

Mathias Kettner: *Fehlerdiagnose und Problembhebung unter Linux* .



Mathias Kettner

Fehlerdiagnose und Problembehebung unter Linux

2., aktualisierte und erweiterte Auflage



Alle in diesem Buch enthaltenen Programme, Darstellungen und Informationen wurden nach bestem Wissen erstellt und mit Sorgfalt getestet. Dennoch sind Fehler nicht ganz auszuschließen. Aus diesem Grund ist das in dem vorliegenden Buch enthaltene Programm-Material mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Autoren und die SUSE LINUX AG übernehmen infolgedessen keine Verantwortung und werden keine daraus folgende Haftung übernehmen, die auf irgendeine Art aus der Benutzung dieses Programm-Materials, oder Teilen davon, oder durch Rechtsverletzungen Dritter entsteht.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Buch berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann verwendet werden dürften.

Alle Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt und sind möglicherweise eingetragene Warenzeichen. Die SUSE LINUX AG richtet sich im Wesentlichen nach den Schreibweisen der Hersteller. Andere hier genannte Produkte können Warenzeichen des jeweiligen Herstellers sein.

Dieses Werk ist urheberrechtlich geschützt.

Alle Rechte, auch die der Übersetzung, des Nachdruckes und der Vervielfältigung des Buches, oder Teilen daraus, vorbehalten. Kein Teil des Werkes darf ohne schriftliche Genehmigung des Verlages in irgendeiner Form (Druck, Fotokopie, Microfilm oder einem anderen Verfahren), auch nicht für Zwecke der Unterrichtsgestaltung, reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

Bibliografische Information Der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.ddb.de> abrufbar.
ISBN 3-89990-127-4

© 2004 SUSE LINUX AG, Nürnberg (<http://www.suse.de>)

Umschlaggestaltung: Fritz Design GmbH, Erlangen

Gesamtlektorat: Nicolaus Millin

Fachlektorat: Joerg Arndt, Anke Boernig, Franz Hassels, Ihno Krumreich, Marco Michna, Sascha Wessels

Satz: L^AT_EX

Druck: Kösel, Krugzell

Printed in Germany on acid free paper.

Inhaltsverzeichnis

1	Los geht's	1
1.1	Die Sache mit der Armbanduhr	1
1.2	Das Besondere an Linux	2
1.3	Den Fehler reproduzieren	3
1.4	Die exakte Ursache finden	5
1.4.1	Der Einblick	5
1.4.2	Spurensuche	6
1.4.3	Die Wirkungskette	6
1.4.4	Der wandernde Fehler	8
1.4.5	Wechselnde Umstände	8
1.4.6	Reine Geisteskraft	9
1.5	Den Fehler beheben	9
1.6	Ein Wort zur 2. Auflage	10
2	Dokumentation und Recherche	13
2.1	Dokumentation	14
2.1.1	Erste und zweite Hand	14
2.1.2	Bücher	15
2.1.3	Man-Seiten	15
2.1.4	GNU Texinfo	18
2.1.5	Artikel aus dem Internet	19
2.1.6	Dokumentation in RPM-Paketen	20
2.1.7	Request For Comments (RFC)	21

2.1.8	Dateien unter <code>/usr/share/doc</code>	21
2.2	Hilfeartikel, HOWTOs und FAQs	22
2.2.1	HOWTOs	22
2.2.2	SUSE-Supportdatenbank (SDB)	22
2.2.3	Die Linux Problem Base	23
2.2.4	FAQs	23
2.3	Newsgruppen und Mailinglisten	23
2.4	Das Linux Documentation Project (LDP)	24
2.5	Quellcode	25
3	Logdateien und Debugausgabe	27
3.1	Der System Logging Daemon	28
3.1.1	Funktionsweise des <code>syslogd</code>	28
3.1.2	Loggen über das Netzwerk	29
3.1.3	Meldungen sortieren	30
3.1.4	Meldungen speichern oder weiterleiten	33
3.1.5	Erzeugen von Logmeldungen aus Shellskripten	34
3.1.6	Erzeugen von Logmeldungen aus C-Programmen	36
3.2	Logdateien analysieren mit <code>logsurfer</code>	36
3.2.1	Funktionsweise von <code>logsurfer</code>	36
3.2.2	<code>logsurfer</code> vorbereiten	37
3.2.3	<code>logsurfer</code> aufrufen	38
3.2.4	<code>logsurfer</code> an <code>syslogd</code> anbinden	39
3.3	Shellskripte verfolgen mit <code>bash -x</code>	41
4	Aktive Diagnose	45
4.1	Dateien	46
4.1.1	Textdateien effizient anzeigen mit <code>less</code>	48
4.1.2	Logdateien anzeigen mit <code>tail -f</code> oder <code>less +F</code>	49
4.1.3	Sich ändernde Ausgaben anzeigen mit <code>watch</code>	50
4.1.4	Dateityp feststellen mittels <code>file</code>	51
4.1.5	Suche innerhalb von Textdateien mit <code>grep</code>	52
4.1.6	Konfigurationsdateien abspecken mit <code>grep -Ev '^(# \$)'</code>	54

4.1.7	Dateien dem Namen und Typ nach finden mit <code>find</code>	56
4.1.8	Schnelles Suchen mit <code>locate</code> statt <code>find</code>	58
4.1.9	Die Nadel finden: <code>grep</code> , <code>find</code> und <code>xargs</code>	60
4.1.10	Dateien aus RPM-Paketen untersuchen mit <code>rpm</code>	61
4.1.11	Fehlende Dateien finden auf der SUSE-CD/DVD	63
4.1.12	Ort eines Befehls feststellen mit <code>which</code>	64
4.1.13	Änderungen in Textdateien aufspüren mit <code>diff</code>	65
4.1.14	Texte in Binärdateien finden mittels <code>strings</code>	66
4.1.15	Binärdateien anzeigen mit <code>hexdump</code>	68
4.1.16	Änderungen in Binärdateien vornehmen	69
4.1.17	Binärdateien vergleichen mit <code>cmp</code>	71
4.2	Festplatten und Dateisysteme	72
4.2.1	Festplatten und Partitionen anzeigen mit <code>fdisk</code>	72
4.2.2	Gemountete Dateisysteme anzeigen mit <code>mount</code>	74
4.2.3	Freien Platz anzeigen mit <code>df</code>	75
4.3	Prozesse	78
4.3.1	Prozessbaum anzeigen mit <code>pstree</code>	78
4.3.2	Prozessliste mit <code>ps</code>	80
4.3.3	Prozessmonitor <code>top</code>	81
4.3.4	Prozess-IDs ermitteln mit <code>pidof</code>	84
4.3.5	Offene Dateien anzeigen mit <code>lsof</code>	84
4.3.6	Alle Informationen aus erster Hand unter <code>/proc</code>	86
4.4	Benutzer und Gruppen	88
4.4.1	Gruppen eines Benutzers anzeigen mit <code>id</code>	88
4.4.2	Benutzer und Gruppe eines Prozesses anzeigen mit <code>ps</code>	89
4.4.3	Alle Gruppen eines Prozesses anzeigen	90
4.5	Der Bootvorgang	91
4.5.1	Der Bootvorgang im kurzen Überblick	91
4.5.2	Bootmeldungen	94
4.5.3	Booten in andere Runlevel	94
4.5.4	Systemrettung im Runlevel <code>S</code>	96
4.5.5	Booten ohne Passwort	98

Inhaltsverzeichnis

4.5.6	Booten ohne Bibliotheken mit der <code>sash</code>	99
4.5.7	Das SUSE-Rettungssystem	102
4.6	Kernel	104
4.6.1	Version, Lebensalter und Speicherverbrauch	105
4.6.2	Module	106
4.6.3	Kernelparameter	108
5	Hardwarediagnose	109
5.1	Prozessor, Speicher und andere innere Werte	110
5.1.1	Der Prozessor und <code>/proc/cpuinfo</code>	110
5.1.2	Informationen aus ACPI	111
5.1.3	ACPI: Prozessor	112
5.1.4	ACPI: Stromverbrauch und Akkus	113
5.1.5	ACPI: Temperatur	115
5.1.6	Hauptspeicher	116
5.1.7	Hardwareuhr und Systemzeit, <code>hwclock</code> und <code>date</code>	116
5.2	Allgemeines zur Peripherie	118
5.2.1	Grundprinzip des Prozessorbus	118
5.2.2	Zugriff auf Peripheriebausteine	119
5.2.3	Interrupts und <code>/proc/interrupts</code>	120
5.2.4	IO-Ports und <code>/proc/ioports</code>	121
5.3	Bussysteme und Steckkarten	122
5.3.1	Grundprinzip des Erweiterungsbus	122
5.3.2	Verschiedene Bussysteme	123
5.3.3	PCI	124
5.3.4	Liste aller PCI-Geräte mit <code>lspci</code>	125
5.3.5	PCI Vendor- und Device-IDs	128
5.4	Externe Schnittstellen	128
5.4.1	Grundprinzip der seriellen Datenübertragung	128
5.4.2	Serielle Schnittstellen testen mit <code>echo</code> und <code>cat</code>	130
5.4.3	Serielle Geräte testen über Interrupts	132
5.4.4	Serielle Geräte testen mit <code>minicom</code>	133
5.4.5	Die parallele Schnittstelle	134

5.4.6	Die Tastatur	136
5.4.7	PS/2-Maus	137
5.4.8	Allgemeines zu USB	138
5.4.9	USB-Diagnose unter Linux	143
5.5	Laufwerke	148
5.5.1	Das Diskettenlaufwerk	148
5.5.2	Allgemeines zu IDE/ATA und ATAPI	151
5.5.3	IDE-Diagnose mit Linux	152
5.5.4	Performancekontrolle bei DMA und PIO	156
5.5.5	Allgemeines zu SCSI	158
5.5.6	Einfache SCSI-Diagnose unter Linux	158
5.5.7	Informationen aus <code>/proc/scsi</code>	160
6	Netzwerkdiagnose	163
6.1	Einteilung in Schichten	163
6.2	Die Hardwareschicht	165
6.2.1	Netzwerkgeräte	165
6.2.2	Aktivierung und Konfiguration der Geräte mit <code>ifconfig</code>	166
6.2.3	Medium und Übertragungsart ermitteln mit <code>mii-diag</code>	167
6.2.4	MAC-Adressen und ARP	169
6.2.5	Punkt-zu-Punkt Verbindungen mit PPP	170
6.3	Das Internetprotokoll (IP)	174
6.3.1	Grundsätzliches zu IP	174
6.3.2	Internetadressen	175
6.3.3	Verbindungstest mit <code>ping</code>	182
6.3.4	Weitere Möglichkeiten von <code>ping</code>	184
6.3.5	Bandbreitenschätzung mit <code>bing</code>	187
6.3.6	Routen verfolgen mit <code>traceroute</code>	189
6.4	TCP und UDP	191
6.4.1	Grundsätzliches zu TCP	191
6.4.2	TCP-Verbindungen testen mit <code>telnet</code>	192
6.4.3	Grundsätzliches zu UDP	196
6.4.4	TCP und UDP testen mit <code>netcat / nc</code>	198

Inhaltsverzeichnis

6.4.5	Sockets und Verbindungen beobachten mit <code>netstat</code>	201
6.4.6	Offene Ports finden mit <code>nmap</code>	204
6.4.7	Netzwerkverkehr mithören mit <code>tcpdump</code>	206
6.5	Namensauflösung mittels DNS	213
6.5.1	DNS-Anfrage in Einzelschritten	214
6.5.2	Probleme mit DNS: Zwei unterschiedliche Symptome	215
6.5.3	Diagnose von DNS-Problemen	216
6.5.4	DNS Anfragen mit <code>tcpdump</code> analysieren	217
6.6	Diagnose von <code>iptables</code>	220
6.6.1	Paketfilter und Firewall	220
6.6.2	Die drei Regelketten	221
6.6.3	Ketten anzeigen	222
6.6.4	Paketzähler	223
6.6.5	Logmeldungen	224
6.7	Weitere Diagnosewerkzeuge	225
6.8	Ein ausführliches Beispiel	226
6.8.1	Das Beispiel	226
6.8.2	Die Wirkungskette	227
6.8.3	Diagnose mit <code>ping</code>	227
6.8.4	Link testen mit <code>mii-diag</code>	229
6.8.5	Routingprobleme	229
6.8.6	Einwahl per DSL	230
6.8.7	DNS	232
6.8.8	POP3-Server	233
6.8.9	Fazit	234
7	Programme und Prozesse	235
7.1	Untersuchen von Binaries	235
7.1.1	Dynamische Bibliotheken unter Linux	235
7.1.2	Verwendete Bibliotheken auflisten mit <code>ldd</code>	238
7.1.3	Binaries analysieren mit <code>objdump</code>	239
7.2	System- und Bibliotheksaufrufe	241
7.2.1	Grundsätzliches zu Systemaufrufen	241

7.2.2	Systemaufrufe protokollieren mit <code>strace</code>	242
7.2.3	Wichtige Systemaufrufe	245
7.2.4	Systemaufrufe: Dateizugriffe	245
7.2.5	Systemaufrufe: Verzeichnisse und Dateiattribute	248
7.2.6	Systemaufrufe: Netzwerk	249
7.2.7	Systemaufrufe: UIDs und GIDs	250
7.2.8	Systemaufrufe: Warten	251
7.2.9	Systemaufrufe: Erzeugen und Beenden von Prozessen	254
7.2.10	Systemaufrufe: Speicherverwaltung und Übriges	257
7.2.11	<code>strace</code> in der Praxis	258
7.2.12	Beispiel: <code>xscanimage</code> als normaler Benutzer verwenden	261
7.2.13	Bibliotheksaufrufe anzeigen mit <code>ltrace</code>	263
7.3	Signale	265
7.3.1	Was sind Signale?	265
7.3.2	Signale senden mit <code>kill</code> und <code>killall</code>	269
7.3.3	Ein kleines Programm zum Testen von Signalen	270
7.3.4	Segmentfehler (segmentation fault)	271
7.3.5	Möglichkeiten, mit einem Segmentfehler umzugehen	272
7.4	Debugging unter Linux	273
7.4.1	Was ist ein Debugger?	273
7.4.2	Der GNU-Debugger <code>gdb</code>	274
7.4.3	Starten von Programmen unter <code>gdb</code>	274
7.4.4	Allgemeines zum Aufrufstapel	276
7.4.5	Aufrufstapel mit <code>gdb</code> untersuchen	278
7.4.6	Ein chirurgischer Eingriff	279
7.4.7	Andocken an laufende Prozesse	281
7.4.8	Untersuchen von Core dumps	282
7.4.9	Die Bedienung des GNU-Debuggers	284
7.4.10	Eine spektakuläre Rettungsaktion	285
8	An die Quellen	291
8.1	Nur kein Respekt!	291
8.2	Interpretersprachen vs. Compilersprachen	292

8.3	Interpretersprachen	293
8.3.1	Allgemeines zu Interpretersprachen	293
8.3.2	Shellskripte	294
8.3.3	Perl	297
8.3.4	PHP	299
8.3.5	Python	300
8.4	Compilersprachen	302
8.4.1	Quellcodes besorgen	303
8.4.2	Entwicklungswerkzeuge vorbereiten	303
8.4.3	Fremde Quellen auspacken	304
8.4.4	In den Quellen recherchieren	309
8.4.5	Fundstellen im Linux-Kernel	314
8.4.6	Übersetzen der Quellen bei Tarbällen mittels <code>make</code>	317
8.4.7	<code>autoconf</code> und <code>automake</code>	319
8.4.8	Übersetzen der Quellen bei Source-RPMs	324
8.4.9	Überprüfen der Kompilation	326
8.4.10	Installieren	326
8.4.11	Den Fehler festnageln	327
8.4.12	Änderungen machen	327
8.4.13	<code>printf</code> -Debugging	328
8.4.14	Debug-Ausgabe in C und C++	329
8.5	<code>printf</code> -Debugging im Kernel	332
8.5.1	Vorbereitungen	332
8.5.2	Den Kernel übersetzen	333
8.5.3	Den Kernel installieren	333
8.5.4	Meldungen ausgeben mit <code>printk</code>	334
8.5.5	Ein komplettes Beispiel	335
8.6	Den Fehler beheben	338
8.6.1	Einen Patch erstellen	338
8.6.2	Den Patch testen	340
8.6.3	Den Patch in das RPM einbauen	341
8.6.4	Dem Autor den Patch schicken	343

9	Performanceprobleme	345
9.1	Zeitmessung	346
9.1.1	Zeitmessung bei Befehlen	346
9.1.2	Differentielle Zeitmessung	349
9.1.3	Zeitmessung bei Serverdiensten	351
9.1.4	Zeitmessung von laufenden Prozessen	353
9.2	Speicherengpässe und Swappen	356
9.3	Festplatten	359
9.3.1	Lese- und Positionierungsgeschwindigkeit	359
9.3.2	Partitionsgrößen	360
9.3.3	Der Einfluss von Caches	360
9.3.4	Raw-Devices	362
9.4	Einfluss des Dateisystems	362
9.5	Netzwerk	364
9.5.1	Bandbreitenmessung mit <code>iptraf</code>	365
9.5.2	Laufzeiten messen mit <code>ping</code>	371
9.5.3	Wartezeiten	373
9.5.4	Zusammenfassung	376
9.6	Kompression von Daten	376
9.6.1	Grundüberlegungen	376
9.6.2	OpenSSH	377
9.7	Verschlüsselung	379
9.7.1	Performance unterschiedlicher Algorithmen	379
9.7.2	Algorithmus wählen bei OpenSSH	380
A	Handwerkszeug	383
A.1	Effizientes Arbeiten mit der Bash	383
A.1.1	Die Tabulatortaste	383
A.1.2	Die Historie	385
A.1.3	Die geschweiften Klammern { und }	386
A.2	Wichtige Funktionen der Shell	388
A.2.1	Backslashes und Anführungszeichen	388
A.2.2	Ausgabeumlenkung	390

Inhaltsverzeichnis

A.2.3	Pipes	391
A.2.4	Kommandoersetzung	391
A.3	Texteditoren	393
A.3.1	vi	394
A.3.2	joe	395
A.3.3	jmacs	396
A.3.4	zile	397
A.3.5	mcedit	398
A.4	Reguläre Ausdrücke	399
A.4.1	Regeln	399
A.4.2	weitere Beispiele	404
A.4.3	einfache und erweiterte reguläre Ausdrücke	404
A.5	Hexadezimalzahlen	405
A.5.1	Der Name	405
A.5.2	Der Sinn	405
A.5.3	Das Umrechnen	406
A.6	Die Verzeichnisstruktur von Linux	407
A.6.1	Der Filesystem Hierarchy Standard	407
A.6.2	Die Rootpartition	408
A.6.3	Das Wurzelverzeichnis	408
A.6.4	Die /usr-Hierarchie	411
A.6.5	Die /var-Hierarchie	413
B	Alle Werkzeuge	415
B.1	Nachinstallieren von Software	415
B.2	Diagnosewerkzeuge	416
B.3	Entwicklungswerkzeuge	423

Kapitel 1

Los geht's

1.1 Die Sache mit der Armbanduhr

Wer liest schon gerne Vorworte und all den anderen Kram, der die ersten zwanzig Seiten eines Fachbuches füllt, bis der erste Absatz zum eigentlichen Thema kommt? Natürlich möchte auch ich ein paar einleitende Worte sagen, verspreche aber, nicht auszuschweifen und so schnell wie möglich in den Stoff dieses Buches einzusteigen. Was erwartet dich also?

Kurz gesagt: Dieses Buch soll dir eine Hilfestellung beim Lösen von *Problemen* sein, die dir bei der Arbeit mit Linux über den Weg laufen. Unter Problem verstehe ich etwas, das dafür sorgt, dass deine Linuxkiste nicht das tut, was du von ihr erwartest. Natürlich sitzt das Problem nicht immer im Computer, sondern auch manchmal davor oder – und das macht uns am meisten zu schaffen – vor einem ganz anderen Computer.

Was ich damit andeuten will, ist meine leidliche Erfahrung, dass du leider am längsten nach denjenigen Fehlern suchen musst, die du gar nicht selbst verursacht hast, sondern ein sogenannter *Hersteller* (bei Linux zum Glück selten), der Linux-Distributor (z. B. SuSE, Red Hat oder Debian) oder einer der vielen fleißigen Programmierer, die uns ihre Werke kostenlos zur Verfügung stellen (und die man deshalb natürlich nicht kritisieren sollte).

Nun bist du bei Linux in der glücklichen Lage, dass du in den meisten Fällen alle nötigen Hilfsmittel an Bord hast, um dem rätselhaften Fehlverhalten selbst auf den Grund zu gehen. Die Möglichkeiten zur treffsicheren Fehlerfindung sind unter Linux ungeahnt und nicht zu vergleichen mit denen anderer bekannter Betriebssysteme. Der Einblick in den Quellcode von Linux und seinen Programmen sei hier nur am Rande erwähnt, auch wenn dieser in einigen Fällen sehr nützlich sein kann. Es ist vor allem die klare und offene Struktur von Linux und die

reichhaltige Vielfalt an Diagnosehilfsmitteln, welche Benutzer anderer Betriebssysteme neidisch machen würden¹.

Manchmal vergleiche ich Linux und ein bekanntes Konsumbetriebssystem mit einem DIN-gerechten Schaltschrank und einer hochintegrierten digitalen Armbanduhr. Verweigert die Uhr ihren Dienst, kannst du nur zwei Dinge ausprobieren:

- ❑ die Batterie wechseln oder
- ❑ die Uhr wegwerfen.

Der Schaltschrank hingegen wird mit einem Schaltplan ausgeliefert. Jedes Teil kannst du einzeln austauschen. Alle Kabel, Schraubklemmen, Relais, Netzteile und andere Bauelemente sind übersichtlich angeordnet und beschriftet. Wenn du dich mit Elektrik etwas auskennst und über ein einfaches Messgerät sowie etwas Geduld und methodisches Vorgehen verfügst, findest du jeden Fehler – und kannst ihn dann auch beheben.

Diese Möglichkeit als *nicht benutzerfreundlich* zu bezeichnen, ist natürlich unklug, denn wenn der Schaltschrank geschlossen ist, tut er seine Dienste mindestens ebenso zuverlässig wie die digitale Uhr. Und auch wenn du kein gelernter Elektriker bist, hast immer noch die Möglichkeit, einen anzuheuern, der dir das Problem löst.

The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong it usually turns out to be impossible to get at and repair.

– Douglas Adams (Mostly Harmless, 1992)

1.2 Das Besondere an Linux

Damit du besser verstehst, was das Besondere an der Fehlersuche unter Linux ist, beleuchte ich erst einmal, wie eine typische Fehlersuche bei einem Konsumbetriebssystem aussieht, welches ich exemplarisch *Bimbaus* nennen will. Bei meinem Kollegen Winni aus dem Bimbaus-Lager beobachte ich folgendes Vorgehen, wenn irgendeine Funktion einmal nicht so arbeitet, wie er es für normal hält:

1. Winni probiert die Funktion noch einmal.
2. Winni probiert Variationen aller Optionen in allen Konfigurationsfenstern.
3. Winni startet das Programm neu.
4. Winni startet den Rechner neu.

¹oder abtrünnig!

5. Winni installiert das Programm neu.
6. Winni installiert – falls vorhanden – eine neuere Version des Programms.
7. Winni installiert Bimbaus und alle benötigten Programme neu.

Die Methode von Winni besteht also darin, Dinge zu *wiederholen* und Varianten *auszuprobieren*. Hat er nach langem Suchen eine Variante gefunden, mit der alles funktioniert, ist er erleichtert und gibt in einer Rundmeldung an seine Kollegen die Anweisung, *jetzt ja nichts mehr anzufassen!*

Warum gefällt mir die Sache nicht so recht?

- Winni hat aus der Sache wenig lernen können.
- Irgendwann bastelt doch wieder jemand herum (vielleicht weil er muss) und das Ganze kracht wieder auseinander. Oder es kracht einfach von selbst auseinander.

Woran das liegt? Winni ist dem Fehler nicht auf den *Grund* gegangen! Er hat nur seit seinen Maßnahmen das Symptom nicht mehr beobachtet. Der Fehler kann also immer noch vorhanden sein, tritt aber aus nicht bekannten Gründen jetzt seltener oder unter anderen Umständen auf.

Ich sage nicht, dass dies die Schuld von Winni ist. Ich kenne sehr erfahrene Administratoren, die nach obigem Schema arbeiten. Man erkennt solche Leute an einem bewundernswerten Faktenwissen, z. B. über die verschiedenen Fehler in diversen Service-Packs oder über die Reihenfolge, in welcher bestimmte Programme installiert werden müssen.

Bei Linux ist die Situation zum Glück etwas anders, weil man hier viel mehr Möglichkeiten hat, einem Fehler wirklich auf den Grund zu gehen. Deshalb empfehle ich folgende Methode:

- 1. Reproduziere den Fehler!**
- 2. Finde die exakte Ursache!**
- 3. Behebe das Problem an der Ursache!**

So wirst du nicht nur sicher sein, dass ein „behobener“ Fehler auch tatsächlich beseitigt ist, sondern du wirst auch im Laufe der Zeit eine Menge über Linux lernen. Du wirst mehr Zeit haben für wichtige Dinge; Zeit, die andere mit dem Ausprobieren von Varianten, dem Neuinstallieren von Programmen und dem Fluchen über Fehler, die doch „*eigentlich schon weg*“ waren, verplempern.

1.3 Den Fehler reproduzieren

Bevor du nach der Ursache eines Fehlers fahnden kannst, musst du erst einmal wissen, unter welchen Umständen der Fehler auftritt. Am besten ist es, wenn du

den Fehler zuverlässig reproduzieren kannst. Das bedeutet, dass du genau weißt, welche Schritte du machen musst und welche Umstände bestehen müssen, damit der Fehler auftritt. Dies herauszufinden, nenne ich „den Fehler festnageln“.

Stell dir vor, du administrierst ein Computernetzwerk. Einer der Anwender berichtet dir in einer E-Mail darüber, dass sein Lieblingsprogramm manchmal abstürzt und erwartet von dir natürlich eine schnelle Lösung.

Aber was kannst du tun? Du startest das Programm, spielst ein bisschen damit herum und teilst dem enttäuschten Anwender mit, dass bei *dir* alles wunderbar funktioniert. Natürlich ist es so unmöglich, die Ursache des Fehlers zu finden.

Was du eigentlich benötigst, ist eine Anleitung, welche dir Schritt für Schritt erklärt, welche Eingaben, Befehle, Menüpunkte usw. du aufrufen musst, damit der Fehler auftritt.

Erst wenn es dir gelingt, den Fehler anhand dieser Anleitung zuverlässig zu reproduzieren, kannst du mit der gezielten Diagnose beginnen. Wenn du dann später den Fehler gefunden und scheinbar behoben hast, kannst du mit derselben Anleitung gegenprüfen, ob der Fehler auch wirklich beseitigt wurde.

Leider gibt es Fehler, bei denen die Umstände, unter denen sie auftreten, entweder nicht ergründet oder nicht kontrolliert werden können. Dabei spielt das Zeitverhalten die zentrale Rolle.

Beispielsweise kann ein Fehler davon abhängen, welches von mehreren Ereignissen als erstes eintritt, wie lange ein Vorgang braucht oder was auf dem Netzwerk gerade passiert. In einem solchen Fall kannst du den Fehler nicht hundertprozentig festnageln. Es kann sein, dass der Fehler nur einmal pro Minute, Stunde oder Woche auftritt oder auch etwa jedes 25ste Mal oder in völlig zufälligen Abständen.

Im Extremfall ist der Fehler nur ein einziges Mal aufgetreten und nicht reproduzierbar, aber von so zentraler Bedeutung, dass unbedingt ausgeschlossen werden muss, dass er noch einmal auftritt. Ein historisches Beispiel ist der missglückte Jungfernflug der ARIANE-5-Trägerrakete, welche sich wenige Sekunden nach dem Start selbst sprengte. Eine Reproduktion des Fehlers schloss sich aus Kostengründen aus. Der Aufwand, der deshalb von der ESA bei der Suche nach der Ursache betrieben werden musste, zeigt die Wichtigkeit der Fehlerreproduktion.

Manchmal kann man einen Fehler in flagranti erwischen, also unmittelbar während des Auftretens. Mit einer sofortigen sorgfältigen Diagnose kann es dir gelingen, die Ursache auf der Stelle zu finden. Eine Reproduktion des Fehlers ist dann nicht mehr notwendig.

1.4 Die exakte Ursache finden

Das Finden der Fehlerursache ist sicherlich die spannendste Aufgabe. Natürlich bedeutet dies, dass man seinen Grips etwas anstrengen muss, aber in diesem Buch werde ich dir ein Arsenal von Werkzeugen und Methoden vorstellen, mit denen du dich auch ohne die Gabe der Hellseherei an die Ursache heranarbeiten kannst.

Findet man die *wahre* Ursache eines richtig knackigen Problems, ist dies ein großes Erfolgserlebnis, vor allem dann, wenn man eindeutig feststellen kann, dass jemand anders Schuld hatte. Alle Variantenprobierer und Neuinstallierer werden so etwas nie erleben.

Abgesehen von den ganzen Linuxtricks und Werkzeugen, die du in diesem Buch findest, gibt es grundsätzliche Methoden, mit denen du einen Fehler einkreisen kannst. Diese will ich nun vorstellen.

1.4.1 Der Einblick

Die Methode des *Einblicks* kann man immer dann anwenden, wenn das Fehlersymptom – sobald es auftritt – für mindestens eine gewisse Weile bestehen bleibt. Dies ist z. B. der Fall bei einem Auto, bei dem das Rücklicht nicht funktioniert. Hier hast du die Möglichkeit, Einblick in die Lage zu nehmen. Du kannst z. B. messen, ob an der Fassung für die Birne Spannung anliegt oder ob die Autobatterie geladen ist. Nimm als Gegenbeispiel den Fall, dass dir dreimal in der Woche im Rücklicht das Birnchen durchbrennt. Ein Einblick ist unmöglich, denn der Fehler – das Durchbrennen – dauert nur einen winzigen Augenblick.

Da die Methode des Einblicks schnell zum Erfolg führen kann, sollte man versuchen, sie wann immer möglich wahrzunehmen. Bei sporadischen Fehlern, die man nicht festnageln *kann*, muss man eventuell lange auf das nächste Auftreten warten. Wenn es dann endlich soweit ist, sollte man alles vorbereitet haben, um Einblick nehmen zu können. Und man muss aufpassen, dass man durch die Diagnose die Fehlersituation nicht verfälscht oder zerstört. Manchmal ist man darauf angewiesen, dass ein Anwender oder Kunde mitspielt und sich bei einem Fehler sofort meldet, ohne selbst am Fehlersymptom herumzudoktern.

Ein Beispiel aus der Computerwelt: Manchmal kommt es vor, dass ein Programm „hängenbleibt“ und nicht mehr auf Benutzereingaben reagiert. Mit den Werkzeugen, die ich später vorstellen werde, kann man untersuchen, was genau das Programm gerade macht oder auf was genau es gerade wartet. Bei einem Programm, das abgestürzt ist, kann man nicht Einblick nehmen, da das Abstürzen selbst ja keine Dauer hat.

1.4.2 Spurensuche

Bei der *Spurensuche* versucht man, die Spuren, die ein Programm hinterlässt, auszuwerten. Dabei will man die Frage klären, was das Programm genau gemacht hat, bis der Fehler auftrat. Dann kann man versuchen, darin Unregelmäßigkeiten zu entdecken, die einen Hinweis auf die Ursache des Fehlers geben.

Um die Spurensuche zu erleichtern, pflegen manche Programme sogenannte Logdateien, in der alle wichtigen Aktionen oder Ereignisse protokolliert werden. Auch Flugzeugbauer verwenden diese Methode in Form des Flugschreibers, der nicht nur Messdaten mitschreibt, sondern auch die Gespräche der Piloten.

Um die Spuren richtig deuten zu können, braucht man ein Verständnis vom gesamten Ablauf, also eine Menge Hintergrundwissen – im Gegensatz zum Einblick, bei dem man nur die aktuelle Situation verstehen muss.

1.4.3 Die Wirkungskette

Bei Fehlern, die man festnageln und reproduzieren kann, ist die *Wirkungskette* die wichtigste und universellste Methode. Sie basiert auf der Art, wie viele komplexe technische Systeme aufgebaut sind – besonders im Computerbereich. Nehmen wir folgendes Beispiel:

Du hast von einem Bekannten den Sommer über eine Fischerhütte in einem abgelegenen norwegischen Dorf gemietet. Der Bekannte ist aber selbst auf Reisen und hat dir mit der Post nur eine Wegbeschreibung und einen Schlüssel geschickt. Die Anreise läuft problemlos, die findest die Hütte, sperrst sie auf, schaltest das Licht ein und – *es bleibt dunkel*.

Was kannst du tun? Den Lichtschalter noch ein paar mal malträtiert? Einen anderen Lichtschalter suchen? Die Birne auswechseln? Nach der Sicherung schauen? Zum Nachbarn gehen und fragen, ob gerade Stromausfall ist?

Um die möglichen Fehlerursachen zu erkennen, muss man sich überlegen, wie die Wirkung des einen technischen Systems die Grundlage für das nächste bildet. Ich habe das hier mal etwas vereinfacht dargestellt:



Dieses Beispiel zeigt zwei wichtige Regeln einer Wirkungskette:

1. Wenn ein Glied fehlerhaft ist, funktioniert dieses *und alle darauf folgenden* Glieder nicht mehr. Ein Beispiel: kommt aus der Sicherung kein Strom, so liefern auch der Schalter und die Lampenfassung keinen Strom, und die Glühbirne leuchtet nicht.

2. Wenn ein Glied funktioniert, beweist dies, dass auch *alle vorhergehenden* Glieder funktionieren. Beispiel: Kommt aus der Lampenfassung Strom, so liefern auch der Lichtschalter, die Sicherung und das Kraftwerk Strom.

Für die Fehlersuche kann man diese beiden Prinzipien ausnutzen. Man pickt sich eine Stelle der Kette heraus und testet die dortige Wirkung. Ist die Wirkung vorhanden, so kann man gemäß Regel 2 einen Fehler in einem Glied bis dorthin ausschließen. Findet man jedoch keine Wirkung vor, so kann man einen Fehler im Rest der Kette ausschließen². Man hat also die Suche auf einen Teil der Kette reduziert: auf eine kürzere Wirkungskette, bei der man wieder nach demselben Schema vorgehen kann.

Wenn man die Teststellen geschickt wählt – nämlich in der Mitte der Kette – kann man jeweils die Hälfte der Ursachen ausschließen! In unserem Beispiel würde das z. B. heißen, dass man eine Steckdose ausprobiert, beispielsweise mit einem mitgebrachten Föhn. Funktioniert der Föhn, so kann man Stromausfall und kaputte Sicherung ausschließen. Funktioniert er nicht, so kann man erst einmal davon ausgehen, dass weder Lichtschalter, Lampe noch Glühbirne schuld sind.

Wichtig ist, dass man versteht, dass durch einen Test an einer Stelle, immer die ganze Wirkungskette bis dorthin geprüft wird. Nimm als Beispiel den Lichtschalter: Prüft man ihn, indem man ihn ausbaut und mit einem Messgerät die Schaltfunktion misst, erfährt man nichts über die restlichen Teile der Wirkungskette. Ist der Lichtschalter zufällig das Problem, hat man zwar Glück gehabt. Ist er es nicht, hat man aber nichts über den Zustand der verbleibenden Komponenten erfahren.

Lässt man hingegen den Lichtschalter im Stromkreis und misst nur, ob beim Ausgang Strom herauskommt, so hat man nach dem Prinzip der Wirkungskette gearbeitet und entweder alle Komponenten vor oder eben nach dem Schalter als Fehlerquellen ausgeschlossen.

Für die Wahl einer günstigen Stelle spielt es natürlich auch eine Rolle, wie einfach ein Test durchzuführen ist. Um festzustellen, ob der Lichtschalter tatsächlich Strom liefert, muss man diesen öffnen und ein Messgerät oder einen Phasenprüfer zur Hand haben. Da ist es schon einfacher, einen Föhn in eine Steckdose zu stecken.

Fazit: Versuche das Prinzip der Wirkungskette zu verstehen, Wirkungsketten in der Praxis zu erkennen und diese mit gut überlegten Tests auszunützen. Vor allem im Bereich der Netzwerke gibt es sehr lange Wirkungsketten, bei denen diese Methode besonders vorteilhaft ist.

²Der Einfachheit halber gehen wir davon aus, dass es nur einen Fehler gibt. Auch wenn das nicht so ist, muss hier mindestens ein Fehler im Anfangsteil der Kette liegen.

1.4.4 Der wandernde Fehler

Das Prinzip des *wandernen Fehlers* ist vor allem in der Elektronik bekannt. Nimm an, du hast eine Telefonanlage, die soeben ihren Geist ausgehaucht hat. Sie besitzt etliche auswechselbare Teile, z. B. einen Epromchip und ein ISDN-Modul. Wie kannst du herausfinden, ob Chip, Modul oder der Rest kaputt ist?

Finde einen Freund, der exakt die gleiche Anlage hat und welche noch funktionstüchtig ist. Nun baue in seine Anlage dein ISDN-Modul ein und umgekehrt. Wenn der Fehler *wandert*, also nun bei der Anlage deines Freundes auftritt, aber deine dafür funktioniert, so ist der Fall klar: das ISDN-Modul ist schuld.

Bleibt der Fehler bei deiner Anlage, weißt du mit gleich großer Sicherheit, dass das ISDN-Modul gut funktioniert und der Fehler in einem anderen Teil zu suchen ist³.

Das Prinzip des wandernden Fehler kann man auch auf den Softwarebereich übertragen. Nimm an, du verwendest zum Abruf deiner E-Mail den Mailer `pine`. Dein Freund, der beim gleichen Provider ist, arbeitet stattdessen mit `elm`. Du hast Probleme, beim Abrufen deiner Mails, er jedoch nicht. Als Test verwendet dein Freund `pine` und du `elm`. Klappt nun bei dir der Abruf aber bei deinem Freund nicht mehr, so hängt das Problem irgendwie mit `pine` zusammen. Bleibt der Fehler jedoch bei dir, so ist der Mailer zumindest nicht schuld daran.

1.4.5 Wechselnde Umstände

Das Prinzip der *wechselnden Umstände* ist verwandt mit dem bereits erwähnten *Festnageln*. Ein Fehler, der nur unter ganz bestimmten Umständen auftritt, verrät seine Ursache gelegentlich – aber nicht immer – durch eben diese Umstände. Dazu musst du solange mit verschiedenen Rahmenbedingungen experimentieren, bis du eine möglichst kleine Änderung bestimmen kannst, die über Fehler oder nicht-Fehler entscheidet.

Hier ist ein Beispiel aus dem Netzwerkbereich: Ich habe einmal für eine Firma eine Firewall eingerichtet. Diese funktionierte gut und der Kunde war zufrieden, bis ich nach etlichen Wochen hörte, dass es einem Mitarbeiter nicht möglich war, eine bestimmte Firma im Internet zu erreichen. Da alle anderen Seiten problemlos funktionierten, nahm ich natürlich an, dass das Problem beim Server jener Firma lag.

Was mich dann stutzig machte war die Tatsache, dass ich die Firma von meinem eigenen Netz aus gut erreichen konnte. Das Problem konnte also nicht bei der

³Und selbst, wenn du den Fehler nicht findest, kannst du wenigstens das ISDN-Modul im Internet versteigern ;-)

Firma liegen, sondern musste mit dem Internetzugang meines Kunden zu tun haben⁴.

Nach einigem Probieren fand ich noch mehr Webauftritte, die nicht erreichbar waren, und konnte die Bedingungen, die zur Nichterreichbarkeit führten, genau feststellen: Alle Rechner, deren Internetadresse mit einer 192 begannen, waren nicht erreichbar. Und hier fiel bei mir der Groschen: Bei der Konfiguration der Firewall hatte ich das Netz, das mit 192.168 beginnt als *intern* klassifizieren wollen, hatte mich jedoch dahingehend verkonfiguriert, dass anstelle dessen das komplette 192er Netzwerk als intern galt.

Fazit: Der entscheidende Hinweis für mich war der genaue *Fehlerumstand*. Allein mit der Aussage „manche Seiten sind nicht erreichbar“, wäre ich nicht (so schnell) darauf gekommen. Als ich aber merkte, dass nur die Adressen Probleme machten, die mit 192 anfangen, konnte ich den Fehler schnell finden.

1.4.6 Reine Geisteskraft

Die Methode der *Reinen Geisteskraft* besteht schlichtweg darin, dass man den Quellcode des fehlerhaften Programms Zeile für Zeile durchgeht und alle Schritte präzise im Kopf nachvollzieht. So versucht man, allein durch die Kraft des Geistes zu ergründen, in welcher der Programmzeilen sich der Fehler versteckt. Folgende Voraussetzungen müssen gegeben sein:

- ❑ Der Programmcode muss offen liegen.
- ❑ Man muss jede Programmzeile zu 100% verstehen.
- ❑ Man muss das Verhalten aller beteiligten Komponenten genau kennen.

Es versteht sich von selbst, dass die Chance, einen Fehler auf diese Art zu finden, rapide mit der Größe des Programmes abnimmt. Man wird also versuchen, mit anderen Methoden den Fehlerort bereits so weit einzugrenzen, dass die zu durchforstende Programmstelle möglichst klein wird. Vielleicht sind es dann nur noch ein paar Programmzeilen, bei denen man sich gut überlegen muss, warum gerade hier das Programm abstürzt.

Da die Methode der Reinen Geisteskraft sehr mühsam und anspruchsvoll ist, solltest du sie nur dann einsetzen, wenn du wirklich keine andere Idee mehr hast, wie du durch geschickte Diagnose den Fehler weiter einkreisen kannst.

1.5 Den Fehler beheben

Natürlich ist es immer am schönsten, wenn man einen Fehler an der eigentlichen *Ursache* behebt. Ein klassischer Programmierfehler lässt sich beispielsweise fast

⁴Wenn man genau hinsieht, erkennt man hier wieder das Prinzip des *wandernen Fehlers*.

immer durch ein Ändern oder Hinzufügen von einer oder zwei Programmzeilen beheben – wenn man den Quellcode zur Verfügung hat.

Wenn ein Beheben nicht möglich ist, kann man *Symptombekämpfung* versuchen, was bedeutet, dass der Fehler zwar nach wie vor auftritt, dessen Wirkung aber durch einen *genialen Trick* vermieden oder repariert wird. Betrachte den Fall, dass ein Serverdienst gelegentlich abstürzt. Eine Symptombekämpfung wäre, ein Programm zu schreiben, das einmal pro Sekunde prüft, ob der Dienst noch läuft und wenn nicht, diesen wieder startet. Genial, oder?

Wenn dir ein Kollege so etwas als Lösung präsentiert, sollten bei dir sofort alle Alarmglocken läuten. Wer immer so arbeitet, schafft nach einiger Zeit ein Flickwerk von Provisorien, das früher oder später an allen Stellen gleichzeitig auseinanderkracht, spätestens aber dann, wenn ein anderer die Wartung übernehmen muss.

Die dritte und eine ebenfalls unbefriedigende Möglichkeit ist die *Vermeidung* des Fehlers. Dies wird erreicht, indem man dafür sorgt, dass die Umstände, die zum Fehler führen, nie eintreten. Wenn der Fehler z. B. immer dann auftritt, wenn man einen bestimmten Menüpunkt aufruft, dann ruft man diesen Menüpunkt in Zukunft eben nicht mehr auf. Dies geht natürlich nur dann, wenn man die Umstände genau kennt und auf die fehlerhafte Funktion verzichten kann!

Fazit: Da du ehrgeizig bist und außerdem das Glück hast, mit Linux arbeiten zu dürfen, kommt für dich natürlich nur die Ursachenbehebung in Frage!

1.6 Ein Wort zur 2. Auflage

Und nun – zum Abschluss meiner Einführung in das Thema dieses Buches – möchte ich noch ein Wort zur zweiten Auflage verlieren.

Bereits wenige Wochen nach dem Verkaufsstart dieses Buches im Februar 2004 wurde sichtbar, dass es eines der erfolgreichsten Titel von SuSE PRESS werden würde. Dass jedoch nach so kurzer Zeit die Startauflage vergriffen sein würde, überraschte sowohl mich, als auch Nico Millin – meinen netten Verleger.

Um den Nachschub für die lesehungrigen Linuxer sicherzustellen, hätte ein Nachdruck vielleicht genügt. Ich nutzte jedoch gerne die Gelegenheit für eine aktualisierte und erweiterte zweite Auflage. Dies hat vor allem zwei Gründe:

- Der große Versionssprung beim Linux-Kernel von 2.4 auf 2.6 ist nun vollzogen. Mit SUSE LINUX 9.1, Mandrakelinux 10, Fedora Core 2 und anderen haben wichtige Distributionen bereits auf den neuen Kernel umgestellt oder werden dies in Kürze tun. Für den Anwender ändert sich dadurch zwar nichts Grundlegendes, aber das ein oder andere Detail ist eben doch etwas anders. Darauf möchte ich in meinem Buch natürlich Rücksicht nehmen.

- Ich hatte die Idee zu einem neuen Kapitel über Performanceprobleme, welches in diesem Buch nun die Nummer 9 tragen darf.

Außerdem wurde das Buch in vielen weiteren Details gepflegt und poliert. Am auffallendsten ist sicherlich das etwas verfeinerte Layout der Beispiele und Kurzreferenzen sowie ein neuer Anhang B mit einer Auflistung aller in diesem Buch vorgestellten Programme nebst der Information, wo du diese auf deiner Distribution findest oder aus dem Internet holen kannst.

Ich danke hiermit allen, die mir bei meiner Arbeit geholfen haben und wünsche viel Spaß beim Lesen und Problemlösen!

München, im Juni 2004

Mathias Kettner

```
Eiger:~ # rm /mnt/?
Eiger:~ # df -i /mnt
Dateisystem          INodes  IBenut.  IFrei  IBen%  Eingehängt auf
/home/mk/part        16      11       5     69%   /mnt
```

Elf Inodes verbraucht das ext2-Dateisystem für interne Zwecke, unter anderem eine für das Wurzelverzeichnis und eine für das Verzeichnis `lost+found`.

Am Ende solltest du wieder aufräumen:

```
Eiger:~ # umount /mnt
Eiger:~ # rm part
```

Übrigens: manche neuere Dateisysteme, wie z. B. das Reiserfs, organisieren die Dateien ohne Inodes und haben gewissermaßen unendlich viele Inodes zur Verfügung.

4.3 Prozesse

Ein Prozess ist wörtlich übersetzt ein *Fortschreiten*. Mit anderen Worten sind die Prozesse das, was im Linuxrechner gerade *abläuft*. Dies darf man nicht mit Programmen verwechseln. Genau gesagt ist ein Prozess der Ablauf eines Programmes, wobei von einem Programm zu einem Zeitpunkt natürlich mehrere Prozesse existieren können.

Man kann das gut mit einem Kochbuch vergleichen: Ein Rezept ist wie ein Programm: Eine Liste von Anweisungen. Aber wenn ich am Freitag abend in der Küche stehe und nach dem Rezept koche, ist das ein Prozess. Klar, dass andere Menschen auf der Welt zur gleichen Zeit nach dem gleichen Rezept kochen können.

4.3.1 Prozessbaum anzeigen mit `pstree`

Um sich einen Überblick über die gerade laufenden Prozesses zu verschaffen, ist ein Aufruf von `pstree` ideal. Es stellt alle Prozesse übersichtlich als Baumstruktur dar. Denn Prozesse existieren bei Linux nicht parallel nebeneinander, sondern jeder Prozess hat einen Vaterprozess, welcher ihn geschaffen hat. Es gibt nur einen einzigen Prozess, welcher als Vater keinen Prozess, sondern den Linux-Kernel selbst hat. Dieser heißt aber nicht Adam, sondern `init`.

```
pstree
zeigt den Prozessbaum, entweder ganz, oder ab
Prozess PID.

pstree [OPTIONEN] [PID]
-----
-p      zeigt Prozess-IDs an
-u      zeigt Benutzerwechsel (UID Transitionen) an
```


Daher ist mein Vorschlag: Versuche nicht, den `ps`-Befehl wirklich zu verstehen, merke dir einfach drei Aufrufvarianten:

- ❑ `ax`,
- ❑ `afx` und
- ❑ `aux`.

Für die andere Fälle verwende `pstree`.

`ps ax` zeigt alle Prozesse an und zwar mit der kompletten Kommandozeile. Die Variante `ps afx` öffnet ein bisschen `pstree` nach, indem es eine Baumstruktur sichtbar macht. `ps aux` zeigt zu jedem Prozess noch ein paar Daten mehr an, darunter die Spalte `USER` (Eigentümer des Prozesses), `%CPU` (aktueller Verbrauch an Rechenleistung) und `%MEM` (aktueller Speicherverbrauch). Sehr sinnvoll ist bei `ps` immer die Kombination mit `grep`:

```
Eiger:~ # ps aux | head -n 1
USER      PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
Eiger:~ # ps aux | grep getty
root      1090  0.0  0.1  1396   584 tty1      S   13:27   0:00 /sbin/mingetty --...
root      1092  0.0  0.1  1396   584 tty3      S   13:27   0:00 /sbin/mingetty tty3
root      1093  0.0  0.1  1396   584 tty4      S   13:27   0:00 /sbin/mingetty tty4
root      1094  0.0  0.1  1396   584 tty5      S   13:27   0:00 /sbin/mingetty tty5
root      1095  0.0  0.1  1396   584 tty6      S   13:27   0:00 /sbin/mingetty tty6
root      2280  0.0  0.1  1752   676 pts/3    S   17:11   0:00 grep getty
Eiger:~ # ps aux | grep wmaker
mk        1138  0.0  0.5  5288  2992 ?        S   13:30   0:01 /usr/X11R6/bin/wmaker
root      2290  0.0  0.1  1752   676 pts/3    S   17:13   0:00 grep wmaker
```

4.3.3 Prozessmonitor `top`

Der beliebte Befehl `top` unterscheidet sich von `ps` und `pstree` dadurch, dass er sich fortlaufend aktualisiert. Erst durch Drücken von **Q** wird `top` beendet.

`top` zeigt alle Prozesse, die auf eine Bildschirmseite passen, sortiert nach deren aktuellem Verbrauch an Rechenleistung. Mit der Option `-i` werden sogar nur die aktiven Prozesse angezeigt, also diejenigen, die nicht gerade auf ein Ereignis warten oder angehalten sind. Damit kannst du auf einen Blick sehen, was gerade im System „passiert“.

Als nette Dreingabe zeigt `top` gleich noch ein paar allgemeine Werte zur aktuellen Speicher- und Rechenauslastung an. Die interessantesten davon sind `us`, `sy` und `id`:

- ❑ `us`
Dies ist die Zeit, die Prozesse im sogenannten Benutzermodus (*user*) verbringen, also in denen sie wirklich rechnen und CPU-Zeit verbrauchen.

4 Aktive Diagnose

❑ sy

Den hier angegebenen Prozentsatz an Zeit benötigt der Linux-Kernel (*system*) für seine Aufgaben, wobei dies durchaus im Auftrag eines Prozesses geschehen kann.

❑ id

Die Zeit, in der die CPU gar nichts macht, wird sie als *idle* bezeichnet. Früher galt das als Verschwendung von Rechenzeit, heutige Prozessoren sparen aber merklich an Strom, wenn sie untätig sind.

Wenn du mehr über die Bedeutung der Performance-Werte wissen willst, so findest du eine genau Beschreibung in Kapitel 9 auf Seite 354.

```
top - 16:30:16 up 6:52, 31 users, load average: 0.00, 0.02, 0.07
Tasks: 174 total, 1 running, 173 sleeping, 0 stopped, 0 zombie
Cpu(s): 1.0% us, 0.3% sy, 0.0% ni, 98.3% id, 0.0% wa, 0.3% hi, 0.0% si
Mem: 516744k total, 505120k used, 11624k free, 25872k buffers
Swap: 1028120k total, 49788k used, 978332k free, 100372k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
3441	root	15	0	185m	42m	143m	S	0.7	8.4	7:16.15	X
12793	mk	17	0	1760	976	1540	R	0.7	0.2	0:00.11	top
3699	mk	16	0	66764	39m	25m	S	0.3	7.8	3:54.24	MozillaFirebird
1	root	16	0	588	144	444	S	0.0	0.0	0:04.37	init
2	root	34	19	0	0	0	S	0.0	0.0	0:00.87	ksoftirqd/0
3	root	5	-10	0	0	0	S	0.0	0.0	0:00.72	events/0
17	root	5	-10	0	0	0	S	0.0	0.0	0:00.51	kblockd/0
27	root	5	-10	0	0	0	S	0.0	0.0	0:00.76	pdflush
29	root	15	0	0	0	0	S	0.0	0.0	0:02.83	kswapd0
190	root	16	0	0	0	0	S	0.0	0.0	0:00.00	kseriod
495	root	6	-10	0	0	0	S	0.0	0.0	0:00.00	kcopypd/0
1301	root	22	0	2500	1264	2288	S	0.0	0.2	0:00.00	hotplug
1302	root	24	0	1360	380	1200	S	0.0	0.1	0:00.00	logger
1318	root	21	0	2500	1260	2288	S	0.0	0.2	0:00.00	pci.agent
1319	root	25	0	1360	380	1200	S	0.0	0.1	0:00.00	logger
1357	root	15	0	0	0	0	S	0.0	0.0	0:00.23	khudb

Neben den von `ps` bekannten Spalten `PID`, `USER`, `%CPU`, `%MEM` und `COMMAND` gibt es folgende Spalten:

❑ PRI

Priorität des Prozesses, sie ergibt sich aus dem `nice`-Wert. Je länger ein Prozess auf etwas wartet, desto höher wird seine Priorität. Beide Werte sind von untergeordneter Bedeutung.

❑ NI

`nice`-Wert des Prozesses.

❑ SIZE

Gesamtgröße des Prozesses, also von Programm, Bibliotheken und Daten belegter Adressraum.

❑ RSS

Aktuell vom Prozess physikalisch belegter Speicher. Teile des Adressraums eines Prozesses können durch das Abbilden einer Datei belegt sein, welche

noch nicht geladen wurde. Dies erklärt, warum `SIZE` und `RSS` unterschiedlich sein können. `RSS` zeigt also, wie viel Speicher der Prozess wirklich kostet.

□ `SHARE`

Der Teil des Speichers von `SIZE`, der gemeinsam mit anderen Prozessen genutzt wird. Dabei handelt es sich in der Regel um dynamische Bibliotheken (shared libraries), welche von mehreren Prozessen gleichzeitig benutzt werden.

□ `STAT`

Aktueller Zustand des Prozesses. Hier ist eigentlich nur der erste Buchstabe interessant: `S` steht für *sleeping* (Prozess wartet), `R` für *running* (Prozess arbeitet), `Z` für einen Zombieprozess (Prozess wurde beendet aber vom Vaterprozess noch nicht quittiert), `T` für *traced* (Prozess wird getracet oder ist angehalten).

Für Ungeduldige gibt es noch die Option `-q`, die dafür sorgt, dass `top` nicht fünf Sekunden bis zur nächsten Anzeige wartet, sondern aktualisiert, so schnell es kann (was dazu führt, dass der `top`-Prozess dann natürlich selbst die Rangliste der CPU-Fresser anführt). Die Option `-n1` hingegen fährt `top` nicht im Dauerbetrieb, sondern macht – genau wie `ps` und `ps tree` – nur eine einmalige Momentaufnahme.

`top` kennt zudem eine ganze Reihe von Tastaturbefehlen, von denen ich einige im Übersichtskasten aufgelistet habe. Mit `H` kannst du eine Hilfe einblenden, welche du mit `←` wieder verlassen kannst.

<code>top</code>	
Prozess- und Speichermonitor mit sich ständig aktualisierender Ausgabe.	
<code>top [OPTIONEN]</code>	
<hr/>	
<code>-i</code>	zeigt nur aktive Prozesse
<code>-q</code>	schnellstmögliche Aktualisierung
<code>-n1</code>	einmalige Momentaufnahme
<hr/>	
<code>H</code>	Hilfe einblenden
<code>F</code>	Spalten ein-/ausblenden
<code>O</code>	Spaltenreihenfolge ändern
<code>></code>	Sortierspalte nach rechts
<code><</code>	Sortierspalte nach links
<code>I</code>	Nur aktive Prozesse zeigen
<code>↑ W</code>	Konfiguration speichern
<code>Q</code>	<code>top</code> beenden

Übrigens: ein Prozess ist ein *Zombie*, wenn er sich beendet hat, sein Vaterprozess ihn aber nicht mittels des Systemaufrufes `waitpid` oder `wait4` wieder eingesammelt hat²⁵. Dies deutet meist auf unsaubere Programmierung im Vaterprozess hin.

²⁵Diese Systemaufrufe erkläre ich in Abschnitt 7.2 auf Seite 241.

4.3.4 Prozess-IDs ermitteln mit `pidof`

Der kleine Befehl `pidof` ist praktisch, wenn du mal schnell wissen willst, welche Prozess-ID (PID) ein Prozess hat, ohne gleich mit `ps tree -p` oder `ps ax` und `grep` herumwurschteln zu müssen. `pidof` erhält einfach den Namen eines Prozesses und spuckt dann alle passenden PIDs aus:

pidof
zeigt PIDs von Prozessen mit einem bestimmten Namen.
`pidof PROZESSNAME`

```
Eiger:~> pidof mingetty
1095 1094 1093 1092 1090
Eiger:~> ps ax | grep mingetty
 1090 tty1      S      0:00 /sbin/mingetty --noclear tty1
 1092 tty3      S      0:00 /sbin/mingetty tty3
 1093 tty4      S      0:00 /sbin/mingetty tty4
 1094 tty5      S      0:00 /sbin/mingetty tty5
 1095 tty6      S      0:00 /sbin/mingetty tty6
2441 pts/3     R      0:00 grep mingetty
```

4.3.5 Offene Dateien anzeigen mit `lsof`

`lsof` bedeutet „list open files“ und zeigt an, welche Prozesse gerade welche Dateien und Verzeichnisse geöffnet haben bzw. verwenden. Das ist deshalb so interessant, weil unter Linux so ziemlich alles eine Datei ist, z. B. auch Netzwerksockets, Pipes und viele Gerätetreiber.

lsof
zeigt an, welche Prozesse welche Dateien offen haben.
`lsof DATEINAME`
`lsof -p P`

`-p P` zeigt Dateien von Prozess P

`lsof` ohne Argumente listet alle offenen Dateien aller Prozesse auf. Beim ersten Mal ist dies sicher sehr spannend. Um sinnvolle Informationen zu erhalten, benötigst du jedoch Optionen, welche die Ausgabe gezielt filtern.

Wichtig sind folgende zwei Varianten:

- ❑ `lsof -p PID` und
- ❑ `lsof /datei/name`.

Die erste Variante zeigt alle offenen Dateien eines Prozesses, die zweite zeigt alle Prozesse, welche eine bestimmte Datei geöffnet haben. Folgendes Beispiel zeigt, was `lsof` zu meinem Prozess `portmap` sagt, welcher bei mir gerade die PID 551 hat:

```
Eiger:~ # lsof -p 551
COMMAND PID USER  FD   TYPE DEVICE        SIZE  NODE NAME
portmap 551  bin  cwd   DIR    3,5         616    2 /
portmap 551  bin  rtd   DIR    3,5         616    2 /
portmap 551  bin  txt   REG    3,5      11280 60202 /sbin/portmap
portmap 551  bin  mem   REG    3,5     104484 10810 /lib/ld-2.3.3.so
portmap 551  bin  mem   REG    3,5     36895 15593 /lib/libwrap.so.0.7.6
portmap 551  bin  mem   REG    3,5     10797 10834 /lib/libutil.so.1
portmap 551  bin  mem   REG    3,5    1349081 10836 /lib/tls/libc.so.6
portmap 551  bin   0u   CHR    1,3             30955 /dev/null
portmap 551  bin   1u   CHR    1,3             30955 /dev/null
portmap 551  bin   2u   CHR    1,3             30955 /dev/null
portmap 551  bin   3u  IPv4   3105             UDP  *:sunrpc
portmap 551  bin   4u  IPv4   3106             TCP  *:sunrpc (LISTEN)
```

Der Prozess `portmap` hat also elf Dateien geöffnet. Etliche der Ausgabespalten kennst du schon von `ps` und `top`.

Die Spalte `FD` zeigt an, in welcher Weise die offene Datei bzw. das Verzeichnis verwendet wird:

- ❑ `cwd`
Das Verzeichnis ist das aktuelle Verzeichnis des Prozesses (*current working directory*).
- ❑ `rtd`
Das Verzeichnis ist das Wurzelverzeichnis des Prozesses (*root directory*). In der Regel ist dies das Verzeichnis `/`. Prozesse, welche mit dem Befehl `chroot`²⁶ in einem „chroot-Jail“ gestartet wurden, haben ein anderes Wurzelverzeichnis.
- ❑ `txt`
Diese Datei enthält das vom Prozess ausgeführte Programm. Besonders interessant ist, dass diese Datei seit dem Start des Programmes umbenannt und sogar gelöscht worden sein kann! Wenn du dies versuchst, wirst du sehen, dass `lsof` bei einer Umbenennung den neuen Namen und nach einem Löschen mit einem (`deleted`) darauf hinweist. Der Kernel löscht die Datei erst dann endgültig, wenn sie von keinem Prozess mehr geöffnet ist.
- ❑ `mem`
Diese Datei wurde mittels des Systemaufrufes `mmap` in die Speicher abgebildet (*gemappt*). Der dynamische Linker `ld.so` macht dies mit den dynamisch nachgeladenen Bibliotheken. In dem Beispiel siehst du, dass der Port-Mapper neben der `libc` auch die `libutil` verwendet.
- ❑ `0u`
Die Datei wurde normal geöffnet und hat den Dateideskriptor 0 (normalerweise Standardeingabe).

²⁶siehe Beschreibung auf Seite 103

- 1u
Die Datei wurde mit dem Deskriptor 1 geöffnet (normalerweise Standardausgabe).
- 2u
Die Datei wurde mit dem Deskriptor 2 geöffnet (normalerweise Standardfehlerkanal).
- 3...
Die Datei wurde mit dem Deskriptor 3 (usw.) geöffnet. Der Buchstabe bedeutet Lesezugriff (r), Schreibzugriff (w) oder wahlweiser Zugriff (u).

Die Spalte `TYPE` gibt den Dateityp an, wobei `DIR` (Verzeichnis, *directory*), `CHR` (zeichenorientiertes Gerät, *character device*) und `REG` (normale Datei, *regular file*) die häufigsten sind. In meinem Beispiel siehst du zwei Dateien vom Typ `IPv4`. Es handelt sich um IP-Sockets, und zwar einmal ein TCP- und einmal ein UDP-Socket.

Mit `lssof` kannst du auch die Frage beantworten, *welcher* Prozess eine bestimmte Datei geöffnet hat. Dazu gibst du den Dateinamen als einziges Argument an. In folgendem Beispiel ermittle ich, welche Prozesse gerade die Bibliothek `libcrypt` verwenden:

```
Eiger :~ # lssof /lib/libcrypt.so.1
COMMAND PID   USER  FD   TYPE DEVICE  SIZE NODE NAME
sshd     631   root  mem   REG   3,8 43371 4654 /lib/libcrypt.so.1
master   678   root  mem   REG   3,8 43371 4654 /lib/libcrypt.so.1
pickup   696  postfix mem   REG   3,8 43371 4654 /lib/libcrypt.so.1
qmgr     697  postfix mem   REG   3,8 43371 4654 /lib/libcrypt.so.1
cupsd    770   root  mem   REG   3,8 43371 4654 /lib/libcrypt.so.1
```

4.3.6 Alle Informationen aus erster Hand unter `/proc`

Wem die bisher vorgestellten Werkzeuge immer noch zu wenig Informationen über die Prozesse liefern, findet im `/proc`-Dateisystem die ganze Wahrheit. Auch wenn heute längst nicht mehr alle Dateien unter `/proc` nur Informationen über Prozesse darstellen, war dies der ursprüngliche Zweck dieses Dateisystems. Das Besondere an diesem Diagnoseinstrument ist, dass es kein Programm, sondern eben ein Dateisystem ist.

Und das funktioniert so: Der Linux-Kernel bereitet allerhand nützliche Informationen in Form von Textdateien auf und stellt sie unter `/proc` dar. Diese Dateien existieren nicht wirklich, sondern werden vom Kernel nur vorgegaukelt. Der Inhalt wird vom Kernel immer in dem Moment erzeugt, in dem man die Datei ausliest. So hat man immer aktuelle Informationen.

Anzeigen kannst du die meist kleinen Dateien bequem mit `cat`.

`/proc` enthält für jeden Prozess ein Unterverzeichnis, dessen Name die PID ist. Dieses enthält eine Hand voll Dateien mit Informationen über den Prozess. Auf meinem Laptop ist momentan gerade 385 die PID vom Prozess `dhcpcd`, dem DHCP-Client-Daemon:

```
Eiger:~ # ls /proc/385
.      auxv      delay      fd          mem         stat        task
..     cmdline   environ    mapped_base mounts      statm       wchan
attr   cwd        exe         maps        root        status
```

Interessant zur Diagnose sind folgende Dateien:

Die Datei `cmdline` enthält die Kommandozeile, mit der der Prozess gestartet wurde.

```
Eiger:~ # cat /proc/385/cmdline
/sbin/dhcpcd-H-D-N-Y-t999999-hlinuxeth0
```

Die einzelnen Argumente werden durch binäre Nullbytes getrennt! Dies macht deshalb Sinn, weil die Argumente Leerzeichen und auch alle anderen denkbaren Sonderzeichen enthalten können. Mit dem `tr`-Befehl²⁷ kannst du die Nullbytes durch Zeilenumbrüche ersetzen und so eine übersichtlichere Ausgabe erzeugen:

```
Eiger:~ # tr \\0 \\n < /proc/385/cmdline
/sbin/dhcpcd
-H
-D
-N
-Y
-t
999999
-h
linux
eth0
```

Das erste Argument (`/sbin/dhcpcd`) ist per Konvention der Name des Befehls, mit dem der Prozess gestartet wurde, dies ist aber nur eine Empfehlung an den Programmierer²⁸. Auf die Programmdatei selbst zeigt die `/proc`-Datei `exe`, die ein symbolischer Link ist:

²⁷`tr`: translate, näheres erfährst du aus der Man-Seite von `tr`.

²⁸In der Man-Seite des Systemaufrufs `execve` sieht man, dass die Programmdatei als erster Parameter und `argv[]` als zweiter Parameter übergeben wird, `argv[0]` ist oben beschriebenes erstes Argument.

4 Aktive Diagnose

```
Eiger:~ # 11 /proc/385/exe
lrwxrwxrwx  1 root  root    0 2002-11-05 18:25 /proc/385/exe -> /sbin/dhccpd
```

Weiterhin interessant ist ein Blick in das *Environment* eines Prozesses. Dies ist eine Liste von Umgebungsvariablen nach dem Schema `NAME=WERT`. Das Environment wird in `environ` aufgelistet und analog zu `cmdline` ausgegeben:

```
Eiger:~ # tr \\0 \\n < /proc/385/environ
CONSOLE=/dev/console
TERM=linux
SHELL=/bin/sh
OLDPWD=/etc/sysconfig/network
INIT_VERSION=sysvinit-2.82
RUN_FROM_RC=yes
REDIRECT=/dev/tty1
COLUMNS=106
PATH=/sbin:/usr/sbin:/bin:/usr/bin:/etc/sysconfig/network/scripts
vga=791
RUNLEVEL=5
PWD=/etc/sysconfig/network
PREVLEVEL=N
LINES=34
HOME=/
SHLVL=4
_=/sbin/startproc
DAEMON=/sbin/dhccpd
```

Es lohnt sich auf jeden Fall, wenn du dich einmal selbst in `/proc` umsiehst. Eine Dokumentation des `proc`-Dateisystems findest du in der Manpage man 5 `proc` oder – wenn du die Kernelquellen installiert hast – in der Datei `/usr/src/linux/Documentation/filesystems/proc.txt`.

4.4 Benutzer und Gruppen

Eine wichtige Eigenschaft eines Prozesses ist, unter welchem Benutzer er läuft. Ferner gehört jeder Prozess zu einer Reihe von Gruppen, von denen eine die Hauptgruppe ist. Beides hat Einfluss auf die Zugriffe auf bestehende Dateien, das Erzeugen von neuen Dateien und andere Vorgänge.

4.4.1 Gruppen eines Benutzers anzeigen mit `id`

Der Befehl `id` zeigt an, zu welchen Gruppen ein Benutzer gehört. Wird als Argument kein Benutzername angegeben, so wird der aktuelle Benutzer angenom-

```
Eiger:~ # nmap -p 1-256,443
```

Die einzige „friedliche“ Option, die ich noch vorstellen will, ist `-sU`. Damit versucht `nmap`, offene UDP-Ports zu finden. Dies ist weit schwieriger und unzuverlässiger als bei TCP, weil UDP ja keinen Verbindungsaufbau kennt und die Gegenseite in der Regel nur dann antwortet, wenn das Testpaket sich an das jeweilige Protokoll des Dienstes hält.

In der Tat macht `nmap` den Scan genau umgekehrt: Es macht sich zunutze, dass ein (kooperativer) Rechner auf ein UDP-Paket an einen nicht offenen Port mit einer ICMP-Meldung vom Typ *port unreachable* antwortet. Alle Ports, von denen keine Fehlermeldung kommt, werden als offen bezeichnet.

Wenn eine Firewall aktiv ist, die allen Netzwerkverkehr auf nicht benutzte Ports abblockt, erscheinen dann aber alle Ports als offen! Die Option `-sU` ist also allenfalls im lokalen Netz nützlich:

```
Eiger:~ # nmap -sU -p 1-100 10.10.0.7
```

```
Starting nmap V. 3.00 ( www.insecure.org/nmap/ )
Interesting ports on Blomberg.zemeplocha.de (10.10.0.7):
(The 98 ports scanned but not shown below are in state: closed)
Port      State  Service
53/udp    open   domain
67/udp    open   dhcpserver

Nmap run completed -- 1 IP address (1 host up) scanned in 11 seconds
```

Fazit: `nmap` ist ein einfach zu bedienendes Diagnosewerkzeug, mit dem du dir schnell einen Überblick verschaffen kannst, welche Netzwerkdienste ein Rechner zur Verfügung stellt. Ferner kannst du einen einfachen Sicherheitstest deines eigenen Servers machen. Ein TCP-Dienst, der von `nmap` nicht angezeigt wird, ist auch mit `telnet` nicht erreichbar.

6.4.7 Netzwerkverkehr mithören mit `tcpdump`

HTTP	FTP	SMTP	POP3	SSH	telnet	...	DHCP	DNS	...	ping	traceroute	...
TCP							UDP			ICMP		
IP												
ARP				PPP								
Ethernet			ISDN			Modem			DSL			

Während `netstat` nur allgemeine Daten über die Verbindungen anzeigt, kannst du mit `tcpdump` alle übertragenen Daten wörtlich „mithören“. Der Netzwerkverkehr wird davon nicht beeinflusst. Innerhalb eines lokalen Netzwerkes kannst du sogar von Rechner A aus Kommunikation zwischen Rechner B und C mithören³²!

³²Bei einem geschwichteten Netzwerk muss A evtl. im gleichen Segment wie B oder C liegen.

Anders als der Name `tcpdump` vermuten lässt, kann dieses Werkzeug auch UDP- und ICMP-Verkehr, IP-Pakete im Allgemeinen und sogar Ethernet-Header anzeigen und aufschlüsseln. Auch kann es für einige Protokolle Datenpakete der Applikationsschicht, wie z. B. DNS-Anfragen, inhaltlich aufschlüsseln.

`tcpdump` ist als Diagnosewerkzeug schwieriger zu erlernen als z. B. `ping` oder `netcat`, aber dafür universell, flexibel und präzise wie ein Schweizer Uhrwerk.

Einführende Beispiele

In folgendem Beispiel verwende ich `tcpdump`, um UDP-Datenpakete anzuzeigen. Die Option `-n` unterdrückt wie üblich die Auflösung von Rechnernamen:

```
Eiger:~ # tcpdump -n udp
tcpdump: listening on eth0
20:00:36.338392 10.10.0.9.32795 > 10.10.0.7.53: 43044+ A? gibtsnix.xyz. (32) (DF)
20:00:36.339173 10.10.0.7.53 > 10.10.0.9.32795: 43044 NXDomain 0/1/0 (107)
20:00:36.339522 10.10.0.9.32795 > 10.10.0.7.53: 43045+ A? gibtsnix.xyz.local. (46)
(DF)
20:00:36.340202 10.10.0.7.53 > 10.10.0.9.32795: 43045 NXDomain* 0/1/0 (120)
20:00:39.629105 10.10.0.9.32795 > 10.10.0.7.53: 6541+ A? www.muen.de. (33) (DF)
20:00:39.630154 10.10.0.7.53 > 10.10.0.9.32795: 6541 2/2/2 CNAME web.muen.de.
[domain]

6 packets received by filter
0 packets dropped by kernel
```

Jede Ausgabezeile entspricht einem IP-Paket und enthält:

1. einen Zeitstempel inkl. Microsekunden (in der ersten Beispielzeile 338392),
2. IP-Adresse (Rechnername) und Portnummer (Dienstname) des Senders (IP-Adresse 10.10.0.9, Quellport 32795),
3. IP-Adresse (Rechnername) und Portnummer (Dienstname) des Empfängers (IP-Adresse 10.10.0.7, Zielport 53),
4. einen variablen Datenteil (43044+ A? gibtsnix.xyz. (32)).
5. eventuell Flags der IP-Pakete. Ein (DF) bedeutet, dass bei dem jeweiligen Paket das *don't fragment*-Bit gesetzt ist, das die Fragmentierung des Paketes durch einen Router verbietet.

Der Aufbau des variablen Datenteils hängt vom Inhalt des Paketes und von den Portnummern ab. `tcpdump` kennt eine ganze Reihe von Protokollen und versucht, die Rohdaten der IP-Pakete nach den Leseregeln der jeweiligen Protokolle aufzuschlüsseln.

In obigem Beispiel sehen wir DNS-Anfragen und -Antworten. Ein Prozess auf Rechner 10.10.0.9 schickt an 10.10.0.7 auf Port 53 (domain) die beiden Anfragen `A? gibtsnix.xyz` und `A? gibtsnix.xyz.local`. Die Antwortet lautet beides Mal `NXDomain` (unbekannte Domain).

Eine dritte Anfrage nach `www.muen.de` wird positiv beantwortet mit `CNAME web.muen.de`. Selbst wenn du das DNS-Protokoll nicht verstehst, kannst du dir ein gutes Bild vom Frage- und Antwortspiel des DNS machen.

Folgendes Beispiel schneidet alle ARP-Anfragen und Antworten im lokalen Netzwerk mit³³:

```
Eiger:~ # tcpdump -n arp
tcpdump: listening on eth0
16:42:21.558038 arp who-has 10.10.0.3 tell 10.10.0.7
16:42:21.558076 arp reply 10.10.0.3 is-at 0:7:95:27:16:74
```

Um 15 Uhr 42, 21 Sekunden und 558038 Mikrosekunden hört die Netzwerkkarte `eth0` eine ARP-Anfrage von Rechner `10.10.0.7`: Er fragt nach der MAC-Adresse von `10.10.0.3`. Gerade mal 38 Mikrosekunden später kommt schon die Antwort: `10.10.0.3` hat die MAC-Adresse `00:07:95:27:16:74`.

Netzwerkkarte angeben

`tcpdump` hört standardmäßig auf der Netzwerkkarte `eth0`. Mit der Option `-i` kannst du eine andere Netzwerkkarte angeben. Es ist damit auch möglich, eine Einwahlverbindung mitzuschneiden, die über PPP läuft³⁴. Du musst nur das entsprechende Gerät mit der Option `-i` angeben, z. B:

```
Eiger:~ # tcpdump -i ppp0
```

Ausführlichkeit der Ausgabe

Mit einer Reihe von Optionen kannst du die Art und die Ausführlichkeit der Anzeige steuern. Die schon erwähnte Option `-n` unterbindet das Umrechnen von IP-Adressen und Portnummern in Namen. Neben einer knapperen Anzeige hat dies einen wichtigen Aspekt: Das Nachschlagen der Namen per DNS erzeugt selbst Netzwerkverkehr, der sich wiederum in der Ausgabe niederschlägt!

Auch mit `-N` kannst du die Ausgabe verkürzen: hier werden die Domainnamen weggelassen und nur die Rechnernamen angezeigt, was im lokalen Netzwerk eigentlich immer ausreicht.

So sieht z. B. ein dreimal erfolgreiches `ping` mit der Option `-N` aus. Da `ping` auf ICMP basiert, werden keine Portnummern ausgegeben, die gibt es nur bei TCP und UDP:

```
Eiger:~ # tcpdump -N
09:59:19.812261 Wethertop > Blomberg: icmp: echo request (DF)
```

³³ARP, Adress Resolution Protocol: siehe Abschnitt 6.2.4 auf Seite 169

³⁴Siehe Abschnitt 6.2.5 auf Seite 170

```
09:59:19.812538 Blomberg > Wethertop: icmp: echo reply
09:59:20.811262 Wethertop > Blomberg: icmp: echo request (DF)
09:59:20.811462 Blomberg > Wethertop: icmp: echo reply
09:59:21.810463 Wethertop > Blomberg: icmp: echo request (DF)
09:59:21.810667 Blomberg > Wethertop: icmp: echo reply
```

Die Option `-q` unterdrückt etliche Informationen, mit denen man so-wieso nur etwas anfangen kann, wenn man das jeweilige Protokoll gut kennt. Bei TCP werden z. B. Informationen über Sequenznummern und Quittierungsfenster angegeben, die du wirklich nur benötigst, wenn du auf tiefster TCP-Ebene arbeitest.

Wenn du *zusätzliche* Informationen über die Hardwareebene benötigst, kannst du diese mit `-e` abonnieren.

Bei Ethernetgeräten zeigt `-e` die MAC-Adresse von Sender und Empfänger sowie die Länge des Ethernetpaketes an.

Als Beispiel habe ich noch einmal das gleiche ping protokolliert wie im letzten Beispiel, diesmal jedoch mit den Optionen `-eqNt`.

Dabei kannst du die MAC-Adresse vom Sender des Ping-Paketes (00:04:76:4c:89:73) und der des Gegenübers (00:a0:cc:5b:f3:34) erkennen, sowie die Länge des Ethernetpaketes von 98 Byte. Ich habe ping mit einer Paketgröße von 56 Bytes Nutzdaten aufgerufen. Daraus kann man 42 Byte Protokoll-Overhead für Ethernet, IP und ICMP ausrechnen.

Die Option `-t` verhindert die Ausgabe des Zeitstempels am Anfang der Zeile.

```
Eiger:~ # tcpdump -eqNt
tcpdump: listening on eth0
0:4:76:4c:89:73 0:a0:cc:5b:f3:34 98: Wethertop > Blomberg: icmp: echo request (DF)
0:a0:cc:5b:f3:34 0:4:76:4c:89:73 98: Blomberg > Wethertop: icmp: echo reply
0:4:76:4c:89:73 0:a0:cc:5b:f3:34 98: Wethertop > Blomberg: icmp: echo request (DF)
0:a0:cc:5b:f3:34 0:4:76:4c:89:73 98: Blomberg > Wethertop: icmp: echo reply
0:4:76:4c:89:73 0:a0:cc:5b:f3:34 98: Wethertop > Blomberg: icmp: echo request (DF)
0:a0:cc:5b:f3:34 0:4:76:4c:89:73 98: Blomberg > Wethertop: icmp: echo reply
```

Die Optionen `-v`, `-vv` und `-vvv` sind lapidar gesagt dafür da, *mehr, noch mehr und noch viel mehr* Informationen anzufordern³⁵. Was dies genau bedeutet, ist von

³⁵Zu merken mit *verbose*, *very verbose* und *very very verbose*

tcpdump

Werkzeug zum Mithören und Analysieren von Netzwerkverkehr

`tcpdump [OPTIONEN] [FILTERAUSDRUCK]`

```
-i G   hört auf Gerät G anstelle von eth0
-n     zeigt IP- und Portnummern statt Namen
-N     zeigt keine Domainnamen
-q     lässt Protokollinformationen weg
-t     lässt Zeitstempel weg
-v     ausführliche Ausgabe
-vv    sehr ausführliche Ausgabe
-vvv   verdammt ausführliche Ausgabe
-e     zeigt Ethernetadressen an
-x     zeigt Hexdump an
-xX    zeigt Hexdump mit ASCII-Interpretation an
-s 0   zeigt komplette Pakete an
```

6 Netzwerkdiagnose

Protokoll zu Protokoll unterschiedlich und in der Man-Seite nachzulesen (`man 1 tcpdump`). Im Zweifelsfall hilft aber auch einfach Experimentieren.

Beispielsweise kannst du dir einmal ansehen, was `traceroute` so alles auf dem Netzwerk veranstaltet, wenn du in einem Fenster eingibst:

```
Eiger:~> traceroute -N 1 -n -F -q 1 213.95.15.200
traceroute to 213.95.15.200 (213.95.15.200), 30 hops max, 40 byte packets
 1  10.10.0.7  0.000 ms
 2  217.5.98.8  56.710 ms
 3  217.237.152.54  57.695 ms
 4  194.25.6.14  57.395 ms
 5  194.59.190.50  57.415 ms
 6  212.218.189.33  73.591 ms
 7  192.135.7.9  75.921 ms
 8  62.128.0.209  81.262 ms
 9  62.128.25.132  87.146 ms
10  *
11  62.128.13.194(N!)  82.969 ms
```

... und dies in einem anderen verfolgt mit:

```
Eiger:~ # tcpdump -ntv 'icmp or udp'
tcpdump: listening on eth0
10.10.0.9.64000 > 213.95.15.200.33434: [udp sum ok] udp 40 (DF) [ttl 1]
(id 7364, len 68)
```

`traceroute` beginnt damit, an den Rechner `213.95.15.200` ein UDP-Paket zu schicken. Der Trick ist dabei, dass der *Time To Live (TTL)*-Zähler auf 1 gesetzt ist. Dieser Zähler wird von `tcpdump` ausgegeben: `[ttl 1]`.

Der erste Router, den das Paket erreicht, ist `10.10.0.7`. Dieser erkennt, dass das UDP-Paket abgelaufen ist, weil TTL bei ihm auf 0 fällt. Er sendet daher eine ICMP-Antwortmeldung vom Typ `time exceeded in-transit`:

```
10.10.0.7 > 10.10.0.9: icmp: time exceeded in-transit [tos 0xc0]
(ttl 255, id 54605, len 96)
```

Jetzt kommt das zweite UDP-Paket, diesmal ist TTL auf 2 gesetzt:

```
10.10.0.9.64001 > 213.95.15.200.33435: [udp sum ok] udp 40 (DF)
(ttl 2, id 7365, len 68)
```

Die Fehlermeldung kommt diesmal vom übernächsten Router, hier im Beispiel `217.5.98.8`:

```
217.5.98.8 > 10.10.0.9: icmp: time exceeded in-transit (ttl 126, id 0,
len 56)
```

Das Spiel geht solange weiter, bis der Zielrechner erreicht wird:

```
10.10.0.9.64002 > 213.95.15.200.33436: [udp sum ok] udp 40 (DF) (ttl 3, id 7366, len 68)
217.237.152.54 > 10.10.0.9: icmp: time exceeded in-transit (ttl 253, id 0, len 56)
10.10.0.9.64003 > 213.95.15.200.33437: [udp sum ok] udp 40 (DF) (ttl 4, id 7367, len 68)
194.25.6.14 > 10.10.0.9: icmp: time exceeded in-transit (ttl 251, id 0, len 56)
10.10.0.9.64004 > 213.95.15.200.33438: [udp sum ok] udp 40 (DF) (ttl 5, id 7368, len 68)
194.59.190.50 > 10.10.0.9: icmp: time exceeded in-transit [tos 0xc0] (ttl 251, id 39247, len 56)
10.10.0.9.64005 > 213.95.15.200.33439: [udp sum ok] udp 40 (DF) (ttl 6, id 7369, len 68)
212.218.189.33 > 10.10.0.9: icmp: time exceeded in-transit (ttl 250, id 0, len 56)
10.10.0.9.64006 > 213.95.15.200.33440: [udp sum ok] udp 40 (DF) (ttl 7, id 7370, len 68)
192.135.7.9 > 10.10.0.9: icmp: time exceeded in-transit [tos 0xc0] (ttl 249, id 63069, len 56)
10.10.0.9.64007 > 213.95.15.200.33441: [udp sum ok] udp 40 (DF) (ttl 8, id 7371, len 68)
62.128.0.209 > 10.10.0.9: icmp: time exceeded in-transit [tos 0xc0] (ttl 248, id 17315, len 56)
10.10.0.9.64008 > 213.95.15.200.33442: [udp sum ok] udp 40 (DF) (ttl 9, id 7372, len 68)
62.128.25.132 > 10.10.0.9: icmp: time exceeded in-transit (ttl 55, id 21034, len 56)
10.10.0.9.64009 > 213.95.15.200.33443: [udp sum ok] udp 40 (DF) (ttl 10, id 7373, len 68)
10.10.0.9.64010 > 213.95.15.200.33444: [udp sum ok] udp 40 (DF) (ttl 11, id 7376, len 68)
62.128.13.194 > 10.10.0.9: icmp: net 213.95.15.200 unreachable - admin prohibited [tos 0xc4] (ttl 245, id 5013, ...)
```

Wenn du über den eigentlichen Inhalt der Datenpakete Bescheid wissen willst und ein Protokoll untersuchst, das `tcpdump` nicht analysieren kann, kannst du auch das komplette Paket als Hexdump ausgeben lassen! Dazu dient die Option `-x`.

Ich empfehle dazu, gleich noch ein `-X` hinterherzuschieben, denn dann bekommst du gleich noch eine ASCII-Interpretation dazugeliefert. Die volle Dröhnung kannst du dir dann mit `tcpdump -evvvvX -s 0` geben, wobei `-s 0` verhindert, dass die Pakete bei 68 Bytes abgeschnitten werden. Das macht `tcpdump` nämlich normalerweise, um Zeit zu sparen.

In folgendem Beispiel kannst du einmal die ganze Wahrheit über ein Paket aus dem NTP³⁶-Protokoll sehen:

```
Eiger:~ # tcpdump -evvvvX -s 0
tcpdump: listening on eth0
11:39:33.485971 0:a0:cc:5b:f3:34 0:4:76:4c:89:73 ip 90: Blomberg.zemeplocha.de.ntp
> Wethertop.zemeplocha.de.ntp: [udp sum ok] v4 server strat 2 poll 6 prec -17 dist
0.087554 disp 0.020935 ref ntp2.ptb.de@3247900450.446489006 orig 3247900773.4854219
85 rec +0.003443000 xmt +0.003617000 (ttl 64, id 61535, len 76)
0x0000 4500 004c f05f 0000 4011 761e 0a0a 0007 E..L._...@.v....
0x0010 0a0a 0009 007b 007b 0038 c9ae 2402 06ef .....{.{.8..$....
0x0020 0000 166a 0000 055c c035 6768 c197 0722 ...j...\.5gh..."
0x0030 724d 1a65 c197 0865 7c44 9dbe c197 0865 rM.e....e|D.....e
0x0040 7d26 41b3 c197 0865 7d31 a8ef }&A....e}1..
```

Paketfilter

Normalerweise zeigt `tcpdump` alle Pakete an, die an der Netzwerkkarte vorbeirauschen. In der Regel interessiert man sich aber nur für ein bestimmtes Protokoll, eine bestimmte Verbindung oder einen bestimmten Rechner. Du kannst `tcpdump` die Arbeit überlassen, die interessanten Pakete für dich herauszufiltern. Dazu gibst du nach den Optionen einen *Filterausdruck* an, wie ich es in einigen der Beispiele schon gemacht habe, ohne darauf hinzuweisen.

³⁶NTP: Network Time Protocol, definiert in RFC 958, <http://www.ietf.org/rfc/rfc958.txt>

Im einfachsten Fall besteht der Filterausdruck nur aus einem einzelnen Wort – wie z. B. `udp`. Es werden dann nur Pakete angezeigt, die der Filter akzeptiert, z. B. zeigt `tcpdump udp` nur UDP-Pakete.

Die nützlichsten Filter habe ich dir im nebenstehenden Kasten zusammengestellt. Manche der Filterausdrücke erfordern nach dem Schlüsselwort eine weitere Angabe.

Beim Filter `net` verwendet man für die Angabe des Netzwerkes die Schreibweise mit dem Schrägstrich, wie ich sie in Abschnitt 6.3.2 auf Seite 175 erklärt habe.

Der Filter `port` akzeptiert sowohl eine numerische Portnummer als auch einen Dienstenamen aus `/etc/services`.

Hier sind einige Beispiele für Filterausdrücke:

Zeige den Netzwerkverkehr von und zu `www.muenchen.de`:

```
Eiger:~ # tcpdump host www.muenchen.de
```

Zeige nur DNS-Abfragen und -Antworten:

```
Eiger:~ # tcpdump port 53
```

oder:

```
Eiger:~ # tcpdump port domain
```

Zeige allen Verkehr von und zum `10.13er` Netz:

```
Eiger:~ # tcpdump net 10.13.0.0/16
```

Vor die Ausdrücke `host` und `port` kannst du noch das Schlüsselwort `src` (Quelle, engl. *source*) oder `dst` (Ziel, engl. *destination*) stellen, womit du die Angabe des Rechners bzw. Ports auf die Quelle oder das Ziel festlegst.

tcpdump – Filterausdrücke

<code>udp</code>	zeigt UDP-Pakete
<code>tcp</code>	zeigt TCP-Pakete
<code>icmp</code>	zeigt ICMP-Pakete
<code>arp</code>	zeigt ARP-Pakete
<code>broadcast</code>	zeigt Broadcastverkehr
<code>host H</code>	zeigt Pakete von oder zu Rechner H
<code>src host</code>	bezieht <code>host</code> nur auf die Quelle
<code>dst host</code>	bezieht <code>host</code> auf das Ziel
<code>net N/B</code>	zeigt Verkehr innerhalb, von oder zu Netzwerkadresse/Bitanzahl
<code>port P</code>	zeigt TCP/UDP-Pakete von oder zu Port oder Dienst P
<code>src port</code>	bezieht <code>port</code> nur auf die Quelle
<code>dst port</code>	bezieht <code>port</code> auf das Ziel

Verknüpfungen

<code>A and B</code>	A und B müssen zutreffen
<code>A or B</code>	A oder B muss zutreffen
<code>not A</code>	negiert den Filterausdruck A
<code>(A)</code>	klammert A als Teilausdruck

Zeige nur Pakete von 10.10.0.7:

```
Eiger:~ # tcpdump src host 10.10.0.7
```

Zeige nur Pakete an Port 80:

```
Eiger:~ # tcpdump dst port 80
```

Wenn dir soviel Auswahl noch nicht genügt, kannst du die Ausdrücke mit `and`, `or`, `not` und Klammern () beliebig verschachteln. Bei der Verwendung von Klammern musst du den ganzen Ausdruck vor der Shell schützen, indem du ihn in Hochkommata setzt.

Zeige alle HTTPS- und HTTP-Pakete:

```
Eiger:~ # tcpdump port https or port http
```

Bei `or` in Verbindung mit `port`, `host` und `net` kannst du abkürzen:

```
Eiger:~ # tcpdump port https or http
```

Zeige HTTP-Verbindung mit Rechner 10.17.18.200:

```
Eiger:~ # tcpdump port http and host 10.17.18.200
```

Zeige UDP-Verkehr der Rechner 10.10.0.7 und 10.10.0.8:

```
Eiger:~ # tcpdump 'udp and host (10.10.0.7 or 10.10.0.8)'
```

Blende Verkehr mit Rechner 10.10.0.7 aus:

```
Eiger:~ # tcpdump not host 10.10.0.7
```

Auf weitere Möglichkeiten, wie z. B. das Abfragen einzelner Bits von Protokollköpfen, verzichte ich hier. Wie immer findest du die Informationen in der sehr ausführlichen Man-Seite von `tcpdump`. Dort findest du auch eine Erklärung der Ausgabeformate verschiedener Protokolle.

6.5 Namensauflösung mittels DNS

HTTP	FTP	SMTP	POP3	SSH	telnet	...	DHCP	DNS	...	ping	tracert	...
TCP							UDP			ICMP		
IP												
ARP						PPP						
Ethernet				ISDN			Modem			DSL		

Ein wichtiger Dienst im Internet ist der *Domain Name Service (DNS)*. Seine primäre Aufgabe ist es, zu einem

8.5 printf-Debugging im Kernel

8.5.1 Vorbereitungen

Das `printf`-Debuggen im Linux-Kernel ist nicht viel schwieriger als in einem normalen Programm. Allerdings funktioniert das Kompilieren und vor allem das Ausführen anders. Und es ist spannender!

Bei der Fehlersuche im Kernel gehst du prinzipiell vor wie bei den C-Programmen. Zuerst kompilierst du einen eigenen Kernel, verwendest ihn oder zumindest das fragliche Modul und reproduzierst den Fehler. Erst wenn sich dein Kernel genauso verhält wie das Original, kannst du mit der eigentlichen Fehlersuche beginnen.

Zunächst musst Du den Quellcode installieren, und zwar genau den richtigen, also genau die Kernelversion, mit der dein System auch läuft. Wenn du den normalen Kernel von deiner Distribution verwendest, musst du auch dort das entsprechende RPM-Paket mit den Kernelquellen suchen und installieren.

Es ist möglich, dass deine Distribution zwei Versionen der Kernelquellen enthält: die von der Distribution angepassten und die originalen ohne Änderungen. Wie in Abschnitt 8.4.3 auf Seite 308 bereits erwähnt, werden die Quellen nicht als SRPM, sondern als normales RPM-Paket installiert. Du findest die Quellen nach der Installation in `/usr/src/linux`.

Als nächstes benötigst du eine *Konfiguration* des Kernels. Diese von Hand zu erstellen erfordert Ausdauer und vor allem umfangreiches Fachwissen. Außerdem benötigst du eine Konfiguration, die exakt derjenigen deines gerade laufenden Kernels entspricht.

Deshalb verwendest du nicht `make menuconfig` oder `make xconfig`, auch wenn du das irgendwo gelesen hast. Stattdessen holst du dir die Konfiguration des gerade laufenden Kernels aus dem `proc`-Dateisystem! Die steckt dort nämlich in der Datei `/proc/config.gz`. Speziell dafür gibt es den Aufruf `make cloneconfig`²²:

```
Eiger:/usr/src/linux # make cloneconfig
```

Wenn du alles richtig gemacht hast, rauschen nun einige tausend Zeilen an dir vorbei, die du getrost alle ignorieren kannst.

²²Das Vorhandensein von `/proc/config.gz` und die Funktion „`make cloneconfig`“ ist leider SUSE-spezifisch und nicht auf allen Distributionen zu finden. Im Zweifelsfall musst du die Dokumentation deiner Distribution durchstöbern, um die Kernelkonfiguration deines aktuellen Kernels zu finden. Diese Datei kopierst du nach `/usr/src/linux/.config` und startest „`make oldconfig`“.

8.5.2 Den Kernel übersetzen

Der Kernel ist jetzt bereit, übersetzt zu werden. Der Befehl dazu lautet²³:

```
Eiger:/usr/src/linux # make clean && make bzImage && make modules
```

Und das kann dauern! Früher waren die Rechner langsamer und einige Stunden Wartezeit musste man schon einplanen. Heute sind die Rechner schneller, der Kernel dafür aber viel größer. Auch heute ist eine Teepause an dieser Stelle sicher keine schlechte Idee.

Mit dem neuen Kernel 2.6 wurde das Build-System gründlich aufgeräumt. Deutlich sichtbar sind die neuen knappen und übersichtlichen Ausgaben zum Kompilervorgang. Es wird nur noch die Datei angezeigt, welche gerade übersetzt wird, und die Art der Operation²⁴:

```
AS      arch/i386/kernel/efi_stub.o
CC      arch/i386/kernel/early_printk.o
SYSCALL arch/i386/kernel/vsyscall-syms.o
LD      arch/i386/kernel/built-in.o
AS      arch/i386/kernel/head.o
CC      arch/i386/kernel/init_task.o
CPP     arch/i386/kernel/vmlinux.lds.s
```

Durch den zusätzlichen Befehl `make modules` werden die Kernelmodule übersetzt. Mit großer Wahrscheinlichkeit wirst du deinen Fehler nämlich in einem Kernelmodul suchen²⁵.

8.5.3 Den Kernel installieren

Ob du den selbstkompilierten Kernel installieren musst, hängt davon ab, ob dein Fehler in einem *Kernelmodul* steckt, oder im *Rumpfkern*. In ersterem Fall ist keine Neuinstallation des Kernel nötig. Wenn das Modul nicht zwingend zum Betrieb notwendig ist und sich mit `rmmod` entladen lässt, musst du noch nicht einmal neu booten und musst auch die Module nicht installieren²⁶.

Unter `/usr/src/linux/arch/i386/boot` findest du den fertig kompilierten Kernel. Der Dateiname ist `bzImage`. Um diesen zu installieren, musst du ihn

²³Bei Kernel Version 2.4 musste man vorher noch ein `make dep` aufrufen. Das ist mit Kernel 2.6 überflüssig geworden.

²⁴Für Neugierige: AS = Assembler, CC = C-Compiler, SYSCALL = Systemaufruf, LD = Linker, CPP = C-Präprozessor

²⁵Zu Kernelmodulen siehe auch Abschnitt 8.5.2 auf Seite 333 und Abschnitt 4.6.2 auf Seite 106.

²⁶Manche Module werden von anderen Modulen benötigt. Möglicherweise musst du also vorher erst eine Reihe anderer Module entladen. Auch musst du dafür sorgen, dass die durch das Modul bereitgestellte Funktionalität nicht benötigt wird, also z. B. das Netzwerk herunterfahren, eine Partition unmounten oder ähnliche Dinge tun.

ins Verzeichnis `/boot` kopieren und zwar so, dass dein Bootloader ihn auch verwendet. Der Standarddateiname heißt `/boot/vmlinuz`. Wenn du als Bootloader `lilo` verwendest, musst du noch `lilo` aufrufen, bei GRUB ist dies nicht nötig. Danach kannst du neu booten und hoffen, dass dein neuer Kernel auch wirklich verwendet wird und außerdem auch noch funktioniert²⁷.

Wenn du nur Änderungen an einem Modul vornehmen willst, ist die Sache einfacher. In diesem Fall kannst du das alte Modul mit `rmmmod` entladen und direkt das kompilierte Modul aus dem Verzeichnis mit dem Quellcode nehmen und mit `insmod` laden. Manchmal lässt sich das Modul nicht entladen:

```
Eiger:~ # rmmmod reiserfs
reiserfs: Device or resource busy
```

In einem solchen Fall kopierst du das selbst kompilierte Kernelmodul an die richtige Stelle in `/lib/modules` und bootest danach neu.

Wenn sich das Modul nicht laden lässt, kann es daran liegen, dass es bereits im Kernel fest einkompiliert ist. In diesem Falle musst du leider den Kernel neu installieren und neu booten.

8.5.4 Meldungen ausgeben mit `printk`

Auch und gerade im Linux-Kernel ist `printf`-Debugging die wichtigste Methode zur Fehlersuche. Der Kernel hat eine spezielle Funktion für diesen Zweck: `printk`. Dies ist sehr nützlich, da es im Kernel keine Standardausgabe wie bei Prozessen gibt. `printk` arbeitet genau wie `fprintf`, nur dass man den ersten Parameter (`stderr`) weglässt. Hier ist ein Ausschnitt aus dem Modul `reiserfs`:

```
printk ("reiserfs: Unrecognized mount option %s\n", this_char);
```

Die Ausgaben von `printk` wandern zum `syslogd` mit der Klasse `kern`. Als Priorität wird standardmäßig 4 (`warning`) verwendet²⁸. Wenn du eine andere Priorität willst, setzt du eine Ziffer in spitze Klammern direkt an den Anfang des Textes:

```
printk ("<3>Diese Meldung kommt mit Prio 3 (err)\n");
```

²⁷Geschickt ist, wenn man immer noch einen Kernel bootfähig parat hat, von dem man sicher weiß, dass er funktioniert. Du kannst den neuen Kernel z. B. nach `/boot/vmlinuz-test` kopieren. Dazu benötigst du einen zusätzlichen Eintrag in der Bootkonfiguration. Bei LILO trägst du dies in der Datei `/etc/lilo.conf` ein und rufst danach den Befehl `lilo` auf. Bei GRUB erfolgt der Eintrag in `/etc/grub/menu.lst` und wird automatisch aktiv.

²⁸Zu den Prioritäten von `syslogd` siehe Abschnitt 3.1.3 auf Seite 30

Nachdem du Änderungen gemacht hast, musst du den Kernel bzw. das Modul neu übersetzen. Das Übersetzen des Kernels geschieht wieder durch ein `make bzImage`. Bei einer Änderung an einem Modul benötigst du `make modules`. Dabei musst du im Verzeichnis `/usr/src/linux` sein.

Danach musst du entweder den Kernel bzw. das Modul installieren und neu booten, oder besser mit `rmmmod` das alte Module entladen und mit `insmod` das neue laden, wenn das geht. Wenn du alles richtig gemacht hast, erscheinen jetzt im Systemlog deine Meldungen von `printk`!

8.5.5 Ein komplettes Beispiel

Um dir zu beweisen, wie einfach es ist, eine Änderung am Kernel zu machen, zeige ich es dir an einem einfachen Beispiel. Ich verwende dazu das Modul `vfat`, welches das Dateisystem VFAT implementiert. VFAT ist eine Erweiterung des ursprünglichen MS-DOS-Dateisystems, mit dem es möglich ist, längere Dateinamen als im Format 8.3²⁹ zu verwenden. Ich werde das Modul so erweitern, dass beim Neuanlegen eines Verzeichnisses eine Kernelmeldung ausgegeben wird.

Als erstes installiere ich die Kernelquellen von meinem SUSE LINUX 9.1, das ich unter `/suse` gemountet habe:

```
Eiger:~ # rpm -hUv /suse/suse/i586/kernel-source-2.6.4-54.5.i586.rpm
kernel-source #####
```

Jetzt erzeuge ich eine Konfiguration, die identisch mit der des laufenden Kernels ist:

```
Eiger:~ # cd /usr/src/linux
Eiger:/usr/src/linux # make cloneconfig
scripts/kconfig/conf -o arch/i386/Kconfig
```

Jetzt fliegen ein paar tausend ähnliche Zeilen vorüber, bis endlich folgende Meldung kommt:

```
*
* Build options
*
Configuration name (CFGNAME) [default] default
Release number (RELEASE) [54.5] 54.5
```

Bei einem Kernel 2.4 musst du nun mit `make dep` die Abhängigkeiten neu berechnen. Beim 2.6er Kernel ist das nicht mehr nötig, und du kannst gleich mit `make clean` weitermachen:

²⁹Beispielsweise ist `maximal8.htm` im Format 8.3; aber nicht `laenger.html`, da hier die Endung `html` hinter dem Punkt vier Zeichen lang ist.

8 An die Quellen

```
Eiger:/usr/src/linux # make clean
CLEAN arch/i386/boot/compressed
CLEAN arch/i386/boot
CLEAN arch/i386/kernel
CLEAN drivers/atm
CLEAN drivers/char
CLEAN drivers/ieee1394
CLEAN drivers/pci
CLEAN drivers/scsi/qla2xxx
CLEAN init
CLEAN kernel
CLEAN lib
CLEAN usr
CLEAN scripts/basic
CLEAN scripts/genksyms
CLEAN scripts/kconfig
CLEAN scripts
CLEAN .tmp_versions include/config
CLEAN include/asm-i386/asm_offsets.h vmlinux System.map include/linux/autoco
nf.h include/linux/version.h include/asm .tmp_kallsyms1.o .tmp_kallsyms1.S .tmp_
kallsyms2.o .tmp_kallsyms2.S .tmp_versions .tmp_vmlinux1 .tmp_vmlinux2
```

Die eigentliche Knochenarbeit für den Rechner kommt jetzt: das Kompilieren des Rumpfkernels und der Module. Ich gebe deshalb beide Befehle in einer Zeile, damit ich meine Teepause nicht nach einer Viertelstunde unterbrechen muss, um den zweiten Befehl einzugeben:

```
Eiger:/usr/src/linux # make bzImage && make modules
CHK include/linux/version.h
UPD include/linux/version.h
SYMLINK include/asm -> include/asm-i386
HOSTCC scripts/basic/fixdep
HOSTCC scripts/basic/split-include
HOSTCC scripts/basic/docproc
```

...usw. Zeit zum Teetrinken. Jetzt werden über 10 Millionen Zeilen Quellcode übersetzt! Das Ende des Kompilervorganges ist ganz unspektakulär und sieht so aus:

```
LD [M] drivers/char/ftape/compressor/zft-compressor.ko
CC lib/zlib_deflate/zlib_deflate.mod.o
LD [M] lib/zlib_deflate/zlib_deflate.ko
CC drivers/net/znet.mod.o
LD [M] drivers/net/znet.ko
```

Das `vfat`-Modul liegt im Verzeichnis `fs/vfat`. Ich probiere zunächst, ob ich das selbstkompilierte Modul laden kann und ob es funktioniert. Mein Testrechner verfügt über eine Partition des Typs `vfat`, die ich immer unter `/data` gemountet habe. Damit ich das Modul `vfat` entladen kann, muss ich diese Partition zunächst unmounten:

```
Eiger:/usr/src/linux # umount /data
Eiger:/usr/src/linux # rmmod vfat
```

Nun kann ich das selbstkompilierte Modul laden. Dazu gebe ich `insmod` direkt den Dateinamen des Moduls:

```
Eiger:/usr/src/linux # cd fs/vfat
Eiger:/usr/src/linux/fs/vfat # insmod vfat.o
```

Das Modul ist geladen. Eine Kontrolle mit `lsmod` beweist das:

```
Eiger:/usr/src/linux/fs/vfat # lsmod | grep fat
vfat                11392  0
fat                  43328  1 vfat
```

Nun müsste sich auch die Partition `/data` wieder mounten lassen:

```
Eiger:/usr/src/linux/fs/vfat # mount /data
Eiger:/usr/src/linux/fs/vfat # mkdir /data/test
Eiger:/usr/src/linux/fs/vfat # rmdir /data/test
```

Gut. Auch das Anlegen und Löschen von Verzeichnissen klappt noch. Mein selbst kompiliertes Modul scheint zu funktionieren. Jetzt kann ich versuchen, eine Änderung zu machen. Dazu öffne ich die Datei `namei.c` im Editor `jmacs` und suche nach `'mkdir'`. Ich finde die Funktion `vfat_mkdir` und füge nach den Variablendeklarationen eine `printk`-Zeile ein³⁰:

```
I A * [namei.c] Row 1133 Col 56 4:54 Ctrl-X H for help

int vfat_mkdir(struct inode *dir, struct dentry* dentry, int mode)
{
    struct super_block *sb = dir->i_sb;
    struct inode *inode = NULL;
    struct vfat_slot_info sinfo;
    struct buffer_head *bh = NULL;
    struct msdos_dir_entry *de;
    int res;

    printk("<3>Jemand legt ein Verzeichnis an!\n");

    lock_kernel();
    res = vfat_add_entry(dir, &dentry->d_name, 1, &sinfo, &bh, &de);
```

Das war's. Nachdem ich gespeichert habe, begeben mich wieder in das Verzeichnis `/usr/src/linux` und rufe `make modules` auf. Das geht bedeutend schneller als beim ersten Mal, weil ja nur das `vfat`-Modul neu übersetzt werden muss:

³⁰In der Programmiersprache C darf der erste Befehl erst nach den Variablendeklarationen kommen. In C++ kann man Befehle und Deklarationen beliebig abwechseln.

```
Eiger:/usr/src/linux/fs/vfat # cd ../../
Eiger:/usr/src/linux # make modules
CHK      include/linux/version.h
make[1]: »arch/i386/kernel/asm-offsets.s« ist bereits aktualisiert.
CC [M]   fs/vfat/namei.o
LD [M]   fs/vfat/vfat.o
```

Jetzt versuche ich, das alte Modul durch das neue zu ersetzen:

```
Eiger:/usr/src/linux # umount /data
Eiger:/usr/src/linux # rmmmod vfat
Eiger:/usr/src/linux # insmod fs/vfat/vfat.o
Eiger:/usr/src/linux # mount /data
```

Ich kann das neue Modul laden und auch /data wieder mounten. Das Spannende kommt jetzt: Ich lege ein Verzeichnis an:

```
Eiger:/usr/src/linux # mkdir /data/test
```

Und jetzt werfe ich mal einen Blick nach /var/log/messages:

```
Wethertop:/usr/src/linux # tail -n 1 /var/log/messages
Jun  8 15:52:54 Eiger kernel: Jemand legt ein Verzeichnis an!
```

Na, wenn das kein Erfolg ist! Aber schwer war es eigentlich nicht, oder? Wie du siehst, kochen auch die Kernelhacker nur mit Wasser.

8.6 Den Fehler beheben

8.6.1 Einen Patch erstellen

Wenn du tatsächlich einen Fehler im Quellcode gefunden hast und weißt, wie du ihn beheben kannst, erstellst du am besten einen Patch. Dies ist eine mit `diff` erstellte Textdatei, welche alle Änderungen an den Quelldateien beschreibt, die notwendig sind, um den Fehler zu beheben. Diesen kannst du dann dem Autor oder Distributor schicken, der ihn dann in die offizielle Version einpflegen kann.

Um einen Patch zu erstellen, gehst du am besten wie folgt vor: Zunächst probierst du die Änderung noch einmal aus und testest, ob der Fehler wirklich damit verschwunden ist. Zum Vergleich probierst du es noch einmal ohne Änderung. Der Fehler muss nun wieder auftreten. Ohne diese Gegenkontrolle kannst du nicht sicher sein, dass deine Änderung den Fehler wirklich behebt!

Nachdem du so lange nach dem Fehler gesucht und mit der Änderung experimentiert hast, kennst du sie sicher auswendig. Deshalb löscht du das Verzeichnis

□ %CPU

Die Spalte %CPU der Prozesstabelle gibt an, wie viel Prozent der *reellen* Zeit der jeweilige Prozess die CPU zur Verfügung hat und nutzt (*user* und *sys* zusammengenommen). Der Wert ist natürlich nur dann größer als 0, wenn der Prozess im Zustand *RUNNING* ist. Die Spalte summiert sich auf zu maximal 100% pro CPU im System.

Anhand dieser Angaben kannst du sehr schnell sehen, welche Prozesse gerade zu welchen Teilen Rechenlast erzeugen. Allerdings handelt es sich hier um eine Momentaufnahme und keine Durchschnittswerte.

Zusammengefasst ergeben sich folgende Schlussfolgerungen:

- Ist die Systemlast größer als die Zahl der Prozessoren, so müssen arbeitswillige Prozesse auf eine freie CPU warten. Das bremst natürlich die Performance.
- Zeit, die die CPU mit *iowait* verbringt, liegt brach. Bei einem Server kann es Sinn machen, zusätzliche Prozesse zu starten, um die Zeit auszunutzen.
- Geht ein nennenswerter Teil der Zeit in Hardwareinterrupts (*hi*), so kannst du versuchen, durch eine bessere Ansteuerung der Peripherie (DMA, anderer Bus) die CPU zu entlasten.

9.2 Speicherengpässe und Swappen

Je mehr Hauptspeicher du in einen Rechner hineinsteckst, desto schneller wird er – *oder?* Naja – auf jeden Fall wird er nicht langsamer, sofern du nicht mit schlechteren Speicherbausteinen nachrüstest.

In der Tat hat die Größe des Arbeitsspeichers einen großen Einfluß auf die Gesamtperformance eines Systems. Aber wie immer steckt auch hier der Teufel im Detail. Aber Auswirkung hat die Größe des Hauptspeichers nun *genau?*

- Ist nicht genug Hauptspeicher vorhanden, um alle Prozesse zu beherbergen, so müssen Teile von Prozessen in *Swapspace* ausgelagert werden, d.h. auf die Festplatte kopiert und von dort bei Bedarf wieder geladen werden. Ein Plattenzugriff dauert um Größenordnungen länger als ein Speicherzugriff.
- Hast du hingegen Hauptspeicher übrig, so kann Linux diesen automatisch dazu nutzen, Festplattenblöcke zu puffern (*cachen*). Die Zahl der Lesezugriffe wird dadurch verringert und das System beschleunigt.
- Einige Applikationen – allen voran Datenbanken – implementieren ihre eigenen Caching-Algorithmen und laufen deutlich schneller, wenn viel Speicher vorhanden ist.

Letztlich läuft sich also alles darauf hinaus, dass Speicher dazu verwendet wird, um Ein-/Ausgabeoperationen (kurz *IO*) zu vermeiden. Dies muss nicht auf die Festplatte alleine beschränkt sein, sondern kann auch Netzwerkverkehr betreffen. Daraus wiederum folgt, dass ein Speicherausbau nur dann hilft, wenn es überhaupt *IO* zu vermeiden gibt!

Ob das der Fall ist, solltest du vor einem Ausbau also unbedingt untersuchen. Dazu können dir die Überlegungen aus dem vorangegangenen Abschnitt dienen. Ein Beispiel: Wenn die CPU immer zu 100% am Anschlag arbeitet, kann es gar nicht sein, dass sie durch *IO* ausgebremst wird!

Wichtig ist es auch, dass du weißt, wie du die Ausgabe des Befehls `free` richtig interpretierst. Dazu ein Beispiel. Die Option `-m` sorgt hier für eine Ausgabe im Megabyte (1024×1024 Bytes):

```
Eiger:~> free -m
              total          used         free       shared    buffers     cached
Mem:           504            290          213           0           40          62
-/+ buffers/cache:      187          316
Swap:          1004            27          976
```

Obiger Rechner verfügt über 512 MB Hauptspeicher. `free` zeigt davon nur 504 an, der Rest verpufft irgendwo im Kernel, als Rundungsungenauigkeit oder ist aufgrund von Schwächen in der PC-Architektur nicht nutzbar. Abgesehen davon gilt die Regel `total = used + free`.

Die Angaben `buffers` und `cached` zeigen, wieviel des zu `used` gerechneten Speichers vom Kernel intern für Puffer bzw. zum Cachen von Festplattenblöcken verwendet wird. Genau genommen kann dieser Speicher als frei bezeichnet werden, weil er vom Kernel nur „ausgeliehen“ ist und bei Bedarf sofort wieder frei gemacht wird.

Diesem Umstand trägt die zweite Zeile Rechnung. Hier wird `used` und `free` anders interpretiert. Diesmal wird `buffers` und `cached` zu `free` hinzugerechnet. Somit weißt du, wieviel Speicher tatsächlich noch für Benutzerprozesse zur Verfügung steht, bevor gewappt werden muss.

Um das Caching des Kernels zu testen, starte ich in einem anderen Fenster folgenden Befehl:

```
Eiger:~> grep -r irgendwas /usr
```

Der rekursive `grep`-Befehl öffnet und liest der Reihe nach alle Dateien unter `/usr`. Die Festplattenblöcke, die der Kernel dabei lädt, wandern in den Puffercache, der aufgrund der Fülle an Hauptspeicher vom Kernel kräftig erweitert wird. Ich warte eine Weile und rufe dann wieder `free` auf:

```
Eiger:~> free -m
              total        used         free       shared    buffers     cached
Mem:           504          501           2           0          53        231
-/+ buffers/cache: 216          287
Swap:          1004           27          976
```

Wie du sehen kannst, hat Linux mittlerweile den kompletten freien Speicher (bis auf 2 MB) als Cache umfunktioniert, welcher jetzt auf 231 MB angewachsen ist. Wenn ich den `grep`-Befehl ein zweites Mal laufen lasse, kann er auf 231 MB gespeicherte Daten zugreifen.

Es versteht sich von selbst, dass eine Speichererweiterung erst dann zum Tragen kommt, wenn der `free`-Zähler der ersten Zeile gegen 0 sinkt. Denn solange noch Speicher brach liegt, kannst du das System durch zusätzlichen Speicher natürlich nicht beschleunigen.

Die dritte Zeile schließlich zeigt die Nutzung des *Swapspace*s. Ein paar MB sind immer reserviert für die interne Verwaltung. Sollte der Wert `used` aber deutlich ansteigen, so bedeutet das, dass der Speicher nicht mehr für alle Prozesse ausreicht. Jetzt gibt es zwei Möglichkeiten:

- ❑ Im besseren Fall werden nur *inaktive* Prozesse ausgelagert. Das ist ein typischer Fall bei einem Arbeitsplatzrechner, wo der Benutzer mehrere größere Programme offen hat und gelegentlich zwischen ihnen wechselt. Er wird feststellen, dass der Wechsel schleppend ist, aber die Programme normal laufen.
- ❑ Im schlimmeren Fall muss der Kernel Teile von *aktiven* Prozessen auslagern. Aber da diese ständig wieder benötigt werden, beginnt ein emsiges Hin- und Herswappen, welches die Systemleistung drastisch reduziert. Da natürlich auch kein Speicher für Festplattenpuffer mehr übrig ist, bedeutet dies Turnübungen für den Lesekopf der Festplatte und eine ständig wartende CPU.
Für einen Server bedeutet dies den Performance-GAU: Das System bricht im schlechtesten Augenblick zusammen – nämlich dann, wenn die meisten Anfragen kommen. Du Server sollte dann so umkonfiguriert werden, dass er weniger Anfragen *parallel* bearbeitet.

Manchmal beginnt ein eigentlich richtig dimensioniertes System trotzdem wie wild zu swappen. Dafür gibt es in der Regel zwei Gründe:

- ❑ Ein einzelner Prozess läuft Amok und frisst immer mehr Speicher. Dies kannst du leicht mit `top` feststellen³.
- ❑ Es werden laufend neue Prozesse erzeugt, die sich nicht mehr beenden. Dies zeigt z. B. ein Aufruf von `ps tree`.

³...sofern du noch Geduld hast, die Anmeldung und das Starten von `top` abzuwarten. Es kann sein, dass der Rechner nur sehr schleppend reagiert.

Index

Symbole

- .so 236
- /bin 408
- /bin/sash 100
- /boot 409
- /dev 409
- /dev/lp0 134
- /dev/psaux 137
- /dev/raw 362
- /etc 408
- /etc/group 89
- /etc/init.d 352
- /etc/init.d/boot 93
- /etc/init.d/rc ... 93
- /etc/inittab 93
- /etc/logsurfer.conf
37
- /etc/mtab 97
- /etc/passwd 89
- /etc/resolv.conf 374
- /etc/services 192
- /etc/sysconfig/sysctl
181
- /etc/sysconfig/syslog
30
- /etc/syslog.conf . 30
- /home 410
- /lib 408
- /lib/modules 106
- /media 411
- /mnt 409
- /opt 410
- /proc 86, 409
- /proc/acpi 111
- /proc/bus/usb 145
- /proc/cmdline 108
- /proc/cpuinfo 110
- /proc/ide 153
- /proc/interrupts 120
- /proc/modules 107
- /proc/net/arp 169
- /proc/net/dev 165
- /proc/scsi 160
- /root 410
- /sbin 408
- /sbin/init 93
- /srv 411
- /sys 146
- /tmp 409
- /usr 409, 411
- /usr/share/doc ... 21
- /usr/src/packages ..
306
- /var 409, 413
- /var/log/boot.msg ..
94
- /var/log/dmesg ... 94
- /var/log/messages ..
28
- [:space:] 403
- # 55
- %prep 307
- _exit 257
- A**
- AAAA 374
- abort 268
- ac 112
- ac_adapter 113
- Accelerated Graphics Port
124
- accept 250
- ACCEPT 222
- ACPI 111
- address resolution
protocol 169
- Adressbus 118
- Adressraum 118
- AGP 124, 127
- Akku 113
- alert 32
- alternate number 140
- APM 111
- apropos 17
- Arbeitsspeicher 356
- arcfour 380
- Armbanduhr 1
- arp 169
- ARP 169
- ARPA Network 21
- Assembler 240
- ATAPI 151
- Aufrufstapel 276
- Ausdruck, regulärer .. 52,
399
- Auslagerungsspeicher ...
106
- auth 31
- authpriv 31

Index

- autoconf 319
- automake 319
- autonegotiation 168
- autoprobe 135

- B**
- backtrace 278
- Bandbreite 187, 364
- Bandbreitenmessung 365
- bash -x 41, 352
- battery 112
- Baudrate 128
- Baumstruktur 78
- Benutzer 88
- Beobachtungsmodus . 50
- Bibliothek 235, 263
- Bibliotheken, dynamische
235
- Binärdatei 51, 66, 68
- Binary 235, 239
- bind 250
- bing 187
- Bit/sec 128
- blowfish 380
- Bogomips 110
- Bonding-Device 166
- Bootkonfiguration 96
- Bootloader 91, 92
- Bootmeldungen 94
- BOOTP 197
- Bootskript 93
- Bootstrap Protocol ... 197
- brace 386
- Bridge 123
- brk 257
- Broadcast 197
- Broadcastadresse 177
- Broadcastping 185
- bt 278
- Buffercache 361
- buffers 106
- Bug 273
- BuildRequires 326
- Bulk Data Transfer ... 141
- Busy wait 252
- button 112
- bzImage 333

- bzip2 304

- C**
- C / C++ 329
- Cache 360
- cachen 356
- call stack 276
- Capabilities 127
- Cardbus 124
- cast 380
- cat 48
- change root 104
- chdir 248
- Chiffretext 379
- chkconfig 41
- chmod 249
- chown 249
- chroot 85, 104
- Class-ID 125
- cloneconfig 332
- close 247
- CLOSE_WAIT 202
- CLOSING 202
- cmdline 87
- cmp 71, 326
- Compilersprache 302
- Config Space 125
- configure 320
- connect 250
- continue 37
- Control Transfer 140
- controlling TTY 80
- core 282
- Coredump 273, 282
- CPU states 354
- crit 32
- cron 31

- D**
- daemon 31
- Daemon 28, 352
- Data Encryption Standard
380
- data flow type 140
- datagram 196
- date 116
- Datei 46
- Dateisystem .. 72, 74, 362
- Dateisystemcheck ... 363
- Dateisystemtyp 74
- Datenbus 118
- Datenflusstyp 140
- Datenrate 364
- dd 247, 348
- ddd 274
- debug 32
- Debugausgaben. 27
- Debugger 273
- debugging symbols .. 274
- Default Gateway 180
- Default-Policy 222
- Defaultroute 180, 230
- des 380
- DES 380
- destination unreachable .
183
- device nodes 409
- device descriptor 140
- Device-ID 125, 128
- df 75
- DHCP 197
- dhcpdump 225
- die 298
- diff 65, 71, 338
- Digitalkamera 144
- direction 140
- Diskettenlaufwerk ... 148
- DMA 120, 156, 348
- dmesg 94
- DNS 213, 232, 372
- Dokumentation 14
- Domain Name Service ...
213
- dont fragment 207
- Doppelkreuz 55
- Downlink 188
- DROP 222
- Durchschnittslast 354
- Dynamic Host
Configuration
Protocol 197
- dynamische Bibliotheken
235

- E**
- EACCES 246
 - egrep 54
 - ehci_hcd 144
 - Eigendokumentation . 14
 - Ein-/Ausgaberichtung .. 118
 - Einblick 5
 - Emacs 70, 396
 - emerg 32
 - endpoint 140
 - Endpunkt 140
 - environ 88
 - Environment 88
 - err 32
 - error_log 299
 - ESTABLISHED 202
 - Etherboot 92
 - ethereal 225
 - exec 37
 - Executable 239
 - execve 256
 - exit 257
 - exitcode 242
 - ext2 74, 77
 - ext2 363
 - ext3 363
- F**
- fan 112
 - FAQ 23
 - fchmod 249
 - fchown 249
 - fdformat 149
 - fdisk 72
 - Fehler beheben 9
 - Fehler, wandernder 8
 - Fehlerrate 365
 - Festplatte 72
 - fgrep 54
 - FHS 407
 - file 51
 - Filesystem Hierarchy
 Standard 407
 - FIN 375
 - FIN_WAIT 202
 - find 56, 58, 60
- G**
- Finite State Machine . 201
 - Firewall 220
 - flow control 129
 - Flusssteuerung 129
 - fork 254
 - Fortschreiten 78
 - Forum 23
 - FORWARD 221
 - fprintf 329
 - free 105, 357
 - Fremddokumentation 14
 - fstat 249
 - ftp 31
 - full duplex 167
 - Funktion 277
- G**
- Gartenzaun 55
 - gdb 274, 284
 - Geisteskraft, reine 9
 - General Public License .. 303
 - Genmask 181
 - Gerätebeschreibung . 140
 - Gerätedatei 409
 - Geräteklasse 141
 - Gerätetreiber 147
 - getgid 250
 - getuid 250
 - ghex 70
 - GID 250
 - Gigabit-Ethernet 368
 - GNU Debugger . 274, 284
 - GPL 303
 - grep 52, 60
 - GRUB 92
 - Gruppe 88
 - gzip 376
- H**
- Halb-Duplex 167, 368
 - half duplex 167
 - halt 94
 - hardware flow control ... 129
 - hardware interrupt .. 355
 - Hardware-Adresse .. 169
 - Hardwarearchitektur 105
 - Hardwareuhr 116
 - Hashmark 55
 - Hauptspeicher .. 116, 356
 - hdparm 361
 - Heap 257
 - Hemiptera 273
 - Henne-Ei-Problem 92
 - hexadezimal 68
 - Hexadezimalzahl 405
 - hexdump 68
 - Hexeditor 69
 - hexl-mode 70
 - hid 144
 - high speed 143
 - Historie 385
 - history 385
 - Homeverzeichnis 410
 - Hop 178
 - host 216
 - Host 139
 - Host Bridge 125
 - Hostadapter 158
 - Hotplug-System 148
 - HOWTO 22
 - HTTP 193
 - Hub 139
 - HUP 267
 - hwclock 116
- I**
- IANA 175
 - ICANN 175
 - ICMP 182
 - id 82, 88
 - IDE 151
 - idle 355
 - ifconfig 166
 - ignore 38
 - illegal instruction 267
 - info 18, 32
 - Info-Seite 18
 - init 78
 - initdefault 93
 - Initial-Ramdisk 92
 - Initrd 92

Index

- Inode 76
INPUT 221
insmod 92, 106
Internet Protokoll ... 164,
174
Internetadresse 175
Interpretersprache ... 293
Interrupt 119, 132
Interrupt Data Transfer ..
141
iowait 355
IP 164, 174
IP-Adresse 175
IP-Masquerading 176
IP_FORWARD 181
ipchains 221
ipfwadm 221
iptables 221
iptraf 365
IPV6 165, 374
ISA Bridge 125
ISA PnP 123
ISA-Bus 123
ISDN-Karte 171
Isochronous Data
Transfer 141
isolinux 92
- J**
jfs 363
jmacs 396
joe 395
Journal 363
Jumper 122
- K**
Kanalgitter 30
kdbg 274
kern 31
Kernel 104, 332
Kernelmodule 106
Kernelparameter 108
keycode 136
khexedit 69
kill 269
KILL 268
killall 270
- Klammern, geschweifte ..
386
kompilieren 317
Kompression 376
Kompressionsrate ... 376
Komprimierung 376
Konfigurationsdateien 54
konsistent 363
Kopf 73
- L**
Last 354
LAST_ACK 203
Latency 127
Latenzzeit 371
Laufwerk 149
Laufzeit 365
lchown 249
ldd 238
LDP 16, 24
less 48
libc_start_main . 279
library 235, 263
LILO 92
Linux Documentation
Project 16, 24
Linux Problem Base .. 23
Linux Standard Base .. 41
Linux-Kernel 314
linux-gate.so.1 . 239
listen 250
LISTEN 202
listen mode 198
ll 64
lo 165
load average 354
local0 31
locate 58
LOG 222
log facility 30
Logdateien 6, 27
logger 34
Loghost 29
Logmeldungen 29
logsurfer 36
loop 77
Loopback-Device 165
- low level format 149
lpr 31
LSB 41, 407
lsmod 106
lsof 84
lspci 125
lstat 249
lsusb 145, 146
ltrace 263
- M**
MAC-Adresse 169
mail 31
Mailinglisten 23
make 317
Makefile 317
man 15
Man-Seite 15
mark 129
match 37
Maus 137
mcedit 398
mccrypt 379
mean deviation 373
Memory Mapped IO . 119
menuconfig 332
mii-diag . 167, 229, 368
minicom 133
MIPS 110
mkdir 248
mke2fs 77
mmap2 257
modprobe 106
Module 106
mount 74
mprotect 257
munmap 257
- N**
Namedpipe 39
Namensauflösung ... 213
nanosleep 254
NAT 176
nc 198
neededforbuild .. 326
netcat 198, 367
netstat 165, 201

-
- Network Address
 - Translation 176
 - network device 165
 - Network File System 371
 - Network Time Protocol .. 198
 - Netzmaske 177
 - Netzwerkadresse 176
 - Netzwerkgerät 165
 - Netzwerkperformance ... 364
 - news 31
 - Newsgruppen 23
 - NFS 371
 - ngrep 225
 - nice 355
 - nmap 204
 - not_match 37
 - notice 32
 - ntop 225
 - NTP 198, 211
 - ntpd 352
 - Nummernzeichen 55
- O**
- O_RDONLY 246
 - objdump 239
 - OHCI 139, 144
 - ohci_hcd 144
 - Oktalzahl 71
 - open 246
 - OpenOffice.org 285
 - OpenSSH 377
 - OUTPUT 221
- P**
- Paketfilter 220
 - Paketlaufzeit 372
 - paketvermittelnd 196
 - parallel 134
 - Parität 129
 - parity 129
 - Partition 72, 360
 - passwd 99
 - Patch 303, 338
 - pause 254
 - PCI 124
 - PCMCIA 124
 - performance 113
 - Performance 345
 - Peripheriebaustein .. 119
 - Perl 297
 - PHP 299
 - PID 79, 84
 - pidof 84
 - pinfo 19
 - ping 182, 372
 - PIO 156
 - PIO-Modus 348
 - pipe 38
 - Plattengeometrie 73
 - PLIP 171
 - PNPBIOS 111
 - Point to Point Protocol ... 171
 - poll 254
 - Polling 119
 - POP3 193
 - port unreachable 206
 - Port IO 119
 - port-sharing 135
 - Portnummer 191, 197
 - Portscanner 204
 - Positionierung 359
 - post mortem 273, 282
 - power 112
 - PPP 170
 - PRI 82
 - printf-Debugging ... 292, 328
 - printk 334
 - Priorität 32
 - processor 112
 - Protokoll 163
 - Protokolloverhead ... 209
 - Prozedur 277
 - Prozess-ID 79, 84, 297
 - Prozessbaum 78
 - Prozessliste 80
 - Prozessmonitor .. 81, 354
 - Prozessor 110, 112
 - Prozessorbus ... 118, 119
 - Prozessorlast 354
 - ps 80
 - PS/2 137
 - pstree 78
 - Puffercache 357
 - Punkt-zu-Punkt
 - Verbindung 170
 - Python 300
- Q**
- Quellcode 291
- R**
- Rücksprungadresse .. 277
 - Raute 55
 - raw 362
 - Raw-Device 362
 - rc2 380
 - read 247
 - readperf 359
 - real 347
 - reboot 95
 - recv 250
 - recvfrom 250
 - Redundanz 376
 - reiserfs 363
 - Reiserfs 78
 - REJECT 222
 - remount 97
 - rename 248
 - Request For Comments .. 21
 - Rescue System 102
 - Resolver 374
 - Rettungssystem 102
 - return 280
 - reverse name lookup 214
 - RFC 21
 - rijndael 380
 - rmdir 248
 - rmmod 107
 - Root Hub 139
 - Rootpartition 408
 - Rootpasswort 98
 - Rootverzeichnis 408
 - round trip time 373
 - route 181
 - Route, Gateway- 179

- Route, lokale 179
Routing 178, 181
Routingprobleme 229
Routingtabelle 179
RPM 20, 61, 306
RS-232 128
RSS 82
rtd 85
RTS/CTS 129
RTT 373
Runlevel 93
RUNNING 354, 356
- S**
SANE 261
sash 99
SASI 158
scanimage 261
Schicht 163
Schnabelkerf 273
Schnittstelle, parallele ...
 134
Schnittstelle, serielle . 128
Schutzblech 138
SCSI 158
SDB 22
search 374
seek 359
segmentation fault .. 268,
 271
segmentation violation ..
 271
Segmentfehler .. 268, 271
SEGV 268, 271
Sektor 73
select 252
send 250
sendto 250
seriell 128
Serverdienste 351
Serversocket 191
setegid 251
seteuid 251
setgid 250
setuid 250
Sharp 55
Shellskript 294
Shellskripte verfolgen 41
Shellvariable 296
Signal 265
signal handler 265
Signalbehandlung ... 270
Signalbehandlungsroutine
 265
Simple Mail Transfer
 Protocol 195
simplex 167
SLIP 171
Small Computer System
 Interface 158
SMTP 195
socket 250
software flow control 129
software interrupt ... 355
Source-RPM 303, 306, 324
space 129
Spec-Datei 306
Speicheradresse 118
Speicherverwaltung . 257
Spurensuche 6
SRPM 306
ssh 259, 378
stack frame 281
Stand-alone Shell 100
Startskript 40
stat 249
Steuerbus 118
Stoppbit 129
storage device 142
strace 241, 242, 258,
 326, 348
strings 66
Stromverbrauch 113
Stromversorgung 142
Suche in Textdateien .. 52
Suche nach Dateien 56, 58
Supportdatenbank 22
SUSE CD 63
SUSE DVD 63
swappen 358
Swapspace .. 74, 106, 356
sy 82, 354
SYN 375
SYN_RECV 202
SYN_SENT 202
sys 347
Sys::Syslog 298
Syslinux 92
syslog() 36
syslogd 28, 39
SYSLOGD_PARAMS ... 30
system call 241
System Logging Daemon
 28
Systemaufruf 241
Systemlast 356
Systemrettung 95
Systemzeit 116
- T**
T10 158
Tabulatortaste 383
tail 49
Taktrate 110
Tarball 303, 304
Tastatur 136
TCP 191
tcpdump 206, 373
TCPMSS 222
telnet 192
Temperatur 115
Terminalprogramm .. 133
Texinfo 18
Texte in Binärdateien . 66
Texteditor 393
tftp 197
thermal
 thermal 112
thermal_zone 115
time 346, 378
time exceeded 189
time to live 189
TIME_WAIT 202
top 81, 354
tr 87
traceroute ... 189, 210
Transmission Control
 Protocol 191
tripledes 380

- Trivial File Transfer
 Protocol 197
- TTL 189
- TTY 80
- twofish 380
- U**
- UDP 196
- Übertragungsgeschwindigkeit 142
- übersetzen 317
- UHCI 139, 144
- uhci_hcd 144
- UID 250
- UID Transition 80
- UltraDMA 153
- Umgebungsvariable .. 88
- umgekehrte Namensauflösung 214
- Umstände, wechselnde 8
- uname 105, 257
- unified diff 339
- Universal Serial Bus . 138
- Unix-Socket 201
- unlink 248
- updatedb 59
- Upstream 370
- uptime 105, 354
- us 81, 354
- USB 138
- USB Host Controller . 139
- usb-ehci 144
- usb-ohci 144
- USB-Stick 144
- usb-storage 143
- usb-uhci 144
- usb_storage 143
- usbcore 144
- usbdevfs 145
- usbmodules 148
- usbserial 144
- usbview 147
- usedforbuild 326
- user 31, 347
- User Datagram Protocol . 196
- using_dma 154
- uucp 31
- V**
- Vaterprozess 78
- Vendor-ID 125, 128
- Verbindung 201
- verbindungsorientiert ... 191
- Verschlüsselung 379
- verschusselt 98
- Verzeichniscache 361
- VESA Local Bus 124
- vfork 256
- vi 394
- VLB 124
- vmlinux 334
- Voll-Duplex 167, 368
- W**
- wait 347
- wait4 257
- waitpid 257
- warning 32
- Warten 251
- Wartezeit 365, 373
- Wartezustand 347
- watch 50
- wc 378
- well known ports 192
- wget 349
- which 64
- Wireless LAN 365
- Wirkungskette 6
- WLAN 365
- Wollmilchsau,
 eierlegende . 111, 138
- word count 378
- write 247
- Wurzelverzeichnis ... 408
- X**
- xargs 60
- xconfig 332
- xev 136
- xntp 352
- XON/XOFF 130
- xscanimate 261
- XT-Bus 123
- Z**
- Zeitmessung 346
- zgrep 54
- zile 397
- Zombie 55, 83, 257
- Zusatztasten 136
- Zylinder 73