. Sowohl Sun als auch HP möchten mit GNOME mittelfristig ihren CDE-Desktop ablösen. Damit hat GNOME das Potenzial, die größere Gesamtverbreitung zu finden.

Die meisten Linux-Distributionen liefern beide Desktops mit aus. Der Benutzer hat also die Wahl zwischen beiden Systemen. Er kann sogar zwischen den beiden Desktops hin- und herschalten.

Die Desktops bringen jeweils gleich ein ganze Reihe kleinerer und größerer Anwendungen mit, die speziell auf den jeweiligen Desktop abgestimmt sind und die Möglichkeiten der jeweiligen Bibliothek ausnutzen. Damit wird der Übergang zwischen dem Desktop und den darauf (oder darunter) laufenden Anwendungen teilweise sehr fließend.

Viele grafische Anwendungen können unter beiden Desktops laufen. Einige Anwendungen sind jedoch von den spezifischen Basisfunktionen eines bestimmten Desktops abhängig und laufen ohne diesen entweder gar nicht oder mit hässlichen Nebenwirkungen. Hierzu zählt z. B. (bisher) **nautilus**.

Virtuelle Desktops

Die Unix-/Linux-Desktops (CDE, KDE, GNOME) bieten die Möglichkeit, statt eines Schreibtisches, gleich mehrere virtuelle Desktops (jeweils gleicher Art) zur Verfügung zu stellen. Als *virtueller Desktop* wird dabei jeweils ein *virtueller Bildschirm* verstanden, dargestellt durch den Bildschirm und die darauf liegenden Ikonen, welche Anwendungen bzw. Programme, Dateien oder andere Objekte (z. B. den Mülleimer) repräsentieren. Der Anwender kann sich damit unterschiedliche Arbeitsumgebungen schaffen. Über die Desktop-Ikonen in der Menüleiste des Systems lässt sich dann schnell von virtuellem Schreibtisch zu virtuellem Schreibtisch schalten. Man bekommt dabei jeweils den gerade selektierten Desktop angezeigt, ohne dass die Übersichtlichkeit durch zu viele Ikonen oder Fenster auf dem einzelnen Desktop verloren geht.



Abb. 3.3: Kontrollleiste mit den Icons der virtuellen KDE-Desktops (hier vier)

Diese virtuellen Desktops können über Sitzungen (ein An- und Abmelden) hinweg bestehen bleiben. Ein Desktop besteht nicht nur aus den Icons auf dem Bildschirm (Desktop) und den dort offenen (aktiven) Anwendungsfenstern, sondern ebenso aus einem Hintergrundbild (oder einer Hintergrundfarbe) und weiteren Charakteristika. Über unterschiedliche Desktop-Hintergründe kann man die Orientierung vereinfachen, auf welchem seiner angelegten Desktops man sich befindet.

Anpassung der grafischen Oberfläche

Die Anpassung der grafischen Oberfläche erfolgt über die Parameter in den entsprechenden Definitionsdateien – recht zahlreichen. Wie bei den Profile-Dateien für die Shell gibt es auch hier gleich mehrere Stellen (Dateien), in denen dies – eher global (für alle Benutzer eines Rechners) und lokal (für den aktuellen Benutzer) oder noch spezifischer für die aktuelle Sitzung das aktuelle Fenster oder den aktuellen Aufruf eines Programms – erfolgen kann. Dabei überdeckt die speziellere Einstellung jeweils die allgemeinere (zu konkreten Details und Beispielen siehe Kapitel 7, Seite 627). Viele dieser Einstellungen lassen sich wiederum entweder durch Editieren der entsprechenden Datei oder über eine grafische Oberfläche (einer Hilfsanwendung) setzen, wobei letzteres sicher die bessere Lösung ist, da die entsprechenden Einstellungsanwendungen (z. B. das KDE- oder GNOME-Kontrollzentrum) die meisten Fehler gleich abfangen oder in ihren Einstellungen erst gar nicht anbieten.



Abb. 3.4: Kontrollzentrum zum Anpassen der Desktops für GNOME (links) und KDE

3.1.9 Information zur aktuellen Umgebung

Eine Reihe von Kommandos erlauben es dem Benutzer, sich über seine aktuelle Umgebung zu informieren.

date	gibt das aktuelle Datum und die Uhrzeit aus.
echo \$variable	zeigt den aktuell gesetzten Wert der Shell-Variablen <i>variable</i> an.
env	zeigt die aktuell globalen Variablen (auch zum Setzen von Variablen).
history	zeigt bei der bash , csh , tcsh und ksh eine Liste der zuletzt ausgeführten Kommandozeilen an.
hostname	gibt den aktuellen Rechnernamen aus.
id	gibt aktuellen Benutzernamen und Nummer aus.
logname	gibt den aktuellen Benutzernamen aus.
printenv	gibt die aktuellen Umgebungsvariablen (<i>Environment</i> -Variablen) aus.
pwd	gibt das aktuelle Arbeitsverzeichnis aus.
set	belegt Umgebungsvariablen oder gibt (ohne Parameter aufgerufen) die aktuellen Umgebungsparameter aus.
setenv	belegt und exportiert Umgebungsvariablen der csh und tcsh oder gibt die aktuell exportierten Umgebungsparameter aus.
top	zeigt die aktuell laufenden Prozesse des Systems mit zahlreichen Zusatzangaben (z.B. unter welchem Benutzer sie laufen). gtop ist eine GUI-Variante davon.
stty	liefert die aktuell gesetzten Charakteristika des Terminals oder setzt diese neu.
tset	erlaubt das Setzen verschiedener Modi der Dialogstation.
tty	liefert den Namen der Dialogstation.
uname -a	liefert Namensangaben zum eigenen System.
uptime	gibt aus, wie lange und unter welcher Last das System schon läuft
users	gibt in einer Kurzform alle gerade aktiven Benutzer aus.
who	gibt die aktiven Benutzer aus und die Dialogstationen, an denen sie arbeiten.
who am I	liefert die eigene, aktuelle Benutzeridentifikation.

date liefert die aktuelle Zeit und das Datum zurück und erlaubt dem Super-User, diese neu zu setzen. Da der Erstellungs- und Modifikationszeitpunkt von Dateien eine wichtige Information ist, sollte es nicht versäumt werden, diese Daten gelegentlich zu überprüfen und zu korrigieren.¹

Mit **who am I** kann ein Benutzer seinen eigenen Benutzernamen abfragen. In der Form mit nur **who** wird zurückgeliefert, welche Benutzer im System an welchen Dia-

1. Dies setzt jedoch Administrationsprivilegien voraus.

logstationen gerade aktiv sind. Das **users**-Kommando ist eine verkürzte Form hiervon und gibt nur die Benutzernamen aus. Das Kommando **logname** gibt im Gegensatz dazu den Namen aus, unter dem sich der Benutzer mit **login** beim System angemeldet hatte. Wurde inzwischen ein **su** oder zusätzliches **login**-Kommando aufgerufen, so unterscheidet sich dieser Name von dem, den **who am I** zurückliefert.

pwd liefert das aktuelle Arbeitsverzeichnis als vollständigen Pfadnamen zurück und zählt damit zu den vermutlich am häufigsten verwendeten Kommandos. In der Korn-Shell und der C-Shell ist es möglich, den aktuellen Pfadnamen in das Prompt mit aufzunehmen.

Mit **env**, **printenv**, **setenv** oder **set** werden die Namen und Werte der in der aktuellen Umgebung definierten Shell-Variablen ausgegeben bzw. definiert. Mit dem Kommando **echo \$variable** lässt sich der aktuelle Wert einer Shellvariablen anzeigen.

Die **bash**, **tcsh**, C- und Korn-Shell erlaubt, zusätzlich einen Kommandospeicher zu führen. In ihm wird eine vorgebbare Anzahl von Kommandozeilen gespeichert. Das Kommando **history** zeigt diese letzten Kommandozeilen mit ihrer Kommandonummer an. Die Kommandos (auch Teile davon) können dann unter Angabe der Kommandonummer erneut aufgerufen werden, ohne dass das vollständige Kommando eingetippt werden muss.

Das Kommando **stty** erlaubt das Setzen neuer Parameter für die alphanumerische Dialogstation. Ruft man das Kommando ohne einen Parameter auf, so erhält man die aktuell gesetzten Werte zurück, während das **ttty**-Kommando den Namen der Dialogstation liefert, an der die Sitzung gerade stattfindet. **tset** ist eine Erweiterung von **stty** (leider keine vollständige) und erlaubt mehr Optionen. Ohne aufgerufene Parameter liefert es wie **stty** die aktuell gesetzten Zeichen für `lösche zeile` (`(kill)`) und `lösche zeile` (bzw. `delete`).

3.1.10 Benutzerkommunikation

Das Linux-System bietet bereits im Standard-Lieferumfang mehrere Formen der Kommunikationen verschiedener Benutzer untereinander. Dazu gehören

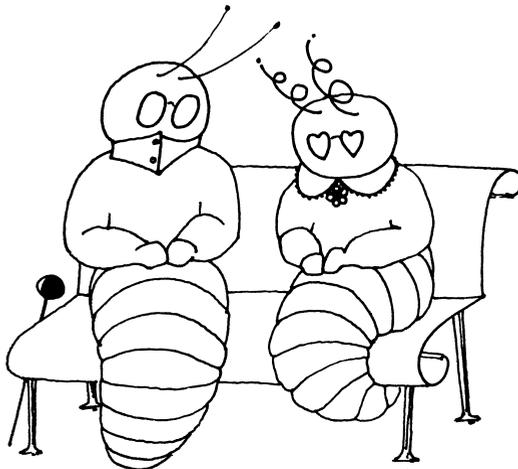
- ▶ Text in der Datei */etc/motd*, die beim **login** ausgegeben wird.
- ▶ Nachrichten eines Benutzers an einen anderen (einseitig) mittels E-Mail. Dazu bietet Linux neben der einfachen, alphanumerischen Version **mail**, **mailx** oder **elm** gleich ein ganzes Spektrum von GUI-Mail-Clients wie etwa **kmail** (KDE), **balsa** (aus GNOME), das Groupware-Programme **evolution** oder die Mail-Clients aus Mozilla oder Netscape.
- ▶ interaktiver Dialog zwischen zwei Benutzern über den Rechner mittels **write**,
- ▶ Nachrichten (des Super-Users) an alle Benutzer mittels **wall**.
- ▶ einen schlichten, bidirektionalen Dialog mittels des Programms **talk**.
- ▶ die dialogartige Kommunikation lokal oder über Internet unter Verwendung des *Internet-Relay-Chats* (kurz: *irc*). Dies beschränkt die Kommunikation nicht mehr auf Unix-/Linux-Systeme; ein IRC-Client steht auf fast allen Systemen zur Verfügung. Ein IRC-Server übernimmt hierbei die Anmeldung der Benutzer und die Vermittlung der Nachrichten.

- ▶ Videokonferenzen, bei denen die Teilnehmer sich nicht nur gegenseitig hören (praktisch telefonieren), sondern auch sehen können. Diese laufen heute zumeist unter Nutzung des Open-H.323-Standards. Solche Konferenzen setzen jedoch ausreichend Netzbandbreite, Soundkarten und Mikrofone und für die Bildübertragung auch Videokameras (oder Webcams) voraus. Ein solcher Client ist *vic*¹ oder *gnomemetting*.²

Für die erste genannte Art von Nachrichten des Systemmanagers (Super-User) an alle Benutzer gibt es folgende Möglichkeiten:

- ▶ Der Super-User schreibt eine Nachricht in die Datei */etc/motd* (*message of the day*). Diese Nachricht wird dem Benutzer beim Anmelden automatisch auf die Dialogstation ausgegeben. Die Meldung sollte entsprechend kurz und für alle Benutzer von Interesse sein.
- ▶ Durch das Kommando *wall*, welches den Text einer angegebenen Datei (oder die nachfolgenden Zeilen bis zu einem `eof`) auf alle angeschlossenen Dialogstationen ausgibt. Hiermit wird man in der Regel alle aktiven Benutzer über bevorstehende Systemänderungen (z. B. das Herunterfahren des Systems) informieren. *wall* kann zwar von allen Benutzern verwendet werden, ist aber nur im Super-User-Modus in der Lage, die Zugriffsrechte der Dialogstationen zu durchbrechen und damit eine Ausgabe auch dann sicherzustellen, wenn Benutzer durch das Kommando *mesg n* Ausgaben anderer Benutzer auf ihre Dialogstation unterbunden haben.

Bei den höher entwickelten Kommunikationsformen wie *talk*, Chats, E-Mail oder der Videokonferenz sind Server als Vermittler involviert. Hier müssen entsprechend die Server aufgesetzt und aktiviert werden (siehe dazu Kapitel 9.10, Seite 811 ff.) oder man verwendet öffentlich zugängliche Server im Internet.



1. Zu finden unter <http://www-nrg.ee.lbl.gov/vic/>.

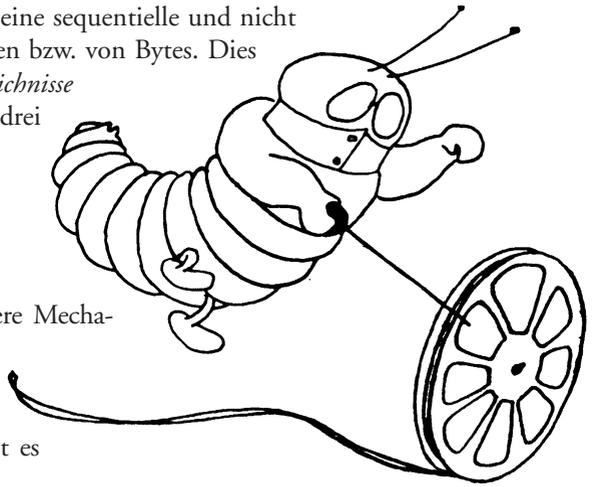
2. Zu finden unter <http://www.gnomemeeting.org>.

3.2 Das Unix-/Linux-Dateikonzept

Eine Datei ist unter Linux zunächst eine sequentielle und nicht weiter strukturierte Folge von Zeichen bzw. von Bytes. Dies gilt für *normale Dateien* und *Verzeichnisse* (*directories*). Insgesamt kann man drei Arten von Dateien unterscheiden:

- ▶ normale Dateien
- ▶ Verzeichnisse
- ▶ Gerätedateien

Daneben stellt der Linux-Kern weitere Mechanismen zur Verfügung, die den direkten Austausch von Daten zwischen Programmen erlauben: *Pipes*, *Streams* und *Sockets*. Zusätzlich gibt es das *Prozessdateisystem*.



Daneben bieten die *Netzwerkdateisysteme* (z.B. NFS, Samba zur Emulation des Windows-Netzwerk-Dateizugriffs (SMB) und *netatalk* durch die Emulation von Apple-Talk und eines Apple-Macintosh-Dateisystems) transparenten Dateizugriff auf Daten anderer Rechner im Netzwerk. Auch per WebDAV kann über ein erweitertes HTTP-Protokoll transparent über Internet bzw. TCP/IP auf die Dateien anderer Rechner zugegriffen werden.

Das UNIX-/Linux-Dateikonzept zeichnet sich durch eine Reihe von Eigenschaften aus. Die wichtigsten dieser Eigenschaften sind:

- ▶ **Hierarchisches Dateisystem**
Die Struktur der Dateiverzeichnisse, Geräteinträge und Dateien auf einem Datenträger mit wahlfreiem Zugriff ist ein (invertierter) Baum. Restriktionen bezüglich Breite und Tiefe des Baums existieren faktisch nicht.
- ▶ **Weitestgehende Geräteunabhängigkeit**
Verzeichnisse (*directories*), normale Dateien und Geräte werden unter Linux syntaktisch gleich und auch semantisch soweit sinnvoll identisch behandelt. Dieses Konzept wird noch durch die Möglichkeit der Intertask-Kommunikation über Pipes ausgedehnt, welche auch über Ein-/Ausgabeoperationen angesprochen werden. Auch für den Geräte- und Dateizugriff über Netz (und damit Rechengrenzen hinweg) gilt diese weitgehende Transparenz.
- ▶ **In hohem Maße adaptiv**
Der Linux-Kern stellt nur wenige, aber flexible Dateioperationen zur Verfügung. Hierdurch werden keine Restriktionen für Erweiterungen vorgegeben. Diese Erweiterungen können dann auf der Ebene der Laufzeitsysteme oder von Datenbanksystemen zur Verfügung gestellt oder durch den Benutzer selbst vorgenommen werden.

► **Die Möglichkeit, mehrere unterschiedliche Dateisysteme zu unterstützen**

Hierzu zählen sowohl mehrere verschiedene lokale Dateisysteme als auch Dateisysteme mit netzweitem Dateizugriff oder CD-ROM-Dateisysteme. Daneben sind Zugriffe auch auf Dateisysteme anderer Betriebssysteme wie etwa OS/2, Microsoft Windows (teilweise nur lesend) oder Apple MAC/OS möglich.

Linux verdeckt die Unterschiede dieser Dateisysteme sehr weitgehend, so dass es für die meisten Anwendungen und Kommandos nicht sichtbar ist, auf welchem Dateisystem sie arbeiten. Gewisse Unterschiede, etwa hinsichtlich der maximal zulässigen Länge der Dateinamen oder der maximalen Größe einer Datei lassen sich jedoch nicht ganz verbergen. Hierauf ist dann in speziellen Situationen zu achten.

3.2.1 Dateiartern

Der Linux-Kern unterstützt außer den Dateiartern *normale Datei*, *Dateiverzeichnis*, *Gerätedatei* sowie *Pipes* keine weiteren Dateistrukturierungen. Derartige Interpretationen sind rein programm- oder datenbankabhängig.

Normale Dateien

Normale Binärdateien sind einfach eine Folge von Bytes. *Normale Textdateien* bestehen aus einer linearen Folge von Zeilen, wobei einzelne Zeilen durch ein *<neue Zeile>*-Zeichen (*new line* = `<lf>` bzw. `\012` im ASCII oder ISO-665971-Code) getrennt sind. Das Linux-System stellt jedoch in der C-Bibliothek oder durch die verschiedenen Laufzeitsysteme der Hochsprachen wie etwa C, C++, Java oder COBOL Routinen zur Verfügung, um auf beliebige Bytefolgen innerhalb einer Datei zuzugreifen, um Zeichen (Bytes) sequentiell zu lesen oder zu schreiben oder um eine durch *<neue zeile>* abgeschlossene Zeile zu holen. Bei Sprachen wie COBOL sind auch indexsequenzielle Zugriffe und Schlüsselwortzugriffe im Laufzeitsystem implementiert.

Verzeichnisse

(Datei-)Verzeichnisse (Kataloge, englisch: *directories*) sind Dateien, welche entweder leer sind (sie enthalten dann lediglich einen Verweis auf sich selbst und auf ihr Vaterverzeichnis¹) oder aber Verweise (englisch: *links*) auf weitere Dateien enthalten. Die Verweisstruktur ist dabei hierarchisch bzw. baumartig. Der Eintrag einer Datei in einem Verzeichnis besteht aus der Knotennummer (*i-node-number*) der Datei und den Zeichen des Dateinamens.² Die Anzahl der Einträge in einem Verzeichnis ist nur durch die maximale Größe einer Datei und des Datenträgers limitiert. Es empfiehlt sich jedoch aus Performance-Gründen, nicht zu viele Dateien nebeneinander in ein Verzeichnis zu legen. Ein Wert von 5 000–10 000 Dateien pro Verzeichnisebene ist hier eine sinnvolle Grenze. Bei der Suche muss (bei den normalen Unix-/Linux-Dateisystemen) nämlich linear durchsucht werden.

1. Die erste Stufe in Richtung der Wurzel des Dateibaums.

2. Diese Beschreibung ist auf die Standard UNIX-/Linux-Dateisysteme ausgelegt.

Gerätedateien

Gerätedateien (englisch: *special files*) sind Einträge, welche für die physikalischen Geräte stehen. Sie werden deshalb hier auch als *Geräteeinträge* (*devices*) bezeichnet. Durch ihre Behandlung als Dateien ergibt sich für den Benutzer kein Unterschied zwischen der Ein-/Ausgabe auf Dateien oder auf physikalische Geräte. *Special files* liegen in der Regel in dem Verzeichnis */dev*. Angelegt werden sie mit dem **mknod**-Kommando. Bei den Gerätedateien kann man nochmals zwischen *realen* und *Pseudogeräten* unterscheiden. Als reale Gerätedatei sei hier ein *special file* verstanden, das für ein tatsächlich vorhandenes Gerät steht, während bei einem *Pseudogerät* der Eintrag nur ein abstraktes Gerät angibt, das seinerseits erst auf ein reales Gerät oder ein anderes Ersatzgerät hinweist oder nur als Funktion im Kernel realisiert ist. So ist die aktuelle Dialogstation */dev/tty* ein Pseudogerät, unter dem jeweils die wirkliche aktuelle Dialogstation angesprochen werden kann, ohne dass man dazu deren konkreten Namen wissen muss, während das Gerät */dev/null* alle Ausgaben einfach wegwirft.

Namen von »special files«

Beispiel für *Geräte* bzw. *special files* sind:¹

Reale Geräte:	
<i>/dev/fd0h1440</i>	Floppy-Laufwerk 0 im HD-Format mit 1440 KB
<i>/dev/lp1</i>	Drucker (<i>line printer</i>) an der Schnittstelle 1
<i>/dev/hdan</i>	Magnetplatte (<i>hard disk</i>) am 1. Anschluss des ersten IDE-Kanals, Partition <i>n</i> . Die Partition 0 meint dabei die gesamte Platte.
<i>/dev/hdbn</i>	Magnetplatte (<i>hard disk</i>) am 2. Anschluss des ersten IDE-Kanals
<i>/dev/hdcn</i>	Magnetplatte (<i>hard disk</i>) am 1. Anschluss des zweiten IDE-Controllers
<i>/dev/sdxy</i>	Magnetplatte (<i>scsi disk</i>) an einem SCSI-Controller, <i>x</i> gibt dabei den SCSI-Kanal und <i>y</i> die Partition auf dem Laufwerk an. Die Partition 0 meint dabei die gesamte Platte.
<i>/dev/stxy</i>	Bandlaufwerk an einem SCSI-Controller, <i>x</i> gibt dabei den SCSI-Kanal und <i>y</i> die Partition auf dem Laufwerk an. <i>/dev/nstxy</i> ist das gleiche Gerät, ohne dass nach der Bandoperation das Band automatisch zurückgespult wird (<i>non-rewind scsi tape</i>).
<i>/dev/ttySn</i>	(alphanumerische) Dialogstation an der seriellen Schnittstelle <i>n</i>
<i>/dev/usb/</i>	Unterverzeichnis, in dem die USB-Geräte liegen
<i>/dev/usb/lp1</i>	Drucker 1 (<i>line printer</i>) an einer USB-Schnittstelle
<i>.../usb/mouse</i>	Maus an einer USB-Schnittstelle
<i>.../usb/scanner0</i>	Scanner 0 an einer USB-Schnittstelle

1. Zahlreiche Geräte (*special files*) sind unter **man 5** detaillierter beschrieben – leider nicht alle.

Pseudogeräte:

<code>/dev/bell</code>	gibt an der aktuellen Dialogstation des Benutzers einen Glockenton aus.
<code>/dev/console</code>	Dies ist die Systemkonsole, d. h. das Gerät, an dem Systemmeldungen ausgegeben werden (etwa fehlender Plattenplatz oder unberechtigter Zugangsversuch zum System).
<code>/dev/dvd</code>	ist ein DVD-Laufwerk. Er verweist per Link auf das 1. reale DVD-Laufwerk (falls vorhanden).
<code>/dev/fd/<i>n</i></code>	gestattet den Zugriff auf den aktuellen Dateideskriptor <i>n</i> eines Programms, ohne dass man den Namen des wirklichen Geräts sowie der Datei kennen muss. 0 ist dabei zumeist die Standardeingabe, 1 die Standardausgabe und 2 die Standardfehlerausgabe.
<code>/dev/kmem</code>	Das Pseudogerät <i>kmem</i> arbeitet wie <i>mem</i> , entspricht jedoch dem virtuellen Betriebssystemspeicher.
<code>/dev/log</code>	Informationen, die hierauf geschrieben werden, gehen in das System-Logbuch.
<code>/dev/mem</code>	Dies ist ein Abbild des physikalischen Hauptspeichers. Es kann dazu benutzt werden, Benutzerdaten zu untersuchen oder zu ändern; dies darf jedoch nur der Super-User.
<code>/dev/mouse</code>	gestattet, die Maus einer grafischen Dialogstation anzusprechen.
<code>/dev/null</code>	steht für das <i>Null-Gerät</i> . Jede Ausgabe darauf wird <i>weggeworfen</i> und jede Eingabe liefert <code><eof></code> (<i>end of file</i>) zurück. Dieses Gerät (Datei) ist für Testzwecke nützlich und dort, wo man eine störende Programmausgabe wegwerfen möchte.
<code>/proc</code>	Dieses virtuelle Dateisystem(es liegt unter Linux nicht in <code>/dev</code>) realisiert einen dynamischen Dateibaum, in dem sowohl zum aktuell laufenden Betriebssystem (Konfiguration, Parameter, Laufzeitwerte) als auch zu den darin laufenden Prozessen Informationen gehalten werden, die nur zur Laufzeit existieren. Siehe dazu auch Seite 143.
<code>/dev/ptsxx</code>	Pseudobildschirm-Anschlüsse, die einzelnen Terminalemulationen unter der grafischen Oberfläche zugeordnet werden
<code>/dev/stdin</code>	Datei oder Gerät, auf der aktuell die Standardeingabe (Dateideskriptor 0) eines Programms liegt
<code>/dev/stdout</code>	Datei oder Gerät, auf der aktuell die Standardausgabe (Dateideskriptor 1) eines Programms liegt
<code>/dev/stderr</code>	Datei oder Gerät, auf der aktuell die Standardfehlerausgabe (Dateideskriptor 2) eines Programms liegt
<code>/dev/systty</code>	Dieses Gerät stellt die physikalische Systemkonsole dar. Findet der <code>init</code> -Prozess beim Starten des Systems die Datei <code>/etc/inittab</code> nicht, so meldet sich das System auf diesem Gerät.

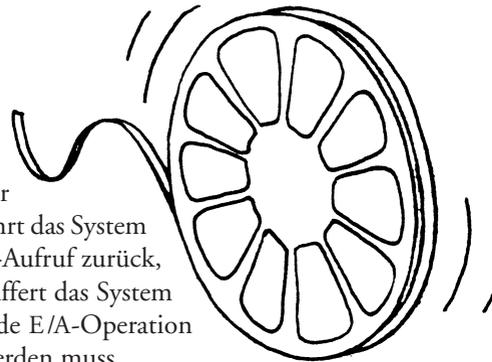
- /dev/swap** Dies ist das *swap device*. Auf diesen logischen Datenträger werden die Programmsegmente ausgelagert, wenn der Hauptspeicher nicht mehr für alle Anforderungen ausreicht und das System einzelne Hauptspeicherseiten vorübergehend auf einen Hintergrundspeicher auslagern muss.
- /dev/tty** Dies ist innerhalb eines Prozesses die virtuelle Dialogstation, von welcher der Prozess gestartet wurde. Dies erlaubt, Nachrichten auf die startende Dialogstation zu senden, auch wenn die Standardausgabe umgelenkt wurde.
- /dev/zero** Dieses Gerät liefert beim Lesen beliebig viele Blöcke mit 0 Bytes zurück.

Bei den *special files* wird unterschieden zwischen den *block devices*, welche ein höhere Abstraktionsebene bieten, die weitgehend die wirklichen physikalischen Eigenschaften des Gerätes abstrahieren. Typische Vertreter sind hier die Magnetplatten, auf die blockweise zugegriffen wird und dies in aller Regel über weitere Zwischenschichten wie z. B. die Dateisysteme. Die zweite Art sind die so genannten *raw devices* oder auch *charakter devices*, d. h. die *rohen* Geräte, die nicht blockweise, sondern zeichenweise angesprochen werden. Sie erlauben den Zugriff auf die physikalische Gerätestruktur (z. B. Blöcke bei Magnetplatten und Bändern).

Bei diesen *raw devices* verbirgt das System somit nicht mehr die gerätespezifischen Eigenschaften vor dem Benutzer. Die Geräteunabhängigkeit geht somit verloren. Mit diesen Geräten wird dann gearbeitet, wenn eine besondere Geschwindigkeit erreicht oder spezielle Geräteeigenschaften ausgenutzt werden sollen. Unter Linux erfolgt die Ein- und Ausgabe auf Geräte in der Regel *synchron*, d. h. bei einer Ein- oder Ausgabe kehrt das System für das Programm sichtbar erst dann aus dem E/A-Aufruf zurück, wenn die Ein- oder Ausgabe beendet ist. Intern puffert das System solche Übertragungen jedoch, so dass nicht für jede E/A-Operation ein wirklicher Ein-/Ausgabevorgang angestoßen werden muss.

Darüber hinaus versucht Linux ein *Vorauslesen*. Dies führt zu einem erhöhten Systemdurchsatz, bedingt jedoch, dass der logische Zustand einer Datei nicht immer mit dem physikalischen Zustand (den Daten auf dem Dateiträger) identisch ist. Dies kann vor allem bei Systemabstürzen fatale Folgen haben.

Wird mit *rohen* Geräten gearbeitet, so entfällt dieser Mechanismus (Pufferung und Vorauslesen); hier entspricht ein logischer Transfer einem physikalischen. Dieser muss dann aber in Einheiten von einem Vielfachen der Geräteblockung (512 Byte, oder in Einheiten von 1 KB bis 8 KB) erfolgen. Die hierbei verwendete Blockgröße ist abhängig vom Dateisystemtyp sowie innerhalb eines Dateisystems von voreingestellten Werten.



Unter Linux werden die *rohen Geräte* im Englischen irreführend als *character oriented*, d.h. als *zeichenorientiert* bezeichnet. Dies hängt damit zusammen, dass der Zugriff auf *rohe Geräte* früher über den gleichen Mechanismus ablief wie der auf zeichenorientierte Geräte (z.B. Dialogstation und Drucker). Blockorientierte Geräte benutzen einen anderen internen Mechanismus. Der Begriff *unstrukturiert* wäre hier sicher korrekter. Das Konzept der Geräte und Gerätenamen befindet sich bei Linux aktuell im Umbruch. Schaut man einmal in das Verzeichnis */dev*, so wird man von der Vielzahl der dort vorhandenen Einträge förmlich erschlagen. Dabei findet man sehr viele gleiche Einträge, die sich nur durch die letzte Nummer oder Buchstaben unterscheiden und für die oft gar kein wirkliches Gerät vorhanden ist. Sie wurden zur Vereinfachung nur *auf Vorrat* angelegt. Das neue Konzept geht dahin, die Geräteeinträge in Form eines Dateisystems zu organisieren, bei dem – analog zu einer Datei – die Einträge dynamisch erst bei wirklichem Gebrauch erzeugt werden. Da das bisherige Konzept von Geräten in */dev* in vielen Programmen noch verankert ist, wird es die alten Einträge noch eine ganze Zeit geben und der Transit zum *Geräte-Dateisystem* sich nur langsam vollziehen.

Pipes sind zwar keine Dateien wie die bisher vorgestellte normale Datei, das Verzeichnis oder die Gerätedatei, sie stellen jedoch einen Mechanismus zur Verfügung, der einen Datenaustausch zwischen zwei Programmen erlaubt, wobei diese normale Ein-/Ausgabeoperationen wie Lesen (**read**) und Schreiben (**write**) benutzen (hier ist der Betriebssystemaufruf **read** bzw. **write** gemeint). Die *Pipe* wird über einen systeminternen Puffer realisiert. Dieser Puffer ist standardmäßig 4 oder 8 KByte groß und wirkt als FIFO-Puffer (*first in first out*). Eine Pipe ist unidirektional, d.h. es kann nur ein Prozess lesen und der andere schreiben. Müssen Daten in beiden Richtungen ausgetauscht werden, so sind zwei Pipes aufzusetzen. Dieses Aufsetzen durch den Systemaufruf **pipe** muss von einem gemeinsamen Vaterprozess der kommunizierenden Prozesse erfolgen.

Diese Einschränkung wird durch den Mechanismus der **Named Pipes** aufgehoben. Liest ein Prozess von einer Pipe, die leer ist, so wird er solange suspendiert, bis ein anderer Prozess etwas in die Pipe geschrieben hat. Möchte ein Prozess in eine volle Pipe schreiben, so wird er ebenfalls suspendiert, bis wieder ausreichend Platz in der Pipe vorhanden ist. Positionierbefehle (**lseek**) auf Pipes liefern einen Fehler. Liest ein Prozess aus einer Pipe, deren anderes Ende nicht (mehr) zum Schreiben geöffnet ist, so erhält er `<eof>` zurück.

3.2.2 Dateiattribute

Normale Dateien und Verzeichnisse (*directories*) haben eine Reihe von Attributen (Metadaten), wovon der Benutzer jedoch die meisten nicht ständig sieht. Zu den wichtigsten Attributen gehören:

- ▶ der Dateiname (ohne Zugriffspfad)
- ▶ der Datei-Zugriffspfad
- ▶ die Länge der Datei (in Byte und in Blöcken zu 512 Byte)
- ▶ die Zugriffsrechte auf die Datei
(auch *protection bits* oder *Dateimodus* genannt)

- ▶ die Knoten- oder Indexnummer (*Inode Number*, eine eindeutige Nummer der Datei in einem Dateisystem)
- ▶ die Anzahl von Verweisen (englisch: *links*) auf die Datei
- ▶ das Datum der Dateierstellung, der letzten Änderung der Datei und des letzten Zugriffs auf die Datei
- ▶ der Dateityp (normale Datei, *special file*, Verzeichnis, symbolischer Link)
- ▶ die Benutzernummer des Besitzers und seiner Gruppe

Spezielle Dateisysteme können weitere Dateiattribute halten.¹ Man nennt dies auch die *Metadaten* der Datei. Sie liegen alle – mit Ausnahme des Dateinamens und des Pfades – im Dateikopf (Inode). Die meisten dieser Attribute werden durch das `ls`-Kommando mit der Option `-lsi` angezeigt (s. `ls`-Kommando auf Seite 345).

Dateinamen

Der Dateiname benennt die in der Datei zusammengefasste Information. Der Name durfte in älteren Unix-Systemen nur bis zu 14 Zeichen lang sein. In aktuellen Unix- und Linux-Systemen darf er bis zu 255 Zeichen lang sein – soweit die Datei auf einem dafür ausgelegten Dateisystem liegt.² Im Prinzip sind alle Zeichen außer Null (`/000`) erlaubt; die neueren Linux-Systeme und moderne Linux-Dateisysteme erlauben sogar die Verwendung von Dateinamen in UTF-8 (Unicode). Jedoch ist es meist sinnvoll, sich beim Zeichenumfang von Dateinamen zu beschränken. Aus praktischen Gründen sollte man sich auf Buchstaben, Ziffern und die Sonderzeichen `>`, `,`, `_` sowie `>-<` beschränken. Das Minuszeichen `>-<` sollte dabei nicht als erstes Zeichen des Dateinamens verwendet werden, da sonst oft Konflikte mit Optionen entstehen; `>/<` in Dateinamen ist noch problematischer, da dies auch als Trenner in Dateipfaden interpretiert wird.

→ Auch sollte man – obwohl dies bei neuen Systemen zulässig ist – mit Umlauten, `>ß<` und anderen europäischen Sonderzeichen in Dateinamen vorsichtig umgehen, da sie leicht zu Kompatibilitätsproblemen führen.

Unix und Linux **unterscheiden Groß-/Kleinschreibung** in Dateinamen. Darauf ist besonders zu achten, wenn man Windows-Systeme gewohnt ist. Der Dateiname ist – wie bereits erwähnt – kein echter Bestandteil einer Datei.

Eine physikalische Datei kann, soweit es sich nicht um eine Verzeichnisdatei handelt, mehrere Dateinamen zugleich besitzen. Die verschiedenen Namen dürfen dabei in unterschiedlichen Ästen eines Dateibaums liegen; sie müssen sich jedoch alle auf dem gleichen logischen Datenträger (korrekter: auf der gleichen logischen Partition bzw. auf dem gleichen Dateisystem) befinden. Die Datei ist dann unter allen diesen Namen ansprechbar. Diese Verweise auf die Datei nennt man in der UNIX-/Linux-Terminologie *Hard Links*.

Das Linux-Dateisystem (und fast alle weiteren moderneren UNIX-Dateisysteme) erlauben daneben auch Namensverweise über die Grenzen eines einzelnen Dateisystems, ja sogar über die eines Rechnersystems im Netz hinaus. Diese Verweise werden als *Symbolic Links* bezeichnet. Mehr dazu später.

1. Insbesondere das später noch beschriebene `xfs`-Dateisystem weist hier eine große Offenheit auf.

2. Siehe hierzu Tabelle 3.2 auf Seite 147.

Im eigentlichen Dateikopf (siehe *Knotennummer*) befindet sich ein Benutzungszähler (*link count*), welcher festhält, wie viele Namensreferenzen (*links*) auf die Datei existieren. Die Datei (der eigentliche Dateiinhalt) wird erst dann gelöscht, wenn alle Dateiverweise (Namenseinträge in Verzeichnissen) gelöscht sind. Die Zählung berücksichtigt dabei keine der später nochmals erläuterten *symbolischen Verweise*.

Während ein Dateiname im Prinzip beliebig lauten darf, verwenden viele Linux-Systeme Konventionen, welche es erlauben, aus der Namensendung auf den Dateiinhalt zu schließen. Hierbei sind z. B. üblich:

.a	für Objektbibliotheken
.bz2	mit bzip2 komprimierte Dateien (auch .bzip2 wird verwendet)
.c	für C-Quelltextdateien
.dvi	DVI-Format (aus T _E X)
.e	für EFL-Quelltextdateien
.f	für FORTRAN-Quelltextdateien (oder <i>.for</i> oder <i>.FOR</i>)
.gz	komprimierte Dateien im GNU-Zip-Format
.h	Header-Dateien – in der Regel für C-Programmmodule
.html	HTML-Dateien
.l	für Listing-Ausgaben von Übersetzern (oder LEX-Quelltextdateien)
.o	für Objektdateien (übersetzte, noch nicht gebundene Module)
.pdf	für (Adobe)-PDF-Dateien (<i>Portable Document Format</i>)
.pl	für Perl-Dateien
.png	Image-Daten (Rasterbilder) im PNG-Format
.ppd	PostScript-Printer-Definition-Dateien
.ps	für PostScript-Dateien
.sh	für Shell-Skript-Dateien
.so	für dynamische Objektbibliotheken
.tar	Archiv-Dateien aus tar. <i>.tbz2</i> sind zusätzlich mit bzip2 komprimiert.
.y	für YACC-C-Quelltextdateien
.Z, .z	mit compress erstellte komprimierte Dateien

Viele Anwendungen legen ihre Dateien mit weiteren, eigenen Namenserweiterungen ab, etwa *datei.sxw* für eine mit dem Writer aus OpenOffice erstellte Textdatei. Erkundigen Sie sich jeweils, welche weiteren Namenskonventionen dieser Art in Ihrem Linux-System gebräuchlich sind.

Im Unterschied zu anderen Betriebssystemen ist diese Namenserweiterung ein normaler Bestandteil des Dateinamens, der nur per Konvention mit einem Punkt vom ersten Teil des Dateinamens abgetrennt ist. Es bestehen weder Einschränkungen für das Trennzeichen (also auch *datei_bak*), für die Anzahl der Verwendung des Trennzeichens (also auch *datei.c.bak*) noch für die Länge der Namenserweiterung (also auch *datei.orig.backup*).

Beginnt ein Dateiname mit einem Punkt, z. B. *.profile*, so wird der Dateiname durch das **ls**-Kommando oder bei der Shell-Expandierung des Metazeichens **^{*}** (am Namensanfang) nicht angezeigt bzw. in der Expansion übernommen. Man hat damit eine Art **verdeckte Datei**. Dies ist häufig praktisch, weil dies per Konvention Dateien sind, die man in den meisten Fällen nicht sehen soll oder nicht sehen will. Durch die Option **ls -a** lassen sich diese Dateien jedoch auch auflisten.

Der Zugriffspfad einer Datei (Path Name)

Die Unix-/Linux-Dateistruktur ist hierarchisch oder baumartig (umgekehrter Baum). Der Ausgangspunkt eines solchen Baums ist die Wurzel (englisch: *root*). Sie wird unter Linux mit `/` angegeben.

Die Wurzel selbst ist eine Verzeichnisdatei. In ihr sind Verweise auf weitere Dateien enthalten. Eine solche Datei kann wiederum ein Verzeichnis sein, das seinerseits auf weitere Dateien verweist; auf diese Weise entsteht die Baumstruktur. Der *Zugriffspfad* (englisch: *path name*) gibt an, wie eine Datei ausgehend von der Wurzel erreicht werden kann. Die vollständige Angabe zur Datei des Kommandos *date* wäre für das Beispiel in Abbildung 3.5 `/usr/bin/date`.

Die Namen der einzelnen Verzweigungsstellen (des dort liegenden Verzeichnisses) werden durch `/` ohne Zwischenraum getrennt.

Wie das Beispiel zeigt, kann ein Dateiname mehrmals im Baum vorkommen (z. B. `test_a`); die Dateien müssen dann jedoch in unterschiedlichen Zweigen des Baumes liegen, d. h. der vollständige Name der Datei inklusive Zugriffspfad muss innerhalb eines Dateibaums eindeutig sein. Die Angabe des Zugriffspfades darf überall dort stehen, wo auch Dateinamen vorkommen können.

Beim **login** bekommt der Benutzer ein Verzeichnis als *aktuelles Verzeichnis* zugewiesen. Dieses steht in der Passwortdatei des Systems als login-Verzeichnis für den Benutzer. Der Name (Zugriffspfad) des aktuellen Verzeichnisses kann mit dem **pwd**-Kommando (*print working directory*) erfragt werden. Spezifiziert der Benutzer einen Dateibezeichner, der nicht mit `/` beginnt, so wird der Pfad zu diesem aktuellen Verzeichnis automatisch vom System vor den Dateibezeichner gesetzt. Ist eine Datei weiter oben in der Baumstruktur oder in einem Seitenast gemeint, so muss der vollständige Zugriffspfad angegeben werden.

Ist das aktuelle Verzeichnis z. B. `/home/neuling`, so kann die Datei `t1.c` entweder unter `/home/neuling/projekt/t1.c` oder verkürzt mit `projekt/t1.p` angesprochen werden, während `tty0` von hier aus nur unter `/dev/tty0` für das Beispiel erreicht wird.

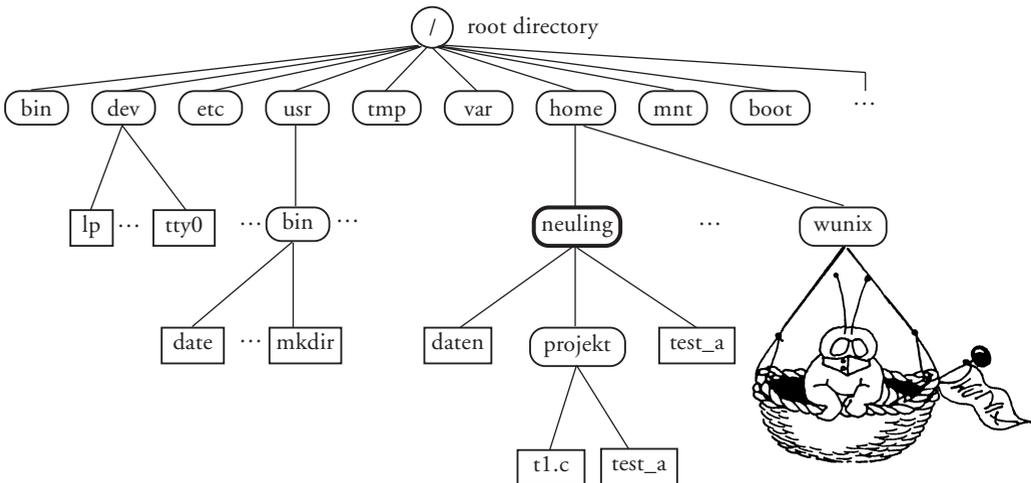


Abb. 3.5: Beispiel für einen Dateibaum

`./` alleine steht für die Wurzel des gesamten Systemdateibaums (englisch: *root directory*). `..` steht für das Väterverzeichnis, `.` für das aktuelle Verzeichnis. Zum Beispiel `./wunix` meint das Verzeichnis oder die Datei `wunix` im übergeordneten Verzeichnis (in Abb. 3.5: `/home`).

Der Benutzer ist in der Lage, mit dem `cd`-Kommando (*change directory*) seine aktuelle Position im Dateibaum zu ändern; er erhält hierdurch jedoch keine neuen Zugriffsrechte! Mit `cd ..` geht er z. B. eine Stufe höher in der Baumhierarchie in Richtung der Baumwurzel – z. B. von `/home/neuling` nach `/home`.

Der Zugriffspfad einer Datei zählt ebenso wie der Dateiname nicht zum festen Bestandteil einer Datei, d. h. er ist nicht im Dateikopf (*Inode*) eingetragen, sondern ist jeweils abhängig davon, wo das Dateisystem (Gerät), auf dem sich die Datei befindet, im Systemdateibaum montiert ist (s. hierzu Abschnitt 3.2.6). Daneben kann, wie bereits beschrieben, eine Datei (in Wirklichkeit ein Verweis auf den Dateikopf) in mehreren Verzeichnissen vorkommen, oder es können mehrere Einträge unter verschiedenen Namen im gleichen Verzeichnis vorhanden sein.

Bei einer Reihe von Kommandos für den *Remote-Zugriff* auf Rechner im Netz (z. B. `r`**cp**, `f`**tp**, `s`**cp**) lässt sich der Pfadname erweitern, indem man dem eigentlichen Pfad den Rechnernamen des Hosts (oder dessen IP-Adresse) voranstellt (syntaktisch separiert durch einen Doppelpunkt) auf den man zugreifen möchte; also z. B. `r`**cp** `sonne:/etc/profile` `./profile`; zum Kopieren der Datei vom Rechner `sonne` in das lokale Verzeichnis.

Wurde der Auto-Mounter installiert, so lässt sich auf *remote*-Dateien (solche auf anderen Rechnern) noch transparenter zugreifen, indem man `/net/host/` dem Pfadnamen voranstellt, wobei `host` hier der Name des Remote-Rechners ist und `net` der Kontrollpunkt für den Auto-Mounter.¹ Dies setzt voraus, dass der genannte Host-Rechner die entsprechenden Verzeichnisse freigegeben/publiziert hat.

Die Dateilänge

Da Dateien auf Speichermedien mit wahlfreiem Zugriff in festen Einheiten (zumeist 0,5–4 KByte) abgelegt werden,² ergeben sich für die Länge einer Datei zwei Größen:

- ▶ die Länge in Bytes und
- ▶ die Länge in Blöcken (z. B. 512 Byte)

Beide Größen lassen sich mit dem Kommando `ls -ls dateiname` anzeigen. Die Länge einer Datei ist zum einen durch das logische Speichermedium begrenzt (d. h. die Datei muss komplett auf einen logischen Dateiträger wie z. B. eine Magnetplatten-Partition bzw. einen später erklärten *Logical Volume* passen) und zum anderen durch bestimmte Systemfaktoren. Die Obergrenze liegt hierbei für Magnetplatten, abhängig vom verwendeten Dateisystemtyp zwischen etwa 2 und 128 Gigabyte pro Datei. Zwei Gigabyte pro Datei waren dabei auch eine gewisse vom Betriebssystem vorgegebene Grenze. Sie ergibt sich dadurch, dass in der Vergangenheit als Datei-Offset beim Lesen und Schreiben in eine Datei ein vorzeichenbehafteter 32-Bit-Wert benutzt wurde und damit die

1. Im Auto-Mounter muss `net` als Kontrollpunkt explizit konfiguriert sein.

2. Die auf den Magnetplatten angelegten Dateisysteme selbst können Blockungsgrößen verwenden, die ein Vielfaches dieser Grund-Blockgröße ausmachen.

maximale Größe auf $2^{31} = 2$ Gigabyte beschränkt war. Aktuelle Unix- und Linux-Systeme bieten daneben Zugriffsfunktionen, welche 64-Bit und damit eine (theoretische) Länge von 2^{63} zulassen – so das Programm diese *Large-File-System-Operations* (kurz: LFS) unterstützt bzw. gegen die entsprechende neue Bibliothek gebunden ist. Für die allermeisten Zwecke stellen jedoch auch 2 Gigabyte pro Datei keine ernsthafte Einschränkung dar – eventuell mit Ausnahme großer Datenbanken oder bei der Bearbeitung von Videodaten. Die Grenze zunehmend abgebaut.

Eine weitere potentielle Beschränkung ergibt sich aus der Größe des Dateisystems. Diese Grenzen sind in Tabelle 3.2 auf Seite 147 aufgeführt. Auch hier sind bei aktuellen Linux-Systemen und bei der Wahl eines geeigneten Dateisystems die Grenzen eher durch die verfügbaren Magnetplatten und deren bezahlbare Kapazität gegeben, denn durch die Implementierung des Systems.

Der Einsatz des **Linux Logical-Volume Managers** (kurz: LVM) gestattet es, Dateisysteme (und damit bedingt auch einzelne Dateien) über mehrere physikalische Platten bzw. Partitionen hinweg anzulegen (zum LVM siehe Seite 143).

Große Dateien – Large File Support (LFS)

Der größte Teil der heutigen Linux-Systeme sind noch 32-Bit-Linux-Systeme auf Intel-PCs. Als Positionszeiger (*Byte-Offset*) für den Dateizugriff verwendete Linux (und zahlreiche ähnliche Betriebssysteme) in der Vergangenheit auf diesen Systemen einen 32-Bit langen Zeiger – mit Vorzeichen. Daraus ergibt sich eine maximale Dateilänge von 2^{31} Byte bzw. 2 GByte. Mit steigender Plattengröße und zunehmendem Speicherbedarf reicht dies in einer Reihe von Fällen nicht mehr aus – insbesondere bei größeren Datenbanken. Bei 64-Bit-Systemen benutzt man hier deshalb gleich einen 64-Bit-langen Offset mit einer maximal möglichen Länge von ca. 8 Exabyte (2^{61} Byte), ein Wert, der noch sehr viele Jahre ausreichend groß sein dürfte. Um aber auch in den viel verwendeten 32-Bit-Systemen große Dateien nutzen zu können – man denke hier nur an einen einstündigen Videofilm – wurden zusätzliche Routinen für Dateioperationen mit großen Dateien geschaffen (dies gilt z.B. auch für Sun Solaris). Dazu mussten nicht nur entsprechende APIs im Linux-Kernel geschaffen werden, sondern auch die Dateisysteme auf 64-Bit-Größen erweitert werden. Dies trifft inzwischen für alle neueren Unix-/Linux-Dateisysteme zu. Diese APIs und ihre Implementierung im Dateisystem wird als *Large-File-Support* kurz (*LFS*) bezeichnet. Er ist im Linux-Kernel seit Version 2.4 standardmäßig vorhanden. Man kann aber davon ausgehen, dass der Dateizugriff – wie bereits in 64-Bit-Linux-Systemen – mittelfristig vollständig auf 64-Bit umgestellt wird, so dass in den Anwendungen nicht mehr spezielle 64-Bit-Routinen vorgesehen werden müssen. Dazu müssen die Programme angepasst, neu kompiliert und gegen 64-Bit-Bibliotheken gebunden werden. Dies wird für den Anwender auf weitgehend transparente Art erfolgen.

Wegen anderer, im Linux-Kernel noch vorhandener Restriktionen ist realistisch die maximale Größe von einzelnen Dateisystemen und damit auch von einzelnen Dateien unter Linux bisher auf ca. 2 Terabyte begrenzt – selbst dann, wenn LFS-APIs und LFS-fähige Dateisysteme verwendet werden.

Wie die Tabelle 3.2 auf Seite 147 zeigt, bieten mit Ausnahme von minix (und bedingt iso9660) bereits alle aktuellen Linux-Dateisysteme heute eine LFS-Unterstützung.

Zugriffsrechte auf eine Datei – der Datei-Modus

Der Benutzer ist in der Lage, die Zugriffsrechte auf seine Dateien (oder Verzeichnisse) festzulegen und zu ändern. Als Zugriffsmöglichkeit wird dabei unterschieden:

- ▶ Lesen (**r** = *read*)
- ▶ Schreiben (**w** = *write*)
- ▶ Ausführen (**x** = *execute*)

Das Recht *Ausführen* (**x**) bedeutet bei normalen Dateien (Dateien, welche ausführbaren Code oder Kommandoprozeduren enthalten), dass es der jeweiligen Gruppe erlaubt ist, dieses Programm zu starten. Bei Dateiverzeichnissen zeigt das **x**-Recht hingegen an, dass ein Zugriff in das Verzeichnis und die tiefer liegenden Dateien möglich ist. Eine *Benutzergruppe* sind alle Benutzer mit der gleichen Gruppennummer.¹

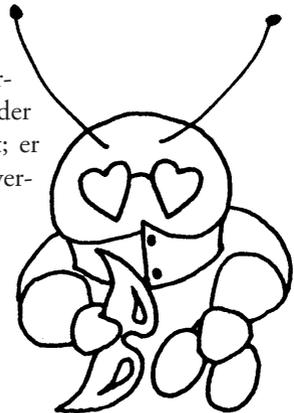
Die Zugriffsrechte können für jede Datei einzeln und für die drei Benutzerklassen getrennt festgelegt werden:

- ▶ den Besitzer (**u** = *login user*)
- ▶ die Benutzer der gleichen Gruppe (**g** = *group*)
- ▶ alle anderen Benutzer (**o** = *other users*)

Die Zugriffsrechte von Dateien werden mit Hilfe des **ls**-Kommandos in der Form **ls -ls dateien** angezeigt und mit dem *Change-Mode*-Kommando (**chmod**) geändert. Die Zugriffsrechte werden im Dateikopf (*Inode*) festgehalten und existieren deshalb auch dann nur einmal und für alle gleich, wenn mehrere Verweise (englisch: *links*) auf eine Datei vorhanden sind.

Hat man das Schreibrecht auf eine Datei, so darf man diese Datei zwar verändern, sie kann jedoch nur dann gelöscht werden, wenn man auch das Schreibrecht für das übergeordnete Verzeichnis besitzt, da zum Löschen der Datei Änderungen in diesem Verzeichnis vorgenommen werden müssen. Hingegen kann man mit dem Schreibrecht auf das Verzeichnis die Datei selbst dann löschen, wenn man keine Schreibrechte auf die darin liegende Datei besitzt, aber zumindest deren Besitzer ist. In diesem Fall fragt das System zurück, ob die Datei wirklich gelöscht werden soll.

Diese teilweise problematische Implementierung lässt sich umgehen. Setzt man für ein Verzeichnis das Attribut **xt** (z.B. mittels **chmod a+xt verzeichnisname**), so darf nur derjenige die in dem Verzeichnis liegenden Dateien löschen oder ändern, der das Schreibzugriffsrecht auf die Datei besitzt; er muss dazu nicht über das Schreibrecht auf das Verzeichnis verfügen.



1. Damit die Wirkung der Gruppennummer aber zum Tragen kommt, müssen alle Benutzer einer Gruppe in der Datei */etc/group* eingetragen sein!

Das Dateidatum

Zu einer Datei gehören drei charakteristische Datumsangaben:

- ▶ das Datum der Erstellung
- ▶ das Datum der letzten Änderung
- ▶ das Datum des letzten Zugriffs

Daneben gibt es unter Linux ein weiteres Änderungsdatum:

- ▶ das Datum der letzten Änderung im Dateikopf

Eine solche Änderung im Dateikopf (dem *Inode*) erfolgt z. B. wenn man nicht auf die Datei selbst zugreift, aber deren Attribute (z. B. durch **chmod**) ändert.

Da zumindest das Datum des letzten Zugriffs auf die Datei auch beim Lesen korrigiert wird, dürfen Dateiträger nicht ohne weiteres schreibgeschützt sein, soweit sie in den Dateibaum eingehängt (englisch: *mounted*) werden. Soll wirklich nur gelesen werden und ist das Datum des letzten Zugriffs nicht wichtig, so kann beim **mount** (s. Abschnitt 3.2.6) die Option **r** (*read only*) diese Datumskorrektur unterbinden. Dies ist beispielsweise bei Dateisystemen auf CD-Platten erforderlich.

Das Datum, welches das Kommando **ls -l...** ausgibt, ist das der letzten Dateiänderung. Mit der Option **ls -lu ...** erhält man das Datum des letzten Dateizugriffs, mittels **ls -lc ...** jenes der letzten Änderung im Dateikopf. Der Systemaufruf **stat** liefert alle drei (ersten) Zeiten zusammen mit weiterer Information über die Datei zurück.

Der Dateibesitzer

Der Dateibesitzer (englisch: *owner*) ist derjenige Benutzer, der die Datei erzeugt hat. Wem dabei das Verzeichnis gehört, in den die Datei eingetragen wird, ist gleichgültig, solange der Erzeuger Schreiberlaubnis für dieses Verzeichnis besitzt. Das Attribut *Dateibesitzer* kann vom Super-User durch das **chown**-Kommando geändert werden.

Die Eigenschaft, Besitzer einer Datei zu sein, ist für die Überprüfung der Zugriffsrechte auf die Datei entscheidend. Das Verändern der Zugriffsrechte ist dabei dem Besitzer sowie dem Super-User vorbehalten. Intern werden nicht Benutzer- und Gruppenname des Dateibesitzers, sondern die entsprechenden Nummern abgespeichert. Die **ls**-Option **-g** erlaubt die Ausgabe des Gruppennamens statt des Benutzernamens. Bei Linux werden bei **ls -l ...** Benutzer- und Gruppenname des Dateibesitzers ausgegeben. Steht statt des Namens eine Nummer, so zeigt dies an, dass unter der Nummer kein Eintrag in der Benutzer- oder Gruppenpasswortdatei vorhanden ist.

Knotennummer einer Datei (Inode Number)

Jedes Dateisystem (dies ist die dateiorientierte Struktur auf einem logischen Datenträger mit wahlfreiem Zugriff) besitzt ein (Dateisystem-)Inhaltsverzeichnis. In ihm sind alle Dateien verzeichnet, die in dem Dateisystem existieren. Dieses Inhaltsverzeichnis wird *Inode List* genannt. Die Elemente der *Inode-Liste* sind die so genannten *Inodes*. Ein solcher *Inode* stellt den *Dateikopf* dar. In ihm sind alle Attribute (Metadaten) enthalten, die einer Datei fest zugeordnet sind (s. Abb. 3.6).

Dateiverzeichnisse (*Directories*) von Speichermedien mit wahlfreiem Zugriff enthalten lediglich den Dateinamen und die *Knotennummer* (englisch: *inode number*) einer Datei. Die Nummer ist ein Index in die *Dateikopfliste* (englisch: *inode list*).

Die nachfolgende Beschreibung bezieht sich auf das Linux-Dateisystem vom Typ `ext2`. Es ist eines des bisher meisteingesetzten Dateisysteme unter Linux und vom Aufbau her sehr stark an das UNIX System V (auch als `us5` bezeichnet) angelehnt. Es ist relativ einfach aufgebaut. Die weiteren von Linux unterstützten Dateisysteme weichen in einigen Details davon ab. Das System erlaubt aber stellvertretend für alle Dateisysteme die Grundmechanismen zu beschreiben.¹

Beim Anlegen einer Datei wird ein neuer *Inode* (Dateikopf) beschafft und der Dateiname zusammen mit dem Index des *Inodes* im entsprechenden Dateiverzeichnis eingetragen.

Neben den Attributen sieht der Dateikopf von `ext2` Platz für 15 Verweise auf den Dateiinhalt vor. Die ersten 12 Verweise zeigen direkt auf die ersten 12 Blöcke der Datei (siehe auch Abb. 3.7). Sind weniger vorhanden, so sind sie entsprechend leer. Ist die Datei länger als 12 Blöcke, so verweist der 13. Zeiger auf einen Block, in welchem weitere Verweise zu finden sind (erste Indirektionsstufe). Die Anzahl der hier möglichen Verweise ist abhängig von der verwendeten Blockgröße im Dateisystem. Reichen die ersten 12 und die hier stehenden Blockverweise nicht aus, um alle Blöcke der Datei zu adressieren, so ist im 14. Eintrag des Inodes ein Verweis auf einen Block zu finden, der Verweise auf indirekte Verweise zeigt (zweite Indirektionsstufe). Reicht auch dies nicht, so enthält der 15. Eintrag einen Verweis auf Verweisblöcke, welche selbst wiederum auf Verweisblöcke zeigen (dritte Indirektionsstufe). Hiermit erreicht man schließlich – bei entsprechender Blockgröße (typisch 1–4 KB) – auch sehr große Dateien. Der Zugriff auf die ersten 12 Blöcke ist dabei am schnellsten (bei der Eröffnung einer Datei wird der

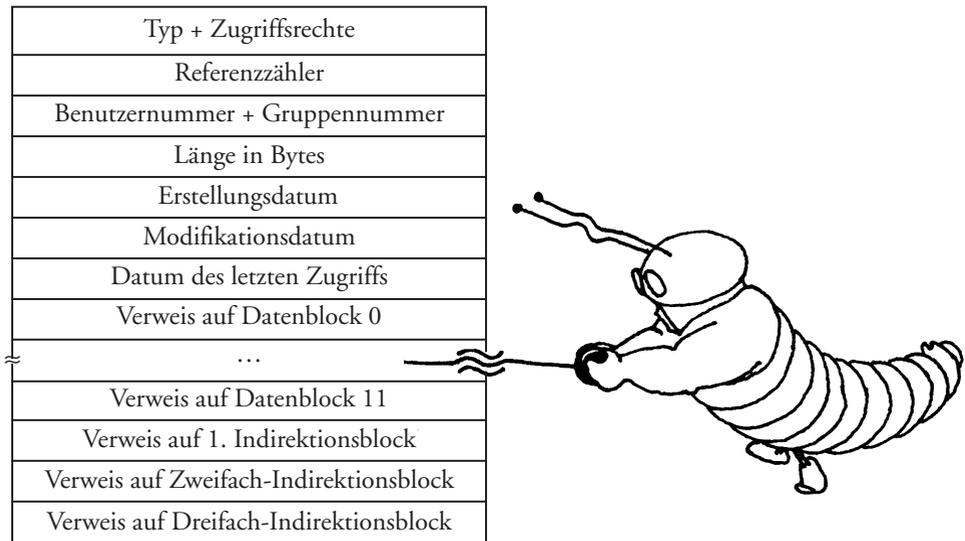


Abb. 3.6: Aufbau eines Dateikopfes (*Inode*) in einem Dateisystem vom Typ `ext2`

1. Eine ausführliche Beschreibung des `ext2`-Dateisystems ist in [EXT2_a] zu finden.

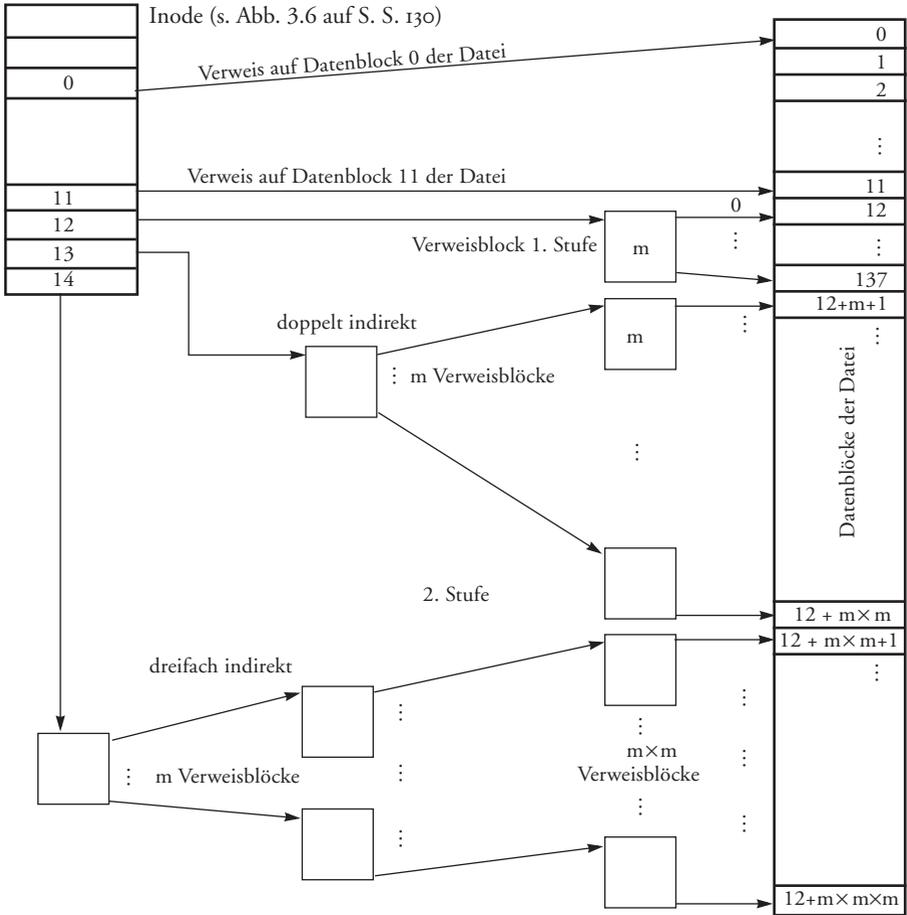


Abb. 3.7: Verweisstruktur einer großen Datei im ext2-Dateisystem

Dateieintrag aus der Indexliste in den Speicher kopiert), während alle weiteren Blöcke zusätzliche Zugriffe notwendig machen. Der Nachteil der dreistufigen Indirektion bei sehr großen Dateien mit dem entstehenden Aufwand an Zugriffen auf die Datenblöcke mit der Verweisstruktur wird teilweise durch die Pufferung von Ein-/Ausgabeblocken durch das System (eine Art Cache für Datenblöcke und ein getrennter Puffer für Inode-Blöcke) aufgefangen.

Abweichend von den eben beschriebenen Indexlisteneinträgen haben bei den *special files* die letzten 15 Verweise keine Bedeutung. Das erste Datum hingegen enthält eine Gerätenummer des Treibers, welcher das physikalische Gerät, mit dem gearbeitet wird, bedient. Diese Gerätenummer besitzt zwei Teile:

- ▶ die Nummer des Gerätetyps (*major device number*)
- ▶ die Nummer des konkreten Gerätes (z.B. Laufwerk) (*minor device number*). In dieser Nummer ist unter Umständen noch Zusatzinformation (z.B. dass das Band beim Schließen der Datei nicht automatisch zurückspulen soll) codiert.

Verweise – Hard- und Symbolic-Links

Linux gestattet Verweise, so genannte *Links*, auf eine Datei anzulegen. Dies geschieht z. B. mit dem **ln**-Kommando. Durch einen solchen Verweis kann eine Datei entweder unter verschiedenen Namen oder von unterschiedlichen Stellen im Dateibaum angesprochen werden. Dabei muss man zwischen so genannten *harten* Verweisen (englisch: *hard links*) und *symbolischen* Verweisen (englisch: *symbolic links*) unterscheiden.

Beim *Hard Link* (und dies ist der Standard beim **ln**-Kommando) wird lediglich im Verzeichnis ein Eintrag mit dem Verweis auf die referenzierte Inode-Nummer angelegt. Ein solcher Verweis kostet damit sehr wenig Platz. Da Inode-Nummern jedoch nur innerhalb eines logischen Dateisystems eindeutig sind,¹ muss bei dieser Art der Verweis (korrekter das Verzeichnis mit dem Verweis) und die referenzierte Datei auf dem gleichen logischen Dateisystem liegen.

Bei *Symbolic Link* oder *symbolischer Verweis* – teilweise auch als *Soft Link* bezeichnet – wird eine Datei angelegt, in welcher der Name der referenzierten Datei steht. Hierbei wird der vollständige Pfadname abgelegt, weshalb bei *Symbolic Links* dieser beim **ln**-Kommando auch vollständig anzugeben ist. Diese Datei erhält jedoch den speziellen Dateityp *Symbolic Link*. Dies gestattet auch Verweise über Dateisystemgrenzen und sogar Rechnergrenzen hinweg. Verschiebt man die referenzierte Datei eines *Symbolic Links* oder benennt sie um, so findet das System die referenzierte Datei nicht mehr! Dieses Problem tritt bei *Hard Links* bei Verschiebung oder Umbenennung innerhalb des Dateisystems nicht auf.

3.2.3 Struktur eines Dateisystems

Ein physikalischer Datenträger wie z. B. eine Magnetplatte kann mehrere *Partitionen* enthalten.² Eine Partition wiederum wird durch ein Dateisystem für eine Dateisystemorientierte Dateiablage zugänglich gemacht. Der eigentliche Zugriff auf den Datenträger wie eine Magnetplatte erfolgt über einen so genannten *Treiber*. Ein Treiber ist der Modul im Betriebssystem, der für den Transfer von Daten zwischen einem Gerät und dem Hauptspeicher zuständig ist. In der Regel gibt es für jede Geräteart einen eigenen Treiber.

Bei Magnetplatten und ähnlichen blockorientierten Geräten erfolgt der Transfer von Daten immer in ganzen Blöcken. Dabei sind nochmals physikalische Blöcke und logische Blöcke zu unterscheiden. Die Größe der physikalischen Blöcke ist vom eingesetzten Medium oder Laufwerk abhängig. IDE-/ATA-Magnetplatten besitzen in aller Regel eine physikalische Blockgröße von 512 Byte. Bei SCSI-Platten kann die Blockgröße teilweise bei der Low-Level-Formatierung zwischen 512 Byte und 8 KByte festgelegt werden.

Oberhalb der *physikalischen Blöcke* legt das Betriebs- und Dateisystem die logischen Blöcke. Ein *logischer Block* ist dabei eine (feste) Folge von physikalischen Blöcken, die (z. B. vom Dateisystem) immer als Ganzes gelesen oder geschrieben werden – also eine

1. Die gleiche Inode-Nummer kann in anderen Dateisystemen nochmals vorkommen.

2. Die Partitionierung einer Platte erfolgt über **fdisk** oder entsprechende grafische Werkzeuge, wie sie z. B. bei United-Linux-Systemen unter YAST2 zu finden sind.

Schicht über den physikalischen Blöcken. Beim Anlegen eines Dateisystems kann man entweder diese logische Blockgröße explizit festlegen oder das Programm, welches das Dateisystem anlegt, berechnet selbst eine geeignete Blockgröße. Typische logische Blockgrößen liegen bei den Linux-Dateisystemen zwischen 1 KB und 4 KB. Eine größere Blockgröße resultiert in schnelleren Übertragungen, führt aber auch zu einem größeren mittleren Speicherverschnitt durch nicht vollständig belegte Datenblöcke.

Dateisystem

Mit *Dateisystem* ist hier ein *logisches Dateisystem* gemeint. Dateisysteme auf Datenträgern mit wahlfreiem Zugriff wie z. B. Magnetplatten und Disketten haben eine einheitliche Struktur. Diese ist von Dateisystemtyp zu Dateisystemtyp unterschiedlich. Das ext2-Dateisystem z. B. kennt (vereinfacht) vier Bereiche (s. Abb. 3.8):

- ▶ Block 0 (*boot block*)
- ▶ Superblock
- ▶ Liste der Dateiköpfe (*Inode List*)
- ▶ Bereich der Datenblöcke

Der erste Block des Dateisystems (die Zählung beginnt bei 0) ist als *boot block* reserviert (historisch bedingt). In ihm kann ein kleines Programm liegen, welches beim Hochfahren des Systems das eigentliche Linux-System in den Hauptspeicher lädt und startet.

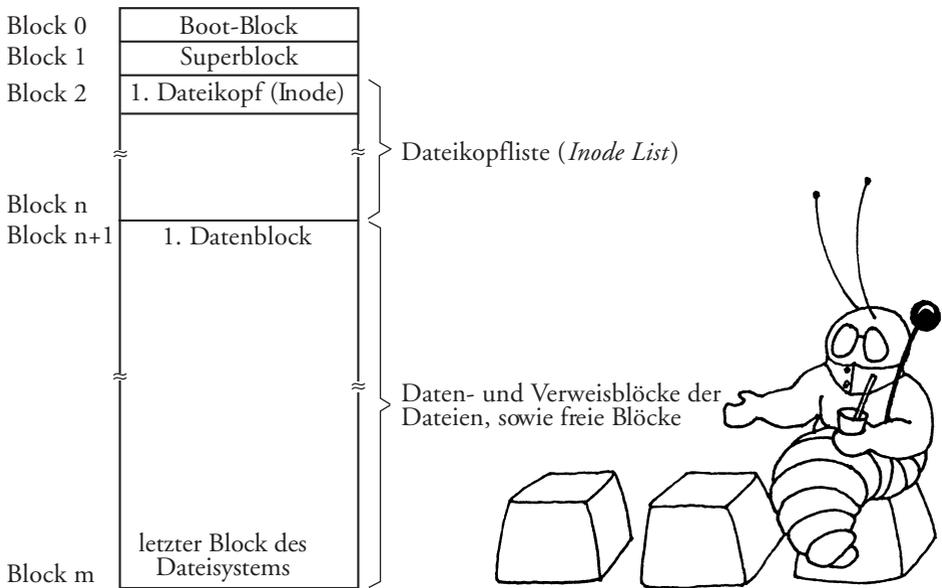


Abb. 3.8: Struktur des ext2-Dateisystems (vereinfacht)

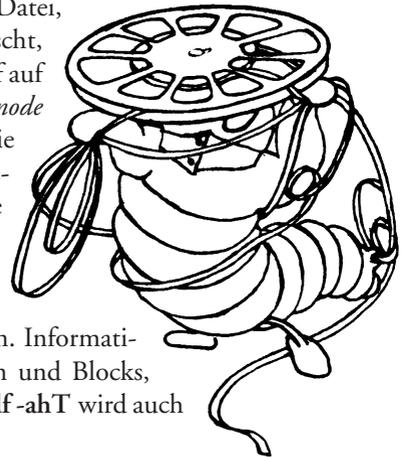
Der erste Block des eigentlichen Dateisystems ist der Block 1. Er wird als **Super Block** bezeichnet. In ihm steht Verwaltungsinformation zum Dateisystem. Hierzu gehören z. B.:

- ▶ Größe des Dateisystems in Blöcken
- ▶ Name des Dateisystems (wird mit `/etc/labelit` angelegt)
- ▶ Zeiger auf das erste Element der Liste der freien Datenblöcke
- ▶ Zeiger auf das erste Element der Liste der freien *Inodes*
- ▶ Datum der letzten Modifikation
- ▶ Indikatoren zum Blockieren des Datenzugriffs bei Korrekturoperationen
- ▶ Kennzeichnung, welche Blockgröße im Dateisystem verwendet wird. Diese kann bei `ext2` 1, 2 oder 4 KB sein. Andere Dateisysteme erlauben auch große Blockgrößen.

Ist ein Dateisystem montiert, so wird dieser *Super Block* ständig im Hauptspeicher gehalten. Der *Super Block* wird dabei jeweils automatisch nach festgelegten Zeiten oder manuell durch das Kommando `sync` auf die Festplatte zurückgeschrieben. Damit können die Zugriffe auf das Dateisystem sehr schnell abgewickelt werden. Nachteil dabei ist jedoch, dass Super Block, wie er im Hauptspeicher steht und den tatsächlichen Zustand des Dateisystems beschreibt, nicht mehr zu jedem Zeitpunkt mit dem Super Block auf der Festplatte übereinstimmt. Kommt es zu einem Stromausfall oder wird das System ohne vorherige Maßnahmen (z. B. `shutdown`) einfach ausgeschaltet, so können aus diesen Inkonsistenzen Datenverluste resultieren. Neuere Entwicklungen bei Dateisystemen (z. B. `ext3`, `reiserfs`, `xfs`) können diese Probleme vermeiden, indem sie permanent zusätzliche Notizen über alle Änderungen führen – man nennt solche Dateisysteme *Journaling Filesystems*, da sie ständig über die Änderungen ein Journal führen.

Da der Superblock für das Dateisystem sehr wichtige Funktionen hat, wird er beim `ext2`-Dateisystem (entgegen der zuvor gegebenen vereinfachten Darstellung) mehrfach gespeichert (jeweils am Anfang eines so genannten *Blockbereichs*). Wird ein Superblock unlesbar, greift der Dateisystemtreiber auf die alternativen Superblöcke zu,

Es existiert genau ein *Inode* für jede Datei (normale Datei, Verzeichnis oder Geräteeintrag). Wird eine Datei gelöscht, d. h. geht die Anzahl von Verweisen auf einen Dateikopf auf 0, so wird auch der Inhalt des *Inodes* gelöscht und der *Inode* in die Liste der freien Dateiköpfe eingefügt. `ext2` legt die Dateikopfliste statisch bei der Erzeugung des Dateisystems an. Dies hat eine gewisse Effizienz. Mit der Länge der Liste der Dateiköpfe ist aber auch die maximale Anzahl von Dateien für das jeweilige Dateisystem beschränkt. Die Größe der Liste wird bei der Initialisierung eines `ext2`-Dateisystems (durch `mk2efs`) angegeben. Informationen über das Dateisystem, belegte und freie Dateien und Blocks, können mit dem Kommando `df` abgefragt werden. Bei `df-ahT` wird auch der Dateisystemtyp mit angezeigt.



Der verbleibende Platz des Systems steht den Datenblöcken der Dateien zur Verfügung. Die Verweislisten der ersten, zweiten und dritten Indirektionsstufe großer Dateien zählen auch hierzu. Die Verwaltung erfolgt über eine Freiblockliste, deren Anfang im Superblock vermerkt ist.

Andere Dateisysteme (etwa `reiserfs`, `JFS` und `xfs`) legen ihre *Inodes* dynamisch an und bieten damit eine höhere Flexibilität hinsichtlich der Speichernutzung und der möglichen Anzahl von Dateien im Dateisystem.

3.2.4 Linux-Dateisysteme

Um Rückwärtskompatibilität zu älteren Dateisystemen zu bieten, unterstützen heutige Linux-Systeme eine ganze Anzahl unterschiedlicher Dateisysteme. Um für die Anwendungen eine weitgehende Transparenz der Unterschiede der Dateisysteme zu erreichen, besitzt Linux den *Virtual File System Switch* – eine Zwischenschicht zwischen Anwendungen und dem Dateisystem, welche die allgemeinen Aufrufe an das Dateisystem in den jeweils spezifischen Aufruf für das angesprochene Dateisystem umsetzt.

Die unterschiedlichen Dateisysteme besitzen spezifische Vor- und Nachteile und typische Einsatzgebiete. Sie unterscheiden sich durch ihre Struktur, die darin zulässigen Längen von Dateinamen, die von ihnen unterstützten Blockgrößen, durch unterschiedliche Geschwindigkeiten und Sicherheitsniveaus und weitere Leistungsmerkmale. Die wichtigsten lokalen Dateisysteme sind hierbei:

- ▶ Standard-Linux-Dateisystem (Kurzform: **ext2**)
- ▶ eine Journaling-Funktion auf das **ext2** aufgesetzt (Kurzform: **ext3**)
- ▶ das Journaling-Dateisystem von Hans Reiser (Kurzform: **reiserfs**)
- ▶ das von IBM-AIX stammende und auf Linux portierte *Journaling File System JFS*
- ▶ das von SGI stammende und auf Linux portierte Dateisystem mit Journaling-Funktionen (Kurzform: **xfs**)
- ▶ Prozessdateisystem
- ▶ CD-Dateisystem (ISO 9660 und andere)

Zusätzlich unterstützen die meisten Linux-Implementierungen weitere Netzwerkdateisysteme wie etwa NFS und SMB/CIFS (das Windows-Netzwerkdateisystem) sowie Dateisysteme anderer Betriebssysteme wie etwa das MS-DOS oder von Mac OS.

Beim Einsatz unterschiedlicher Dateisysteme oder gar von Nicht-Linux-Dateisystemen sind natürlich die speziellen Charakteristika der Systeme und deren Restriktionen zu beachten, wie etwa maximale Namenslängen, maximal zulässige Größen für einzelne Dateien und des ganzen Dateisystems, sowie deren Blockungs- und Pufferungsmechanismen. Bei einigen fremden Dateisystemen ist aktuell auch nur ein Lesezugriff möglich, nicht jedoch das Schreiben. Dies gilt z. B. noch (bei Linux 2.4x) auf Partitionen unter dem Windows NTFS-System (NTS5).

Journaling Dateisysteme

Unter Linux werden, wie bei anderen Betriebssystemen auch, bei Operationen auf dem Dateisystem die Daten nicht sofort auf den Datenträger hinaus geschrieben, sondern aus Performance-Gründen zunächst im Hauptspeicher gepuffert. So können Daten, die gerade erzeugt wurden, sehr schnell wieder aus dem Hauptspeicher gelesen werden. Auch die Metadaten – Inodes, Superblock, Freiblocklisten und Ähnliches – werden so gepuffert. Eine neu angelegte Datei existiert so zunächst (zumindest in Teilen) nur im Hauptspeicher. Beim Schreiben auf die Platte werden die Schreibaufträge zusätzlich in eine Reihenfolge gebracht, welche ein schnelles, zusammenhängendes Schreiben benachbarter Blöcke oder nahe beieinander liegender Blöcke erlaubt. Bei einem Stromausfall oder Systemabsturz kann es dabei zu wesentlichen Inkonsistenzen des Dateisystems kommen, etwa weil mit Daten belegte Blöcke noch nicht in der Bitblock-

Liste auf der Platte vermerkt sind oder weil ein Inode noch auf alte Datenblöcke verweist. Beim Einhängen (**mount**) eines Dateisystems überprüft Linux deshalb zunächst, ob das Dateisystem regulär ausgehängt wurde (per **umount**). Ist dies nicht der Fall, überprüft Linux über die Filesystem-Check-Funktion (**fsck**), welche Inkonsistenzen vorhanden sind. Es versucht dann (optional) diese zu beheben. Die Prüfung und Behebung kann bei großen Dateisystemen erheblich Zeit kosten – 30 Minuten und mehr sind hier durchaus realistische Zeiten. Bei Server-Systemen, die nach einem Ausfall wieder schnell verfügbar sein müssen, ist dies oft ausgesprochen problematisch.

Hier setzen so genannte *Journaling Filesystems* an – Dateisysteme, die über alle Änderungen ab dem letzten konsistenten Stand ein Journal führen.¹ Beim Start durchlaufen sie das Journal und führen alle dort aufgeführten (verändernde) Dateioperationen nun aus. Am Ende der Abarbeitung befindet sich das Dateisystem wieder in einem konsistenten Zustand. Diese Abarbeitung kann, wenn das Journal nicht zu lang ist, sehr schnell erfolgen – in aller Regel innerhalb weniger Sekunden. Ein spezieller Prozess (und das Dateisystem selbst) sorgt dafür, dass das Journal eine bestimmte Größe nicht überschreitet. In diesem Fall werden die Daten eben in der korrekten Reihenfolge auf die Platte geschrieben, gelöscht und ein neues Journal aufgesetzt.

Die Journaldaten selbst werden zunächst auch im Hauptspeicher gehalten und in kürzeren Intervallen als die restlichen Daten auf die Platte geschrieben. Das Journal muss dabei nicht einmal auf dem gleichen Speicher wie das Dateisystem liegen. Es kann z.B. auf einer schnellen, batteriegepufferten RAM-Disk gespeichert sein. Nicht alle Dateisysteme erlauben ein solches *ausgelagertes Journal*.

Ein Journaling Filesystem gewährleistet nicht, dass keine Daten bei einem Crash verloren gehen. Jene im Hauptspeicher ohne Plattenkopie sind auch hier weg. Es versucht jedoch zu gewährleisten, dass eine Dateioperation (z.B. das Löschen einer Datei) vollständig oder gar nicht ausgeführt ist und damit ein konsistenter Stand erreicht wird. Der Fokus der Systeme mit Journaling liegt bisher auf der Konsistenz der Metadaten (Inodes, Belegt-Block-Listen, ...). Einige Systeme können neben den Metadaten auch die eigentlichen Datenblöcke mit in die Journal-Funktion einbeziehen. Dies gilt z.B. für das **ext3**-System, wenn es per **mount**-Option **data=journal** eingehängt wird. Dies reduziert jedoch etwas die Performance des Systems. Ein Datenverlust beim Crash ist damit immer noch nicht vollständig ausgeschlossen.

Linux bietet eine Reihe unterschiedlicher Journaling Dateisysteme, welche durchaus unterschiedliche Funktionen haben. **ext3** ist eine um Journaling-Funktionen erweiterte Version des Standard-Dateisystems **ext2**. Der Vorteil liegt bei **ext3** darin, dass es – unter temporärem Verlust des Journalings – auch wieder als **ext2** eingehängt werden kann.

Das ReiserFS ist ein seit längerem verfügbares und recht stabiles Dateisystem mit Journaling, während das von IBM stammende Dateisystem JFS noch relativ neu ist und (Stand Ende 2002) weder seine Performance-Ziele noch seine Zielstabilität richtig erreicht hat. Ausgesprochen auf Skalierbarkeit ausgelegt und ebenfalls recht stabil ist das von SGI stammende **xf**s.

1. Ein guter Überblick zu den Linux Journaling Files System wird in [Journal] gegeben.

Dateifragmente und kurze Dateien

Große Blockgrößen erhöhen die Performance von Dateisystemen, insbesondere beim Lesen großer Dateien, da weniger separate Plattenzugriffe notwendig sind. Zugleich steigt damit aber der Plattenplatzverschchnitt, da man davon ausgehen kann, dass im Mittel jeweils ein halber Block jeder Datei unbesetzt ist. Bei vielen kleinen Dateien kann dies einen erheblichen Prozentsatz des gesamten Speicherbedarfs ausmachen. Einige Dateisysteme (z. B. `reiserfs`) bieten deshalb die Möglichkeit, mehrere kleine Fragmente in einem (von entsprechend mehreren Dateien geteilten) Block abzulegen. Man spricht hier auch von *Dateifragmenten*. Dies reduziert den Speicherverschnitt, erhöht aber etwas den Zugriffs- und Organisationsaufwand für das Dateisystem. Es kann deshalb vorteilhafter sein, kleine Dateien – und gerade das Linux-Basissystem mit seinen zahlreichen kleinen Skripten und Initialisierungsdateien hat recht viele davon – auf ein eigenes Dateisystem mit kleiner Blockgröße zu legen (so die Blockgröße beim Dateisystem gewählt werden kann) und Datenbanken und große Dateien auf ein separates Dateisystem mit großer Blockgröße.

Das ext2-Dateisystem

Das traditionelle Dateisystem von Linux wird als `ext2`-System (*extended file system version 2*) oder als `ext2fs` bezeichnet. Es gestattet eine Blockgröße von 1 KByte bis 4 KByte; auf den 64-Bit-Systemen wie HP/DEC Alpha sind es 8 KByte. Dateinamen können bis zu 255 Zeichen lang sein. Mit `ext2ed` steht zusätzlich eine Version zur Verfügung, welche die gespeicherten Daten komprimiert. Die Struktur der Inodes sowie das Layout des `ext2`-Dateisystems wurde bereits in den Graphiken von Abb. 3.6 und Abb. 3.7 sowie in Kapitel 3.2.3 beschrieben.

Performance-Nachteile ergeben sich bei `ext2` und dem darauf aufsetzenden `ext3` bei sehr großen Verzeichnissen (Directories), (typisch ab etwa 5000 Einträgen), da hier eine lineare, verkettete Liste durchsucht werden muss, während Dateisysteme wie etwa `reiserfs` oder `xfs` die Verzeichniseinträge in so genannten *B-Bäumen* anlegen, in denen die Suche sehr viel schneller erfolgen kann. Da die Inode-Tabelle beim Erstellen des Dateisystems statisch angelegt wird, kann es vorkommen, dass noch Datenblöcke auf dem Dateisystem frei sind, aber keine Inodes mehr zum Anlegen weiterer Dateien vorhanden sind oder dass eine große, nicht benötigte Inode-Tabelle unnötig Platz für Datenblöcke belegt.

Neben den *normalen* (auf Seite 122 aufgeführten) Dateiattributen, kennt `ext2/etx3` eine Reihe weiterer Dateiattribute, welche mit `lsattr`¹ abgefragt und mit `chattr` gesetzt werden können. Dazu gehört z. B. das Attribut, dass beim Löschen einer Datei die Blöcke nicht nur wie sonst üblich freigegeben, sondern zuvor noch mit Nullen überschrieben werden – eine Sicherheit bei vertraulichen Daten.

Für `ext2` stehen zahlreiche Werkzeuge zum Anlegen und Parametrisieren (`mke2fs`), zur Konsistentprüfung (`e2fsck`), Sicherung (`dump`), zum Zurückladen (`restore`), für manuelle Korrekturen und Änderungen (`debugfs`) sowie zum Optimieren (`tune2fs`) zur Verfügung. `ext2` bietet wie die meisten anderen moderneren Linux-Dateisysteme Unterstützung für Dateien mit einer Größe von mehr als 2 GB (*LFS = Large File Support*).

1. Siehe Seite 351 zur Beschreibung von `lsattr` und Seite 226 für `chattr`.

Das ext3-Dateisystem

ext3 ist eine Erweiterung des **ext2**-Dateisystems um Journaling-Funktionen. Der Vorteil von **ext3** besteht darin, dass es auf das sehr stabile **ext2** aufsetzt und dass Migrationen von **ext2** nach **ext3** problemlos per `tune2fs -j /dev/dateisystem` möglich sind. Auch lässt sich nachträglich ein **ext3** immer noch als **ext2**-System per **mount** einhängen. Ansonsten übernimmt **ext3** hierdurch auch praktisch alle Limitation von **ext2** – z. B. das statische Anlegen von Inodes und die verketteten Dateieinträge in den Verzeichnissen/*Directories*. Mit der **mount**-Option `data=journal` führt **ext3** nicht nur ein Journaling der Metadaten, sondern auch der Datei-Datenblöcke selbst durch und erreicht dabei eine höhere Datensicherheit bei Abstürzen – ohne allzu große Performance-Verluste.

Das ReiserFS-Dateisystem

Das ursprünglich von Hans Reiser entwickelte Dateisystem **reiserfs** ist eines der ältesten Journaling-Dateisysteme unter Linux und inzwischen ausgereift und sehr stabil. Dabei ist die Entwicklung bei weitem nicht abgeschlossen. Zahlreiche weitere Funktionen sind einkonzipiert und neue sollen noch entwickelt werden. Durch eine gute Finanzierung erlebt es eine dynamische Weiterentwicklung. Die Stärke von **reiserfs** liegt in der effizienten Handhabung großer Dateien und Dateisysteme und großer Verzeichnisse. Für Verzeichnisse werden B-Bäume (*balanced trees*) eingesetzt, bei denen die Suche sehr viel schneller geht als mit den verlinkten Listen des **ext2/etx3**. **reiserfs** erlaubt zusätzlich eine effiziente Handhabung kleiner Dateien, von denen mehrere in einen Dateisystemblock gepackt werden können. Kleine Dateien und Dateifragmente packt **reiserfs** in den 4-KB-Blöcken zusammen und erreicht damit eine höhere Speichereffizienz – etwas auf Kosten der Performance. Mit der **mount**-Option `notail` lässt sich dies aber unterdrücken.

Während Version 3 von **reiserfs** das Journaling nur für Metadaten durchführt, erlaubt die seit 2003 verfügbare Version 4 auch das Journaling von Datenblöcken. Version 4 bietet zusätzlich Schnittstellen (APIs), um Transaktion auf Dateien auszuführen. Wie für **reiserfs** gibt es zahlreiche Werkzeuge, bisher jedoch kein **dump/restore**.¹

Für detailliertere Information und die aktuelle Version des ReiserFS sei auf [ReiserFS] verwiesen.

Das xfs-Dateisystem

Das ursprünglich unter dem Unix-/IRIX laufende Dateisystem von SGI wurde von SGI auf Linux portiert und freigegeben. Es handelt sich um ein *Journaling*-, 64-Bit-Dateisystem und dürfte aktuell (Stand Anfang 2003) das am besten skalierende und für viele Dateien und große Dateien am performantesten laufende Dateisystem unter Linux sein. *Extended Attributes* erlaubt weitere Metadaten (>Name=Wert-Paare) in den Inodes unterzubringen (bis zu 64 KB pro Inode). Diese können über spezielle Funktionen gesetzt und abgerufen werden. Da es sich bereits originär um ein 64-Bit-Dateisystem han-

1. **dump** führt ein sehr schnelles und vollständiges Sichern des **ext2/ext3**-Dateisystems durch, **restore** das Zurückkladen.

delt, kann eine Datei (theoretisch) 2^{63} Bytes bzw. 8 Exabyte (=8 192 Petabyte) groß sein. Durch Limitationen des Linux-Kernels ist die reale maximale Größe bisher jedoch lediglich 2 TB. Dieses Limit gilt auch für die Größe des gesamten Dateisystems. Dies sollte jedoch für die meisten Installationen reichen.

xfs kommt mit einer ganzen Reihe von Werkzeugen zur Erstellung, Parametrisierung, Vergrößerung und Verkleinerung sowie mit spezifischen **dump**- und **restore**-Programmen.

xfs ist in einigen Linux-Distributionen noch nicht vorhanden und setzt dann (Stand: Anfang 2003) eine ganze Reihe von Kernel-Patches voraus. Mittelfristig dürfte es jedoch im Standard-Source-Tree von Linux landen.

Weitere kommerzielle Dateisysteme für Linux

Mit dem Einsatz von Linux in unternehmenskritischen Anwendungen (z. B. SAP R/3) und der Zunahme der kommerziellen Bedeutung von Linux, haben einige Firmen begonnen, ihre speziellen Dateisysteme und andere Speicher-Management-Werkzeuge auf Linux zu portieren und dort gegen Lizenzkosten zu vermarkten. So bietet z. B. die Firma Veritas mit seiner *Veritas Linux Foundation Suite* ihr vxfs Dateisystem auch unter Linux an. vxfs ist ein sehr stabiles, performantes Journaling Dateisystem, welches auch für Plattformen wie IBM-AIX, Sun Solaris, HP/UX und Windows 2000 verfügbar ist. Die Veritas *Foundation-Suite* bringt dafür einen eigenen Logical Volume Manager mit.

Zusätzlich kann vxfs als Client für ein HSM-System (*Hierarchical Storage Management*) eingesetzt werden. Bei einem HSM werden Daten, auf die einige Zeit nicht mehr zugegriffen wurde, automatisch auf ein langsames Speichermedium ausgelagert, z. B. auf Band oder eine optische Jukebox mit optischen Datenträgern wie CD oder DVD. Gleiches passiert, wenn das primäre, schnelle Speichersystem einen gewissen Füllgrad überschreitet. Werden die Daten wieder benötigt, verlagert das HSM-System die Daten weitgehend transparent wieder auf das lokale System zurück.

Andere kommerzielle Systeme sind das *Global File System* (GFS) der Firma Sistina oder das GPFS (*General Parallel File System*) von IBM, welches speziell auf die effiziente, schnelle Übertragung von Streaming-Daten ausgelegt ist.

Netz-Dateisysteme

Unter *Netz-Dateisystem* (*Network File System*) ist ein Dateisystem zu verstehen, bei dem in einem Rechnernetz von einem anderen Rechner auf das lokale Dateisystem eines Rechners zugegriffen werden kann. Man nennt dies auch *Sharing*, da sich die verschiedenen Rechner den Zugriff auf das lokale Dateisystem teilen. Der Rechner (Host), auf dessen lokalen Platten das Dateisystem liegt, muss dazu das gesamte Dateisystem oder einen Ausschnitt daraus im Netz publizieren (freigeben und bekannt machen).

nfs – das *Network File System* (kurz NFS) – ist das ursprünglich von der Firma Sun entwickelte Dateisystem mit Zugriffen auf Dateien anderer Rechner in einem lokalen Netzwerk. nfs ist inzwischen ein offener Standard und wird vor allem auf Unix- und Linux-Systemen als Netz-Dateisystem stark eingesetzt. Es steht jedoch (zumeist als optionales Paket) auch auf anderen Betriebssystemen wie Windows oder Mac OS zur Verfügung. Unter NFS können Dateibäume, welche von einem anderen Rechner publiziert

wurden, wie lokale Dateisysteme per **mount** eingehängt und genützt werden. Die aktuellste Version davon ist NFS-3.¹

Daneben bietet Linux über das Samba-Paket die Unterstützung des SMB- und CIFS-Protokolls² und kann damit lokal vorhandene Dateibäume als Netzlaufwerke für Windows-Systeme zur Verfügung stellen. Auch der Zugriff von Linux auf einen Windows-*Share* (von Windows freigegeben Verzeichnisse) ist bei entsprechender Kernel-Generierung und einem entsprechenden **mount** möglich – oder über spezielle Linux-Clients.

Über das **netatalk**-Paket, Teil der meisten Linux-Distributionen, simuliert Linux das Apple-Talk-Protokoll und kann in einem Apple-Netz Verzeichnisse für den Zugriff für Macintosh-Systemen freigeben. Dies setzt man vorwiegend für die älteren Mac-OS-Systeme ein, da das aktuelle Mac OS X selbst das NFS-System unterstützt und damit offener und mit weniger Restriktion Dateisysteme publiziert und *geshared*³ werden können. Neben Dateisystemen erlaubt das **netatalk**-Protokoll unter Linux auch das Publizieren und Sharing von Linux-Druckern.

Das Novell-NetWare-Protokoll und damit der Zugriff auf entsprechend (von Linux) freigebende Verzeichnisse wird bei Linux **ncpfs** realisiert. Es ist jedoch standardmäßig auf den meisten Distributionen nicht installiert.

WebDAV das *Web Distributed Authoring and Versioning* Protokoll ist eine Erweiterung des Internet-HTTP-Protokolls – zumeist realisiert über einen Web-/WebDAV-Server. Es erlaubt einen weitgehend transparenten Zugriff über Internet (und zumeist über Firewalls hinweg) auf ein vom WebDAV-Server publiziertes Verzeichnis und den darin vorhandenen Dateibaum. Allerdings sind dazu spezielle Zugriffsroutinen zu nutzen. Der zugreifende Client (das zugreifende Programm) muss deshalb nicht über die normalen Zugriffsroutinen, sondern über WebDAV-orientierte Funktionen zugreifen. WebDAV ist speziell auf das verteilte, kooperative Bearbeiten von Dokumenten und Dateien ausgerichtet. Es bietet neben den reinen Dateizugriffen auch einfache Dokument-Management-Funktionen wie *Check-Out* und *Check-In*. Diese Funktionen sollen bei einem kooperativen Editieren von Dateien/Dokumenten verhindern, dass mehrere Personen zugleich ein Dokument editieren und sich später ihre Änderungen gegenseitig überschreiben. Kommerzielle Desktop-Publishing-Anwendungen unterstützen deshalb zunehmend WebDAV. Dazu zählen z. B. die neueren MS-Office-Anwendungen (ab Office 2000) und zahlreiche Adobe-Produkte wie etwa Photoshop (ab Version 6), Frame-Maker (ab Version 7) oder Illustrator (ab Version 9). Für sie, unter Windows oder Mac OS laufend, kann Linux mittels des Apache-Servers *shared Ressources* zur Verfügung stellen.

Da die von Linux realisierten Netz-Dateisysteme auf den lokalen Linux-Dateisystemen aufsetzen (sofern Linux den Plattenplatz zur Verfügung stellt), sind in gewissem Umfang die Möglichkeiten der beiden Dateisysteme aufeinander abzustimmen. Potenziell addieren sich hier die einzelnen Restriktionen.

1. Linux 2.4.x hat noch nicht den vollen Umfang von NFS-3 implementiert. Dies sollte aber bis Ende 2003 abgeschlossen sein.
2. SMB = *Server-Message-Block*-Protokoll, ein Microsoft-Windows-Protokoll für Netzwerkzugriffe auf Dateisysteme und Drucker. CIFS = *Common Internet File System* (Protokoll).
3. Mit *sharing* ist hier der gemeinsame Zugriff über Netz verschiedener (Rechner-)Systeme auf lokale Dateisysteme verstanden.

Ältere Dateisysteme

Ältere Linux-Systeme unterstützen noch einige ältere Linux-Dateisysteme, welche aber inzwischen keine Bedeutung mehr haben und in aktuellen Linux-Systemen teilweise auch nicht mehr benutzt werden können. Zu diesen Dateisystemen gehört z.B. das **minix**-Dateisystem. Es war das erste unter Linux laufende System und stammt aus der Minix-Entwicklung von Professor Tannenbaum. Wegen seiner Limitationen (z.B. eine maximale Dateisystemgröße von 64 MByte und kurze Dateinamen) sollte es heute nicht oder nur auf kleinen, dedizierten Systemen eingesetzt werden.

Auch der minix-Nachfolger unter der Bezeichnung **ext** (*extended file system*) muss als veraltet und nicht mehr geeignet angesehen werden. Es wurde durch das aktuelle **ext2** (die zweite Version des **ext**) vollständig ersetzt. Eine andere Weiterentwicklung von minix war **xiafs**. Auch seine Entwicklung ist inzwischen eingestellt und aktuelle Linux-Kernel (seit Version 2.1.12) unterstützen es nicht mehr.

Fremde Dateisysteme

Aus Kompatibilitätsgründen und um das Zusammenspiel mit anderen Betriebssystemen zu vereinfachen, unterstützt Linux den Zugriff auf eine Reihe fremder, d.h. unter anderen Betriebssystemen übliche Dateisysteme. Die häufigste Nutzung dürfte dabei der Zugriff auf Dateisysteme des Microsoft-Windows-Systems sein. Zum Spektrum der unterstützten Nicht-Linux-Dateisysteme gehören z.B.¹

- msdos** Dies ist das alte MS-DOS FAT16 Dateisystem, welches heute zumeist nur noch auf Floppies zum Datenaustausch genutzt wird. Die Dateinamen sind hier auf 8+3-Zeichen (8 Zeichen, gefolgt optional von einem Punkt und einer 3-Zeichen-langen Endung) beschränkt.
- umsdos** (*UNIX on DOS*) ist eine Linux-Erweiterung des msdos-Systems. Linux fügt hier – über spezielle Erweiterungen, welche die Kompatibilität unter DOS nicht stören – lange Dateinamen, Benutzer- und Gruppen-Ids beim Dateibesitzer sowie Posix-konforme Zugriffsrechte hinzu.
- vfat** ist das modernere VFAT-Dateisystem der Windows-Systeme. Es bietet lange Dateinamen und erlaubt auch größere Dateisystemgrößen.
- ntfs** ist das native Dateisystem von Windows-NT5, -2000 und -XP. Leider ist hier bisher nur ein lesender Zugriff möglich.
- hfs** ist das (alte) *Hierarchical-File-System* (unter MacOS 8.x/9.x). Linux kann mit dem hfs-Modul solche Datenträger bzw. Dateisysteme per **mount** einhängen und darauf lesend und schreibend zugreifen.
- hpf** ist das *high-performance-file-system* bzw. das Standarddateisystem von OS/2. Unter Linux kann nur lesend darauf zugegriffen werden.

1. Nicht alle der hier aufgeführten Systeme sind standardmäßig installiert bzw. als Kernel-Erweiterung vorhanden. Sie sind auch nicht in allen Linux-Distributionen vorhanden, sondern müssen teilweise aus dem Netz geladen und integriert werden.

- sysv** steht für das ›*UNIX System V*‹-Dateisystem und erlaubt den Zugriff auf Datenträger des Xenix-Systems sowie des Systems-V/386 und des Coherent-Dateisystems. Dieses System wird auch unter SCO UNIX viel verwendet. Diese Systeme verlieren an Bedeutung.
- ncpfs** ist die Unterstützung des Novell-Netware-Netz-Dateisystems. Für seine Nutzung müssen spezielle Zusatzprogramme und -module unter Linux installiert werden.
- ufs** das unter dem BSD-Unix-System,¹ unter SunOS, OpenBSD, NetBSD und FreeBSD vielfach eingesetzte UNIX-Dateisystem UFS. Die aktuelle Linux-Unterstützung sieht hierfür lediglich einen lesenden Zugriff vor.
- iso9660** ein Dateisystem für CD-ROMs. Hierbei handelt es sich um Dateisysteme nach dem ISO-9660- bzw. High-Sierra-Standard. Auch die Erweiterung nach dem Rock-Ridge-Format wird unterstützt. Die Implementierung bietet auch die Unterstützung des unter Windows viel verwendeten Joliet-Formats, welches z. B. lange Dateinamen zulässt. Selbst das unter Mac OS bevorzugte HFS-CD-Format wird hier weitgehend transparent unterstützt.
- udf** (*universal disk format*) ein Dateisystem, welches bei DVDs und CD-RW-Systemen eingesetzt wird.
- cramfs** ist ein compressed (komprimiertes) ROM-Dateisystem, mit dem sich mehr Daten auf eine CD packen lassen als im üblichen ISO9660-Format.

Der Zugriff auf fremde Dateisysteme geht aber auch in der anderen Richtung, d. h. von einem anderen Betriebssystem auf einen unter Linux erstellten Datenträger oder lokal vorhandenes Linux-Dateisystem. Für Mac OS steht z. B. ein Macintosh-Plug-In (**MountX**) zur Verfügung, mit dem auf **ext2**-Dateisysteme/-Datenträger zugegriffen werden kann. Auch für Windows-95-Systeme gibt es einen **ext2**-Treiber (**FSDEXT2**), jedoch lediglich für einen Lesezugriff. **explore2fs** (siehe [Explore2fs]) geht hier weiter und erlaubt unter Windows (von 95 bis XP) Zugriff auf (lokale) **ext2**-Dateisysteme. Diese werden zwar unter Windows nicht als Dateisystem eingehängt, man kann aber wie in einem grafischen FTP-Browser per Navigation und per *Drag&Drop* darauf zugreifen – in beiden Richtungen und sowohl lesend als auch schreibend. Da dieses System jedoch noch in einem frühen Stadium ist (Beta-Test) ist, sollte man es mit Vorsicht einsetzen.

Einen Schritt weiter geht der bei Druck dieses Buchs noch im Beta-Test befindliche **ext2**-Treiber von FreeSourceCodes.org (siehe [EXT2-NT]). Er erlaubt unter Windows NT 4.0, **ext2**-Systeme als NT-Laufwerk einzuhängen und von Anwendungen transparent darauf zuzugreifen.

Neben diesen kostenlosen Lösungen gibt es auch eine Reihe kommerziell vertriebener Produkte wie etwa **Ext2FS Anywhere** der Firma Paragon Software.

Eine ausführliche Liste von Möglichkeiten von Zugriffen auf unterschiedliche Dateisysteme auf unterschiedlichen Plattformen ist unter [FSHowTo] zu finden.

1. BSD = *Berkeley System Distribution*. Siehe dazu die Beschreibung auf Seite 24 und 32.

Das Prozessdateisystem (/proc)

Auf den Adressraum von Prozessen (und dem Betriebssystem) direkt wie auf eine Datei zugreifen zu können, bietet zum Testen eine Reihe von Vorteilen. Auch bestimmte Linux-Programme wie etwa **ps**, **top** oder **gtop** nutzen einen solchen Zugang auf System- und Prozessdaten. Linux stellt dafür das so genannte *Prozessdateisystem* zur Verfügung. Hierbei handelt es sich nicht um ein Dateisystem im herkömmlichen Sinne, sondern um einen vom Systemkern vorgetäuschten Dateibaum, der in */proc* beginnt. Hier ist jeder aktuell aktive bzw. gestartete Prozess durch eine Pseudo-Datei oder ein Pseudo-Verzeichnis vertreten. Die Verzeichnisnamen von Prozessen entsprechen den Prozessnummern. Darin sind die wesentlichen Daten des Prozesses als Datei angelegt, in denen z. B. die Umgebungsvariablen des Prozesses unter **environ** oder die Aufruffolge unter **cmdline** und der vom Prozess belegte Speicher unter **mem** zu finden sind. Zum Schutz der Daten ist hier der Zugriff jedoch eingeschränkt – mit **root** als Besitzer.

Das Prozessdateisystem belegt keinen Platz auf der Platte – die eigentlichen Daten liegen im Hauptspeicher und im Swap-Bereich.

Weitere Dateisysteme

Die Linux-Entwickler sind eine ausgesprochen dynamische Entwicklergemeinde, in der ständig neue Ideen, Konzept und Implementierungen entstehen. Einige davon betreffen auch Dateisysteme. Nicht alle davon erreichen einen Produktstatus oder die Integration in den Standard-Linux-Kernel. Es gibt mehrere Entwicklungen für verschlüsselte Dateisysteme (z. B. unter Verwendung des *Loopback-Devices*, das FSFS (*Fairly Secure File System*)).¹ Auch das bereits erwähnte Gerätedateisystem ist in Teilen noch in der Entwicklung.

LVM – der Logical Volume Manager

In größeren Systemen benötigt man häufig Dateisysteme, welche größer als eine physikalische Platte sind. Daneben ergibt sich der Bedarf, Speicherbereiche neu zuzuteilen, zu vergrößern und zu verkleinern, ohne dass dazu das System neu gestartet werden und ohne dass man dazu die Partitionierung aufwändig ändern muss. Dies ermöglicht der seit dem Linux-Kernel 2.4 verfügbare *Logical Volume Manager* – kurz LVM.²

Er abstrahiert Speicherbereiche – hier *Volumes* genannt – von der physikalischen Schicht und erlaubt z. B., dass sich ein *Volume* und das darauf befindliche Dateisystem über mehrere physikalische Partitionen und Datenträger bzw. Magnetplatten erstrecken kann. Andererseits lässt sich der vom LVM verwaltete Bereich in zahlreiche kleine *Volumes* aufteilen, ohne dass man sich dabei mit den Details und den Limitationen von Plattenpartitionen herumschlagen muss.

Der Linux-LVM erlaubt zusätzlich in recht transparenter Art und Weise das Spiegeln (*Mirroring*) von *Volumes*. Hierbei wird der Inhalt des *Volume* automatisch auf ein weiteres System gespiegelt, so dass im Fall eines System- oder Plattenausfalls die Daten in aktueller Form nochmals vorhanden sind.

1. Zu verschlüsselten Dateisystemen unter Linux siehe auch [CryptFs].

2. Eine ausführliche Beschreibung des Linux LVM ist unter [LVM-1] und [LVM-2] zu finden.

Der LVM hat zwei Abstraktionsstufen und eine eigene (notwendige) Terminologie. In der ersten Stufe fasst er reale physikalische Speichereinheiten wie ganze Platten (mit einer einzigen Partition) oder Partitionen einer Platte – beides wird als *Physical Volume* (kurz PV) bezeichnet – zu großen logischen (virtuellen) Speicherbereichen zusammen – so genannte *Volume Groups* (kurz VG). Dieser Volume Group können also mehrere Partitionen und mehrere Platten angehören. Man kann einer Volume Group auch später noch neue *physical Volumes* (Partitionen) hinzufügen. Zugleich werden beim Anlegen Speicherblockgrößen festgelegt – so genannte *Extends*. Im Standardfall sind dies 4-MB-Einheiten.¹ Speicher für ein Dateisystem kann nur aus einem Vielfachen dieser Extends vergeben werden. Eine *Volume Group* ist damit eine gewisse Verwaltungseinheit mit eigenem Namen unter dem LVM. LVM 1.05 erlaubt bis zu 99 solcher Volume Groups.

Aus einer Volume Group kann der Speicher nun an *virtuelle Partitionen* vergeben werden – hier *Logical Volumes* genannt (oder kurz LV). Die Größe ist jeweils ein Vielfaches der Extend-Größe. Auf diese *Logical Volumes* schließlich legt man das eigentliche Dateisystem an – z. B. ein ext2- oder ein reiserfs-Dateisystem. Mit LVM 1.05 sind bis zu 256 solcher Logical Volumes pro Volume Group möglich – also z. B. sehr viel mehr als Partitionen bei der normalen Platten-Partitionierung.² Auch können sich nun virtuelle Partitionen (*Logical Volumes*) über mehrere Platten erstrecken. Man kann sogar ein *Striping* einsetzen, bei dem explizit die Blöcke eines Logical Volumes stückweise in *Stripes* (*Streifen*) über mehrere Platten verteilt werden, um durch parallele Zugriffe eine höhere Performance für ein hier liegendes Dateisystem zu erreichen.

Das Logical Volume kann nun wie eine Platten-Partition dazu genutzt werden, um darauf ein Dateisystem anzulegen oder um – wie bei Datenbanken wie z. B. Oracle üblich – der Datenbank einen Speicherbereich für direkte (*raw*) Blockzugriffe zur Verfügung zu stellen.

Wird der Platz auf einem Logical Volume zu klein, so kann man ihm im laufenden Betrieb aus dem noch verfügbaren Speicher-Pool der gleichen Volume Group weitere Extends zuweisen, ohne dass dazu das System heruntergefahren oder eine Neupartitionierung notwendig ist. Man kann aber auch einen Logical Volume verkleinern – so noch nicht der ganze Platz im Dateisystem vergeben ist – und den frei gewordenen Platz einem anderen Logical Volume der gleichen Volume Group zuteilen. Ebenso lassen sich weitere *Physical Volumes* (z. B. neue Platten) in die Volume Group aufnehmen und mit deren Extends der Platz auf den *Logical Volumes* vergrößern. Man erhält damit eine große Flexibilität beim Speichermanagement. LVM besitzt dabei Werkzeuge, um die Daten eines Logical Volumes zur Laufzeit und transparent für die laufenden Anwendungen zu verschieben, um so z. B. eine Platte frei zu bekommen, damit sie dann einer anderen Volume Group zuteilt oder das Laufwerk ausgetauscht werden kann.

Da das Vergrößern und Verkleinern von Logical Volumes auch Eingriffe in das darauf befindliche Dateisystem bedingt, muss das Dateisystem auf eine entsprechende Zusammenarbeit mit dem LVM vorbereitet sein. So erfordert eine Veränderung bei z. B.

-
1. Möchte man sehr große Speichereinheiten handhaben, sollte man die Extend-Größe heraufsetzen, da im LVM 1.0 lediglich 65 536 Extends verwaltet werden können, d. h. der maximal verwaltbare Speicher bei 4-MB-Extends also auf 256 GB beschränkt ist.
 2. Die Gesamtzahl aller *Logical Volumes* in allen *Volume Groups* ist jedoch auch auf 256 beschränkt.

ext2/etx3 einen Neustart des Systems, während reiserfs und xfs dies im laufenden Betrieb durchführen können.

Ein zusätzliche Funktion des Linux LVMs sind so genannte *Snapshots*. Sie benutzt man zur konsistenten Sicherung eines Dateisystems, ohne dass dieses dazu angehalten werden muss. Dabei wird das Dateisystem kurzfristig eingefroren, um einen konsistenten Stand zu erreichen. Nun kann die Sicherung auf Band oder auf einen anderen Speicherbereich beginnen und das Dateisystem weiter laufen. Alle nun erfolgenden Änderungen an dem Dateisystem werden nun vorübergehend nicht mehr in das Dateisystem selbst geschrieben, sondern in einen dafür bereitgestellten Snapshot-Bereich (einen Logical Volume). Ist die Sicherung des ursprünglich Volumens abgeschlossen, so erfolgt der zweite Schritt des Snapshots. Nun überträgt das System die Änderungen aus dem Snapshot-Bereich wieder in das ursprüngliche Dateisystem, während zugleich (oder danach) auch das Snapshot-Volume gesichert werden kann. Anschließend wird das Snapshot-Volume wieder freigegeben. Zu beachten ist, dass mit dieser Technik ein konsistenter Stand des Dateisystems gesichert werden kann; zum Snapshot-Zeitpunkt müssen die Daten auf dem Dateisystem jedoch nicht unbedingt aus Sicht der darauf arbeitenden Anwendungen konsistent sein. Um dies zu erreichen, muss man unter Umständen die darauf arbeitenden Anwendungen zuvor herunter fahren oder ihnen den Befehl geben, einen konsistenten Datenzustand herzustellen.

Software-RAID

RAID steht für *Redundant Array of Inexpensive (Independent) Disks*. Hierbei werden mehrere Platten zusammenschaltet, um damit eine höhere Performance durch parallele Zugriffe, eine höhere Sicherheit durch redundante Speicherung oder beides durch eine Kombination zu erreichen. Im Regelfall wird dies durch spezielle Plattencontroller oder ganze Plattensysteme erreicht, die jedoch teuer sind. Alternativ kann die RAID-Funktion auch über Software im Betriebssystem übernommen werden. Linux erlaubt sowohl die Nutzung von RAID-Controllern als auch die Emulation per Software im Kernel. Die Linux-Software-RAID kann in einem der beiden Modi RAID-0 oder RAID-1 betrieben werden.

RAID-0 verteilt die Daten über mehrere Platten (minimal 2) und erreicht dabei sowohl beim Lesen als auch beim Schreiben eine bessere Performance (fast die doppelte). Zeigt eine der beteiligten Platten jedoch einen Defekt, fällt faktisch das gesamte auf dem RAID-Segment liegende Dateisystem aus. Man arbeitet hier also mit einem höheren Ausfallrisiko.

Beim RAID-1-Modus werden die Daten des Dateisystems parallel und redundant auf mehrere Platten geschrieben (zumeist 2). Dies schafft Sicherheit beim Ausfall einer der Platten, da das System diese dann ausblendet und die Daten von den verbleibenden holt und dort weiterarbeitet. Bringt man danach ein Ersatzlaufwerk wieder online, so synchronisiert das RAID-System die Daten von den vorhandenen Systemen (allmählich) wieder auf dieses Laufwerk. Während die Schreibperformance etwas unter RAID-1 leidet, ist die Leseleistung etwas besser als ohne RAID.

›Memory-Mapped‹ Dateien

Linux übernahm die Möglichkeiten des Berkeley-Unix-Systems, Dateien (oder Teile daraus) in den Adressraum eines oder mehrerer Programme einzubinden. Dazu wird die Datei zunächst geöffnet und danach mit dem `mmap`-Aufruf in den Adressraum eingebunden. Der Vorteil liegt darin, dass dann auf die Dateikomponenten mit normalen Speicherzugriffen zugegriffen werden kann – so als handle es sich um normalen Arbeitsspeicher – ohne dass programmtechnisch dazu jeweils zuvor ein Lesen in einen Speicherbereich und später ein Zurückschreiben erfolgen muss. Mehrere Programme können damit auch auf eine Art *Shared Memory* (d.h. einen gemeinsamen Speicherbereich) zugreifen; an Sohnprozesse kann so ein gemeinsamer Speicherbereich für einen effizienten Datenaustausch vererbt werden.

3.2.5 Anlegen und Prüfen von Dateisystemen

Bevor Dateien in einem Dateisystem abgelegt werden können, muss zunächst das Dateisystem – bzw. seine Struktur- und seine Informationseinheiten – auf dem Datenträger angelegt werden. Dies erfolgt auf PC-Systemen in drei oder vier Schritten:

- ▶ **Formatieren des Datenträgers** oder der Magnetplatte:
Dies ist nicht bei allen Datenträgern und Platten notwendig oder möglich. Zum Beispiel stehen in der Regel keine Werkzeuge zur Basisformatierung von IDE-Platten zur Verfügung – sie werden vom Hersteller vorformatiert.
- ▶ **Anlegen der Partition-Struktur** (*Partitioning*):
Unter Linux erfolgt dies entweder per `fdisk`, `cfdisk` oder über eine grafische Oberfläche, welche dann ihrerseits diese Funktion benutzt.
- ▶ Möchte man den *Logical Volume Manager* (LVM) verwenden, so muss entweder die ganze Platte (ohne eine vorhergehende Partitionierung) oder die betreffende Partition in eine *Volume Group* des LVMs eingebracht werden. Danach ist ein *Logical Volume* anzulegen, auf dem später das Dateisystem angelegt werden soll.
Möchte man die Linux-Software-RAID-Funktion benutzen, so ist das RAID-Segment vor dem Anlegen des eigentlichen Dateisystems zu erstellen, da es dann eine neue logische Partition zur Verfügung stellt.
- ▶ **Anlegen des eigentlichen Dateisystems** auf einer Partition oder – bei Verwendung des LVMs – auf einem Logical Volume:
Dies erfolgt mit Hilfe des *Make-File-System*-Kommandos `mkfs`. Da jeder Dateisystemtyp spezifische Funktionen und Optionen hat, gibt es praktisch für jedes unter-

Anmerkungen zur Tabelle 3.2 auf Seite 147:

* Dateien müssen, soweit es sich nicht um *Sparse Files* (Dateien, mit leeren Blöcken ohne Inhalt) handelt, natürlich immer kleiner als das Dateisystem sein, auf dem sie liegen. Bei Dateien größer als 2 GB muss für eine korrekte Verarbeitung der Large-File-System-Support sowohl im Betriebssystem vorhanden als auch im Programm umgesetzt sein.

** Der Linux-Kernel besitzt in der Regel Restriktionen, welche die reale Größe eines Dateisystems weiter einschränken. In der Kernel-Version 2.4.x sind dies auf 32-Bit-Systemen zumeist 2 TB.

Tabelle 3.2: Die wichtigsten Merkmale der verschiedenen Linux-Dateisysteme

Funktion:	minix	ext2	ext3 (??)	reiserfs (3.6)	JFS (1.0x)	xfx (1.0x)	iso9660
Dateinamenlänge (Bytes)	14	255	255	255	255	255	8/255
mögliche Blockgröße	0,5 kB	1, 2, 4, (8) kB	1, 2, 4, (8) kB	4 kB	0,5–4 KB	4 kB	1 kB
maximale Dateigröße*	64 MB	16 GB–2 TB	2–16 TB	4 GB	4 PB	4 PB	< 4 GB
maximale Dateisystemgröße**	64 MB	2 TB	2 TB	16 TB	512 TB–4 PB	512 TB–2 TB	4 GB
Journaling Metadaten	–	–	ja	ja	ja	ja	–
Journaling Datenblöcke	–	–	optional	ab Version 4	–	–	–
Unterstützung LVM/SW-RAID	–	LVM/RAID	LVM/RAID	LVM/RAID	LVM/RAID	LVM/(RAID)	–
Dateifragmente	–	–	–	ja	–	–	–
Komprimierung	–	ext2d	–	–	–	–	cramp ??
Quota-System möglich	–	ja	ja	ja	ja	ja	–
Erweiterte Metadaten	–	–	–	ab Version 4	–	ja	–
spezielle Eignung	sehr kleine Systeme	Root- und Boot-Dateisystem	Root-Dateisystem	große Verzeichn., viele kleine Dateien	kompatibel mit ADX-JFS	sehr große Dateien	nur für CD/DVD vorgesehen
besondere Merkmale	veraltet	Standard-Dateisystem unter Linux	setzt Journaling auf ext2 auf	stabil + performant	Portierung von IBM AIX	stark skalierend	

stützte Dateisystem ein spezifisches mkfs – z. B. **mkex2fs** für ein Dateisystem vom Typ ext2 oder **mkreiserfs** für eines vom Typ reiserfs.

Alternativ ist das Anlegen des Dateisystems auch über die entsprechende grafische Oberfläche der Linux-Distribution. Bei SuSE-Linux ist dies z. B. eine Funktion des YaST2. Hierbei sind die Art des gewünschten Dateisystems sowie eine Reihe von Parametern wie etwa Dateisystemgröße anzugeben, die wiederum vom Typ des Systems abhängig sind. Beim Anlegen eines neuen Dateisystems werden alle in diesem Bereich liegenden Daten zerstört.

Danach muss das Dateisystem mittels des im nächsten Abschnitt beschriebenen **mount**-Kommandos dem System bekannt gemacht bzw. *eingehängt* werden. Linux montiert nur *saubere Dateisysteme*. Darunter versteht man ein Dateisystem, das entweder neu angelegt wurde oder nach der letzten Benutzung korrekt demontiert (mittels des Kommandos **umount**) und dabei als *sauber* gekennzeichnet wurde. War das Betriebssystem abgestürzt oder wurde das Dateisystem auf eine andere unkorrekte Art aus dem System genommen, so muss zunächst eine Konsistenzprüfung erfolgen. Hierzu steht das Programm **fsck** (*File System Check*) zur Verfügung – jeweils ebenso in Dateisystem-spezifischen Varianten (etwa **e2fsck**, **reiserfsck**, ...). Das fsck-Programm behebt eventuelle Inkonsistenzen und setzt das Dateisystem auf den Status *Clean*. Beim Hochfahren des Linux-Systems überprüft Linux zunächst alle zu montierenden Dateisysteme automatisch auf den Zustand *clean* und ruft, sofern ein anderer Status gefunden wird, automatisch **fsck** auf.

Da auch bei jeweils sauber aus- und eingehängten Dateisystemen potenziell die Gefahr von Inkonsistenzen besteht – verursacht etwa durch Hard- oder Software-Probleme – führt Linux im Dateisystem einen Zähler mit, welcher die **mount**-Operation seit der letzten **fsck**-Prüfung verfolgt. Nach einer vordefinierten Anzahl von mounts führt Linux dann trotz des Status *clean* einen **fsck**-Lauf durch und setzt danach den Zähler zurück. Im Standardfall wird bei ext2 nach 25 **mount**-Operationen dies durchgeführt. Beim Anlegen des Dateisystems können jedoch andere Werte vorgegeben oder die Prüfung auf diese Art unterdrückt werden. Zusätzlich zum **mount**-Zähler kann auch eine maximale Zeitspanne vorgegeben werden, nach der spätestens ein **fsck** erfolgen soll.

3.2.6 Demontierbare Dateisysteme

Das Gerät bzw. Laufwerk mit dem darauf vorhandenen Dateisystem, auf welchem sich das Linux-System befindet, wird als **root device** bezeichnet.¹ Der auf dem *root device* liegende Dateibaum ist nach dem Start dem System bekannt und zugreifbar. Neben diesem Dateisystem gibt es jedoch normalerweise Dateien auf weiteren Dateisystemen, welche man entweder beim Systemstart oder später einhängen möchte. Hierzu können z. B. Dateisysteme auf Magnetplatten, CDs, DVDs oder anderen Datenträgern gehören. Inzwischen präsentiert sich auch der Speicher einer digitalen Kamera häufig als ei-

1. Für das Dateisystem, von welchem der Linux-Kernel selbst gestartet wird (in der Regel unter **/boot**), gibt es einige Restriktionen die sicherstellen, dass dieses Dateisystem vom Boot-Loader gelesen werden kann! Hier sollte man deshalb in der Regel ein ext2-Dateisystem verwenden.

genständiges Dateisystem, welches per **mount** eingehängt und per **umount** wieder entfernt werden kann.

Daneben möchte man bei Netzwerk-Dateisystemen wie NFS oder Samba/CIFS nach dem Starten des Systems häufig Teile der Dateisysteme anderer Rechnersysteme zugreifbar machen.

Das Dateisystem, welches sich auf diesen Datenträgern befindet, kann man dem System durch das **mount**-Kommando bekannt machen und als Teilbaum in den Systemdateibaum montieren. Entsprechend sind beim **mount**-Kommando folgende minimale Angaben notwendig, die jeweils durch weitere Optionen ergänzt werden können:

- ▶ der Typ des Dateisystems, soweit er nicht automatisch aus einer Beschreibungsdatei (*/etc/vfstab*) ermittelt werden kann
- ▶ das logische Gerät, auf welchem sich das neue Dateisystem befindet
- ▶ das Verzeichnis, in dem der neue Dateibaum eingehängt werden soll



mount -t msdos /dev/fd0 /media/floppy

→ hängt das DOS-FAT16-Dateisystem (vom Typ *msdos*), das sich auf dem ersten Floppy-Laufwerk befindet, in das Verzeichnis */media/floppy* ein. Abb. 3.9 und Abb. 3.10 verdeutlichen dies. Abb. 3.9 zeigt dabei die beiden Dateibäume vor dem **mount**-Kommando und Abb. 3.10 danach.

Unter dem KDE-Desktop geht es bei der Floppy noch einfacher, indem man hier einfach einen Klick auf das Floppy-Symbol auf dem Desktop ausführt. Hiermit wird die Floppy unter */media/floppy* eingehängt.

Erst nach dem Montieren kann mit den normalen Linux-Dateioperationen (wie **create**, **open**, **read**, **write** usw.) auf diese Dateien zugegriffen werden. Der Zugriffspfad der Dateien besteht nun aus dem Zugriffspfad des Knotens, in den das neue System eingehängt wurde, gefolgt von dem Zugriffspfad innerhalb des montierten Systems. Befindet sich auf der Floppy z. B. eine Datei mit dem Namen */projekt/dat.1*, so ist sie nun unter dem Namen */media/floppy/projekt/dat.1* zu erreichen.

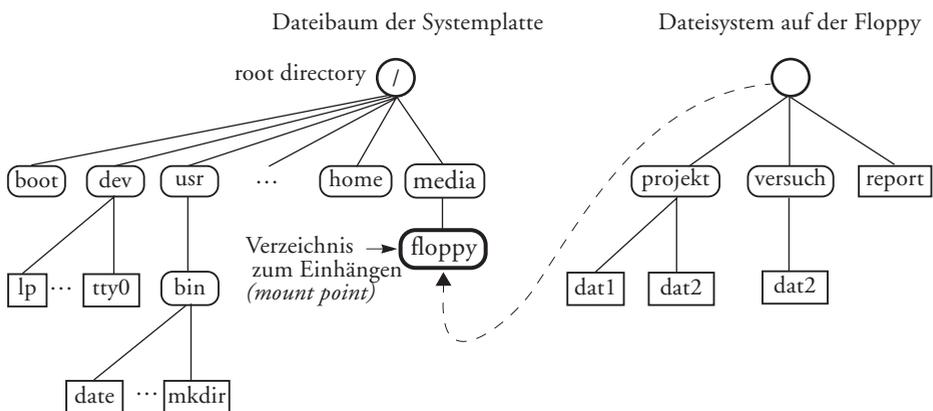


Abb. 3.9: Ausschnitt aus den Dateibäumen der beiden Dateisysteme

Nach der Ausführung des **mount**-Kommandos sieht der Dateibaum wie folgt aus:

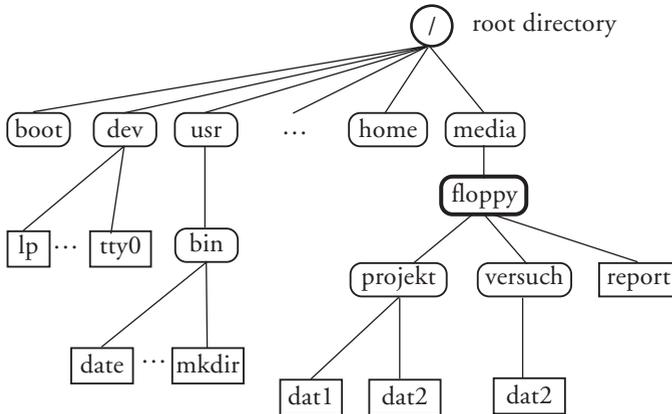


Abb. 3.10: System-Dateibaum nach der mount-Operation

Eine auf dem montierten Gerät liegende Datei unterscheidet sich nun nicht mehr von einer Datei auf dem *root device*. Eine Ausnahme gilt für Verweise (ein Eintrag im Verzeichnis) auf Dateien, sofern nicht mit symbolischen Verweisen gearbeitet wird. Bei *harten Links* gilt, dass alle Einträge und die Datei selbst auf dem gleichen logischen Datenträger liegen müssen.

Wären in dem Verzeichnis, in welchem das neue System eingehängt wurde, bereits Dateien vorhanden, so werden sie durch das eingehängte System überdeckt, solange das Dateisystem eingehängt ist.

Das Entfernen eines solchen Dateisystems erfolgt durch das **umount**-Kommando. Hierbei ist als Parameter nur der Name des logischen Datenträgers anzugeben.



umount /dev/fd0

→ entfernt das Dateisystem auf der Floppy aus dem Systembaum.

Einige der Optionen des **mount**-Kommandos sind dateisystemspezifisch. Da das Betriebssystem nicht in allen Fällen eindeutig bestimmen kann, um welchen Dateisystemtyp es sich bei dem zu montierenden Volume handelt, sollte man möglichst dem **mount**-Kommando den Systemtyp als Parameter mitgeben (in der Form *-Fds-typ*). Wird er nicht angegeben, so versucht das System den Typ aus der Datei */etc/vfstab* zu entnehmen.

Da das Ein- und Aushängen von Dateisystemen eine Gefahr für die Sicherheit und Konsistenz eines Systems darstellen kann, sind in den meisten Linux-Systemen das **mount**- und **umount**-Kommando Befehle, die nur von privilegierten Benutzern ausgeführt werden können – es sei denn, in der Datei */etc/fstab* ist für das entsprechende Gerät eine entsprechende **user**-Option gesetzt.¹

Möchte man vermeiden, dass Dateien auf dem montierten Dateiträger versehentlich oder absichtlich gelöscht bzw. geändert werden, so erlaubt die **mount**-Option **ro**, den Volume im *Read-Only-Modus*, d.h. nur zum Lesen zu montieren.

1. Siehe hierzu die Beschreibung von */etc/fstab* in Kapitel 9.2 auf Seite 766.

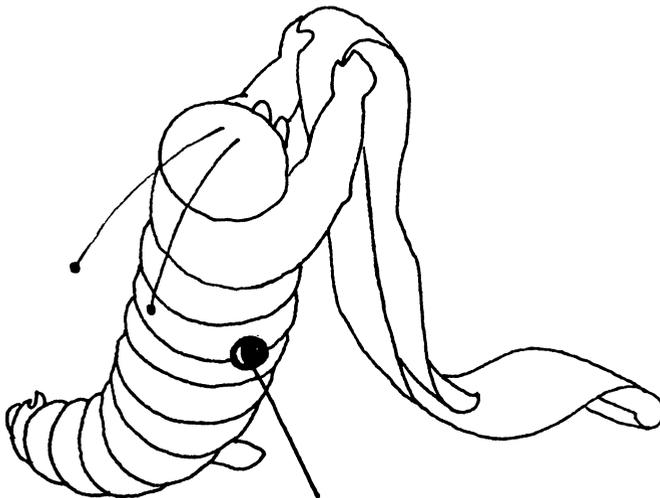
3.2.7 Das Quota-System bei Dateisystemen

In größeren Installationen möchte man den Platzverbrauch von Benutzern begrenzen können, so dass nicht ein einzelner Anwender versehentlich oder böswillig den gesamten verfügbaren Plattenspeicher belegt. Das Linux-System gestattet dies über das so genannte *Quota-System*. Dieses erlaubt, benutzerspezifische Grenzen für den Plattenspeicher vorzugeben. Dabei kennt das System zwei Grenzen:

- a) Eine *harte Grenze*, die nie überschritten werden darf. Das System meldet beim Überschreiten Fehler und verweigert die Schreib- oder Kopieroperation.
- b) Eine *weiche Grenze*. Sie darf vorübergehend überschritten werden. Geschieht dies, so wird der Anwender durch eine Nachricht informiert und es beginnt ein Zeitzähler zu laufen. Wird innerhalb des festgesetzten Zeitlimits die Grenze wieder nach unten durchschritten, erfolgt weiter nichts. Reduziert der Benutzer jedoch innerhalb der vorgegebenen Zeit seinen Plattenspeicherbedarf nicht, so tritt nach dem Zeitablauf die untere Grenze als harte Grenze in Funktion und der Anwender kann keine weiteren Speicher mehr belegen. Fehlermeldungen informieren ihn darüber.

Das Quotensystem lässt sich nicht nur individuell für einzelne Benutzer festlegen, sondern auch Dateisystem-/Partition-spezifisch. Damit ist es z. B. möglich, einem Anwender auf den Platten seines eigenen Systems beliebig viel Speicher zu gestatten (bis an die Grenzen seiner Platten), auf einem zentralen Fileserver jedoch Quoten vorzugeben. Das Quotensystem setzt den Einsatz eines Quoting-fähigen-Dateisystems voraus – ist also dateisystemabhängig (siehe Tabelle 3.2, Seite 147).

Damit das Quota-System korrekt arbeitet, müssen im Linux-Kernel die entsprechenden Module vorhanden sein, was bei SuSE und RedHat bereits im Standard-Kernel der Fall ist. Zusätzlich muss der Quota-Daemon laufen. Dies ist ein spezieller Überwachungsprozess, der beim Systemstart automatisch gestartet werden sollte.



3.2.8 Dateiorientierte Kommandos

Zu den dateiorientierten Kommandos sollen hier all jene gezählt werden, welche zum Neuanlegen, Kopieren, Ausgeben und Löschen von Dateien und Dateiverzeichnissen notwendig sind, welche das Abfragen und Ändern der Dateiattribute erlauben und welche die Sicherung und die Konsistenzprüfung von Dateien und Dateisystemen zulassen. Dabei sollen hier nur die wichtigen Kommandos erwähnt werden.

Kommandos zur Dateiausgabe und Dateianzeige

a2ps	konvertiert Dateien aus zahlreichen Formaten nach PostScript.
cat	Ausgabe oder Konkatenation von Dateien
fold	Ausgabe von Dateien mit überlangen Zeilen
lp	Ausgabe von Dateien über den <i>lp-Print-Spooler</i> . GUI-Varianten sind z. B. gtklp , kprinter oder xpp .
lpr	Aufruf des <i>lp-Print-Spoolers</i> in der BSD-LPD-Variante
lpstat	liefert Statusinformation zum Print-Spooler und seinen Aufträgen.
less, more	seitenweise Ausgabe von Dateien auf die Dialogstation
od, hexedit	erstellt einen oktalen bzw. hexadezimalen Auszug (<i>Dump</i>) einer Datei.
pr	seitenweise Ausgabe von Dateien mit einer Überschrift und Seitennummerierung
head	Ausgabe der ersten Zeilen einer Datei
split	zerteilt eine Datei in mehrere einzelne Dateien gleicher Größe
tail	Ausgabe der letzten Zeilen einer Datei

Das Programm **cat** kann sowohl zur Ausgabe als auch zum Zusammenhängen von Dateien verwendet werden. Sein Haupteinsatz ist die Ausgabe von kurzen Dateien auf die Dialogstation. Bei Sichtgeräten hat man dabei jedoch das Problem, dass **cat** nicht nach einer Seite anhält, sondern die Ausgabe fortlaufend erfolgt. Das Programm **more** und das mächtigere **less** erlauben hier eine komfortable seitenweise Ausgabe von einfachen Textdateien, wobei, soweit nicht von einer Pipe eingelesen wird, auch ein Überspringen von Seiten oder Rückwärtsblättern möglich ist. Sie werden auch als *Pager* bezeichnet. Für komplexere Formate wie etwa PostScript, PDF oder HTML gibt es die später noch aufgeführten speziellen Anzeigeprogramme (auch *Viewer* genannt).

Auch die GUI-Dateimanager **konquerer** (aus dem KDE-Paket) und **nautilus** aus GNOME sind in der Lage, Dateien anzuzeigen, nutzen dafür aber in der Regel – wie man es von Web-Browsern her kennt – dateitypspezifische Anzeigeprogramme, die man selbst entsprechend konfigurieren (zuordnen) kann.

Will man Text auf eine druckende Dialogstation oder einen Zeilendrucker ausgeben, so erweist sich **pr** als zweckmäßig, da es die Ausgabe in Seiten unterteilt, die Seiten durchnummeriert und optional mit einem Titel versieht. Zum Drucken sollte das **lp**-Kommando verwendet werden. Es erlaubt ein abgesetztes Drucken (*spooling*) der Aufträge in der korrekten Reihenfolge. Der Status eines Auftrags kann später mit **lpstat** oder **lpq** abgefragt und Aufträge mit **cancel** wieder storniert werden. Die zum Berkeley-UNIX kompatible Variante des **lp**-Kommandos ist **lpr** mit den Programmen **lpc**, **lpq** und **lprm** zur Administration der Aufträge im *Print-Spooler*. GUI-Versionen sind hier **klpq**, **gtklpq**, **xpdq** oder **kjobviewer**.

fold stellt einen typischen Filter dar, den man in der Regel einem anderen Ausgabeprogramm vorschaltet, um überlange Zeilen in mehrere kürzere zu zerteilen. **od**, **hexedit** oder die GUI-Variante **khxedit** wird man dann benutzen, wenn man die Struktur einer Datei mittels eines Dateiauszugs, im Computerjargon *dump* genannt, analysieren möchte, oder um nichtdruckbare Zeichen in einer Datei aufzudecken. Letzteres geht auch mit der **-v**-Option des **cat**-Kommandos.

head und **tail** sind dann nützlich, wenn man in eine größere Anzahl von Dateien kurz hineinschauen möchte, um sich einen Überblick zu verschaffen. **head** zeigt dabei die ersten paar Zeilen und **tail** die letzten Zeilen der Datei an. Mit der Option **-f** versehen, kann **tail** dynamisch eine Datei anzeigen, in die gerade aus einem anderen Programm geschrieben wird.

Zuweilen ist es notwendig, sehr große Dateien in mehrere Einzeldateien zu zerteilen, da einige Editoren nur Dateien bis zu einem implementierungsabhängigen Limit verarbeiten können, oder um sehr große Dateien auf mehrere Datenträger (Disketten) aufzuteilen. Hierzu ist das Programm **split** geeignet. Anschließend können derartig aufgeteilte Dateien mit **cat** wieder zusammengesetzt werden.

Einen fast eigenen Bereich stellen die Programme dar, welche eine Konvertierung von Textdateien nach PostScript oder PDF durchführen und dabei noch eine gewisse Formatierung (ein *Pretty-Printing*) erlauben. Hier sind z. B. **a2ps**, **enscript** und **mpage** zu nennen. **gs** – der Ghostscript-Interpreter – erlaubt dann PostScript und PDF-Dateien zu rastern (in ein Pixel-Image), zu wandeln oder für verschiedene Drucke aufzubereiten. **gv** und **gsview** sind GUI-Front-Ends, um PostScript und PDF auf dem Bildschirm anzuzeigen. **ggv** ist die GNOME- zu **gv** und **kghostview** die KDE-Variante. Auch der von Adobe stammende Acrobat Viewer ist als **acroread** verfügbar und dürfte für PDF-Dateien einer der besten Viewer sein.

Informationen über Dateien

df	gibt die Anzahl von freien Blöcken eines Dateiträgers aus.
du	gibt die Anzahl der durch einen Dateibaum belegten Blöcke aus.
file	versucht eine Klassifizierung (Art des Dateiinhalts) von Dateien.
find	sucht nach Dateien mit vorgegebenen Charakteristika.
ls	liefert das Inhaltsverzeichnis eines Dateiverzeichnisses.
pwd	liefert den Zugriffspfad des aktuellen Dateiverzeichnisses.
quot	liefert eine Aufstellung über die Dateibelegung aller Benutzer.
type	zeigt den vollen Pfad zu einem Programm an.

Das meistbenutzte Datei-Informationskommando ist **ls**, welches vollständige und partielle Inhaltsverzeichnisse von Verzeichnissen oder im ausführlichen Format (Option **-l**) auch (mit Hilfe der Option **-R**) Dateibäume ausgibt und die meisten Dateiattribute anzeigt. Die Option **-F** zeigt bei jeder Datei mit einem Sonderzeichen an, um welchen Dateityp es sich handelt. Varianten von **ls** mit sehr ähnlicher Funktion sind **vdirc** und **dir**.

Die Anzahl der durch Dateien oder Dateibäume belegten Blöcke liefert **du**, während **df** die Anzahl der noch freien Datenblöcke und Dateiköpfe für einen ganzen Datenträger (Dateisystem) ausgibt. **/usr/sbin/quot** liefert die Blockbelegung eines Dateisystems nach Benutzern oder Benutzergruppen aufgeteilt. Das Programm **file** versucht, mit Hilfe der Informationen aus der Datei */etc/magic* den inhaltlichen Typ der Datei zu er-

mitteln und erlaubt damit eine erste, schnelle Analyse, wenn man es mit einer unbekanntem Datei zu tun hat bzw. wenn man einfach feststellen möchte, ob es sich um eine druckbare Datei handelt.

Sucht man eine bestimmte Datei oder mehrere Dateien mit vorgebbaren Attributen (z. B. Dateien, welche seit dem 1.12. 2002 oder innerhalb des letzten Tages modifiziert wurden oder die älter als 100 Tage sind), so bietet sich hierfür das Programm **find** an. Es lässt auch das Durchsuchen ganzer Dateibäume zu.

pwd schließlich nennt das aktuelle Verzeichnis, wenn man nicht sicher ist, wo man sich im Dateibaum befindet.

Mit **type** kann der vollständige Zugriffspfad auf ein Programm ausgegeben werden und damit u. a. schnell ermittelt werden, ob ein bestimmtes Programm existiert und wo es gefunden werden kann.

Verzeichnis- und dateisystemorientierte Kommandos

cd	setzt neues Verzeichnis als aktuelles Verzeichnis ein.
fuser	gibt alle Prozesse an, die eine bestimmte Datei oder ein angegebenes Dateisystem momentan benutzen.
mkdir	legt ein neues Verzeichnis an.
mkfs	legt ein neues Dateisystem auf einem Datenträger an.
mknod	schafft einen neuen Gerätenamen.
mount	hängt ein Dateisystem in den Systembaum ein.
rm	löscht Dateien, mit der Option -i erst nach einer Rückfrage.
rmdir	löscht ein leeres Verzeichnis.
rm -r	löscht (rekursiv) ein Verzeichnis und alle darin enthaltenen Dateien und Verzeichnisse.
umount	entfernt das Dateisystem auf einem Datenträger aus dem Systemdateibaum.

cd ist ein häufig benutztes Kommando, um in ein neues aktuelles Verzeichnis zu wechseln, wobei man damit jedoch noch nicht die Schreibberechtigung dafür hat. Neue Verzeichnisse werden mit **mkdir** angelegt. Sie bekommen dabei Standarddateiattribute. Will man diese ändern, so sind hierfür die Programme **chmod**, **chown** und **chgrp** vorhanden. Gelöscht werden kann ein Verzeichnis dann wieder mit **rmdir**, allerdings nur, wenn es leer ist, während **rm** Dateien (oder auch Geräteknotten) zu löschen erlaubt. Rekursiv (mit der Option **r**) können dabei auch ganze Dateibäume und damit auch nicht-leere Verzeichnisse gelöscht werden. In Wirklichkeit wird mit **rm** jedoch nur die Referenz auf den Indexknoten gelöscht. Erst mit der letzten Referenz wird dann auch der Indexknoten freigegeben und damit die Datei gelöscht.

Bevor man auf einem neuen dateistrukturierten Datenträger wie Magnetplatte oder Floppydiskette Dateien anlegen kann, muss eine initiale Dateistruktur darauf angelegt werden. Dies erfolgt mit **mkfs**, wobei man dieses Kommando dem Systemverwalter vorbehalten sollte. Eine Ausnahme stellen hier die entfernbaren Datenträger wie Floppy, Wechselpplatten wie ZIP-Datenträgern oder MO-Datenträgern sowie CD-RW und DVR±R oder DVR±RW. Von **mkfs** gibt es dateisystemspezifische Varianten wie etwa **mkfs.ext2** zum Anlegen eines **ext2**-Dateisystems. Für die anderen Dateisysteme gibt es **mkfs** jeweils mit den Endungen **.ext3**, **.reiserfs**, **.jfs**, **.xfs**, **.mdos**, ...

Möchte man ein Dateisystem in den Systemdateibaum einhängen, so geschieht dies mit **mount**. Erst danach kann man mit den normalen Dateioperationen auf die Dateien dieses Systems zugreifen. **umount** ist komplementär dazu und entfernt einen Dateibaum auf einem entfernbaren Datenträger wieder aus dem Systemdateibaum. Dabei wird sichergestellt, dass alle noch intern gepufferten Datenblöcke für diesen Datenträger auf das Medium hinausgeschrieben werden. Das Aushängen ist jedoch nur möglich, wenn kein Prozess mehr auf dem Dateisystem arbeitet. Beide Kommandos brauchen zumeist nicht durch den Benutzer aufgerufen zu werden, sondern werden entweder beim Hoch- und Runterfahren des Systems automatisch oder durch den Systemverwalter ausgeführt.

Das Kommando **lsof** zeigt hierzu an, welche Prozesse eine vorgegebene Datei bearbeiten oder auf Dateien eines Dateisystems operieren.

mknod schließlich erlaubt es, neue Geräteknoten anzulegen – d.h. externe Namen für intern generierte Geräte. Dies geschieht in der Regel in dem Verzeichnis */dev*. Das bessere Verfahren ist das Anlegen dieser Einträge über die GUI-Werkzeuge der Systemverwaltung.



Modifikation von Dateiattributen

chattr	erlaubt, besondere Dateiattribute in einem ext2/etx3 -Dateisystem zu ändern (lsattr zeigt sie an).
chmod	erlaubt, die Zugriffsrechte (Mode) einer Datei zu ändern.
chown	ändert den Besitzereintrag einer Datei.
chgrp	ändert die Gruppennummer einer Datei.
ln	gibt einer Datei einen weiteren Namen (Verweis, <i>link</i>).
mv	ändert den Namen einer Datei.
touch	ändert das Datum der letzten Dateiänderung.
rename	benennt eine Datei oder ein Verzeichnis um
rm	löscht eine Datei (Referenz) aus dem Verzeichnis.
umask	setzt die Standardzugriffsrechte beim Anlegen einer neuen Datei.

Diese Kommandos erlauben es, die meisten Dateiattribute zu ändern. Mit **chmod** können die Zugriffsrechte der einzelnen Benutzergruppen einzeln oder für alle gemeinsam festgelegt werden, während man den Eintrag des Dateibesitzers mit **chown** bzw. **chgrp** für die Gruppe ändern muss. **touch** setzt das aktuelle Datum oder ein angegebenes Datum als *Datum der letzten Dateiänderung* ein, ohne sonstige Modifikationen an der Datei vorzunehmen, erlaubt jedoch über Optionen auch das Ändern des Erstellungsdatums sowie das Datum des letzten Zugriffs. Damit kann auch eine Datei neu und leer angelegt werden.

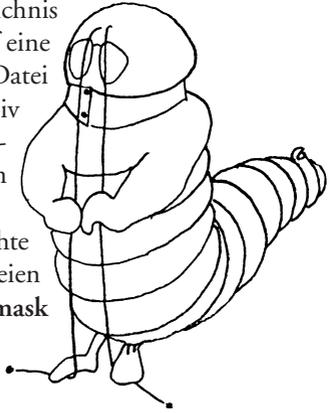
Mit Hilfe des **mv**-Kommandos kann man einer Datei (auch bei Verzeichnissen erlaubt) einen neuen Namen geben (der alte existiert danach nicht mehr), während mit **ln** einer Datei ein zusätzlicher Name verliehen bzw. eine weitere Referenz auf den Dateikopf (*Inode*) in das Verzeichnis eingetragen wird. **ln** gestattet unter Verwendung der Option **-s** die Vergabe von *symbolischen Verweisen*, die auch *Links* über Dateisystem-

grenzen und – bei entsprechender Vernetzung – auch über Rechnergrenzen hinweg zulässt.

Das **rename**-Kommando gestattet sicherer und logischer die Umbenennung von Dateien und Verzeichnissen.

rm schließlich löscht den Dateieintrag in einem Verzeichnis (den Verweis auf den Dateikopf). Ist der letzte Verweis auf eine Datei gelöscht (Anzahl der *Links* = 0), so wird auch die Datei selbst gelöscht. Benutzt man die Option **-r**, so wird rekursiv gearbeitet und es können ganze Dateibäume gelöscht werden. Verzeichnisse (sie müssen dazu leer sein) kann man auch mit **rmdir** löschen,

Das **umask**-Kommando verändert nicht die Zugriffsrechte einer Datei, sondern gestattet die Rechte neu angelegter Dateien mittels einer Maske vorzugeben. Alle in der Maske von **umask** gesetzten Rechte werden danach beim Anlegen einer Datei gelöscht bzw. **nicht** vergeben. Die Zugriffsrechte der Dateien können nachträglich natürlich verändert werden.



Sichern und Zurückladen von Dateien

bzip2	komprimiert Dateien (ähnlich wie compress und gzip).
cp	kopiert einzelne Dateien.
compress	komprimiert Dateien.
cp	kopiert einzelne Dateien.
cpio	sichert Dateien auf andere Datenträger und kann diese Dateien auch selektiv wieder zurücklesen.
dd	kopiert Datenblöcke physikalisch, wobei vielerlei Konvertierungen vorgenommen werden können.
dump	sichert ganzes Dateisystem, welches später mit restore zurückgespielt werden kann.
gzip	ist die unter Linux meistbenutzte GNU-Variante von compress und pack . Noch kompaktere Dateien soll bzip2 erzeugen.
pcat	führt eine cat -Operation auf eine mit pack komprimierte Datei aus.
restore	lädt mit dump gesicherte Dateien wieder ein. (*B*)
tar	sichert Dateibäume und erlaubt das selektive Wiedereinlagern der gesicherten Dateien.
uncompress	dekomprimiert eine mit compress komprimierte Datei.
unpack	dekomprimiert eine mit pack komprimierte Datei.
zcat	führt eine cat -Operation auf eine mit compress , gzip oder bzip2 komprimierte Datei aus.

cp ist die einfachste Art der Dateisicherung und kopiert entweder eine Datei in eine andere oder mehrere Dateien in ein neues Verzeichnis, wobei diese dort unter dem gleichen Namen angelegt werden. Sollen ganze Dateibäume kopiert werden, so ist hierfür die Option **-R** zu verwenden.

dd ermöglicht beim Kopieren eine Reihe von Konvertierungen (Blockungen, Codekonvertierungen, Längenbeschränkung). Daneben ist es mit **dd** möglich, sehr schnell physikalische Kopien von ganzen Datenträgern zu erstellen.

Das **dump**-Programm erlaubt es, ganze Datenträger entweder komplett oder inkrementell (d.h. ab einem bestimmten Änderungsdatum) zu sichern. Die **dump**-Termine können dabei in einer speziellen Datei festgehalten und damit eine Art Sicherungsautomatismus realisiert werden. **restore** gestattet die Information von den Sicherungsbandern wieder einzulagern. Da sowohl **dump** als auch **restore** Dateisystemspezifika kennen müssen, werden jeweils spezifische Versionen für unterschiedliche Dateisysteme benötigt. So stehen z.B. **xfsdump** und **xfsrestore** zur Verfügung. Anfang 2003 gab es aber leider noch keine Version für das Reiser-Dateisystem. **dump** und **restore** erlauben sowohl ein inkrementelles Sichern, bei dem jeweils nur die Dateien gesichert werden, welche sich seit dem letzten Sicherungspunkt geändert haben, als auch Vollsicherungen.

Sollen ganze Dateibäume oder Dateisysteme gesichert oder auf ein anderes System transportiert werden, so stehen hierfür **tar** und **cpio** als weitaus populärste Programme zur Verfügung. Beide erlauben sowohl das Sichern und die Erstellung eines Inhaltsverzeichnisses der gesicherten Information als auch das Wiedereinlagern. Das Programm **tar** hat sich dabei als Standardprogramm zum Austausch von Datenträgern zwischen verschiedenen Linux-Installationen etabliert.

cpio schreibt ähnlich wie **tar** auf Band oder einen anderen Datenträger. Ihm müssen dabei die Namen der zu sichernden Dateien explizit angegeben werden. Diese werden für einen Dateibaum zumeist mit **find** ermittelt und dann an **cpio** übergeben. **cpio** liest diese Namen von der Standardeingabe. Es erlaubt auch das Wiedereinlesen.

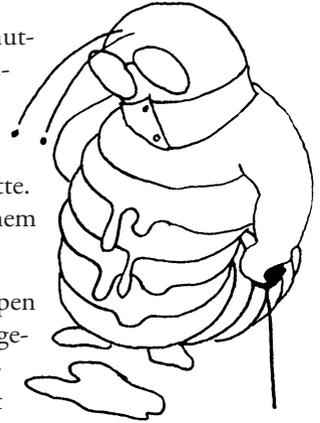
Die Programme **compress** und – unter Linux mehr eingesetzt – **gzip** und **bzip2** komprimieren Dateien nach unterschiedlichen Algorithmen und ersetzen – ohne weitere Optionen – die Originaldatei durch die komprimierte, kennzeichnen sie jedoch durch die Endung **.Z** bzw. **.z**. Das Dekomprimieren erfolgt durch **uncompress** bzw. **gunzip** und **bunzip2**. Die Programme **zcat** und **bzcat** geben die komprimierten Dateiinhalte analog zu **cat** aus, ohne zuvor explizit eine dekomprimierte Datei zu erzeugen.

Möchte man Dateien aus Sicherheitsgründen verschlüsseln, so kann dazu **pgp** eingesetzt werden. Die Datei wird damit so chiffriert, dass nur jemand damit etwas anfangen kann, der den Schlüssel kennt, um sie zu dechiffrieren. Auch die Editoren **ed**, **edit**, **ex** und **vi** erlauben über die Option **-x**, beim Sichern die Datei über einen Schlüssel zu chiffrieren bzw. eine chiffrierte Datei unter Angabe des Schlüssels zu dechiffrieren. Dies geschieht ebenfalls unter Verwendung von **crypt**. Der **crypt**-Mechanismus steht jedoch aus lizentechnischen Gründen nicht überall mit dem gleichen Verschlüsselungsverfahren zur Verfügung. Das ursprünglich darin eingesetzte DES-Verfahren darf nämlich nur bedingt aus den USA exportiert werden.

Konsistenzprüfung von Dateisystemen

Die folgenden Kommandos sind nicht für die alltägliche Benutzung, sondern von allem für die Anwendung durch den Systemverwalter gedacht:

fsck	Konsistenzprüfung des Dateisystems
sync	schreibt alle gepufferten Blöcke auf die Festplatte.
debugfs	erlaubt interaktiv bestimmte Korrekturen in einem ext2-Dateisystem vorzunehmen.



Hinsichtlich der Robustheit der älteren Linux-Dateisystemtypen gegenüber Software- oder Hardwarefehlern zeigt Linux nicht gerade seine besten Seiten – neuere Systeme weisen hier eine wesentliche höhere Robustheit auf. Das Programm **fsck** bietet eine vollständige Prüfung und Behebung von Problemen. Bei größeren Systemen sollte ein Aufruf dieses Programms deshalb Teil der Prozedur zum Hochfahren des Systems sein.

Aus Effizienzgründen puffert Linux Datenblöcke im Hauptspeicher. Hierdurch stimmt die Information auf dem Datenträger mit dem logischen Zustand nicht immer überein. Das Programm **sync** schreibt alle gepufferten noch ausstehenden Ausgaben auf die Datenträger. Aus Sicherheitsgründen sollte man daher vor dem Ausschalten eines Systems ein **sync**-Kommando ausführen. Im Standardfall geschieht dies bei Verwendung des **shutdown**- oder **reboot** Kommandos automatisch.

Linux überprüft vor dem Einhängen eines Dateisystems, ob es sauber ausgehängt wurde. Es aktiviert im Fehlerfall eine automatische Überprüfung mittels **fsck**. Da die Konsistenzprüfung und Fehlerbehebungen im Dateisystem dateisystemspezifisch sind, gibt es für jedes (der neuen Dateisysteme) eine eigene **fsck**-Variante. Hierzu gehören z. B. **fsck.ext2**, **e2fsck**, **fsck.ext3** für Systeme vom Typ **ext2/ext3**, **fsck.reiserfs** für das Reiser-Dateisystem, **fsck.jfs** für ein JFS- und **fsck.xfs** für ein XFS-Dateisystem, sowie **fsck.minix** für ein **minix**- und **fsck.cramfs** für ein CRAM-Datei-System. Selbst DOS- und NTFS-Systeme lassen sich mit **fsck.msdos**, **fsck.vat** (FAT-16, FAT-32) und **ntfsfix** prüfen und reparieren. Der File-System-Check sollte in aller Regel auf **ausgehängtem** Dateisystem erfolgen.

Mit entsprechendem Wissen und Vorsicht lassen sich mit **debugfs** einige Korrekturen in einem Dateisystem vom Typ **ext2** vornehmen – z. B. das Rückgängigmachen einer Dateilöschung. **lsdel** liefert dazu die Inode-Nummern der gelöschten Dateien. Da hier die Dateinamen nicht angezeigt werden (sie sind nicht Teil des Inodes), muss man sich am Löschmodum, dem Dateibesitzer oder Größe den *passenden* Inode ermitteln. Zumeist wird **debugfs** aber eher dazu benutzt, um sich bestimmte Strukturen anzusehen und Informationen herauszuholen, die über andere Programme nicht verfügbar sind.

tune2fs hingegen erlaubt bei einem **ext2**-Dateisystem gewisse Optimierungen vorzunehmen, indem man bestimmte Parameter des Systems ändert. Hierzu zählt z. B. die Anzahl der **mount** oder die maximale Zeitspanne, nach denen ein File-System-Check erzwungen wird. Auch für das Reiser-Dateisystem gibt es mit **reiserfstune** und **debugreiserfs** entsprechende Lösungen.

3.3 Kommandos, Programme, Prozesse

Um es vorweg zu nehmen: Einen Unterschied zwischen einem *Programm* und einem *Linux-Kommando* gibt es bei Linux nicht. In der Regel bezeichnet man die von Linux zur Verfügung gestellten Programme als *Kommandos*, während bei einem vom Benutzer oder Drittanbietern erstellten Programm häufiger der Begriff *Programm* oder *Applikation* verwendet wird. Von Kommandoprozeduren und wenigen Kommandos abgesehen (hierzu gehört z. B. `cd`), welche die Shell selbst behandelt, ruft diese zur Durchführung der Kommandoaufgabe ein Programm mit dem Namen des Kommandos auf, welches die gewünschte Funktion ausführt.

So hat auch ein Benutzerprogramm, sofern es sich an die übliche Kommandosyntax hält (z. B. `>-<` kennzeichnet Parameter als Option), das Aussehen eines Linux-Kommandos. Es wird wie ein Linux-Kommando durch die Angabe des Programmnamens bzw. durch Angabe des Programmdateinamens aufgerufen. Da die Shell, bei entsprechend aufgesetztem Suchpfad, darüber hinaus die Programmdatei zuerst im aktuellen Benutzerverzeichnis sucht (s. hierzu Abschnitt 3.1.6), wird bei Namensgleichheit von Benutzerprogramm und Linux-Kommando das Benutzerprogramm ausgeführt. Hierbei ist zu beachten, dass einige Linux-Kommandos keine echten Programme sind, sondern *Kommandoprozeduren*, welche ihrerseits andere Linux-Kommandos aufrufen und sich darauf verlassen, dass die Linux-Version verwendet wird. Aus diesem Grund sollte der Benutzer z. B. einige Programmnamen vermeiden, da sie von der Shell intern verwendet werden. Hierzu gehören z. B. `break`, `cd`, `continue`, `echo`, `eval`, `exec`, `exit`, `export`, `expr`, `hash`, `login`, `newgrp`, `pwd`, `read`, `readonly`, `return`, `set`, `shift`, `test`, `times`, `trap`, `type`, `ulimit`, `umask`, `unset`, `wait`. Das Spektrum ist abhängig von der verwendeten Shell. Deren interne Kommandos lassen sich bei der `bash` per `enable` abfragen.

Ein **Prozess** ist unter Linux eine selbstständig ablauffähige Verwaltungseinheit des Betriebssystems, die ein Programm ausführt, so als ob der Prozessor nur diesem Programm zur Verfügung stünde. Die Emulation der *Pseudoprozessoren* der einzelnen Prozesse durch zeitlich verzahnte Ausführung ist Sache der Ablaufsteuerung des Betriebssystems.

Unter Linux und in praktisch allen aktuellen Unix-Systemen gibt es neben den Prozessen auch so genannte *Threads*. Sie werden auch als *light-weight*, d. h. *leichtgewichtige Prozesse* bezeichnet. Sie erlauben Prozesse nochmals in Teilprozesse zu untergliedern, die parallel ablaufen können. Alle Threads eines Prozesses laufen in einem einheitlichen Adressraum und haben eine Reihe gemeinsamer Ressourcen. Das Thread-Konzept gestattet die Synchronisation der Threads (eines Prozesses). Der Vorteil von Threads gegenüber mehreren getrennten Prozessen besteht in einer geringeren Prozessumschaltzeit und den gemeinsam zugreifbaren Ressourcen. Die Nutzung von Threads bei der Programmierung von Applikationen geht jedoch recht langsam voran. Dies liegt u. a. daran, dass bei Verwendung von Threads spezielle Programmierrichtlinien zu beachten sind und die Fehlersuche in solchen Programmen schwieriger ist als ohne Threads.



3.3.1 Prozesskenndaten

Man kann den Prozess vereinfacht als das eigentliche Programm im Sinne einer Programmiersprache und einer Programmumgebung betrachten. Zu dieser Programmumgebung gehören u. a. Speicherbelegung, Registerinhalte, geöffnete Dateien, das aktuelle Verzeichnis und die für den Prozess sichtbaren Umgebungsvariablen (wie z. B. HOME und PATH). Die Menge der Umgebungsvariablen wird auch als *Environment*, die Variablen als *Environment-Variablen* bezeichnet.

Der Adressraum eines Prozesses (der zu einem Prozess gehörige logische Speicherbereich) unterteilt sich in Benutzer- und Systemdaten. Zu den Systemdaten gehören

- ▶ prozessspezifische Systemdaten
- ▶ Systemkeller (für jeden Prozess)

Der Benutzeradressraum untergliedert sich in drei getrennte Bereiche, unter Linux **Segmente** genannt:

- ▶ Das **Textsegment**, in dem der Programmcode liegt und welches, soweit möglich, schreibgeschützt ist und, falls es keine Modifikationen an sich selbst vornimmt, auch mehrfach (von mehreren gleichen Prozessen) benutzt werden kann: Im Fall von *Shared Libraries* (entsprechend programmierten Programmbibliotheken) können auch mehrere unterschiedliche Prozesse diese gemeinsam (*shared*) benutzen.
- ▶ Das **Datensegment**, in dem alle anderen Benutzerdaten des Prozesses liegen: Dieses Segment wird nochmals unterteilt in einen initialisierten und einen nicht initialisierten Datenbereich. Letzterer wird auch *bss-Segment* genannt.
- ▶ Das **Kellersegment**, in dem der Benutzerkeller und die Verwaltungsdaten liegen (englisch: *stack segment*)

Hierzu kann ein vierter Segmenttyp hinzukommen:

- ▶ **Shared-Memory-Segmente**: Dies sind Speicherbereiche (Datensegmente), die der Prozess mit anderen Prozessen teilt und auf die alle zugreifen können.

Ein Prozess kann die Größe seines Keller- und Datensegmentes in der Regel bis zu einem systemspezifischen Limit dynamisch vergrößern. Das Kommando **size** gibt über die Anfangsgröße dieser Segmente eines Programms Auskunft.

Ein Teil der Kenndaten eines Prozesses wird vom **ps**-Kommando im ausführlichen Format (Option **-l**) angezeigt. Hierzu gehören z. B.

- ▶ die Prozessnummer des Prozesses und des Vaterprozesses,
- ▶ die Benutzernummer und Gruppennummer, unter welcher der Prozess abläuft,
- ▶ die Priorität des Prozesses,
- ▶ die physikalische Adresse des Prozesses im Hauptspeicher oder die Blockadresse des Prozesses im Auslagerungsbereich,
- ▶ der Prozesszustand,
- ▶ die Dialogstation, von welcher der Prozess gestartet wurde, und
- ▶ die vom Prozess verbrauchte Rechenzeit.

Ein Beispiel der Ausgabe des **ps**-Kommandos ist auf Seite 388 zu finden.

Die Prozessnummer (PID)

Intern, und bei Hintergrundprozessen auch für den Benutzer sichtbar, wird ein Prozess durch eine *Prozessnummer*, kurz *PID* (*Process Identification*) benannt. Diese *PID* wird vom System fortlaufend vergeben und ist systemweit eindeutig, d.h. es wird sichergestellt, dass nicht zwei gleichzeitig existierende Prozesse die gleiche Nummer erhalten. Startet ein Benutzer einen Hintergrundprozess durch ein angehängtes ›&x‹ hinter dem Kommando, so wird ihm bei erfolgreichem Start die *PID* des gestarteten Prozesses auf der Dialogstation ausgegeben. Diese Prozessnummer wird dann benötigt, wenn er diesen Prozess mit Hilfe des **kill**-Kommandos abbrechen möchte.

Die Prozessnummer des Vaterprozesses (englisch: *Parent Process Identification* genannt, kurz *PPID*) gibt an, von welchem Prozess der jeweilige Prozess gestartet wurde. Bei einem von der Dialogstation gestarteten Kommando z.B. ist dies die Prozessnummer des auf dieser Dialogstation aktiven Shell-Prozesses. Die Prozesse **vhand** mit der *PID* 0 und **init** mit der *PID* 1 haben eine besondere Bedeutung (s. Abschnitt 3.3.2).

Benutzer- und Gruppennummer eines Prozesses

Benutzer- und Gruppennummern werden vom System dazu verwendet, um Prozesse einem Benutzer oder einer Benutzergruppe zuzuordnen (z.B. für Abrechnungszwecke), und um die Zugriffsrechte des Prozesses auf Dateien und andere Objekte (z.B. Shared Memory) zu überprüfen. Ein Prozess besitzt zwei Arten von Benutzer- und Gruppennummern:

- ▶ die *effektive Benutzer- und Gruppennummer* und
- ▶ die *reale Benutzer- und Gruppennummer*

Die *effektive* Benutzer- bzw. Gruppennummer wird für die Überprüfung der Dateizugriffsrechte verwendet. Die *reale* Benutzer- und Gruppennummer ist jeweils die Nummer, die der aufrufende Benutzer besitzt. Ist bei einem aufgerufenen Programm im Dateizugriffseintrag an der Stelle des *Ausführungsrechts* ›x‹ eingetragen, so wird beim Programmstart (Prozessgenerierung) die *effektive* und *reale* Benutzer- und Gruppennummer auf die jeweilige Nummer des aufrufenden Benutzers gesetzt.

Steht an der Stelle des Ausführungsrechts einer Programmdatei für den Dateibesitzer jedoch statt des ›x‹ ein ›s‹ (für *set user ID*), so wird die Benutzernummer des Programmdateibesitzers als *effektive* Benutzernummer eingesetzt, während die *reale* Benutzernummer die des Aufrufers bleibt. Auf diese Weise können Operationen an Dateien vorgenommen werden, welche dem Programmbesitzer gehören, auf die der aufrufende Benutzer jedoch eigentlich keine Zugriffsrechte hat. In der Regel wird dies dazu benutzt, um durch das Programm kontrolliert Veränderungen von geschützten Dateien vorzunehmen.

Wird neben dem *s-Attribut* für den Besitzer auch (oder) das *s-Attribut* für die Gruppe im Modus einer Programmdatei gesetzt, so wird entsprechend die Gruppennummer der Programmdatei (bzw. deren Besitzer) als *effektive Gruppennummer* bei der Programmausführung eingesetzt. Das *s-Attribut* wird auch als *Set-User-ID-Bit* (*Set-UID-Bit*) und *Set-Group-ID-Bit* (*Set-GID-Bit*) bezeichnet.

Ein Beispiel für die Anwendung des *Set-User-ID-Attributs* ist die Passwortdatei */etc/passwd*, auf die, um Missbrauch zu vermeiden, nur der Super-User Schreiberelaubnis besitzen darf. Da dort auch das vom Benutzer definierte und änderbare Passwort eingetragen wird, ist es notwendig, dass ein nichtprivilegiertes Benutzer, wenn auch kontrolliert, die Passwortdatei ändern kann. Dies geschieht durch das Programm `passwd`. Im Dateimodus (Dateizugriffsattribut) dieses Programms ist entsprechend das `s`-Attribut bei Besitzer und Gruppe gesetzt, so dass bei der Ausführung als *effektive Nummern* die Benutzer- und Gruppennummer des Super-Users eingesetzt wird und damit z.B. schreibend auf die Passwort-Datei zugegriffen werden kann (der Super-User `root` ist als Besitzer von */bin/passwd* eingetragen).

Da solche Programme potenziell ein Sicherheitsrisiko darstellen können, insbesondere wenn sie von fremden montierten Datenträgern stammen, erlaubt die `mount`-Option `-o nosuid`, beim Montieren eines Dateisystems anzugeben, dass bei allen Programmen, die von diesem Dateisystem herunter geladen werden, das Set-UID-Bit und das Set-GID-Bit automatisch gelöscht wird.

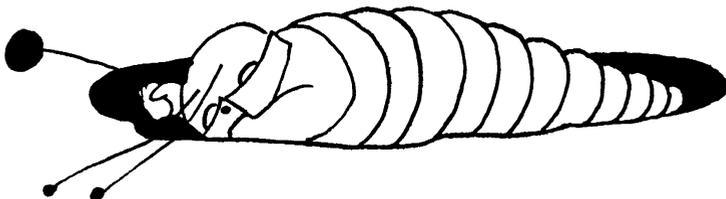
Beim Laden eines Programms wird dieses zunächst über den Hauptspeicher in den Auslagerungsbereich (*swap area*) geschaufelt und von dort gestartet.¹

Dieses Umkopieren bei jedem Programmstart kann durch das so genannte *save-text-Attribut* teilweise vermieden werden. Steht beim Ausführungsrecht ein `xt`, so ist im Dateimodus das *sticky bit* gesetzt (auch *save text bit* genannt). Hierdurch wird das Textsegment des Programms auch dann noch im Swap-Bereich belassen, wenn bei dem von mehreren Benutzern gemeinsam benutzbaren (*sharable*) Programm der letzte Benutzer seine Programm-Inkarnation beendet hat. Es braucht damit beim nächsten Start des Programms nicht erneut aus der Programmdatei gelesen, sondern kann direkt aus dem Swap-Bereich eingelagert werden. Das Setzen dieses Attributes ist, da der Swap-Bereich knapp sein kann, dem Super-User vorbehalten!

Prozesszustände

Ein Prozess kann sich in drei Grundzuständen befinden:

- ▶ **aktiv**, d.h. rechnend (*running*)
- ▶ **rechenwillig**, aber die CPU nicht besitzend (*suspended*)
- ▶ auf ein Ereignis **wartend** (*waiting*)



1. Dies gilt für die Standard-UNIX-Systeme von USL. Einige Systeme erlauben auch das segmentweise Laden von Programmen aus der Programmdatei heraus. Die Auslagerungen auf den Swap-Bereich (eigentlich den Paging-Bereich) erfolgt dann erst, wenn Segmente ausgelagert werden müssen.

Ein rechnender Prozess verliert immer dann die CPU, wenn er sie entweder freiwillig abgibt, um auf ein Ereignis (z. B. die Beendigung einer Ein- oder Ausgabe) zu warten, wenn ein Ereignis eintritt, auf das ein Prozess höherer Priorität gewartet hat, oder wenn seine Zeitscheibe abgelaufen ist. Wird der Prozess dabei durch einen anderen mit höherer Priorität verdrängt, so geht er in den Zustand *rechenbereit* über. Man bezeichnet diesen Prozess dann auch als *suspendiert*. Das `ps`-Kommando zeigt darüber hinaus weitere Zwischenzustände an, z. B. wenn ein Prozess beendet ist, aber noch nicht aus dem Speicher gelöscht wurde.

Die Prozesspriorität

Da unter Linux ständig mehrere Prozesse um die Zuteilung der CPU konkurrieren, muss im System eine Steuerung implementiert sein, nach der eine Prozessauswahl getroffen wird. Diese Steuerung wird *Scheduling-Algorithmus* genannt und die Terminierung, das zeitweise Verdrängen (*Suspendierung*), die Auswahl und Aktivierung des nächsten rechenbereiten Prozesses entsprechend *Scheduling*.

Das Linux-System verwendet beim Scheduling einen prioritätsgesteuerten Algorithmus. Es wird jeweils demjenigen rechenbereiten Prozess als nächstem die CPU zugeteilt, der die höchste Priorität besitzt. Dabei haben Prozesse, die sich im Systemmodus befinden, eine höhere Priorität als solche im Benutzermodus. Ein Prozess befindet sich dann im Systemmodus, wenn er eine Systemfunktion aufgerufen hat und diese noch nicht beendet ist.

Linux kennt zwei Prioritätsklassen:

- ▶ Time-Sharing-Prozesse
- ▶ Systemprozesse

Die Klasse der Systemprozesse – ist nicht weiter beeinflussbar und speziellen Systemprozessen vorbehalten. Prozesse aus einer der beiden anderen Klassen können nicht in die Systemprozess-Klasse verschoben werden. Sie haben Vorrang vor Time-Sharing-Prozessen. Es gibt ein Linux-Projekt und bereits spezialisierte Linux-Versionen, welche noch eine dritte Klasse von Realzeitprozessen bieten.¹ Sie erlaubt, zeitkritische Prozesse in einer eigenen Klasse laufen zu lassen, die sicherstellt, dass Realzeit-Prozesse Vorrang vor Time-sharing-Prozessen haben.

Time-Sharing-Prozesse

Die *Time-Sharing-Prozesse* bilden den Standardfall. Alle Linux-Kommandos und die meisten Applikationen laufen in dieser Prozessklasse. Das System versucht hier, die verfügbare Prozessorzeit möglichst gleichmäßig auf alle Prozesse zu verteilen.

Um eine einseitige Vergabe der CPU-Zeit an Prozesse hoher Priorität zu verhindern, wird die Priorität eines Prozesses in gewissen Zeitintervallen neu berechnet. In diese Berechnung gehen die im letzten Zeitintervall verbrauchte CPU-Zeit ein, die Größe des Prozesses und die Zeit, für die der Prozess verdrängt war (d. h. die CPU nicht erhielt).

Das `ps`-Kommando zeigt zwei Prioritäten an:

1. UNIX System V.4 kennt solche Realzeitprozesse.

► **Die aktuelle Priorität**

Dies ist die Priorität, die der Prozess augenblicklich besitzt und die bei der nächsten CPU-Vergabe für das Scheduling verwendet wird.

► **Die nice-Priorität**

Dies ist die Grundpriorität, die dem Prozess beim Start mitgegeben und als Steigerungswert bei der jeweiligen Prioritätsberechnung verwendet wird. Das Spektrum reicht von -20 (die höchst mögliche nice-Priorität) bis 19 (die niedrigste nice-Priorität).

Bei den als Priorität angegebenen Zahlen bedeutet eine hohe Zahl eine niedrige Priorität! Möchte man einen Prozess mit niedriger Priorität im Hintergrund ablaufen lassen, so kann man ihn mit dem **nice**-Kommando starten. Der Super-User kann als Priorität einen negativen Wert angeben und damit dem gestarteten Prozess eine höhere Priorität verleihen.

Prozessauslagerung (Swapping und Paging)

Bevor ein Prozess *rechnend* werden kann (d.h. die CPU zugeteilt bekommt), müssen die entsprechenden Code-/Text-, Keller- und Datensegmente im Hauptspeicher sein. Bei Paging-Systemen (wie Linux) reicht es, wenn einige Segmente des neuen Prozesses in den freien Hauptspeicher passen. Reicht der momentan freie Hauptspeicher dazu nicht aus, so müssen die Segmente anderer Prozesse auf Hintergrundspeicher ausgelagert werden, damit ausreichend freier Hauptspeicherplatz entsteht. Dieses Aus- und spätere Wiedereinlagern wird als *swapping* bezeichnet, der Bereich auf dem Hintergrundspeicher, auf den ausgelagert wird, als *Swap Space* (hier: *Swap-Bereich*) und das logische Gerät, auf das ausgelagert wird (in der Regel eine Magnetplatte), als *Swap Device*. Das *Swap Device* ist als Geräteknoten im Verzeichnis */dev* unter dem Namen */dev/swap* angelegt.

Das Auslagern von Prozessen geschieht durch einen eigenen Prozess, der in älteren Systemen den Namen **swapper** und die Prozessnummer **0** besaß. In neuen Unix-Systemen trägt dieser Ein-Auslagerungsprozess die Bezeichnung **vhand**. Unter Linux ist es der Prozess **kswapd**. Der wird beim Systemstart erzeugt und bleibt danach ständig aktiv.

Um zu verhindern, dass ein Prozess ständig nur ein- und ausgelagert wird, ohne ausreichend CPU-Zeit zu erhalten, ist der Auslagerungsalgorithmus so aufgebaut, dass ein Prozess nur dann ausgelagert wird, wenn er bereits eine gewisse Zeit im Hauptspeicher war.

Ob ein Prozess ausgelagert ist oder sich im Hauptspeicher befindet, ist an dem Prozesszustand (Spalte unter **F** beim ausführlichen **ps**-Kommando) zu erkennen. Ist ein Prozess im Hauptspeicher, so gibt die Prozessadresse seine Position im Hauptspeicher an; ansonsten steht hier die Adresse des Prozesses im *swap space*.

Bei einem virtuellen System – unter Linux der Standard – braucht nicht das gesamte Programm in den Hauptspeicher zu passen, sondern nur ein Teil. Das Programm wird hierzu in kleine Stücke unterteilt, so genannte Seiten oder englisch *pages*. In der Regel sind nur die Seiten (Code, Daten und Stack) aktuell im Hauptspeicher, in denen das Programm gerade arbeitet. Muss wegen Hauptspeicherknappheit ausgelagert werden, so werden nur einzelne Seiten (je 4 oder 8 KBytes groß) aus- und später wieder einge-

lagert. Beim Einlagern werden nicht alle ausgelagerten Seiten wieder hereingelesen, sondern nur die Seite, welche gerade benötigt wird. Man nennt diesen Mechanismus *Demand Paging*. Den Hintergrundspeicher, auf den Seiten ausgelagert und von dem sie später wieder eingelesen werden, nennt man *Paging Area*. Aus historischen Gründen wird er aber unter Linux *Swap-Space* genannt. Die Programme können bei diesem Verfahren wesentlich größer als der physikalisch vorhandene Hauptspeicher sein. Die maximale Programmgröße ist dabei vom virtuellen Adressraum der Maschine und dessen Implementierung im Linux-Kernel abhängig. Der übliche virtuelle Adressraum der Linux-Systeme liegt zwischen 4 und etwa 256 Gigabyte. Dabei ist zu beachten, dass der *Swap Space* auf der Platte größer als das größte Programm sein muss, da hier auch noch Segmente des Betriebssystemkerns und anderer Programme Platz haben müssen!

Die kontrollierende Dialogstation eines Prozesses

Der Name */dev/tty* steht innerhalb eines Prozesses als Pseudogerät für die *kontrollierende Dialogstation*. Diese Station entspricht der Standardein- und Ausgabe sowie der Standardfehlerdatei, sofern diese nicht umgelenkt sind. Das Pseudogerät */dev/tty* ist jedoch unabhängig von einer eventuellen Umsteuerung der Dialogstation zugeordnet, von der das Programm oder sein Vaterprozess aufgerufen wurde.

Die reale *kontrollierende Dialogstation* ist diejenige Dialogstation (*terminal file*), die ein Prozess als erste zum Lesen und (oder) Schreiben öffnet. Nur von dieser Dialogstation aus können über die entsprechenden Tasten die Signale an ihn geschickt werden. Alle Prozesse, die auf diese Weise eine gemeinsame *kontrollierende Dialogstation* besitzen, werden als *Prozessfamilie* oder *Prozessgruppe* bezeichnet. Über den Signal-Mechanismus ist es möglich, ein Signal an alle Prozesse der gleichen Prozessfamilie zu senden.

3.3.2 Prozesskommunikation, Prozesssynchronisation

Unter Linux kann ein Prozess weitere Prozesse anlegen, die dann asynchron von diesem abgearbeitet werden. In vielen Fällen wird jedoch ein solcher neu angelegter Prozess eine bestimmte Funktion ausführen, auf deren Beendigung der erzeugende Prozess wartet. Dies setzt eine Interprozesskommunikation voraus, die vom Linux-Kern implementiert wird.

Die Erzeugung eines neuen Prozesses geschieht durch den Systemaufruf *fork* (oder *vfork*).¹ Der neue Prozess ist dabei eine genaue Kopie des aufrufenden Prozesses, wobei selbst Daten, Befehlszähler, offene Dateien und Priorität identisch sind. Der aufrufende Prozess wird nun als *Vaterprozess* (englisch: *parent*) bezeichnet, der neu erzeugte als *Sohnprozess* (englisch: *child*).² *fork* ist ein Funktionsaufruf und liefert dem Vaterprozess bei erfolgreichem Start des Sohnprozesses dessen Prozessnummer (PID) zurück, während der Sohnprozess (beide stehen nun hinter dem *fork*-Aufruf) die Prozessnummer 0 zurückgeliefert bekommt. Von nun an laufen beide Prozesse unabhängig und asyn-

-
1. Daneben gibt es noch einen erweiterten Systemaufruf *clone*. Hierbei können die Prozesse noch gemeinsame Datenbereiche haben.
 2. Frauen mögen diese Übersetzung entschuldigen.

chron weiter, sofern nicht der Vaterprozess durch einen **wait**-Aufruf auf die Beendigung des Sohnprozesses wartet. Für die durch den **fork**-Aufruf geerbten Dateien besitzen Vater- und Sohnprozesse nur einen gemeinsamen Lese-Schreibzeiger. Liest oder schreibt einer der Beteiligten von einer solchen (auf eine solche) Datei, so wird der Zeiger für alle diese Prozesse verändert.

Soll nicht ein identischer Sohnprozess gestartet werden, sondern ein anderes Programm (der aufrufende Prozess soll jedoch weiterhin bestehen), so geschieht dies in zwei Schritten:

- a) Durch **fork** wird ein Sohnprozess als Kopie gestartet.
- b) Der Sohnprozess erkennt an der vom **fork**-Aufruf gelieferten Prozessnummer 0 seinen Sohnstatus und überlagert sich durch **exec** mit dem neuen Programm.

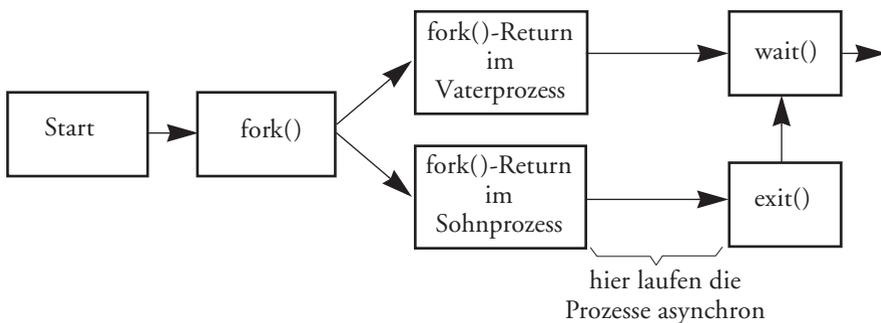


Abb. 3.11: Erzeugen eines Sohnprozesses über `fork()` und Warten auf dessen Beendigung

Beim **exec**-Aufruf werden die Segmente des aufrufenden Prozesses durch die des neu generierten ersetzt. Dem neuen Prozess kann dabei die aktuelle Systemumgebung des aufrufenden Prozesses mitübergewen werden. Die Prozessnummer des Prozesses bleibt erhalten.

Ein Prozess terminiert sich über einen **exit**-Aufruf oder durch einen Prozessabbruch. Dabei kann ein Statuswert (der so genannte *Exit-Status*) zusammen mit der Prozessnummer des beendeten Sohnprozesses an den eventuell auf die Terminierung des Sohnprozesses wartenden Vaterprozess weitergereicht werden. Ein solcher Wert wird auch dann übergeben, wenn sich der Sohnprozess nicht selbst terminiert hat, sondern durch den Benutzer oder einen Programmfehler abgebrochen wurde. Somit liefert jeder Prozess bei seiner Beendigung einen Wert zurück. Per Konvention ist dies 0, falls der Prozess erfolgreich seine Aufgabe durchführen konnte und ungleich 0 in allen anderen Fällen. Auf diese Weise ist eine, wenn auch sehr beschränkte Prozesssynchronisation möglich.

Stirbt ein Vaterprozess, bevor alle seine Sohnprozesse beendet sind, so erbt der Prozess **init** (mit der PID 1) die verbleibenden Sohnprozesse und wird damit deren Vaterprozess.

Wird ein Sohnprozess beendet, so werden alle Signale an den Prozess deaktiviert, alle noch offenen Dateien des Prozesses geschlossen, sowie der vom Prozess belegte Speicher und weitere Ressourcen werden freigegeben. Die Sohnprozesse des beendeten Pro-

zesses werden dem `init`-Prozess zugeordnet. An den Vaterprozess wird nun ein ›*Sohnprozess beendet*‹-Signal (SIGCHLD) geschickt. Der Prozesskontrollblock des beendeten Prozesses kann aber solange nicht aus dem Hauptspeicher geräumt werden, bis der Vaterprozess diese Terminierung zur Kenntnis genommen hat – entweder über ein Return aus der `wait`-Funktion oder über die Signalbehandlung des SIGCHLD-Signals. Dieser Zustand (der Sohn ist terminiert, kann aber noch nicht ausgeräumt werden) wird als *Zombie-Zustand* bezeichnet. Der Prozess erscheint hier als <defunct> in der Liste des `ps`-Kommandos. Der Prozess `init` wartet deshalb ständig auf die Terminierung eines Sohnprozesses. Ein Prozess kann mit der `wait`-Funktion nicht auf die Beendigung eines bestimmten Sohnprozesses warten, sondern nur auf die irgendeines Sohnprozesses. Er bekommt jedoch als Funktionsergebnis mitgeteilt, welcher Prozess terminiert hat. Daneben erhält er den Funktionswert des Sohnprozesses, auch *Exit-Status* genannt.

Der Vaterprozess kann sich auch durch ein Signal SIGCLD (siehe weiter unten) über die Beendigung eines Sohnprozesses informieren lassen.

Signale

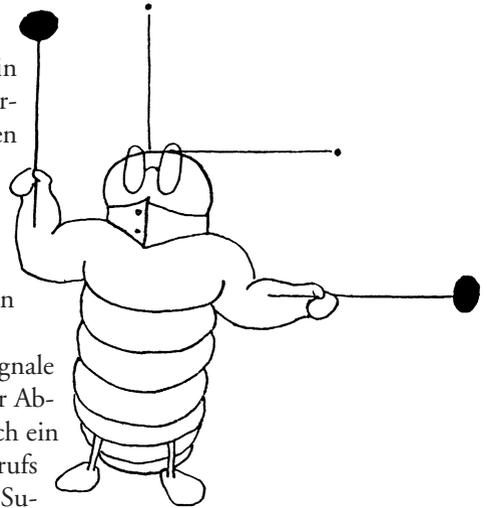
Eine weitere Möglichkeit der Prozesssynchronisation stellen **Signale** dar. Ein **Signal** ist ein asynchrones Ereignis und bewirkt eine Unterbrechung auf der Prozessebene. Signale können entweder von außen durch den Benutzer an der Dialogstation (z.B. durch Eingabe der <unterbrechung>-Taste) oder durch das Auftreten von Programmfehlern (Adressfehler, Ausführung einer ungültigen Instruktion, Division durch Null usw.) erzeugt werden.

Auch externe Unterbrechungen können Signale hervorrufen; z.B. das SIGKILL-Signal oder der Ablauf einer mit `alarm` gesetzten Zeitscheibe. Auch ein anderer Prozess kann, mittels des Systemaufrufs `kill (pid, signal_nr)` ein Signal senden. Nur der Super-User darf jedoch Signale an fremde Prozesse schicken.

Die in Tabelle A.15 auf Seite 866 mit *Core* markierten Signale führen, wenn sie vom Programm nicht explizit abgefangen werden, zu einem Programmabbruch.

Der Systemaufruf ›`signal (signal_nr, funktion)`‹ erlaubt einem Programm anzugeben, dass beim Auftreten des Signals `signal_nr` die Funktion `funktion` angesprungen werden soll. Ist (`funktion = SIG_IGN`), so wird keine Funktion angesprungen, sondern das Signal ignoriert.

Wird ein Signal an den Prozess mit der PID `0` gesendet, so wird es an alle Prozesse der gleichen Prozessfamilie gegeben. Das Signal SIGKILL (9) kann nicht abgefangen oder ignoriert werden und führt in jedem Fall zum Programmabbruch. Somit kann durch `kill 9 pid` ein Benutzer seine eigenen Prozesse abbrechen. Der Super-User ist dabei auch in der Lage, fremde Prozesse zu terminieren. Die Signale SIGUSR1 und SIGUSR2, welche keine feste Bedeutung haben, stehen für eine sehr einfache Interpro-



zesskommunikation zur Verfügung. Daneben sind auch die Realzeit-Signale zwischen SIGRTMIN und SIGRTMAX nutzbar.

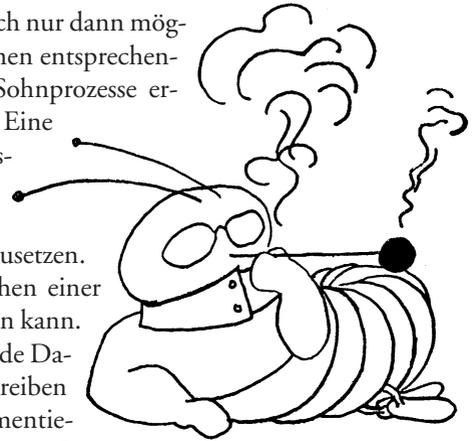
In einem Programm kann mit **sighold(signal)** ein *kritischer Abschnitt* begonnen und mit **sigrelse(signal)** beendet werden. In diesem Abschnitt wird jeweils das angegebene Signal, sofern es auftritt, zurückgehalten, bis der Abschnitt beendet ist.

Die üblichen Signale mit ihren Nummern und symbolischen Namen sind im Anhang A.6 in der Tabelle A.15 auf Seite 866 zu finden.

Pipes

Eine weitere Art der Kommunikation, über die zwei Prozesse Daten austauschen können, sind **Pipes**. Dies ist jedoch nur dann möglich, wenn ein Vaterprozess die Pipe durch einen entsprechenden Systemaufruf aufbaut und dann zwei Sohnprozesse erzeugt, die von ihm die Pipe-Dateien erben. Eine Pipe hat dabei nur eine Eingabe- und eine Ausgabeseite und ist somit unidirektional. Sollen Daten in beiden Richtungen ausgetauscht werden, so sind entsprechend zwei Pipes aufzusetzen.

Eine Pipe hat für den Prozess das Aussehen einer Datei, auf die er schreiben oder von der er lesen kann. Außer dem Positionieren (*seek*) kann darauf jede Dateioperation durchgeführt werden. Beim Schreiben sowie beim Lesen gibt es jedoch eine implementierungsabhängige Beschränkung der Art, dass eine Operation nicht mehr als die Größe des Pipe-Puffers (in der Regel 4 KB oder 8 KB) übertragen kann.



Named Pipes

Eine Pipe ist im Standardfall eine Art temporäre Pufferdatei, die nur solange lebt, wie einer der beteiligten Prozesse lebt. Sobald ein Prozess diese Pipe schließt und der andere weiterhin darauf zugreift, bekommt der zweite Prozess einen Fehler gemeldet. Bei dieser Art von Pipe müssen die beteiligten Prozesse auch stets entweder einen gemeinsamen Vaterprozess haben, der die Pipe aufgesetzt hat, oder sie stehen in einer Vater-Sohn-Beziehung, wobei ebenfalls der Vaterprozess die Pipe angelegt hat.

Die so genannte **named pipe** stellt eine Erweiterung dieses Mechanismus dar. Eine solche *named pipe* besitzt einen mit **mknod** angelegten Geräteeintrag vom Typ FIFO (für *First In First Out*) und hat damit einen entsprechenden externen Namen, unter dem sie angesprochen werden kann (beim **ls**-Kommando wird sie durch ein **p** als Typangabe angezeigt). Mittels dieses Namens können nun mehrere Prozesse miteinander kommunizieren, ohne einen gemeinsamen Vaterprozess zu haben. Dies wird in der Regel dazu benutzt, ein Serverkonzept zu realisieren. Der Dienstprozess liest dabei seine Aufträge aus der *named pipe*, während die Auftragsprozesse ihre Aufträge in die *named pipe* schreiben. Hierzu ist es notwendig, dass zwischen den Prozessen Einigkeit bezüglich der Größe des Auftrags textes besteht, damit der Dienstprozess den Auftrag mit einem Lesen aus der Pipe ausfassen kann. Ansonsten kann es zur Vermischung der ver-

schiedenen Eingaben kommen. Die Einträge von *named pipes* liegen entgegen der üblichen Konvention zumeist nicht im Verzeichnis */dev*, sondern im Verzeichnis des Serverprozesses. Die Serverprozesse **lpsched** und **cron** sind typische Beispiele für auf diesem Prinzip operierende Dienstprozesse.

Flexibler und mächtiger als *named pipes* sind die nachfolgend beschriebenen *Sockets* und *Streams*. Sie haben weitgehend *named pipes* bei der Kommunikation zwischen Clients und Servern abgelöst.

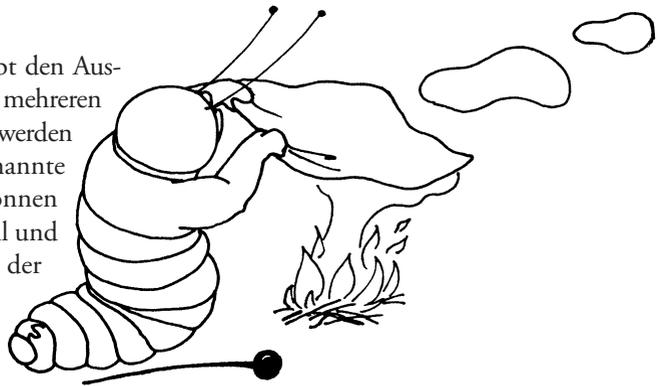
Mit UNIX System V.3 wurden vier neue Mechanismen zur Intertaskkommunikation verfügbar, welche alle auch in Linux vorhanden sind:

- ▶ Nachrichten (englisch: *messages*)
- ▶ Semaphore
- ▶ Speicher, auf den mehrere Programme zugreifen können (*Shared Memory*)
- ▶ Streams und
- ▶ Sockets (aus dem Berkeley-System übernommen)

ipcs erlaubt, den Implementierungs- bzw. Generierungsstand dieser Mechanismen abzufragen. Der Zugriffsschutz erfolgt bei allen vier Mechanismen über ein Verfahren, das weitgehend dem Zugriffsschutz von Dateien entspricht. Auch hier können für jedes Element die Zugriffsrechte für Besitzer (Erzeuger), die Benutzer der gleichen Gruppe sowie alle anderen Benutzer angegeben als auch Lese- und/oder Schreibrecht vorgegeben werden. Das Bit für **x** (*execute*) im Moduswort hat dabei keine Bedeutung.

Nachrichten (Messages)

Der *message*-Mechanismus erlaubt den Austausch von Nachrichten zwischen mehreren Programmen. Diese Nachrichten werden an Nachrichtenspeicher, so genannte **message queues** geschickt und können von dort abgeholt werden. Anzahl und Größe der Speicher werden bei der Systemgenerierung festgelegt. Eine Nachricht besteht aus dem Nachrichtentext und einem Nachrichtentyp.



Die Nachrichtentypen haben keine feste Bedeutung, sondern es ist dem Programmierer überlassen, ihnen Funktionen zuzuordnen. Ein Prozess kann nun eine Nachricht aus einer *message queue* anfordern und dabei einen Nachrichtentyp vorgeben. Fehlt die Angabe eines Typs, so wird die nächste in der Warteschlange vorhandene Nachricht zurückgegeben; ansonsten werden die Nachrichten in der Reihenfolge ihres Eintreffens ausgegeben. Ein Prozess kann beim Anfordern einer Nachricht angeben, ob er suspendiert werden will, sofern noch keine Nachricht vorliegt, oder ob der Funktionsaufruf sogleich zurückkehrt und diesen Umstand durch einen Fehlercode anzeigt.

Semaphore

Semaphore sind Zustandsvariablen. Der Zugriff auf sie ist nur mittels spezieller Funktionen (Betriebssystemaufruf **semop**) möglich. Die hier implementierten Semaphore sind nicht binär, sondern können mehrere Werte annehmen. Prozesse können den Wert von Semaphoren abfragen oder darauf warten, dass ein Semaphore einen vorgegebenen Wert annimmt. Mit einer Operation können auch Funktionen auf mehrere Semaphore zugleich ausgeführt werden. In der Regel wird man Semaphore zur Synchronisation beim Zugriff auf kritische Betriebsmittel verwenden.

Gemeinsamer Datenspeicher (Shared Memory)

Der *Shared-Memory*-Mechanismus erlaubt mehreren Prozessen, auf einen Speicherbereich gemeinsam zuzugreifen. Es handelt sich dabei um Datenspeicher. Um auf einen gemeinsamen Datenspeicherbereich zugreifen zu können, muss zunächst ein Prozess diesen Speicherbereich anlegen. Danach müssen alle Prozesse, die Zugriff darauf haben möchten, den Speicherbereich in ihren Adressraum einfügen (Funktion **shmat** für *shared memory attach*) und angeben, in welchen Adressbereich der Speicher abgebildet werden soll. Hierbei ist auch die Art des gewünschten Zugriffs (Lesen und/oder Schreiben) anzugeben. Danach kann bis zu einem Lösen (Funktion **shmdt**) der Speicherzuordnung der Speicherbereich wie ein normaler Datenspeicher behandelt werden.

Die Implementierung von *Shared Memory* ist stark von der Struktur der vorhandenen Speicherverwaltungseinheit (*memory management unit*) und von der Kernel-Konfiguration abhängig, so dass Größe und Stückelung solcher Speicherbereiche variieren können. Der Anwendungsprogrammierer sollte aus diesem Grund den Mechanismus mit Vorsicht verwenden, wenn sehr große Bereiche als Shared-Memory verwendet werden sollen.

Gemeinsamer Programmspeicher – Shared Libraries

Der Mechanismus der *Shared Libraries* ist inzwischen weit verbreitet und für zahlreiche Bibliotheken der Standard. Er bietet zwar keine Möglichkeit der Kommunikation zwischen Prozessen, soll hier jedoch auch kurz erwähnt werden.

Wird ein Programm oder Linux-Kommando mehrmals vom gleichen oder von unterschiedlichen Benutzern gestartet, so wird das Textsegment (Codesegment) nur einmal im Hauptspeicher gehalten, vorausgesetzt, dass der Code *reentrant* geschrieben ist, was in der Regel zutrifft. Beim Programmieren solcher Teile sind eine Reihe von Einschränkungen hinsichtlich globaler Variablen und deren Platzierung zu beachten. Das Konzept führt zu einer Platzeinsparung im Hauptspeicher und reduziert die Ladezeiten des Programms. Kommen jedoch gleiche Codestücke bzw. Prozeduren in unterschiedlichen Programmen vor, so arbeitete dieser Mechanismus bisher nicht. Bedenkt man, dass jedoch einige Funktionen (z. B. die des C-Laufzeitsystems) in fast allen Programmen vorkommen, so lohnt es sich, diese Funktionen nur einmal für alle sie nutzenden Programme im Hauptspeicher zu halten. Dies kann in Form von *Shared Libraries* geschehen. Beim Binden der Programme muss dieser Umstand explizit angegeben werden. Ähnlich wie bei den gemeinsamen Datenbereichen greifen dann alle so gebundenen Programme auf diese Funktionen zu. Allerdings ist hier, im Gegensatz zum *Shared-*

Memory-Mechanismus, kein explizites Anlegen des Speicherbereichs und Abbilden (*attach*) in den Programmadressraum notwendig. Diese Aufgabe übernimmt das System beim Start der Programme automatisch.

Die Speicherplatzersparungen, die auf diese Weise erzielt werden können, sind natürlich stark vom Umfang und der Anzahl der gemeinsam verwendeten Routinen abhängig, dürften jedoch am Beispiel der C-Grundbibliothek bei ca. 8 bis 16 KB je Programm liegen. Bei den wesentlich umfangreicheren X11-Bibliotheken kann die Einsparung bereits ein Megabyte betragen. Ein weiterer Vorteil der *Shared Libraries* liegt darin, dass auch Plattenplatz eingespart wird, da nun der entsprechende Code nicht mehr in den einzelnen Programmdateien vorhanden sein muss. Das Laden der Programme kann damit auch schneller erfolgen, da weniger Code zu laden ist.

Bei den *Shared Libraries* unterscheidet man nochmals zwischen statisch und dynamisch gebundenen Bibliotheken. Bei den statisch gebundenen Bibliotheken werden alle Referenzen bereits zur Bindezeit aufgelöst. Dies spart beim Programmstart Zeit und stellt weniger Anforderungen an die Programmierung und den Aufbau der Bibliothek. Bei den dynamisch gebundenen Bibliotheken erfolgt die Auflösung der Referenzen zum Zeitpunkt des Programmstarts oder erst zur Laufzeit beim ersten Ansprechen einer Referenz. Dies gestattet ein Austauschen der Bibliotheken, ohne dass dazu eine neue Version des Programms erstellt werden muss. Zudem müssen die Bibliotheken hier erst geladen werden, wenn sie (vom ersten) Programm angesprochen werden.

Streams

Die *Streams* wurden unter UNIX als ein Teil der als *Network Support Services* bezeichneten Funktionen bzw. Mechanismen zur Unterstützung von Rechnernetzen eingeführt. Sie sind ebenso in Linux vorhanden. Da der *Streams*-Mechanismus auch außerhalb von Netzdiensten als ein eleganter Mechanismus zur Kommunikation zwischen Programmen dienen kann und auch zur Abwicklung der Terminalprotokolle verwendet wird, soll er hier kurz erläutert werden.

Ein *Stream* ist ein Pseudotreiber im Betriebssystemkern, wobei der Begriff *Pseudo* hierbei verwendet wird, weil zunächst hinter dem Treiber kein physikalisches Gerät steht, sondern nur eine Reihe von Softwarefunktionen. Der Treiber stellt dabei eine Schnittstelle zwischen Benutzerprogramm und dem Betriebssystem zum Austausch von Daten(strömen) zur Verfügung und zwar in beiden Richtungen und voll duplex (d. h. in beiden Richtungen zugleich).

Der *Streams*-Treiber erlaubt dabei in wohl definierter und kontrollierter Weise den Aufbau eines Datenstroms sowie den eigentlichen Datentransfer. So können neben den Funktionen **putmsg**, **getmsg** und **poll**, welche nur auf *Streams* definiert sind, die Standard-Ein/Ausgabefunktionen wie **open**, **close** sowie **read**, **write** und **ioctl** auf *Streams* angewendet werden.

Ein mit Hilfe des *Streams*-Mechanismus aufgebauter Datenweg besteht aus folgenden Komponenten (siehe Abb. 3.12):

- ▶ dem Stream-Kopf (*Stream Head*)
- ▶ einem oder mehreren optionalen Verarbeitungsmoduln
- ▶ einem an den *Stream* angekoppelten Treiber

Der Treiber kann dabei ein Gerätetreiber für ein physikalisches Gerät oder wiederum ein Pseudotreiber sein. Eine mögliche Funktion eines Verarbeitungsmoduls kann z. B. in einem Netzwerk die Abarbeitung eines Netzprotokolls sein oder im Terminaltreiber die Behandlung von Zeilen und von Zeichen mit besonderer Bedeutung.

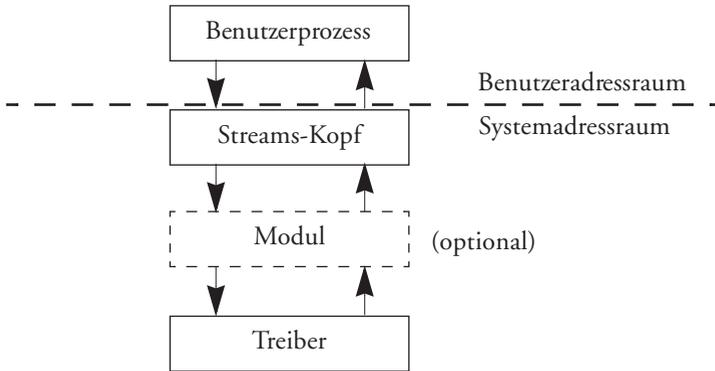


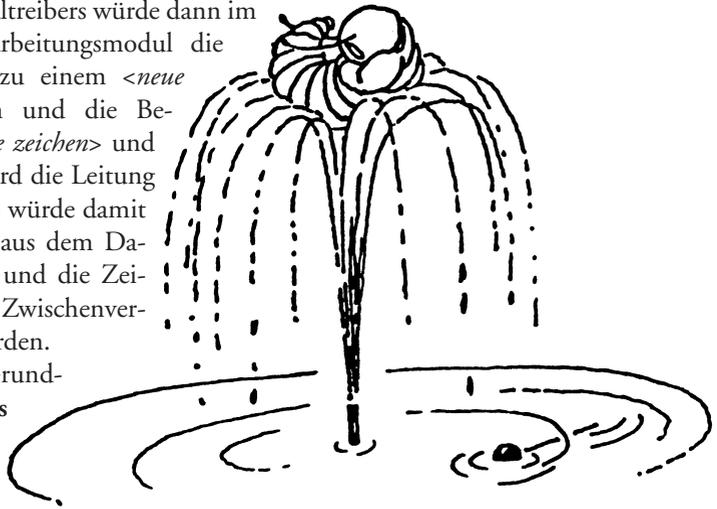
Abb. 3.12: Schemabild eines Streams

Eine wesentliche Eigenschaft des Streams-Mechanismus ist der, dass Verarbeitungsmodule dynamisch in den Verarbeitungsstrom eingeschaltet und wieder entfernt werden können

Am Beispiel des Terminaltreibers würde dann im *normalen Modus* der Verarbeitungsmodul die Zeichen der Eingabe bis zu einem *<neue zeile>*-Zeichen aufsammeln und die Behandlung der Tasten *<lösche zeichen>* und *<lösche zeile>* ausführen. Wird die Leitung in den *raw mode* versetzt, so würde damit dieser Verarbeitungsmodul aus dem Datenstrom (*Stream*) entfernt und die Zeichen der Eingabe ohne eine Zwischenverarbeitung weitergereicht werden.

Neben den genannten Grundfunktionen erlauben **Streams** das Multiplexen und Demultiplexen von Daten sowie asynchrone Ein/Ausgabe.

Im Kern stehen dem Stream-Treiber Funktionen wie z. B. das Anfordern und die Freigabe von Puffern, Datenflusssteuerungen und einem *Streams Scheduler* zur Verfügung.



Sockets

Der Mechanismus der *Sockets* wurde im Berkeley-Unix-System primär zur Kommunikation zwischen Prozessen über ein Rechnernetz eingeführt¹ ebenso wie der Mechanismus der *Streams* im USL-System. Die Programme des inzwischen zur Kommunikation zwischen den Systemen unterschiedlicher Rechnerhersteller zum Industriestandard gewordenen TCP/IP-Pakets stützen sich z. B. auf *Sockets* ab.²

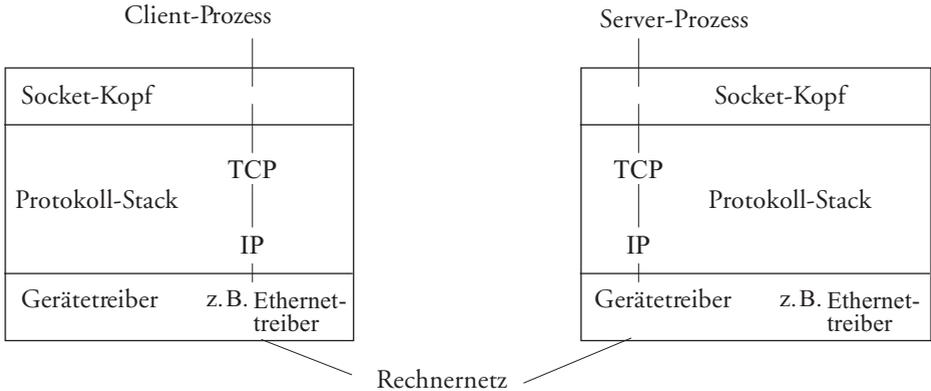


Abb. 3.13: Das Socket-Modell am Beispiel von TCP/IP

Ein *Socket* kann als Datenendpunkt zur Kommunikation zwischen Prozessen betrachtet werden. Der *Socket*-Mechanismus sieht dabei sowohl einen Datenaustausch auf dem lokalen als auch mit einem über ein Netz ansprechbaren Rechnersystem vor. *Sockets* sind wie *Streams* bidirektionale Datenpfade. Der vom Benutzer aus sichtbare Teil der Kommunikation besteht wie bei *Streams* aus drei Teilen:

- ▶ dem Socket-Kopf (*Socket Layer*)
- ▶ dem Protokollteil (*Protocol Layer*)
- ▶ dem Gerätetreiber (*Device Layer*)

Der *Socket*-Kopf bildet die Schnittstelle zwischen den Betriebssystemaufrufen und den weiter unten liegenden Schichten. Bei der Systemgenerierung wird festgelegt, welche Kombinationen von *Socket*, Protokoll und Treiber möglich sind.

Sockets mit gleichen Charakteristika bezüglich der Adressierung und des Protokoll-adressformats werden zu Bereichen, so genannten *Domains*, zusammengefasst. Die Linux *System Domain* dient dabei z. B. der Kommunikation zwischen Prozessen auf der lokalen Maschine, die *Internet Domain* für die Kommunikation über ein Netzwerk unter Verwendung des DARPA-Protokolls.

1. Erlauben jedoch wie *Streams* auch eine Kommunikation zwischen Prozessen auf dem gleichen Rechner.
 2. In System V.4 werden *Sockets* über Emulationsbibliotheken unterstützt, die ihrerseits auf *Streams* aufsetzen.

Sockets werden nochmals in unterschiedliche Typen untergliedert. Der so genannte *Stream-Typ* stellt eine virtuelle, gesicherte, verbindungsorientierte Kommunikation zur Verfügung (eine TCP-Kommunikation), der Typ *Datagram* eine Verbindung für Datagramme (eine UDP-Kommunikation); d.h. es wird eine Nachricht an einen oder mehrere Adressaten abgeschickt, der Empfang ist jedoch nicht gesichert und die Reihenfolge der Nachrichten ist nicht garantiert.

Eine Kommunikation läuft in der Regel so ab, dass ein Serverprozess mittels des Aufrufs **socket** einen Kommunikationspunkt aufbaut. Dabei können Typ und Bereich angegeben werden. Mit **bind** kann nun ein Name an den *Socket gebunden* werden. Ein Kundenprozess (*Client Process*) koppelt sich ebenfalls mit **socket** an einen (lokalen) Kommunikationspunkt (*Socket*) und beantragt mit **connect** einen Verbindungsaufbau zum *Socket* des Servers.

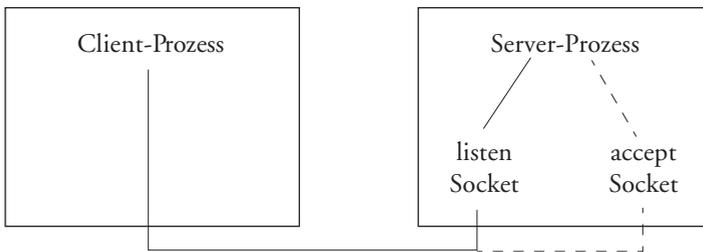
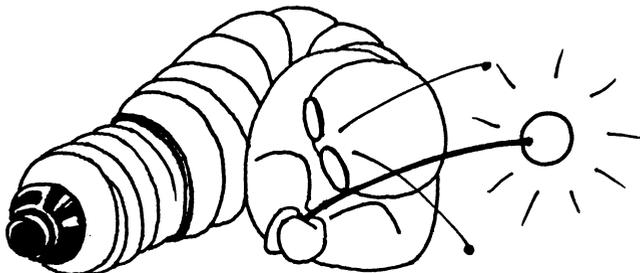


Abb. 3.14: Client-Server-Kommunikation

Der Serverprozess macht mit **listen** dem System bekannt, dass er Verbindungen akzeptieren will und gibt die Länge einer Warteschlange an. Mit **accept** wartet er darauf, dass ein *Client-Prozess* eine Verbindung anfordert.

Der **accept**-Aufruf liefert nach einem Verbindungsaufbau dem Server einen neuen *Socket-Deskriptor* (analog zu einem Dateideskriptor) für einen (anderen) *Socket* zurück, über den nun die Kommunikation mit dem *Client* erfolgen kann. Der Austausch von Daten ist danach mit **send** und **rcv** oder mittels **write** und **read** über diesen *Socket* möglich. Wie man in Bild Abb. 3.13 sieht, sind der *Socket*, an dem der Server-Prozess auf Verbindungen wartet, und der *Socket*, über den der Serverprozess nach einem Verbindungsaufbau mit dem *Client* kommuniziert, auf der Serverseite nicht identisch. Der Aufruf **shutdown** schließlich baut die Verbindung wieder ab.

Eine gute Beschreibung der *Sockets* ist in [Bach] zu finden. Die hier gegebene Erklärung und die Zeichnungen basieren auf dem Buch von Bach.



3.4 Reguläre Ausdrücke in Dateinamen und Suchmustern

Häufig möchte man Operationen wie z.B. Sichern, Löschen oder Sortieren auf eine Reihe von Dateien ausführen oder in Texten, statt nach einer festen Zeichenkette mit einem Zeichenmuster suchen, in dem noch gewisse Freiheitsgrade vorhanden sind. Für beide Anforderungen stehen unter Linux so genannte *reguläre Ausdrücke* zur Verfügung, die weitgehend einheitlich von den verschiedenen Linux-Programmen interpretiert werden. So erlauben die Shell und einige andere Programme, die mit Dateinamen operieren (z.B. **find**), *reguläre Ausdrücke* in Dateinamen; die meisten Editoren und Suchprogramme erlauben *reguläre Ausdrücke* in Suchmustern.

3.4.1 Metazeichen in regulären Ausdrücken

Ein *regulärer Ausdruck* ist eine Folge von *normalen Zeichen* und *Metazeichen*. Ein normales Zeichen (z.B. der Buchstabe `x`) steht für das entsprechende Zeichen bzw. den Buchstaben selbst. Ein *Metazeichen* – teilweise auch *Jokerzeichen* genannt – ist ein Zeichen, welches nicht das entsprechende Zeichen darstellt, sondern eine erweiterte Bedeutung besitzt. Das Fragezeichen `?` z.B. steht in der Angabe von Dateinamen für *Ein beliebiges (auch nicht druckbares) einzelnes Zeichen*.

Ein *regulärer Ausdruck* ist ein Muster, mit dem die in Frage kommenden Objekte verglichen werden (z.B. die Namen aller Dateien im aktuellen Verzeichnis oder der Text eines Zeilenbereichs bei einem Editorsuchkommando). Passt das Muster auf eines der untersuchten Objekte, so spricht man von einem *Treffer* (englisch: *match*). Häufig passen mehrere der untersuchten Objekte auf einen regulären Ausdruck. Bei dem Aufruf

```
rm ?ab?
```

z.B. wird die Zeichensequenz `?ab?` von der Shell als Muster für Dateinamen betrachtet und wie folgt interpretiert:

Setze statt des Parameters die Namen aller Dateien ein, die dem Muster ?ab? entsprechen. Dies sind alle Dateien, deren Namen vier Zeichen lang sind. Das erste Zeichen ist beliebig (erstes >?<). Das zweite Zeichen des Namens muss ein >a< und das dritte Zeichen ein >b< sein. Das vierte Zeichen darf wieder beliebig sein.

Meint man in einem solchen Ausdruck das Zeichen selbst und nicht die Metabedeutung, so muss man das Metazeichen *maskieren*. Dies kann durch das Voranstellen des Fluchtzeichens `\` erfolgen. Meint man das Fluchtzeichen selbst, so ist dies ebenfalls zu maskieren und muss dann als `\\` angegeben werden. Bei der Shell kann ein ganzer Ausdruck durch eine `"..."`- oder `'...'`-Klammerung maskiert werden. Bei den Klammern `"..."` findet keine Expandierung von Dateinamen mehr statt, wohl aber noch die Ersetzung von Shell-Variablen durch ihren Wert oder die Auswertung von Kommandos in ``...``-Klammern. Letztere ist identisch mit `$(...)`-Klammern.¹ Bei der `'...'`-Klammerung findet keinerlei Auswertung mehr durch die Shell statt.

1. Siehe hierzu die Erklärung in Kapitel 6.2.9 auf Seite 547.

Folgende Metazeichen stehen zur Bildung von Suchmustern zur Verfügung:

Tabelle 3.3: Metazeichen in Dateinamen und Suchmustern

Bedeutung	Metazeichen	
	bash im Dateinamen	im Suchmuster
Beliebiges einzelnes Zeichen	?	. (Punkt)
Beliebige Zeichenkette (auch die leere)	*	.*
Beliebige Wiederholung des vorangestellten Zeichens (auch keine)	<i>fehlt</i>	*
Beliebige Wiederholung des vorangestellten Zeichens (mindestens 1)	<i>fehlt</i>	+ ^a
0 oder 1 Wiederholung des vorangestellten Zeichens	<i>fehlt</i>	?
n bis m Wiederholungen des vorangestellten Zeichens ^b	<i>fehlt</i>	\{ n,m \}
Eines der Zeichen aus ...	[...]	[...]
Eines der Zeichen aus dem Bereich ...	[a-e]	[a-e]
Eines der Zeichen aus den Bereichen ...	[a-eh-x]	[a-eh-x]
Alle Zeichen außer ...	[! ...]	[^ ...]
Fluchtsymbol	\	\
Unterdrückung der Interpretation	'...'	
Unterdrückung der Dateinamen-Expansion	"..."	

a. Nur bei den Programmen **awk/gawk** und **grep**.

b. Ist nur \{ n \} angegeben, so ist damit ›Genau n mal‹ gemeint. Fehlt die Angabe minimal, so wird ›1‹ angenommen; fehlt maximal, so wird beliebig oft › ∞ ‹ angenommen. ›*‹ ist damit äquivalent zu: \{0,\}, ›+‹ ist äquivalent zu \{1,\}, und ›?‹ ist äquivalent zu \{0,1\}.

Beim Suchen mit regulären Ausdrücken wird versucht, eine möglichst lange Zeichenkette als Treffer zu bilden. In der Zeile

Mutter und Vater ...

würde zum Beispiel das Suchmuster ›**t.*r**‹ die Zeichenkette **ter und Vater** als Treffer haben und nicht das kürzere (auch passende) Textstück *ter* aus *Mutter*. Bei Dateinamen werden alle passenden Namen verwendet.

Die Maskierungszeichen \, ", ' werden von der Shell entfernt, bevor der Parameter dem aufgerufenen Programm übergeben wird. Beim Aufruf

```
rm "*"
```

bekommt das **rm**-Kommando also nur die Zeichenkette ***** als Parameter übergeben. Bei **rm ab?** wäre dies die Zeichenkette: **a|b|?** (jedes Kästchen ist ein Zeichen).

- Bei der Shell findet keine Expandierung der Dateinamen in der Umlenkungs-komponente eines Kommandos statt! Die Anweisung `>ls > * <` erzeugt demnach eine Datei mit dem Namen `>*<`.
- Bei der Shell wirkt ein mit Punkt beginnender Dateiname als Verdeckung des Dateinamens, so dass die Namen dieser Dateien nicht für die normale Jokerzeichen/Metazeichen-Erweiterung herangezogen werden – es sei denn, man gibt im Shell-Parameter explizit den Punkt vorne an. So setzt die Shell z.B. in der Folge `>ls .a* <` für `>.a* <` die Namen aller Dateien ein, welche mit `>.a <` beginnen.

Zum Suchen mit Zeichenketten in Texten gibt es einige Erweiterungen, die für Dateinamen nicht sinnvoll sind:

Tabelle 3.4: Zusätzliche Metazeichen im Suchmuster

Bedeutung	Metazeichen
Zeichenkette am Anfang der Zeile	<code>^muster</code>
Zeichenkette am Ende der Zeile	<code>muster\$</code>
Zeile bestehend aus ...	<code>^muster\$</code>
Zeichenkette am Anfang eines Wortes ^a	<code>\<muster</code>
Zeichenkette am Ende eines Wortes	<code>muster\></code>
Alternative: <code>muster_1</code> oder <code>muster_2</code>	<code>muster_1 muster_2</code>

- a) Ein *Wort* ist eine Folge von Buchstaben und Ziffern ohne Leerzeichen, Tabulatorzeichen oder Sonderzeichen wie `.`, `_` usw. darin.

Bei den Editoren (mit Ausnahme des `sed`, bei dem `>\n <` im Suchtext *neue Zeile* bedeutet) sowie bei den `grep`-Programmen ist das Zeilenende eine Grenze beim Mustervergleich. Es ist dabei nicht möglich, nach einem Muster zu suchen, welches sich über eine Zeilengrenze erstreckt! Das *<neue Zeile>*-Zeichen kann also auch nicht mit `>.<` oder `>*<` gefunden werden!

Shell-spezifische Metazeichen

Die `bash` und einige andere Shells (C- und Korn-Shell, ...) kennen noch zwei weitere Metazeichen in Dateinamen:

Tabelle 3.5: Zusätzliche Shell-spezifische Metazeichen in Dateinamen

- `~` steht für den Namen des HOME-Verzeichnisses des Benutzers (Aufrufers). In der Form `>~name <` wird das Login-Verzeichnis des angegebenen Benutzers eingesetzt.
- `{x,y,...}` Hierdurch werden mehrere Namen generiert – für jede durch Kommata getrennte Zeichenkette einen.
z.B. `>man.{1,10} <` → `>man.1, man.10 <`.

Beispiele für reguläre Ausdrücke in Dateinamen

- rm ?** → löscht alle Dateien des aktuellen Verzeichnisses, deren Namen genau ein Zeichen lang sind.
- ls /usr/gast/.*** → erstellt ein Inhaltsverzeichnis aller Dateien des Verzeichnisses */usr/gast*, deren Namen mit einem Punkt beginnen.
- cat *out*** → gibt den Inhalt aller Dateien des aktuellen Verzeichnisses auf die Dialogstation aus, in deren Namen *out* vorkommt. Dateien, deren Namen mit einem Punkt beginnen, werden nicht mit in den Vergleich einbezogen!
- rm mod.?** → löscht alle Dateien, deren Namen mit *mod.* beginnen und fünf Zeichen lang sind.
- rm \?!\?!** → löscht die Datei mit dem Namen *?!?!?*.
- rm '???'** → ist in der Wirkung äquivalent zu *rm \?!\?!?*
- ls *** → wird von der Shell zu *ls alt a1 neu* expandiert, wenn nur die Dateien *alt*, *a1* und *neu* im aktuellen Verzeichnis existieren.
- cp [a-m]* l** → kopiert alle Dateien des aktuellen Verzeichnisses, deren Namen mit *a*, *b*, ... bis *m* beginnen, unter dem gleichen Namen in das Verzeichnis *l*.
- find . -name '*c' -print** → ruft das *find*-Kommando auf. Durch *'...'* wird hier die Zeichenkette **c* vor der Shell-Interpretation geschützt und unverändert, jedoch ohne die Zeichen *'...'* als Parameter an das *find*-Kommando weitergereicht. **find** selbst interpretiert nun **c* als Metazeichen und ist damit eine Ausnahme unter den Linux-Kommandos.
- ls ~** → ersetzt *~* durch das HOME-Verzeichnis des aktuellen Benutzers, so dass beim Aufrufer *karl* z. B. *ls /home/karl/* herauskommen könnte.
- rm ~otto/abc** → ersetzt *~otto* durch das HOME-Verzeichnis des Benutzer *otto*, so dass z. B. *rm /home/otto/abc* herauskommen könnte.
- mkdir a{1,2,bc}** → wird zu *mkdir a1 a2 abc* expandiert.

Beispiele für reguläre Ausdrücke in Textmustern

Textmuster müssen in der Regel begrenzt werden. In diesen Beispielen geschieht dies durch die in den Editoren üblichen Zeichen */*. Bei den **grep**-Programmen werden sie in der Regel mit *" ... "* geklammert.

- /^Auf /** → sucht nach dem Wort *Auf*, welches am Anfang einer Zeile (*>^<*) steht. Eine Zeile, die nur aus dem Wort *Auf* ohne nachfolgendes Leerzeichen besteht, gilt hierbei nicht als Treffer. Das bessere Suchmuster wäre hier */^Auf>/*

- `/^Ende$/` → sucht das Wort *Ende*, welches am Anfang einer Zeile beginnt und an deren Ende abschließt.
- `/^$/` → steht für eine leere Zeile.
- `/[aA]nfang/` → sucht nach der Zeichenkette *anfang* oder *Anfang*.
- `/[0-9][0-9]*/` → meint eine beliebig lange Ziffernfolge.
- `/\|` → meint das Zeichen `|`.
- `/\|` → meint das Zeichen `|`. Da es hier Begrenzerfunktion hat, muss es durch `|` maskiert werden.
- `/a\.b*/` → steht für die Zeichenkette *a.b**.
- `/a.b*/` → sucht nach einer Zeichenkette, welche mit *a* beginnt. Dieser darf sich ein beliebiges weiteres Zeichen anschließen, dem können 0 (d.h. keines) oder mehr *b*-Zeichen folgen. Wird mindestens ein weiteres *b* verlangt, so muss das Muster *a.bb** lauten!
- `/a?$/` → sucht nach der Zeichenkette *a?* am Ende einer Zeile. Das Fragezeichen hat hier, im Gegensatz zur Shellinterpretation keine Sonderfunktion.
- `/<ein/` → sucht nach der Zeichenfolge `<ein` am Anfang eines Wortes.
- `/<ein>/` → meint *ein* als Wort. Das Suchmuster `/ ein /` würde das Wort *ein* am Anfang einer Zeile oder an deren Ende nicht finden! Da ein Wort eine Folge von Buchstaben und Ziffern ist, trifft das obige Muster auch auf eine Zeichenkette wie *ein&* zu.

Bei den Operationen *Suchen-und-Ersetzen* der Editoren **ed**, **vi/vim**, **ex** oder **emacs** unterscheidet sich die Interpretation von Zeichen als Metazeichen im Suchmuster von der im Ersetzungsteil. Die Syntax des *Suchen-und-Ersetzen*-Befehls des **ed** sieht z. B. wie folgt aus:

```
[bereich]s[suchmuster]ersetzungsmuster/
```

Im Suchmuster haben die oben genannten Metazeichen die angegebene Bedeutung. Hinzu kommt noch die *Metaklammer* `(...)`. Sie klammert im Suchmuster einen Teilausdruck. Den *n*-ten Teilausdruck des Suchmusters, bzw. das darauf passende gefundene Teilmuster, kann man im Ersetzungsteil durch `\n` mit *n* = 1, ... angeben, wie die nachfolgenden Beispiele demonstrieren:

`s/(Linux)\-Version/\1/` → ersetzt (z. B. in **ed**) *Linux-Version* durch *Linux*. `>Version<` im Text allein stehend bleibt hierbei unverändert.

`s/([Ee])nviroment/\1nvironment/` → ersetzt *enviroment* durch *environment* und *Enviroment* durch *Environment*.

`s/([A-Z])\.\([0-9]\)/\2.\1/` → ersetzt z. B. `>B.3<` durch `3.B<`. Während hier der Punkt im Suchteil noch durch `|` maskiert sein muss, um nicht als

›Jedes beliebiges einzelne Zeichen‹ interpretiert zu werden, hat er im Ersetzungsteil keine Metazeichen-Bedeutung mehr.

Im Ersetzungsteil haben nur die in Tabelle 3.6 aufgeführten Zeichenfolgen eine erweiterte Bedeutung. Alle anderen Zeichen (z. B. ›*‹, ›.‹, ›[‹) stehen im Ersetzungsteil für das entsprechende Zeichen selbst.

Tabelle 3.6: Metazeichen im Ersetzungsteil

Bedeutung im Ersetzungsteil	Metazeichen
der n -te gefundene Teilausdruck	$\backslash n$
die zuletzt verwendete Ersetzungszeichenkette	\backslash
die gesamte gefundene Zeichenkette	$\&$

Die nachfolgenden Zeilen zeigen weitere Beispiele für Such- und Ersetzungsmuster am Substitutionskommando z. B. von **vim** oder **ex**.

`s/[0-9][0-9]*/&./` → setzt z. B. in **vim** hinter eine Ziffernfolge einen Punkt.

`s/#$/&&&/` → ersetzt ›#‹ durch ›##‹ und ›\$‹ durch ›\$\$‹.

`s/und/\&/` → ersetzt ›und‹ durch das Zeichen ›&‹.

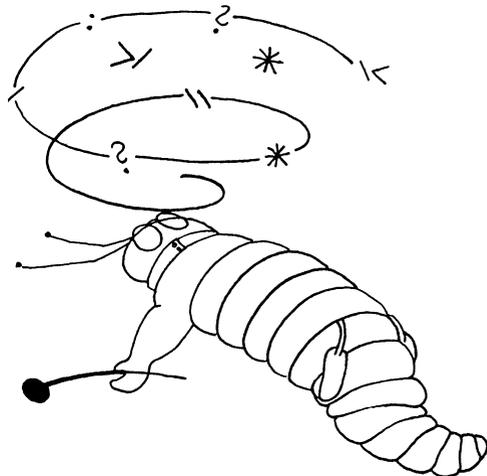
`s/(Linux)\-(System\)/\2 \1/` → ersetzt ›Linux-System‹ durch ›System Linux‹.

`s/Maurel/Maurer/`

`s/Hans Maurel/Hans -/` → ersetzt zunächst ›Maurel‹ durch ›Maurer‹ und danach ›Hans Maurel‹ durch ›Hans Maurer‹ (z. B. in **ex** oder **vi**).

`1,$s/Tilde/\~/g` → ersetzt im ganzen Text alle ›Tilde‹ durch ›~‹. Hier muss im Ersetzungsteil die Tilde durch ›\‹ maskiert werden.

`1,$s/Tilde/& (\~/)g` → ersetzt im ganzen Text alle ›Tilde‹ durch ›Tilde (~)‹.



3.4.2 Tabelle der regulären Ausdrücke in Linux

Leider kennen nicht alle Programme, die entsprechende Suchfunktionen benutzen, alle oben angeführten Metazeichen/Jokerzeichen. Die nachfolgende Tabelle versucht deshalb, eine Übersicht der einzelnen Möglichkeiten zu geben. Es sei jedoch ausdrücklich darauf hingewiesen, dass sich dies von Linux-System zu Linux-System und teilweise auch von Shell zu Shell unterscheiden kann!

Tabelle 3.7: Die Verarbeitung von Metazeichen der wichtigsten Linux-Kommandos

Metabedeutung	Dateinamen	sed, grep	awk	ed	ex, vi
Funktion im Suchmuster					
beliebiges Zeichen	?
belieb. Zeichenkette (auch leere)	*	.*	.*	.*	.*
beliebige Wiederholung (auch keine)	fehlt	*	*	*	*
beliebige Wiederholung (mindestens 1)	fehlt	fehlt	+	\{1\}	fehlt
keine oder 1 Wiederhol.	fehlt	fehlt	?	\{0,1\}	fehlt
n -malige Wiederholung	fehlt	fehlt	\{ n \}	\{ n \}	fehlt
n - bis m Wiederholung	fehlt	fehlt	\{ n, m \}	\{ n, m \}	fehlt
Zeichen aus ...	[...]	[...]	[...]	[...]	[...]
kein Zeichen aus ...	[!...]	[^...]	[^...]	[^...]	[^...]
am Zeilenanfang		^muster	^muster	^muster	^muster
am Zeilenende		muster\$	muster\$	muster\$	muster\$
am Wortanfang	xyz*				\<muster
am Wortende	*xyz				muster\>
$a1$ oder $a2$	fehlt		$a1 a2$	$a1 a2$	fehlt
Funktion im Ersetzungsmuster					
n -ter Teilausdruck		\ n ¹	\ n ¹	\ n	\ n
gefundene Zeichenkette		& ζ ¹	& ζ ¹	& ζ	& ζ
vorhergehende Ersetzung					~

$\text{>}a1 \text{ oder } a2\text{<}$ bedeutet *›Entweder ein Text, der auf den (regulären) Ausdruck $a1$ passt oder einer, der auf den Ausdruck $a2$ passt‹.*

1. Nicht bei **grep** und **egrep**.

3.5 Internationalisierung und lokale Anpassungen

Unix – und als Nachimplementierung auch GNU/Linux – sind originär englischsprachige bzw. amerikanische Systeme. Dies zeigt sich nicht nur an den Namen der Kommandos und Begriffe, sondern auch an dem zunächst ausschließlich verwendeten ASCII-Zeichensatz (*American Standard Code for Information Interchange*). Bei Unix wurde erst relativ spät die Möglichkeit für andere oder erweiterte Zeichensätze, anderssprachige Meldung oder der Mehrsprachigkeit angegangen. Dies erfolgte in mehreren, relativ langwierigen Iterationen.

Die Linux-Entwicklung startete später und übersprang einige dieser Iterationen. Eine vollständige Internationalisierung oder Mehrsprachigkeit ist aber noch nicht erreicht und der Realisierungsstand ist recht unterschiedlich für verschiedene Teilsysteme. So findet man in GNOME 1.4 in deutschen Systemen teilweise eine wilde Mischung aus Deutsch und Englisch bzw. Amerikanisch, es geht jedoch Schritt für Schritt weiter.

In den letzten Jahren wurde hier sehr viel erreicht. Dies ist durchaus anzuerkennen, zumal das Thema objektiv gesehen ausgesprochen komplex ist. Neben der reinen Übersetzung einzelner Texte müssen der verwendete Zeichensatz (ASCII, ISO-8859-x, UTF-8), die zur Darstellung eingesetzten Schriften (Fonts) und teilweise sehr unterschiedlichen andere Konventionen berücksichtigt werden – z. B. das Schreiben von links nach rechts oder von rechts nach links. Wir fokussieren uns bei der Betrachtung hier ganz egozentrisch auf die besonderen europäischen oder deutschen Bedürfnisse und Anpassungen.

Was ist Internationalisierung und Lokalisierung?

Unter *Internationalisierung* versteht man, dass ein Programm so programmiert ist, dass es Meldungen technisch in mehreren Sprachen ausgeben und Eingaben sprachabhängig verarbeiten kann. Dafür stützt es sich in der Regel auf spezielle Bibliotheksfunktionen, die ihm den größten Teil der Arbeit abnehmen, indem sie z. B. bei Ausgaben auf sprachabhängige Meldungsdateien zurückgreifen, beim Sortieren von Texten die Sortierreihenfolge der eingestellten Sprache berücksichtigen und z. B. bei der Ausgabe eines Datums die in der jeweiligen Sprach- oder Landeskonvention übliche Datumsdarstellung erzeugen. Dies wird unter Linux als *Internationalization* oder kurz **I18n** bezeichnet (da in dem Wort *Internationalization* zwischen dem **I** und dem abschließenden **n** 18 Zeichen liegen). Man nennt dies auch *National Language Support* – oder kurz **NLS**.

Der zweite Schritt der Internationalisierung nach der entsprechenden Programmierung ist die Erstellung der übersetzten Meldungen für die unterstützten Sprachen – Ausgangsbasis ist zumeist Englisch.¹ Ist keine spezielle Sprachangabe eingestellt, so wird in aller Regel Englisch verwendet. Mit entsprechenden Werkzeugen, welche abhängig von der verwendeten Programmiersprache und anderen Techniken sind, kann auch ein Anwender weitere Sprachen/Übersetzungen hinzufügen. Dies wird unter Linux als *Localization* bzw. *Localization* bezeichnet – oder kurz **L10n** (hier stehen 10 Zeichen zwischen dem **L** und dem **n** von *Localization*).

1. Ein in der Linux-Welt sehr willkommener Beitrag, der auch von Menschen ohne Programmiererfahrung erbracht werden kann, ist die Anfertigung weiterer Übersetzungen.

3.5.1 Einstellungen zur Lokalisierung

Die meist benutzte Stelle für die Einstellungen für Sprache und andere lokale Konventionen ist die Umgebungsvariable `LANG` (angezeigt mit `>echo $LANG<`). In ihr lässt sich die Sprache und das Land einstellen. Die allgemeine Syntax für `LANG` lautet:

```
sprache_land[,zeichensatz][@modifikation]
```

Die hinteren Teile sind dabei optional. Die Komponenten (außer *sprache*) müssen dabei jeweils durch die oben angezeigten Trennzeichen (`_`, `,`, `@`) eingeleitet werden.

Die Komponente *sprache* wird durch den zwei Buchstaben langen Code nach ISO 639-1 angegeben, das Land entsprechend dem ISO-3166-Code.¹

Für Deutschland ist die typische Belegung von `LANG` z. B. `>de_DE<`, während es für Österreich `>de_AT<` und die Schweiz `>de_CH<` ist. Für die französische Schweiz wäre es `>fr_CH<`.

Der nächste wesentliche Parameter ist der geeignete Zeichensatz. Er muss die in der gewählten Sprache benötigten Zeichen enthalten – für Deutsch z. B. die Umlaute und das `>ß<`. Für die meisten westeuropäischen Länder kann dies z. B. ISO-8859-1 sein, der eine kompakte Codierung von 8 Bit pro Zeichen verwendet. Benötigt man das Eurozeichen, so ist statt dessen ISO-8859-15 zu verwenden. Eine Alternative ist hier UTF8 (aus dem Unicode-Repertoire). In der Angabe in `LANG` wird der Zeichensatz durch einen Punkt abgesetzt (also z. B. `>de_DE.utf8<` oder `>de_DE.ISO-8859-15<`).

Schließlich sind hier noch Sonderfunktionen möglich – hier *modifier* genannt – etwa in der Form `de_DE@euro`. Damit ist gemeint, dass ein Zeichensatz mit dem Eurozeichen (z. B. ISO-8859-15) zu verwenden ist. Um die Sache einfacher (oder komplizierter) zu machen, gibt es zusätzlich noch Synonyme oder Abkürzungen, welche dann auf eine eben beschriebene Code-Kombination abgebildet werden. Ein wichtiges Synonym ist `POSIX`. Man benutzt es häufig in Shell-Prozeduren, um eine wohl definierte Ablaufumgebung zu haben – unabhängig von dem gerade eingestellten Wert von `$LANG`. Die Sprache ist hier Englisch. `POSIX` (oder `>C<`) entspricht weitgehend `>en_USA.ASCII<`.

Diese Angaben sind bereits nützlich, aber nicht für alle Fälle ausreichend detailliert und noch nicht ausreichend flexibel, da man zuweilen z. B. durchaus unterschiedliche Konventionen mischen möchte – z. B. deutsche Dialoge und Fehlermeldungen, aber englische Währungskonventionen, da man als Deutscher in England arbeitet oder für ein englisches Unternehmen in Deutschland. Aus diesem Grund hat man die Anpassungen an lokale Gegebenheiten in mehrere Kategorien unterteilt und innerhalb jeder Kategorie mehrere Elemente, die nochmals spezifische Festlegungen treffen.

Eine der einfachen Kategorien ist die für das lokale (bevorzugte) Papierformat. Diese Kategorie `LC_PAPER` hat drei Elemente: **height**, **width** und **paper-codeset**, welche die Höhe und Breite des Standardpapierformats angeben, sowie der Standardzeichensatz zum Drucken. Beim Format A4 wäre z. B. `height=297` und `width=210` (in Millimeter) und ein passender Zeichensatz für deutsche Texte ISO-8859-1 oder ISO-8859-15 (letzterer mit dem `€`- und `¢`-Zeichen).²

1. Für einige Beispiele dieser Codes siehe Anhang A.5 auf Seite 864.

2. Zu ASCII, ISO-8859-1 und ISO-8859-15 siehe Anhang A.9, Seite 871 ff.

Um die Einstellungen zu vereinfachen, lässt sich für eine ganze Kategorie eine Einstellung vornehmen, indem man der Kategorie-LC-Variablen (z. B. LC_TIME) einen Sprachwert analog zu LANG zuweist. So lässt sich z. B. trotz deutscher Sprache ein amerikanisches Papierformat (über die Variable LC_PAPER) oder englische Währungskonventionen zuweisen (z. B. indem man in LC_CURRENCY den Wert en_UK setzt).

Zu den wichtigsten LC_Klassen gehören:

LC_ALL	überschreibt die nachfolgenden Variablen.
LC_COLLATE	legt die Sortierreihenfolge fest (wie sie z. B. in den C-Funktionen strcol() und strxfrm() benutzt werden).
LC_CTYPE	legt fest, welche Zeichen zu den verschiedenen Zeichenklassen gehören (Buchstaben, Großbuchstaben, Kleinbuchstaben, Interpunctszeichen usw.) und wie sie z. B. in den C-Funktionen isupper() , toupper() , mblem() und wctomb() behandelt werden.
LC_MESSAGES	definiert die Sprache der Programm Meldungen und für Rückfragen (z. B. y/n bzw. j/n).
LC_MONETARY	beschreibt die Währungsdarstellung.
LC_NUMERIC	legt das Zahlenformat fest für die Ein- und Ausgabe in den C-Funktionen printf() und scanf() .
LC_PAPER	definiert das Standardpapierformat.
LC_NAME	definiert das Format von Namen.
LC_ADDRESS	Format für die Darstellung von Adressen (wenig benutzt)
LC_TELEPHONE	Format von Telefonnummern
LC_MEASUREMENT	Format (und Kürzel) für Maßeinheiten
LC_TIME	bestimmt das Format für Zeitangaben (wie sie z. B. mit der C-Funktion strftime() ausgegeben werden).
LC_TYPE	wird nur noch von älteren Programmen ausgewertet und entspricht dort der Funktion von LC_CTYPE. Sie muss den Namen des Zeichensatzes enthalten (also z. B. ISO-8859-1).

Hier gibt es noch eine Reihe weiterer LC-Variablen und nicht alle Programme berücksichtigen alle oben aufgeführten Einstellungen.

Innerhalb der Kategorien gibt es die einzelnen Elemente mit eigenen Namen und Werte. So ist **name_mr** das Element aus der Kategorie LC_NAME, welche die Anrede für einen Herrn (*mister*) enthält. Ist die Variable LC_ALL besetzt, so dominiert sie alle anderen Einstellungen.

Die verschiedenen Einstellungen, insbesondere der Wert LC_MESSAGES kann natürlich nur dann effektiv sein, wenn für das betreffende Programm entsprechend der eingestellten Sprache übersetzte Meldungen vorhanden sind. Sie sind in der Regel unter `/usr/share/locale` im Verzeichnis der entsprechenden Sprache zu finden (z. B. `de/LC_MESSAGES/`). Sie liegen dort in einem speziellen, kompakten Format vor. Fehlen sie, so kann dies zwei unterschiedliche Gründe haben:

- a) Es gibt für das Programm überhaupt noch keine entsprechende Übersetzung oder
- b) es gibt die Übersetzung, aber sie ist nicht installiert.

Zuweilen kann man damit leben, dass man die Meldungen nicht in der bevorzugten, sondern in der nächstbesten Sprache erhält. Hierzu lässt sich in der (globalen) Variablen `LANGUAGE` die Vorzugsreihenfolge der Sprachen festlegen. Für einen Schweizer könnte dies z. B. wie folgt aussehen: ›set LANGUAGE = "it_CH:fr_CH:de_CH:no"‹. Das aufgerufene Programm würde dann zunächst nach italienischen Meldungen suchen. Fehlen diese würde nacheinander nach französischen, deutschen und schließlich durch ›no‹ auf die Standardsprache des Programms zurückgegangen. Einige Programme werden nur `LANGUAGE` statt der `LC`-Variablen aus, andere in Ergänzung zu den `LC`-Variablen.

Die von den C-Bibliotheken, welche die sprachspezifischen Verarbeitungen durchführen, verwendete Auswertungsreihenfolge sieht wie folgt aus:

LC_ALL	übersteuert alle nachfolgenden.
↓	
LC_xxx-Variablen	erlauben granulare Einstellungen.
↓	
LANG	erlaubt einfaches Setzen für alle LC_xxx-Variablen.
↓	
LANGUAGE	mit mehreren möglichen Rückfallpositionen

Beim Setzen der lokalen Einstellungen ist zu beachten, dass durchaus nicht alle Möglichkeiten frei kombiniert werden können – dies wäre für die Entwickler teilweise auch extrem schwierig zu testen. So muss der angegebene Zeichensatz zunächst lokal über eine entsprechende so genannte *keymap* definiert und die Tastatureingabe zu einem zugeordneten Zeichen vorhanden sein (siehe Abschnitt 3.5.4, Seite 189). Für die Ein- und Ausgabe muss zusätzlich ein passender Fonts (Schriftsatz) installiert und zugewiesen sein. So lassen sich zahlreiche Zeichen des vollständigen Unicodezeichensatzes mit den üblichen Schriften nicht darstellen und selbst wenn sie sich auf dem Bildschirm darstellen lassen, ist der Druck auf einem Drucker unter Umständen nicht möglich. Die möglichen (im System aktuell vorhandenen) Kombinationen lassen sich per `locale -a` anzeigen. So steht z. B. der Alias *deutsch* oder *german* für ›de_DE.ISO-8859-1‹.

In der Datei `/usr/share/locale/locale.alias` findet man weitere Lokalisierungs-Alias-Namen.

Leider benutzen nicht alle Linux-Kommandos das hier beschriebene Schema. Einige fragen nur einen Teil der Einstellungen ab, andere gar keine (und sind rein amerikanisch), und wieder andere verwenden andere Variablen und Einstellungen – letztere erfreulicherweise in abnehmender Zahl und Art.

Dies alles hört sich ausgesprochen kompliziert an und ist es in einigen Aspekten auch. In der Praxis hat man aber zumeist wenig Probleme, solange die Wünsche oder Kombinationen nicht zu ausgefallen sind und man eine geeignete Distribution für das jeweilige Land zur Verfügung hat. Die meisten Einstellungen lassen sich dabei bereits aus wenigen Angaben ableiten und werden oft bereits bei der Installation brauchbar gesetzt. Sie sollen sich den gesetzten Lokalisierungen und die möglichen Werte der lokalen Installation durch ein bisschen Experimentieren mit dem Programm `locale` (siehe Seite 324) einmal ansehen.

Die entsprechende Belegung der Umgebungsvariable LANG kann entweder in den systemweit geltenden Initialisierungsskripten (z. B. in */etc/profile*) für die **bash** vorgenommen werden oder jeweils in den entsprechenden Dateien im Login-Verzeichnis des Benutzers. Möchte man bestimmte Programme immer abweichend von den eigenen Standardeinstellungen aufrufen, so kann man diese Definition auch in den jeweiligen Startup-Dateien (zumeist rc-Dateien) vornehmen. Alternativ lässt sich ein Programm auch in folgender Form aufrufen:

```
LANG=sprache_land kommando [optionen] [parameter]
```

Hier wird LANG als Umgebungsvariable nur für genau das aufgerufene Programm und nur für diesen Aufruf entsprechend belegt.

Das Kommando **localedef** (siehe Seite 325) erlaubt, eine erstellte Definitionsdatei in ein Format zu konvertieren, welches von der **locale**-Funktion genutzt werden kann. Diese **locale**-Funktion ist ein C-Modul, welcher die entsprechenden (binären) Definitionen auswertet und in Programmen, welche diese Funktion nutzen, die entsprechenden Funktionen der Lokalisierung realisiert.

Die globale, systemweite Einstellung ist über das bisher beschriebene Verfahren hinausgehend abhängig von der Linux-Distribution. So erfolgt sie für RED HAT¹ (und dem darauf basierenden Mandrake in der Datei */etc/sysconfig/i18n* (dort wird LANG und LANGUAGE gesetzt). Für SUSE erfolgt die Definition in */etc/sysconfig/language* über die Variable RC_LANG und in den RC_LC_xxx-Variablen.² Bearbeiten kann man bei SuSE diese Datei mit dem **sysconfig**-Editor. LANGUAGE ist bei SUSE im Standardfall nicht definiert, lässt sich dort aber z. B. in */etc/profile.local* setzen.

Bei SuSE gibt es eine weitere Besonderheit. Hier wird in der Variablen ROOT_USES_LANGUAGE die Spracheinstellung für den Benutzer *root* festgelegt. Diese Variable wird in */etc/sysconfig/language* definiert. Es ist zumeist sinnvoll, diesen Wert auf **ctype** oder **no** stehen zu lassen (was etwa »en_US« entspricht bzw. den POSIX-Einstellungen).

3.5.2 Lokalisierung unter KDE

Während GNOME im Desktop und in den meisten Programmen der oben beschriebenen Konvention in der Nutzung der Umgebungsvariablen LANG, LANGUAGE, LC_ALL und LC_xxx folgt, hat KDE ein etwas abweichendes Verfahren. Hier lassen sich die Sprache und die analogen detaillierteren Einstellungen über das KDE-Kontrollzentrum (**kcontrol**) einstellen – für alle KDE-Programme.

Dort findet man unter dem Reiter *Index* → *Persönliche Einstellungen* → *Land & Sprache* unter den Reitern *Regionales* (die Sprache), *Zahlen*, *Währung*, *Zeit & Datum* sowie *Sonstige* (Papierformat und Maßsystem) die globale und die individuellen Einstellmöglichkeiten für die Lokalisierung (siehe Abb. 3.15). Die Einstellun-

-
1. Bei Red Hat erlaubt das X-Programm **locale_config**, die Einstellungen für LANG systemweit vorzunehmen.
 2. Die RC_-Variablen sind nur Vorlagen. Sie werden von Startskripten für die Besetzung der LANG- und LC-Variablen herangezogen.

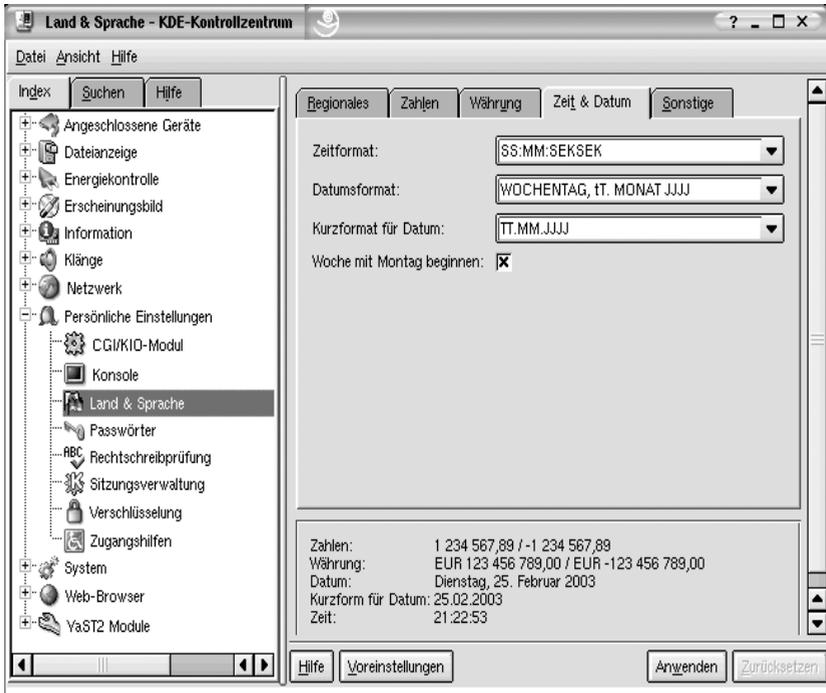


Abb. 3.15: Lokalisierung im KDE-Kontrollzentrum

gen hier wirken sich erst in nachfolgend aufgerufenen Programmen aus. Unter dem Reiter *Regionales* lassen sich mehrere Sprachen in die Liste aufnehmen. Dies entspricht dem Mechanismus von LANGUAGE, d. h. ist fur ein Programm die zuerst aufgefuhrte bersetzung nicht vorhanden, so wird nach der nachsten in der Liste gesucht.

Jedoch lassen sich auch unter KDE individuelle Abweichungen fur einzelne Programme erreichen. Die Steuervariable unter KDE ist `KDE_LANG`. So lasst sich z. B. aus der/dem Konsole/Terminalfenster heraus mit `›KDE_LANG=en_US konqueror‹` der KDE-Dateimanager **konqueror** (bei Grundeinstellung `de_DE`) mit der Spracheinstellung *englisch* starten, ohne dass man dazu zuvor im KDE-Kontrollzentrum die Sprache umstellen und sie spater wieder zuruckstellen muss.

Die KDE-Grundeinstellungen fur den einzelnen Benutzer sind in dem verdeckten Verzeichnis `~/.kde/share/config` (im Login-Verzeichnis des Benutzers) abgelegt.¹ In der Datei `kdeglobals` liegen die globalen Einstellungen des KDE-Desktops. Dort findet man die Spracheinstellungen unter der Sektion `›[Locale]‹`.

Hier gibt es fur die meisten KDE-Programme Konfigurationsdateien mit der Endung `rc`. Setzt man in die entsprechende Datei folgende Anweisungen:

```
[ LOCALE ]
Charset=iso8859-15
Country=de
Language=en_US
```

1. Unter Umstanden muss man auch in den Verzeichnissen unter `~/.kde2` oder `~/.kde3` suchen.

So wird künftig das entsprechende Programm mit diesen speziellen Lokalisierungseinstellungen gestartet. Im KDE-Kontrollzentrum lassen sich für die von einigen KDE-Programmen angebotene Rechtschreibprüfung zusätzlich Einstellungen wie das (sprachspezifische) Wörterbuch, der Zeichensatz und das zu verwendende Rechtschreibprogramm (`ispell` oder `aspell`) vornehmen.

3.5.3 Die richtigen Fonts zur Darstellung

Wie bereits erwähnt, muss man schließlich dafür sorgen, dass zur Bildschirmdarstellung und zu Druckausgabe eine Schrift (englisch: *font*) verwendet wird, welche die verwendeten Zeichen auch darstellen kann. Dies erfolgt jedoch nicht über das hier beschriebene Schema mit LC- (oder anderen) Umgebungsvariablen, sondern ist in den Darstellungs- und Druckanwendungen zu berücksichtigen. Während die meisten bei uns verwendeten Schriften die Zeichen des ISO-8859-1-Codes abdecken können, sind die zusätzlichen Zeichen des ISO-8859-15 (z. B. €, š, Š, ž, Ž, œ, Œ, ...) erst in neueren Schriften zu finden und es sind uns keine Schriften bekannt, die das volle Spektrum der Zeichen aus dem UTF-16- oder UTF-8-Zeichensatz abdecken. Linux bringt einige ISO-8859-1/15-fähige Schriften mit – zu erkennen an der entsprechenden Nummer im Namensteil – und auch Fonts, welche die üblichen europäischen Zeichen aus UTF-8 enthalten. Bei den Schriften sind zwei Bereiche zu unterscheiden:

- ▶ Schriften in den Textkonsolen außerhalb der grafischen Oberfläche und
- ▶ Schriften in X-Anwendungen, also auch unter den grafischen Desktops von KDE, GNOME, anderen Desktops oder unter den reinen Window-Managern.

Für die *Textkonsolen* werden Tastaturlayout und Schriftzeichenzuordnung mit den Programmen `loadkeys` und `setfont` durchgeführt.

Für X Window und die darauf aufsetzenden Desktops GNOME oder andere – sowie für Programme unter einem reinen X Window-Manager – werden Zeichensatz und Schriftart gemeinsam in den X-Ressourcen eingestellt (siehe Kapitel 7.4, Seite 689 ff.). KDE hat hier eigene Einstellungen.

Einige Programme benötigen zum Arbeiten mit speziellen Codierungen (insbesondere mit UTF-8)¹ und Zeichensätzen spezielle Aufrufoptionen oder speziell gesetzte Umgebungsvariablen. So sollte das Programm `less` z. B. zur Ausgabe einer Unicode-Datei die Umgebungsvariable `CHARSET` gesetzt haben, und `xterm` lässt sich für UTF-8 über das Shell-Skript `uxterm` aufrufen, in dem sowohl die Option `-T 'xterm UTF-8'` als auch eine UTF-8-fähiger Font per `-fn -Misc-Fixed-Medium-R-Normal ...-ISO10646-16` gesetzt wird. Weitere Informationen zu Fonts- und Font-Einstellungen sind im Kapitel 7.4 (S. 689) zu finden.

1. UTF-8 ist eine spezielle Codierung von *Unicode*. Unicode entspricht ISO-10646, weshalb in einigen Unicode-fähigen Fonts der Namensanteil ISO10646 vorkommt.

3.5.4 Das richtige Tastaturlayout

Ein wesentlicher Punkt für eine Lokalisierung ist die Wahl bzw. die Einstellung des korrekten Tastaturlayouts. So möchte man auf einer deutschen Tastatur beim Anschlagen von *y* eben ein *y* und kein *z* (und umgekehrt) erhalten. Bei den meisten Linux-Distributionen wird die globale Tastatureinstellung bereits bei der Installation abgefragt.

Für X Windows ist sie in `/etc/X11/XF86Config` in der Sektion `InputDevice` mit der Option `XkbLayout` festgelegt. Für die Textkonsole des Kernels erfolgt die Einstellung über die Programme `loadkeys` und `setfont`.

`loadkeys` lädt dabei die so genannte *Keymap* für den Kernel bzw. die Textkonsole des Kernels. Die *Keymap* ist eine Umsetzungstabelle, welche die von der Tastatur kommenden Tasten-Codes einem Zeichen des Zeichensatzes zuordnet. Die unterschiedlichen Tastaturlayouts bestehen also faktisch aus nichts anderem als unterschiedlichen Zuordnungstabellen. Die Tabellen bzw. *Keymaps*, von denen man für spezielle Zwecke natürlich auch eigene anlegen kann,¹ sind in der Regel in einem Unterverzeichnis `/usr/share/kbd/keymaps` oder im Verzeichnis `/usr/src/linux/drivers/char` zu finden. Die Tabelle (Datei) `/etc/default/keymap.map` stellt dabei die Standardtabelle für das jeweilige System (den Kernel) dar.

`setfont` lädt den Font (die Schriftdefinition) für die Textkonsole des Kernels. Damit lassen sich hier unterschiedliche Schriftgrößen und unterschiedliche Zeichensätze einstellen. Die Fonts liegen in der Regel im Verzeichnis `/usr/share/kbd/consolefonts`, die Unicode-Fonts in `/usr/share/kbd/unimaps`.

Die Standardbelegungen werden aber in praktisch allen Distributionen bei der Installation abgefragt und lassen sich später über die Administrationsoberfläche einstellen, so dass man in der Regel nicht in diese Tiefen steigen muss. Bei SUSE ist dies z. B. in YaSTE2 möglich. Hinterlegt werden diese Standardeinstellungen leider wieder an distributionsspezifischen Stellen: bei SuSE in der Datei `/etc/rc.config`, bei Red Hat wird die Einstellung in `/etc/sysconfig/keyboard` in den Variablen `KEYBOARDTYPE` und `KEYTABLE` hinterlegt.

Die Tastatureinstellungen lässt sich aber statt mit dem oben beschriebenen Verfahren in den meisten aktuellen Linux-Distributionen sowohl bei der Installation in einen grafischen Dialog auswählen als auch später noch über die Administrations- und Konfigurationsoberfläche. Bei SuSE erfolgt dies z. B. unter YaST2 über *System* → *Tastaturbelegung auswählen*.

KDE hat eigene Möglichkeiten der Tastatureinstellung über das KDE-Kontrollzentrum, wie bereits Abschnitt 3.5.2 (Seite 186) beschrieben. Hier lassen sich mehrere Tastaturlayouts wählen und relativ einfach zwischen diesen wechseln.

Kompositionen

Die typischen 101–104 Tasten einer normalen Tastatur reichen bei weitem nicht aus, um z. B. die Zeichen des ISO-8859-1/15-Zeichensatzes direkt abzubilden. Linux bietet deshalb (wie andere Systeme auch) die Möglichkeit, bestimmte Zeichen durch eine Komposition zu erzeugen, d. h. dadurch, dass man nacheinander bestimmte Tasten drückt und daraus ein kombiniertes Zeichen entsteht. So ergibt z. B. `<compose><^><O>` das

1. Der Aufbau der Keymap-Dateien ist unter `man 5 keymaps` zu finden.

Zeichen › \hat{O} ‹ oder $\langle compose \rangle \langle \wedge \rangle \langle \hat{A} \rangle$ das Zeichen › \hat{A} ‹. Die $\langle compose \rangle$ -Taste ist natürlich eine Metataste, und es ist festzulegen, welche Taste oder Tastenkombination diese Funktion haben soll. Die Standardeinstellung für Taste oder Tastenkombination für die $\langle compose \rangle$ -Taste (in den *keymaps* trägt sie die Bezeichnung *Multi_key*) ist distributionsabhängig – \uparrow -[AltGr] funktioniert jedoch bei mehreren Distributionen.¹ Bei Red Hat kann zusätzlich die [rechte Windows-Taste] (auf moderneren PC-Tastaturen) als $\langle compose \rangle$ genutzt werden. SuSE hat im Standardfall neben \uparrow -[AltGr] auch \uparrow -[rechte Windows-Taste] und \uparrow -[rechte Windows-Taste] sowie \uparrow -[rechte Strg-Taste] hier definiert.

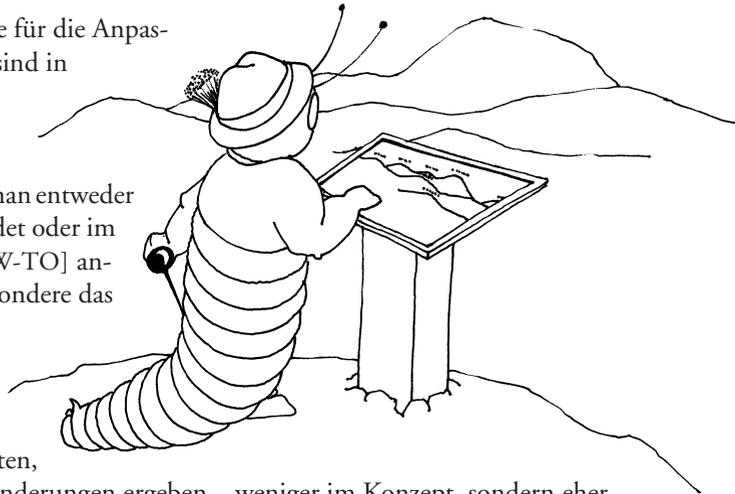
Für X Window wird die $\langle compose \rangle$ -Taste in */etc/sysconfig/keyboard* festgelegt. Die Belegung kann mit `xmodmap -pk | grep Multi_key` angezeigt werden. Mit dem Programm `xev` (aufgerufen aus einem `xterm`-Fenster) sieht man, welche Taste welchen *Key-Code* hat und welche Funktion ihr zugeordnet ist (man drückt dazu die entsprechende Taste. `xev` zeigt *X-Events* an. Das Drücken und Loslassen einer Taste ist ein solcher *Event*).

Daneben gibt es die Tasten mit der etwas irreführenden Bezeichnung *deadkeys* (*tote Tasten*), die keineswegs tot sind, sondern Tasten, deren Zeichen (z. B. die Tilde $\grave{\text{~}}$) in der Ausgabe über das nachfolgende Zeichen gedruckt wird (also z. B. $\langle \grave{\text{~}} \rangle \langle A \rangle \rightarrow \hat{A}$). Ist in *keymaps* *nodedeadkey* festgelegt, so ist dieses Kombinieren unterdrückt, und die Tasten wie \wedge , $\grave{\text{~}}$, $\acute{\text{^}}$, $\grave{\text{~}}$, \circ , ¨ ergeben gleich das vorstehende Zeichen und keine kombinierten Zeichen ohne vorherige $\langle compose \rangle$ -Taste. Mit dieser geht es jedoch.

Das €-Zeichen wird – den richtigen Zeichensatz vorgesetzt – bei PC-Tastaturen per [AltGr]-[E] und das ¢-Zeichen per [AltGr]-[C] eingegeben (bei Verwendung der üblichen, aktuellen *keymaps*).

Weitere Hinweise zur Lokalisierung

Zahlreiche weitere Hinweise für die Anpassung an die lokale Sprache sind in entsprechenden HOWTO-Artikeln zu finden. Für die deutsche Sprache wäre dies *German-HOWTO*, welche man entweder als Teil der Distribution findet oder im Internet unter den in [HOW-TO] angegebenen Adressen. Insbesondere das dort aufgeführte Papier von M. Kuhn kann nützlich sein. Das Lokalisierungsthema ist noch nicht abgeschlossen und es ist zu erwarten,



dass sich hier noch einige Änderungen ergeben – weniger im Konzept, sondern eher was die Einstellmöglichkeiten und die Oberflächen betrifft und insbesondere, was die Unterstützung in den Programmen betrifft und die Anzahl und den Umfang der unterstützten Sprachen in den Programmen.

1. Bei diesen Kombinationen muss die Shift-Taste immer zuerst gedrückt werden!

3.6 Textkonsolen

Bildschirme sind fast immer zu klein, um alle Informationen, Fenster und andere Elemente halbwegs übersichtlich anzuordnen. Linux mit seinen alphanumerischen und grafischen Oberflächen bietet dafür mehrere Lösungsansätze, die auch kombiniert verwendet werden können:

- ▶ virtuelle Arbeitsflächen (siehe hierzu Seite 632)
- ▶ virtuelle Desktops (siehe Seite 633)
- ▶ virtuelle Textkonsolen

Textkonsolen passen zwar just nicht in das Konzept grafischer Oberflächen, aber durchaus in das Konzept *virtueller Bildschirm*.

Linux kennt die Möglichkeit mehrerer virtueller Bildschirme auch außerhalb der grafischen Oberfläche – oder parallel zu dieser. Dies sind die so genannten Textkonsolen: im Prinzip sind nichts anderes, als mehrere parallel laufende alphanumerische Bildschirme, welche von Linux-Basis-System zur Verfügung gestellt werden. Eine solche Textkonsole füllt immer jeweils den ganzen Bildschirm aus und ähnelt weitgehend einem alphanumerischen Terminalfenster unter der grafischen Linux-Oberfläche, jedoch ohne Fensterrahmen, ohne Fensterattribute und andere Merkmale von grafischen Fenstern.

Man gelangt entweder direkt bei der Anmeldung in eine solche Textkonsole – z. B. wenn kein grafischer Login-Manager läuft und nur die alphanumerische Oberfläche aktiviert ist oder über spezielle Tastenkombinationen aus der grafischen Linux-Oberfläche heraus.¹ In diesem Fall wird die grafische Oberfläche vorübergehend komplett ausgeblendet und die Textkonsole übernimmt den Bildschirm. Hierbei sind in einer Standardkonfiguration in der Regel bis zu sechs solcher Textkonsolen möglich, zwischen denen sich schnell hin- und herschalten lässt.

Per `[Alt]-[Strg]-[Fn]` wechselt man von der aktuellen Textkonsole – oder einer grafischen Oberfläche – zur Konsole n ($1 \leq n \leq 6$).² Öffnet man eine Textkonsole zum ersten Mal, so erscheint der alphanumerische Login-Bildschirm bei dem man sich anmeldet. Anmeldungen mit unterschiedlichen Benutzernamen sind in den verschiedenen Konsolen möglich. Mit dem Anmelden bekommt man die in `/etc/passwd` definierte Login-Shell, ist also auf der Shell- bzw. Kommandozeilenebene. Abmelden kann man sich hier entweder per `exit` oder `logout` oder per `[Strg]-[D]`.

Aus einer Textkonsole gelangt man per `[Alt]-[Strg]-[F7]` in die grafische Oberfläche zurück – sofern diese zuvor gestartet war.

Ein Wechsel von der grafischen Oberfläche in eine Textkonsole kann auch dann sinnvoll sein, wenn sich Probleme mit dem X-Display-Treiber ergeben oder KDE oder GNOME einmal (ausnahmsweise) *aufgehängt* haben.

Das Arbeiten nur mit der Textkonsole und ganz ohne grafische Oberfläche erfordert wesentlich weniger Hauptspeicher und wesentlich geringere Rechenleistung, so dass

1. Wählt man in einem grafischen Login-Manager die Einstellung `failsafe`, so gelangt man auch in ein Textkonsol-ähnliches alphanumerisches Terminal-Fenster. Dies läuft jedoch unter X Window!

2. `[Fn]` ist hier die Funktionstaste n .

man hier für spezielle Zwecke auch ältere PCs einsetzen kann und mit schlanken Konfigurationen auskommt.

In welcher der Textkonsolen man sich gerade befindet ist u. a. über das Kommando `tty` ersichtlich. Es gibt (ohne weitere Parameter aufgerufen) den Namen der aktuellen Dialogstation aus. `/dev/tty2` ist dabei z. B. die Textkonsole 2. Daneben wird die Konsole auch in der Regel als Teil der Login-Aufforderung angezeigt.

Textkonsolen verwenden (potentiell) eigene Tastaturbelegungen (Tastaturlayout) und Schriften (Fonts). Wie zuvor bereits beschrieben lässt sich das Tastaturlayout von Textkonsolen per `loadkeys` einstellen und der darin verwendete Font per `setfont` festlegen.

