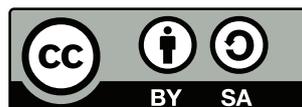
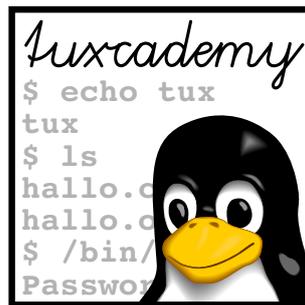




Version 4.0

Linux für Fortgeschrittene

Der Linux-Werkzeugkasten



tuxcademy – Linux- und Open-Source-Lernunterlagen für alle
www.tuxcademy.org · info@tuxcademy.org



Diese Schulungsunterlage ist inhaltlich und didaktisch auf Inhalte der Zertifizierungsprüfung LPI-102 (LPIC-1, Version 4.0) des Linux Professional Institute abgestimmt. Weitere Details stehen in Anhang C.

Das Linux Professional Institute empfiehlt keine speziellen Prüfungsvorbereitungsmaterialien oder -techniken – wenden Sie sich für Details an info@lpi.org.

Das tuxcademy-Projekt bietet hochwertige frei verfügbare Schulungsunterlagen zu Linux- und Open-Source-Themen – zum Selbststudium, für Schule, Hochschule, Weiterbildung und Beruf.
Besuchen Sie <https://www.tuxcademy.org/>! Für Fragen und Anregungen stehen wir Ihnen gerne zur Verfügung.

Linux für Fortgeschrittene Der Linux-Werkzeugkasten

Revision: grd2:73e2754304ddc5c7:2015-08-05

grd2:fbdd62c7f8693fb:2015-08-05 1–12, B–C

grd2:CPfT8uJKUsRd4Q2vmo5B7V

© 2015 Linup Front GmbH Darmstadt, Germany

© 2015 tuxcademy (Anselm Lingnau) Darmstadt, Germany

<http://www.tuxcademy.org> · info@tuxcademy.org

Linux-Pinguin »Tux« © Larry Ewing (CC-BY-Lizenz)

Alle in dieser Dokumentation enthaltenen Darstellungen und Informationen wurden nach bestem Wissen erstellt und mit Sorgfalt getestet. Trotzdem sind Fehler nicht völlig auszuschließen. Das tuxcademy-Projekt haftet nach den gesetzlichen Bestimmungen bei Schadensersatzansprüchen, die auf Vorsatz oder grober Fahrlässigkeit beruhen, und, außer bei Vorsatz, nur begrenzt auf den vorhersehbaren, typischerweise eintretenden Schaden. Die Haftung wegen schuldhafter Verletzung des Lebens, des Körpers oder der Gesundheit sowie die zwingende Haftung nach dem Produkthaftungsgesetz bleiben unberührt. Eine Haftung über das Vorgenannte hinaus ist ausgeschlossen.

Die Wiedergabe von Warenbezeichnungen, Gebrauchsnamen, Handelsnamen und Ähnlichem in dieser Dokumentation berechtigt auch ohne deren besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne des Warenzeichen- und Markenschutzrechts frei seien und daher beliebig verwendet werden dürften. Alle Warennamen werden ohne Gewährleistung der freien Verwendbarkeit benutzt und sind möglicherweise eingetragene Warenzeichen Dritter.



Diese Dokumentation steht unter der »Creative Commons-BY-SA 4.0 International«-Lizenz. Sie dürfen sie vervielfältigen, verbreiten und öffentlich zugänglich machen, solange die folgenden Bedingungen erfüllt sind:

Namensnennung Sie müssen darauf hinweisen, dass es sich bei dieser Dokumentation um ein Produkt des tuxcademy-Projekts handelt.

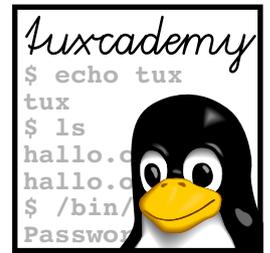
Weitergabe unter gleichen Bedingungen Sie dürfen die Dokumentation bearbeiten, abwandeln, erweitern, übersetzen oder in sonstiger Weise verändern oder darauf aufbauen, solange Sie Ihre Beiträge unter derselben Lizenz zur Verfügung stellen wie das Original.

Mehr Informationen und den rechtsverbindlichen Lizenzvertrag finden Sie unter <http://creativecommons.org/licenses/by-sa/4.0/>

Autoren: Tobias Elsner, Anselm Lingnau

Technische Redaktion: Anselm Lingnau (anselm@tuxcademy.org)

Gesetzt in Palatino, Optima und DejaVu Sans Mono

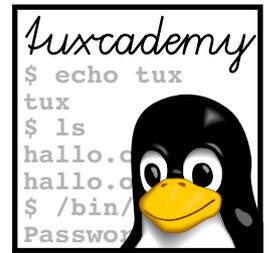


Inhalt

1	Allgemeines über Shells	13
1.1	Shells und Shellskripte	14
1.2	Shelltypen	14
1.3	Die Bourne-Again-Shell	17
1.3.1	Das Wichtigste.	17
1.3.2	Login-Shells und interaktive Shells	18
1.3.3	Dauerhafte Konfigurationsänderungen	20
1.3.4	Tastatur-Belegung und Abkürzungen	21
2	Shellskripte	23
2.1	Einleitung	24
2.2	Aufruf von Shellskripten	24
2.3	Aufbau von Shellskripten	26
2.4	Shellskripte planen	27
2.5	Fehlertypen	28
2.6	Fehlererkennung	29
3	Die Shell als Programmiersprache	33
3.1	Variable	34
3.2	Arithmetische Ausdrücke	40
3.3	Bearbeitung von Kommandos	41
3.4	Kontrollstrukturen	42
3.4.1	Überblick.	42
3.4.2	Der Rückgabewert von Programmen als Steuergröße	42
3.4.3	Alternativen, Bedingungen und Fallunterscheidungen	44
3.4.4	Schleifen	49
3.4.5	Schleifenunterbrechung	51
3.5	Shellfunktionen	54
3.6	Das Kommando exec.	55
4	Praktische Shellskripte	59
4.1	Shellprogrammierung in der Praxis	60
4.2	Rund um die Benutzerdatenbank	60
4.3	Dateioperationen	65
4.4	Protokolldateien	67
4.5	Systemadministration	73
5	Interaktive Shellskripte	77
5.1	Einleitung	78
5.2	Das Kommando read.	78
5.3	Menüauswahl mit select	80
5.4	»Grafische« Oberflächen mit dialog	84

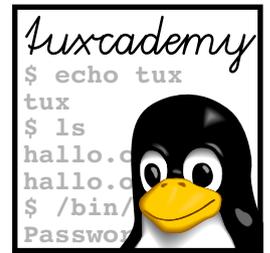
6 Der Stromeditor sed	93
6.1 Einsatzgebiete	94
6.2 Adressierung	94
6.3 sed-Anweisungen	96
6.3.1 Ausgeben und Löschen von Zeilen	96
6.3.2 Einfügen und Verändern	97
6.3.3 Zeichen-Transformationen	98
6.3.4 Suchen und Ersetzen	98
6.4 sed in der Praxis	99
7 Die awk-Programmiersprache	105
7.1 Was ist awk?	106
7.2 awk-Programme.	107
7.3 Ausdrücke und Variable	109
7.4 awk in der Praxis	113
8 SQL	123
8.1 Warum SQL?	124
8.1.1 Überblick.	124
8.1.2 SQL einsetzen	126
8.2 Tabellen definieren	127
8.3 Datenmanipulation und Abfragen.	129
8.4 Relationen	133
8.5 Praktische Beispiele	136
9 Zeitgesteuerte Vorgänge – at und cron	141
9.1 Allgemeines.	142
9.2 Einmalige Ausführung von Kommandos	142
9.2.1 at und batch	142
9.2.2 at-Hilfsprogramme	144
9.2.3 Zugangskontrolle.	145
9.3 Wiederholte Ausführung von Kommandos	145
9.3.1 Aufgabenlisten für Benutzer	145
9.3.2 Systemweite Aufgabenlisten	147
9.3.3 Zugangskontrolle.	148
9.3.4 Das Kommando crontab	148
9.3.5 Anacron	148
10 Lokalisierung und Internationalisierung	153
10.1 Überblick.	154
10.2 Zeichencodierungen.	154
10.3 Spracheneinstellung unter Linux	158
10.4 Lokalisierungs-Einstellungen	160
10.5 Zeitzonen	163
11 Die Grafikoberfläche X11	169
11.1 Grundlagen	170
11.2 X11 konfigurieren.	175
11.3 Displaymanager	182
11.3.1 Grundlegendes zum Starten von X	182
11.3.2 Der Displaymanager LightDM	183
11.3.3 Andere Displaymanager	185
11.4 Informationen anzeigen	186
11.5 Der Schriftenserver	188
11.6 Fernzugriff und Zugriffskontrolle	191

12 Linux für Behinderte	193
12.1 Einführung	194
12.2 Tastatur, Maus und Joystick	194
12.3 Die Bildschirmdarstellung	195
A Musterlösungen	197
B Reguläre Ausdrücke	213
B.1 Überblick.	213
B.2 Extras	214
C LPIC-1-Zertifizierung	217
C.1 Überblick.	217
C.2 Prüfung LPI-102	218
C.3 LPI-Prüfungsziele in dieser Schulungsunterlage.	218
D Kommando-Index	223
Index	225



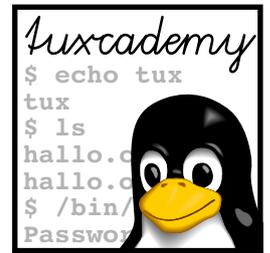
Tabellenverzeichnis

3.1	Reservierte Rückgabewerte bei der Bash	42
5.1	Interaktions-Elemente von dialog	85
6.1	Von sed unterstützte reguläre Ausdrücke und ihre Bedeutung (Auswahl)	96
10.1	Die wichtigsten Teile von ISO/IEC 8859	155
10.2	LC_*-Umgebungsvariable	161
10.3	Die Sommerzeit in Deutschland	164
B.1	Unterstützung von regulären Ausdrücken	215



Abbildungsverzeichnis

3.1	Ein einfaches Init-Skript	48
4.1	Welche Benutzer haben eine bestimmte primäre Gruppe? (Verbesserte Version)	62
4.2	In welchen Gruppen ist Benutzer <i>x</i> ?	64
4.3	Massen-Suffixänderung für Dateinamen	67
4.4	Mehrere Protokolldateien überwachen	70
4.5	df mit Balkengrafik für die Plattenauslastung	74
5.1	Ein Menü mit dialog	85
5.2	Eine Version von wmm mit dialog	89
8.1	Eine Datenbanktabelle: Berühmte Raumschiffkommandanten im Kino	124
8.2	Berühmte Raumschiffkommandanten im Kino (normalisiert)	125
8.3	Das komplette Schema für die Datenbank	128
8.4	Das Skript calendar-upcoming	139
11.1	X Window System als Client-Server-System	170



Vorwort

Diese Lernunterlage richtet sich an fortgeschrittene Linux-Benutzer und gibt ihnen das Rüstzeug zur noch produktiveren Anwendung des Systems. Aufbauend auf der tuxcademy-Lernunterlage *Einführung in Linux* oder vergleichbaren Kenntnissen führt sie in die fortgeschrittene Verwendung der Bourne-Again-Shell und die Shellprogrammierung ein, die mit zahlreichen praktischen Beispielen illustriert wird. Weitere Themen sind der Stromeditor `sed`, die Programmiersprache `awk` und die Grundlagen von SQL zum Zugriff auf relationale Datenbanken.

Den Kursumfang komplettieren eine Diskussion von `at` und `cron` zur zeitgesteuerten Ausführung von Programmen, eine Einführung in die Internationalisierung und Lokalisierung von Linux-Systemen sowie die Verwendung und Administration der grafischen Oberfläche X11 und schließlich ein Überblick über die Hilfsmittel, die Linux Behinderten zur Verfügung stellt.

Dieser Kurs setzt voraus, dass Teilnehmer den Umgang mit der Linux-Kommandozeile sowie die wichtigsten Kommandos beherrschen (der Umfang orientiert sich lose an den Anforderungen der LPI-Prüfung LPI-101). Auch der Umgang mit einem Texteditor muss geläufig sein. Kenntnisse der Systemadministration sind nicht zwingend erforderlich, aber schaden auch nicht; einige Inhalte in dieser Unterlage sind allerdings nur für Systemadministratoren interessant.

Der erfolgreiche Abschluss dieser Lernunterlage oder vergleichbare Kenntnisse sind Voraussetzung für das erfolgreiche Studium der tuxcademy-Lernunterlage *Linux-Administration II* sowie darauf aufbauender weiterführender Linux-Themen und für eine Zertifizierung beim *Linux Professional Institute*.

Diese Schulungsunterlage soll den Kurs möglichst effektiv unterstützen, indem das Kursmaterial in geschlossener, ausführlicher Form zum Mitlesen, Nach- oder Vorarbeiten präsentiert wird. Das Material ist in Kapitel eingeteilt, die jeweils für sich genommen einen Teilaspekt umfassend beschreiben; am Anfang jedes Kapitels sind dessen Lernziele und Voraussetzungen kurz zusammengefasst, am Ende finden sich eine Zusammenfassung und (wo sinnvoll) Angaben zu weiterführender Literatur oder WWW-Seiten mit mehr Informationen.

Kapitel

Lernziele

Voraussetzungen



Zusätzliches Material oder weitere Hintergrundinformationen sind durch das »Glühbirnen«-Sinnbild am Absatzanfang gekennzeichnet. Zuweilen benutzen diese Absätze Aspekte, die eigentlich erst später in der Schulungsunterlage erklärt werden, und bringen das eigentlich gerade Vorgestellte so in einen breiteren Kontext; solche »Glühbirnen«-Absätze sind möglicherweise erst beim zweiten Durcharbeiten der Schulungsunterlage auf dem Wege der Kursnachbereitung voll verständlich.

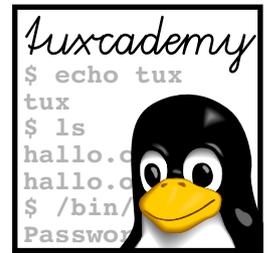


Absätze mit dem »Warnschild« weisen auf mögliche Probleme oder »gefährliche Stellen« hin, bei denen besondere Vorsicht angebracht ist. Achten Sie auf die scharfen Kurven!



Die meisten Kapitel enthalten auch Übungsaufgaben, die mit dem »Bleistift«-Sinnbild am Absatzanfang gekennzeichnet sind. Die Aufgaben sind nummeriert und Musterlösungen für die wichtigsten befinden sich hinten in dieser Schulungsunterlage. Bei jeder Aufgabe ist in eckigen Klammern der Schwierigkeitsgrad angegeben. Aufgaben, die mit einem Ausrufungszeichen (»!«) gekennzeichnet sind, sind besonders empfehlenswert.

Übungsaufgaben



1

Allgemeines über Shells

Inhalt

1.1	Shells und Shellskripte	14
1.2	Shelltypen	14
1.3	Die Bourne-Again-Shell	17
1.3.1	Das Wichtigste.	17
1.3.2	Login-Shells und interaktive Shells	18
1.3.3	Dauerhafte Konfigurationsänderungen	20
1.3.4	Tastatur-Belegung und Abkürzungen	21

Lernziele

- Grundlagen über Shells und Shellskripte lernen
- Login-, interaktive und nichtinteraktive Shellprozesse unterscheiden können
- Die Methoden zur Konfiguration der Bash beherrschen

Vorkenntnisse

- Vertrautheit mit der Kommandooberfläche von Linux
- Umgang mit Dateien und einem Texteditor

1.1 Shells und Shellskripte

Shell Die **Shell** erlaubt Ihnen den direkten Umgang mit dem Linux-System: Sie können Kommandos formulieren, die die Shell auswertet und ausführt – meist durch Starten von externen Programmen. Man spricht von der Shell auch als »Kommandointerpreter«.

Die meisten Shells haben ferner Eigenschaften von Programmiersprachen – Variable, Kontrollstrukturen wie Verzweigungen und Schleifen, Funktionen und mehr. Das macht es möglich, auch komplexe Folgen von Shellkommandos und externen Programmaufrufen in Textdateien abzulegen und als **Shellskripte** interpretieren zu lassen. Damit lassen sich schwierige Operationen wiederholbar und dauernd auftretende Vorgänge mühelos gestalten.

Das Linux-System verwendet Shellskripte für viele interne Aufgaben. Zum Beispiel werden die »Init-Skripte« in `/etc/init.d` in aller Regel als Shellskripte realisiert. Das gilt auch für viele Systemkommandos. Linux erlaubt es, den Umstand, dass ein »Systemprogramm« gar kein direkt ausführbares Binärprogramm ist, sondern ein Shellskript, vor seinen Benutzern zu verstecken, jedenfalls soweit es um die Aufrufsyntax geht – wenn Sie es genau wissen wollen, können Sie natürlich herausfinden, was Sache ist, zum Beispiel mit dem Kommando `file`:

```
$ file /bin/ls
/bin/ls: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),>
< for GNU/Linux 2.2.0, dynamically linked (uses shared libs),>
< stripped
$ file /bin/egrep
/bin/egrep: Bourne shell script text executable
```

Übungen



1.1 [!2] Welche Vorteile könnte es haben, ein Kommando als Shellskript und nicht als Binärprogramm zu realisieren? Welche Nachteile? Unter welchen Umständen würden Sie sich für ein Shellskript anstatt eines Binärprogramms entscheiden?



1.2 [3] Wie viele Kommandos in den Verzeichnissen `/bin` und `/usr/bin` Ihres Linux-Systems sind als Shellskripte realisiert? Geben Sie eine Kommando-Pipeline an, mit der Sie diese Frage beantworten können (Tipp: Verwenden Sie die Kommandos `find`, `file`, `grep` und `wc`.)

1.2 Shelltypen

bash Die Standardshell unter Linux ist die *Bourne-Again-Shell* (`bash`) des GNU-Projekts. Sie ist weitestgehend kompatibel zur ersten brauchbaren Shell der Unix-Geschichte (der Bourne-Shell) und zum POSIX-Standard. Traditionell sind auch andere Shells üblich, etwa die C-Shell (`csh`) von BSD und deren Weiterentwicklung, die Tenex-C-Shell (`tcsh`), die Korn-Shell (`ksh`) und einige mehr. Diese Shells unterscheiden sich mehr oder weniger stark in ihren Eigenschaften und Möglichkeiten und in ihrer Syntax.



Man kann bei Shells im Wesentlichen zwei große inkompatible Strömungen identifizieren, nämlich die Bourne-artigen Shells (`sh`, `ksh`, `bash`, ...) und die C-Shell-artigen Shells (`csh`, `tcsh`, ...). Die Bash versucht, die wichtigsten C-Shell-Eigenschaften zu integrieren.

Welche Shell für was? Sie können als Benutzer selbst entscheiden, welche Shell Sie für was verwenden möchten. Entweder Sie starten die Shell Ihrer Wünsche durch den entsprechenden

Programmaufruf oder Sie stellen sich die Shell als Login-Shell ein, die das System beim Anmelden für Sie startet. Zur Erinnerung, der Typ der Login-Shell wird in der Benutzerdatenbank (meist in `/etc/passwd`) festgelegt. Ihren Login-Shell-Eintrag können Sie mit Hilfe des Kommandos `chsh` abändern:

```
$ getent passwd tux
tux:x:1000:1000:Tux der Pinguin:/bin/bash:/home/tux
$ chsh
Password: geheim
Changing the login shell for tux
Enter the new value, or press return for the default
  Login Shell [/bin/bash]: /bin/csh
$ getent passwd tux
tux:x:1000:1000:Tux der Pinguin:/bin/csh:/home/tux
$ chsh -s /bin/bash
Password: geheim
$ getent passwd tux
tux:x:1000:1000:Tux der Pinguin:/bin/bash:/home/tux
```

Mit der Option `-s` können Sie direkt die gewünschte Shell angeben; wenn Sie das nicht tun, werden Sie interaktiv gefragt.



Für `chsh` stehen Ihnen alle Shells zur Verfügung, die in der Datei `/etc/shells` erwähnt werden (jedenfalls soweit sie tatsächlich als Programme auf Ihrem System installiert sind – viele Distributoren tragen alle Shells der Distribution in die Datei ein, egal ob Sie das Paket eingespielt haben oder nicht.)

Sie können herausfinden, welche Shell Sie gerade verwenden, indem Sie sich den Wert der Shellvariablen `$0` anschauen: Aktuelle Shell

```
$ echo $0
/bin/bash
```

Unabhängig davon, mit welcher Shell Sie interaktiv arbeiten, können Sie natürlich auch frei wählen, welche Shell(s) Sie für Ihre Shellskripte nutzen. Hier kommen verschiedene Erwägungen ins Spiel:

- Es ist natürlich verlockend, die interaktiv benutzte Shell auch für Shellskripte einzusetzen. Schließlich brauchen Sie nur das, was Sie sonst auf der Kommandozeile tippen würden, in eine Datei zu schreiben. Das hat nur zwei mögliche Nachteile: Zum einen setzt das voraus, dass überall da, wo Sie ein Shellskript später laufen lassen wollen, Ihre Shell installiert ist – je nachdem, welche Shell Sie benutzen, ist das kein Problem oder doch. Zum anderen sind manche Shells zwar interaktiv gut, aber nicht besonders für Skripte geeignet, oder umgekehrt.



Die C-Shell hat zwar für den interaktiven Gebrauch ihre treue Fan-Gemeinde, für den Einsatz in Shellskripten ist sie jedoch aufgrund diverser Implementierungsfehler und syntaktischer Inkonsistenzen universell verpönt (siehe hierzu [Chr96]). Obwohl die Tenex-C-Shell da nicht ganz so schlimm ist, sind gewisse Vorbehalte sicherlich nach wie vor angebracht.

- In vielen Fällen tun Sie gut daran, bei den Anforderungen an die zu verwendende Shell Bescheidenheit walten zu lassen, etwa indem Sie sich in Ihren Skripten auf den von POSIX genormten Funktionsumfang für Shells beschränken und die (komfortablen) Bash-Erweiterungen links liegen lassen. Dies ist zum Beispiel wichtig für Init-Skripte – in vielen Linux-Systemen ist die Bash zwar die Standard-Shell, aber es lassen sich Umgebungen denken, in denen sie zugunsten etwas weniger fetter Shells weggelassen wird.

Ein Skript, das sich als »Bourne-Shell-Skript« ausgibt und vom System auch so gestartet wird, sollte also keine »Bashismen« enthalten; wenn Sie wirklich Bash-Eigenschaften ausnutzen, sollten Sie Ihr Skript explizit als »Bash-Skript« kenntlich machen. Wie das geht, sehen Sie in Kapitel 2.



Im Zuge des Einsatzes von Linux als *embedded system* (also extrem abgespeckte Umgebung etwa als Betriebssystem für einen Router oder digitalen Videorecorder) und mit dem zunehmenden Interesse an Dingen wie der `initrd` sind »alternative« Shells wieder in den Vordergrund gerückt, die sich benehmen wie die »POSIX-Shell« und die Bash-Erweiterungen nicht anbieten. Einige Namen sind `ash`, `dash` oder `busybox` (letztere bringt außer der Shell-Funktionalität auch noch die meisten sonst externen Hilfsprogramme eingebaut mit). Ebenfalls erwähnenswert ist die `sash` oder *System Administrator's Shell*, die ebenfalls die wichtigsten externen Kommandos eingebaut hat und, statisch gelinkt, gerne als »Notnagel« für harte Fälle der Systemrettung vorgehalten wird.

Portabilität: Shell vs. Programme Übrigens: Häufig ist nicht die Shell das Problem, das die Portabilität eines Shell-Skripts einschränkt, sondern die externen Programme, die das Skript aufruft. Diese müssen auf dem »Zielsystem« natürlich auch vorhanden sein und sich genauso benehmen wie auf Ihrem System. Solange Sie nur Linux verwenden, ist das in der Regel nicht so schlimm – aber wenn Sie auf einem Linux-System Skripte für proprietäre Unix-Systeme erstellen, können Sie Überraschungen erleben: Die GNU-Versionen der Standardwerkzeuge wie `grep`, `cat`, `ls`, `sed`, ... haben nämlich nicht nur mehr und leistungsfähigere Optionen als die Implementierungen, die Sie auf vielen proprietären Unixen finden, sondern oft auch weniger Fehler und willkürliche Grenzen (etwa bei der Länge von Eingabezeilen). Auch hier ist also oft Mäßigung angesagt, wenn Sie nicht sicher sind, dass Sie immer die GNU-Werkzeuge zur Verfügung haben.



Betrachten Sie zur Illustration ein beliebiges `configure`-Skript eines freien Programmpakets (`configure`-Skripte sind Shellskripte, die auf maximale Portabilität getrimmt sind). Tun Sie das im eigenen Interesse aber nicht unmittelbar vor oder nach der Nahrungsaufnahme.

Die folgenden Kapitel beschränken sich auf die Bourne-Again-Shell, was aber nicht heißt, dass Sie das Gelernte nicht auch zum Großteil auf andere Shells anwenden können. Spezielle Bash-Eigenschaften sind, wo möglich, als solche gekennzeichnet.

Übungen



1.3 [!1] Versuchen Sie herauszufinden, welche Shells auf Ihrem System installiert sind.



1.4 [!2] Rufen Sie – falls Sie sie haben – die Tenex-C-Shell (`tcsh`) auf, geben Sie dort ein Kommando mit einem kleinen Tippfehler ein und drücken Sie . – Wie kommen Sie wieder zu Ihrer alten Shell zurück?



1.5 [!1] Weisen Sie sich eine andere Shell als Login-Shell zu (zum Beispiel die `tcsh`). Prüfen Sie, ob es geklappt hat, etwa indem Sie sich auf einer anderen virtuellen Konsole anmelden. Ändern Sie danach Ihre Login-Shell wieder zur Bash zurück.



1.6 [2] Installieren Sie die `sash` (falls Ihre Distribution sie anbietet) und führen Sie einen neuen Benutzer `rootsash` ein, der die `sash` als Login-Shell hat, aber sonst die Rechte von `root` genießt.

1.3 Die Bourne-Again-Shell

1.3.1 Das Wichtigste

Die Bourne-Again-Shell (oder Bash) wurde im Rahmen des GNU-Projektes der *Free Software Foundation* von Brian Fox und Chet Ramey entwickelt und vereinigt Funktionen der Korn- und der C-Shell.



Da die Korn-Shell eine Weiterentwicklung der Bourne-Shell ist und die Bash quasi eine Rückbesinnung der traditionell eher mit der C-Shell assoziierten BSD-Welt auf die Bourne-Konzepte darstellt, ist der Name »Bourne-Again-Shell« – englisch-phonetisch nicht von der »wiedergeborenen Shell« zu unterscheiden – angebracht.

Einen ausführlichen Überblick über die Möglichkeiten der Bash im interaktiven Gebrauch gibt die Linup-Front-Schulungsunterlage *Linux-Grundlagen für Anwender und Administratoren*. Wir wiederholen hier nur kurz das Wichtigste:

Variable Die Bash unterstützt – wie die meisten Shells – die Verwendung von **Variablen**, die gesetzt und wieder abgerufen werden können. Variable werden auch zur Konfiguration zahlreicher Aspekte der Shell selbst verwendet, beispielsweise sucht die Shell ausführbare Programme für Kommandos in den Verzeichnissen, die in PATH aufgezählt werden, oder orientiert sich für die Ausgabe einer Eingabeaufforderung am Wert der Variablen PS1. Mehr über Variable finden Sie im Abschnitt 3.1.

Aliasnamen Das Kommando alias gibt Ihnen die Möglichkeit, eine längere Kommando-Folge abzukürzen. Beispielsweise definieren Sie durch

```
$ alias c='cal -m; date'
$ type c
c is aliased to `cal -m; date`
```

das neue »Kommando« c. Immer, wenn Sie jetzt »c« aufrufen, wird die Bash das Kommando cal mit der Option -m gefolgt vom Kommando date für Sie ausführen. Dieses **Alias** können Sie auch wieder in anderen Alias-Definitionen benutzen. Sie können sogar durch ein Alias ein bestehendes Kommando »umdefinieren«. Mit

```
$ alias rm='rm -i'
```

wird beispielsweise das Kommando rm entschärft. Allerdings ist das von zweifelhaften Nutzen: Haben Sie sich einmal an die »sichere« Variante gewöhnt, so kann es sein, dass Sie auch dort damit rechnen, wo das Alias nicht gesetzt ist. Es ist daher besser, wenn Sie sich bei potenziell gefährlichen Kommandos einen neuen Namen ausdenken.

Durch alias (ohne Argumente) können Sie sich alle zur Zeit definierten Aliase anzeigen lassen, und mit unalias können Sie einzelne oder alle Aliase löschen.

Funktionen Wenn Sie mehr als reine textuelle Ersetzung wie bei Aliasen benötigen, so können Ihnen Funktionen weiterhelfen. Mehr über Funktionen erfahren Sie in Abschnitt 3.5.

Das Kommando set Mit dem Bash-internen Kommando set können Sie nicht nur alle Shell-Variablen anzeigen (wenn Sie es ohne Parameter aufrufen), sondern auch Options-Schalter der Bash setzen. So können Sie mit »set -C« (oder gleichwertig: »set -o noclobber«) verhindern, dass Sie durch Ausgabe-Umlenkung versehentlich eine existierende Datei überschreiben.

Mit »set -x« (oder »set -o xtrace«) können Sie sehen, was die Bash für Schritte unternimmt, um zu ihrem Ergebnis zu kommen:

```
$ set -x
$ echo "My $HOME is my castle"
+ echo 'My /home/tux is my castle'
My /home/tux is my castle
```

Wenn anstelle des »-« bei den Optionen ein »+« gesetzt ist, wird die entsprechende Option abgeschaltet.

1.3.2 Login-Shells und interaktive Shells

Shell ist nicht gleich Shell. Natürlich unterscheidet sich die Bash von einer C-Shell oder Korn-Shell, aber schon das Verhalten eines Bash-Prozesses kann von dem eines anderen Bash-Prozesses abweichen, je nachdem wie die Bash jeweils aufgerufen wurde.

Prinzipiell gibt es drei Varianten: Login-Shell, interaktive Shell und nichtinteraktive Shell. Diese Formen unterscheiden sich darin, welche Konfigurationsdateien eingelesen werden.

Login-Shell Diese Form der Shell erhalten Sie direkt nach dem Anmelden am System. Das die Shell startende Programm, also `login`, »su -«, `ssh` usw., gibt der Shell ein »-« als erstes Zeichen des Programm-Namens mit. Dadurch weiß die Shell, dass sie eine Login-Shell sein soll. Wenn Sie sich frisch angemeldet haben, sollte es etwa so aussehen:

```
$ echo $0
-bash
$ bash
$ echo $0
bash
```

Aufruf der Bash von Hand; keine Anmeldung

Alternativ können Sie die Bash auch mit der Option `-l` aufrufen, damit diese sich als Login-Shell fühlt.

Jede Bourne-artige Shell (nicht nur die Bash) führt als erstes die Kommandos in der Datei `/etc/profile` aus. Dadurch können systemweit beim Anmelden zum Beispiel Umgebungsvariable oder die `umask` gesetzt werden.



Wenn Sie auf Ihrem System sowohl die Bash als auch die Bourne-Shell verwenden, müssen Sie darauf achten, in `/etc/profile` nur Konstruktionen zu verwenden, die auch die Bourne-Shell versteht. Alternativ dazu können Sie von Fall zu Fall prüfen, ob gerade eine Bourne-Shell oder eine Bash die Datei bearbeitet. Wenn es sich um eine Bash handelt, ist die Umgebungsvariable `BASH` definiert.

`.profile` Danach sucht die Bash die Dateien `.bash_profile`, `.bash_login` und `.profile` im Heimatverzeichnis des Benutzers. Nur die *erste* gefundene Datei wird abgearbeitet.



Auch dieses Verhalten hat seinen Ursprung in der Bourne-Shell-Kompatibilität der Bash. Wenn Sie auf Ihr Heimatverzeichnis von verschiedenen Rechnern aus zugreifen können, von denen manche die Bash und andere nur die Bourne-Shell unterstützen, können Sie Bash-spezifische Konfigurationseinstellungen in der Datei `.bash_profile` hinterlegen, so dass diese eine Bourne-Shell, die nur `.profile` liest, nicht aus dem Tritt bringen. (Aus der `.bash_profile` können Sie dann die `.profile` einlesen.) – Alternativ dazu können Sie in der Datei `.profile` den oben angedeuteten Ansatz mit der `BASH`-Variable benutzen.



Der Name `.bash_login` kommt aus der C-Shell-Tradition. Eine etwa vorhandene `.login`-Datei für die C-Shell lässt die Bash aber links liegen.

Beenden Sie eine Anmelde-Shell, so arbeitet sie die Datei `.bash_logout` im Heimatverzeichnis ab, sofern sie existiert.

Interaktive Shell Wenn Sie die Bash ohne Dateinamen-Argumente (aber möglicherweise mit Optionen) aufrufen und ihre Standard-Ein- und -Ausgabe mit einem »Terminal« verbunden ist (`xterm` und Konsorten reichen), so versteht sie sich als interaktive Shell. Als solche liest sie beim Starten die Dateien `/etc/bash.bashrc` und `.bashrc` in Ihrem Heimatverzeichnis und führt die darin enthaltenen Kommandos aus.

interaktive Shell
/etc/bash.bashrc
.bashrc



Ob eine interaktive Shell `/etc/bash.bashrc` liest, ist in Wirklichkeit eine Konfigurationsoption. Bei den wesentlichen Distributionen, beispielsweise den SUSE-Distributionen und Debian GNU/Linux, ist diese Option allerdings eingeschaltet.

Beim Verlassen einer interaktiven Shell, die keine Anmelde-Shell ist, werden keine Dateien ausgewertet.

Nichtinteraktive Shell Nichtinteraktive Shells werten gar keine Dateien beim Starten oder Beenden aus. Zwar können Sie einer solchen Bash durch die Umgebungsvariable `BASH_ENV` einen Dateinamen zum Auswerten mitgeben, diese Variable ist aber zumeist nicht gesetzt.

Eine Shell ist dann nichtinteraktiv, wenn sie benutzt wird, um ein Shell-Skript auszuführen, oder wenn ein Programm sich einer Shell bedient, um ein anderes Programm zu starten. Das ist der Grund, warum Aufrufe der Art

```
$ find -exec ll {} \;  
find: ll: No such file or directory
```

fehlschlagen: `find` startet eine nichtinteraktive Shell, um `ll` auszuführen. Obwohl `ll` auf vielen Systemen verfügbar ist, handelt es sich nur um ein Alias für »`ls -l`«. Als Alias muss es aber in jeder Shell definiert werden, da Aliase nicht »vererbt« werden. Eine Konfigurationsdatei mit Aliasdefinitionen wird aber normalerweise nicht eingelesen.

Distributionspezifische Besonderheiten Die strikte Trennung zwischen Anmelde-Shell und normalen interaktiven Shells erzwingt, dass Sie gewisse Einstellungen sowohl in `.profile` als auch in `.bashrc` vornehmen müssen, damit sie in jeder Shell wirksam werden. Um diese fehleranfällige Doppelarbeit zu vermeiden, haben viele Distributionen in der mitgelieferten Datei `.profile` eine Zeile wie

```
## ~/.profile  
test -r ~/.bashrc && . ~/.bashrc
```

die das Folgende bewirkt: Wenn `.bashrc` existiert und lesbar ist (`test...`), dann (`&&`) wird die Datei `.bashrc` abgearbeitet (`».«` ist ein Shell-Kommando, das die Datei so einliest, als würde ihr Inhalt an dieser Stelle eingetippt).

Möglicherweise hat Ihre Distribution die Bash auch so übersetzt, dass sie noch andere Dateien einliest. Dies können Sie mit `strace` überprüfen; es listet Ihnen alle Systemaufrufe auf, die ein anderes Programm benutzt, darunter auch den `open`-Aufruf zum Öffnen einer Datei. Erst damit können Sie wirklich sicher sein, welche Dateien ausgewertet werden.

Übungen



1.7 [!2] Überzeugen Sie sich davon, dass Ihre Bash die Dateien `/etc/profile`, `/etc/bash.bashrc`, `~/.profile` und `~/.bashrc` wie beschrieben verwendet. Untersuchen Sie alle Abweichungen.



1.8 [1] Woran merkt ein Shellprozess, dass er eine »Login-Shell« sein soll?

1.9 [3] Wie können Sie eine Shell als »Login-Shell« verwenden, die nicht in `/etc/shells` steht?

1.3.3 Dauerhafte Konfigurationsänderungen

Individuelle Änderungen Individuelle Anpassungen Ihrer Arbeitsumgebung halten nur bis zum Ende des Shell-Prozesses vor. Wollen Sie Ihre Änderungen auch beim nächsten Anmelden vorfinden, so müssen Sie dafür sorgen, dass die Shell sie automatisch beim Start ausführt. Umgebungsvariablen, Aliase, Shell-Funktionen, die `umask` usw. müssen in einer der in Abschnitt 1.3.2 genannten Dateien gesetzt werden – die Frage ist nur, in welcher.

Im Fall von Aliasen ist die Antwort leicht: Da sie nicht vererbt werden können, müssen sie von jeder Shell neu gesetzt werden. Daher müssen Sie Aliase in der Datei `~/.bashrc` definieren. (Damit der Alias aber auch in der Anmelde-Shell funktioniert, müssen Sie ihn *zusätzlich* in die Datei `~/.profile` eintragen, wenn dort nicht, wie oben beschrieben, die Datei `~/.bashrc` eingelesen wird.)

Gute Kandidaten für die Datei `.bashrc` sind auch Variablen, die das Verhalten der Shell steuern (`PS1`, `HISTSIZE` u. a.), aber sonst nicht von Interesse sind, d. h. eben keine Umgebungsvariablen. Wenn Sie wollen, dass jede Shell »frisch« startet, so müssen Sie diese Variablen jedesmal neu setzen, was daher in `.bashrc` geschehen sollte.

Anders sieht die Sache bei Umgebungsvariablen aus. Diese werden üblicherweise einmal gesetzt, genau wie Änderungen der Tastaturbelegung und ähnliches. Es genügt daher, sie in `.profile` zu definieren. Bei Konstruktionen wie

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

die die Variable `PATH` um eine weitere Komponente verlängert, verbietet sich die Datei `~/.bashrc`, weil sonst der Wert der Variable immer weiter verlängert würde.



Wenn Ihr Rechner in den Runlevel 5 startet, also die Anmeldung über den X-Displaymanager abgewickelt wird, haben Sie nicht direkt eine Anmelde-Shell. Damit die Einstellungen trotzdem berücksichtigt werden, sollten Sie diese in die Datei `~/.xsession` eintragen oder aus dieser heraus Ihre `.profile` einlesen.

Systemweite Änderungen Als Systemadministrator können Sie Einstellungen für alle Benutzer in den Dateien `/etc/profile` und `/etc/bash.bashrc` vornehmen. Damit diese Anpassungen eine Software-Aktualisierung des Systems überleben, haben viele Distributoren besondere Vorkehrungen getroffen: SUSE zum Beispiel empfiehlt die Verwendung der Dateien `/etc/profile.local` und `/etc/bash.bashrc.local`, die von den entsprechenden Schwesterdateien eingelesen werden.

`/etc/skel` Eine andere Möglichkeit der globalen Änderung stellt das Verzeichnis `/etc/skel` dar, das »Rohgerüst« eines Heimatverzeichnisses, das neue Benutzer kopiert bekommen, wenn Sie `useradd` mit der Option `-m` aufrufen. Alle darin enthaltenen Dateien und Verzeichnisse werden damit zur Grundausstattung des neuen Heimatverzeichnisses.

Wenn Sie in `/etc/skel` eine Datei `.bashrc` des Inhalts

```
## Systemweite Einstellungen; bitte nicht veraendern:
test -r /etc/bash.local && . /etc/bash.local

## Individuelle Einstellungen hier einfüegen:
```

ablegen, so können Sie Änderungen am System vornehmen (in `/etc/bash.local`), die für alle Benutzer wirksam werden.

Natürlich können Sie in `/etc/skel` auch noch vorgefertigte Konfigurationsdateien für beliebige andere Programme, Verzeichnisstrukturen usw. ablegen.

Übungen



1.10 [!1] Installieren Sie das Alias `hallowelt` für das Kommando »echo Hallo Welt« so, dass es in Ihrer Login-Shell und allen interaktiven Shells zur Verfügung steht.



1.11 [!1] Installieren Sie das Alias `hallowelt` für das Kommando »echo Hallo Welt« so, dass es in allen Login-Shells und interaktiven Shells im ganzen System zur Verfügung steht.



1.12 [2] Sorgen Sie dafür, dass Sie `hallowelt` auch in Ihren nichtinteraktiven Shells aufrufen können.

1.3.4 Tastatur-Belegung und Abkürzungen

Die Bash verwendet verschiedene Tastaturkürzel, um das Editieren von Kommandos bei der Eingabe zu erlauben und besondere Eigenschaften anzusteuern.

Noch weitergehende Einstellungsmöglichkeiten erlaubt Ihnen die Datei `.inputrc` `.inputrc` in Ihrem Heimatverzeichnis bzw. `/etc/inputrc` für systemweite Einstellungen. Diese Datei ist die Konfigurationsdatei für die Readline-Bibliothek, die von der Bash und anderen Programmen benutzt wird. Beispielsweise liegt es an der Readline-Bibliothek, dass `[Strg]+[r]` das Rückwärtssuchen in der History erlaubt.

Die Einstellungen für Readline gelten sowohl für die virtuellen Terminals als auch für die grafische Oberfläche. Die ganzen Details verrät Ihnen `readline(3)`, wir beschränken uns hier auf einige Beispiele. Wenn Sie die Zeilen

Control-t:	tab-insert
Control-e:	"cal\C-m"

in die Datei `.inputrc` eintragen, dann wird das Drücken von `[Strg]+[t]` einen Tabulator-Vorschub erzeugen, was Sie normalerweise in der Bash nicht bekommen, da die Tabulator-Taste schon belegt ist. Das Drücken von `[Strg]+[e]` startet ein Makro, in diesem Fall die Zeichen »cal« gefolgt von `[Strg]+[m]` (dafür steht »\C-m«), was dem Drücken von `[←]` entspricht.

Die Änderungen in der Datei werden allerdings nur wirksam, wenn Sie die Umgebungsvariable `INPUTRC` auf `$HOME/.inputrc` gesetzt haben und eine neue Bash starten oder wenn Sie das Kommando `bind -f ~/.inputrc` ausführen.

Kommandos in diesem Kapitel

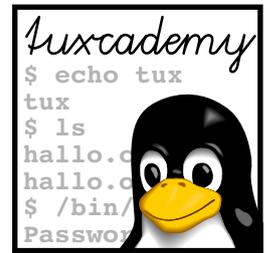
ash	Eine leichtgewichtige POSIX-konforme Shell	ash(1)	16
busybox	Eine Shell, die Versionen vieler Unix-Werkzeuge integriert hat	busybox(1)	16
chsh	Ändert die Login-Shell eines Benutzers	chsh(1)	14
dash	Eine leichtgewichtige POSIX-konforme Shell	dash(1)	16
file	Rät den Typ einer Datei anhand des Inhalts	file(1)	14
find	Sucht nach Dateien, die bestimmte Kriterien erfüllen	find(1), Info: find	14
sash	„Stand-Alone Shell“ mit eingebauten Kommandos, zur Problembehebung	sash(8)	16
strace	Protokolliert die Systemaufrufe eines Programms	strace(1)	19

Zusammenfassung

- Die Shell erlaubt den kommandoorientierten Umgang mit dem Linux-System. Die meisten Shells haben Eigenschaften von Programmiersprachen und gestatten die Erstellung von »Shellskripten«.
- Shellskripte werden im Linux-System an vielen Stellen verwendet.
- Es gibt eine Vielzahl von Shells. Die Standardshell bei den meisten Linux-Distributionen ist die »Bourne-Again-Shell«, kurz »Bash«, aus dem GNU-Projekt.
- Die Bash kombiniert Eigenschaften der Bourne- und Korn-Shell mit der der C-Shell.
- Die Bash (wie die meisten Shells) verhält sich unterschiedlich, je nachdem, ob sie als Login-Shell, als interaktive oder nichtinteraktive Shell gestartet wurde.
- Benutzerspezifische Voreinstellungen für die Bash können in einer der Dateien `~/.bash_profile`, `~/.bash_login` oder `~/.profile` sowie in der Datei `~/.bashrc` gemacht werden.
- Systemweite Voreinstellungen für alle Benutzer können in den Dateien `/etc/profile` und `/etc/bash.bashrc` gemacht werden.
- Abweichende Tastaturbelegungen sind in `~/.inputrc` und `/etc/inputrc` konfigurierbar.

Literaturverzeichnis

- Chr96** Tom Christiansen. »Csh Programming Considered Harmful«, Oktober 1996. <http://www.faqs.org/faqs/unix-faq/shell/csh-whynt/>



2

Shellskripte

Inhalt

2.1	Einleitung	24
2.2	Aufruf von Shellskripten	24
2.3	Aufbau von Shellskripten	26
2.4	Shellskripte planen	27
2.5	Fehlertypen	28
2.6	Fehlererkennung	29

Lernziele

- Einsatzgebiete und grundlegende Syntax von Shellskripten kennen
- Shellskripte aufrufen können
- #!-Zeilen verstehen und formulieren können

Vorkenntnisse

- Vertrautheit mit der Kommandooberfläche von Linux
- Umgang mit Dateien und einem Texteditor
- Shell-Vorkenntnisse (etwa aus Kapitel 1)

2.1 Einleitung

- Vorteil** Der unschlagbare Vorteil von Shellskripten ist: Wenn Sie mit der Shell umgehen können, dann können Sie auch programmieren! Shellskripte bieten sich immer dann an, wenn es darum geht, eine Aufgabe zu automatisieren, die Sie auch interaktiv »von Hand« hätten erledigen können. Umgekehrt können Sie die Kontrollstrukturen für Shellskripte auch jederzeit auf der Kommandozeile benutzen. Dies vereinfacht nicht nur das Testen von Konstruktionen immens, es macht oft ein Skript komplett überflüssig.
- Einsatzgebiete** Die Einsatzgebiete von Shellskripten sind vielfältig. Überall, wo dauernd die gleichen Kommandos eingegeben werden müssten, lohnt es sich, ein Shellskript zu schreiben (etwa bei der regelmäßigen Suche nach Dateien mit bestimmten Eigenschaften). Meist werden Shellskripte aber verwendet, um komplexe Aufgaben zu vereinfachen (zum Beispiel automatisches Starten von Netzwerkdiensten). In der Regel geht es dabei nicht darum, ausgefeilte bzw. schöne Programme zu schreiben, sondern die eigene Arbeit zu vereinfachen. Was nicht heißt, dass Sie einen schlechten Programmierstil entwickeln sollten – Kommentare und Dokumentation sind spätestens dann sinnvoll, wenn auch andere Personen Ihre Skripte nutzen und verstehen sollen.
- Nachteile** Allerdings sollten Sie Shellskripte auch nicht überstrapazieren: Überall da, wo komplexere Datenstrukturen benötigt werden oder es auf Effizienz oder Sicherheit ankommt, sind Sie mit echten Programmiersprachen meist besser beraten. Neben den »klassischen« Sprachen wie C, C++ und Java kommen hier auch die modernen »Skriptsprachen« Tcl, Perl oder Python in Frage.

2.2 Aufruf von Shellskripten

Ein Shellskript können Sie auf verschiedene Art und Weise starten. Am einfachsten übergeben Sie seinen Namen einer Shell als Aufrufparameter:

```
$ cat skript.sh
echo Hallo Welt
$ bash skript.sh
Hallo Welt
```

Das ist natürlich unbefriedigend, da Benutzer Ihres Skripts zum einen wissen müssen, *dass* es sich um ein Shellskript handelt (und nicht etwa ein ausführbares Maschinenprogramm oder ein Skript für die Programmiersprache Perl), und zum anderen, dass es mit der Bash ausgeführt werden muss. Es wäre schöner, wenn der Aufruf Ihres Skripts aussähe wie der jedes anderen Programms. Dazu müssen Sie das Skript mit `chmod` für ausführbar erklären:

```
$ chmod u+x skript.sh
```

Anschließend können Sie es mit

```
$ ./skript.sh
```

direkt starten.



Shellskript-Dateien müssen nicht nur ausführbar sein, sondern auch lesbar (im Sinne des `r`-Rechts). Bei Maschinenprogrammen, die im Binärformat vorliegen, genügt Ausführbarkeit.

Wollen Sie auf das unschöne »./« verzichten, so müssen Sie dafür sorgen, dass die Shell Ihr Skript finden kann. Zum Beispiel können Sie ».« – das aktuelle Arbeitsverzeichnis – in Ihren Suchpfad für Kommandos (Umgebungsvariable `PATH`) aufnehmen. Das ist aber einerseits aus Sicherheitsgründen keine gute Idee (erst

recht nicht für root) und andererseits unpraktisch, weil es ja sein könnte, dass Sie Ihr Skript mal aus einem anderen Verzeichnis heraus aufrufen wollen. Am besten legen Sie sich für oft gebrauchte Shellskripte ein Verzeichnis `$HOME/bin` an und nehmen das explizit in Ihren `PATH` auf.

Die dritte Methode, ein Shellskript aufzurufen, besteht darin, das Skript statt in einem Kindprozess in der aktuellen Shell auszuführen. Dazu können Sie das Kommando »source« bzw. seine Kurzschreibweise ».« verwenden:

```
$ source skript.sh
Hallo Welt
$ . skript.sh
Hallo Welt
```

(Bitte beachten Sie, dass bei der Kurzschreibweise hinter dem Punkt ein Leerzeichen folgen muss.) Einem so gestarteten Skript steht der komplette Kontext der aktuellen Shell zur Verfügung. Während ein in einem Kindprozess gestartetes Skript zum Beispiel das aktuelle Verzeichnis oder die *umask* Ihrer interaktiven Shell nicht direkt verändern kann, können Sie das über ein mit `source` gestartetes Skript durchaus. (Mit anderen Worten: Ein mit `source` aufgerufenes Skript wird so ausgeführt, als würden Sie die Zeilen in der Datei direkt in Ihre interaktive Shell eintippen.)



Wichtig ist diese Methode beispielsweise dann, wenn Sie aus einem Shellskript heraus auf Shellfunktionen, -variable oder Aliase zugreifen wollen, die in einem anderen Shellskript definiert werden – etwa im Sinne einer »Bibliothek«. Würde jenes Shellskript wie üblich in einem Kindprozess ausgeführt, dann hätten Sie nichts von den Definitionen darin!

Bei der Namensgebung Ihrer Shellskripte sollten Sie aussagekräftige Bezeichnungen wählen. Es bietet sich zusätzlich an, Datei-Endungen wie ».sh« oder ».bash« zu verwenden, damit man auf den ersten Blick erkennen kann, dass es sich um Shellskripte handelt. Natürlich können Sie auch auf die Endung verzichten. Insbesondere in Experimentierphasen ersparen Sie sich durch die Endung aber viel Ärger, denn so naheliegende Namen wie `test` oder `script` werden schon von Systemkommandos benutzt.

Namensgebung



Wie schon erwähnt, sollten Sie die Endung ».sh« für Skripte reservieren, die auch mit der Bourne-Shell oder Kompatiblen laufen, also keine speziellen Bash-Eigenschaften voraussetzen.

Übungen



2.1 [!2] Erstellen Sie eine Textdatei namens `meinskript.sh`, das zum Beispiel mit »echo« eine Meldung ausgibt, und machen Sie diese Datei ausführbar. Überzeugen Sie sich, dass die drei Aufrufmöglichkeiten

```
$ bash meinskript.sh
$ ./meinskript.sh
$ source meinskript.sh
```

im Sinne der obigen Beschreibung funktionieren.



2.2 [!1] Mit welcher Methode liest die Login-Shell die Dateien `/etc/profile` und `$HOME/.bash_profile` – Kindprozess oder »source«?



2.3 [2] Ein Benutzer kommt mit der folgenden Beschwerde zu Ihnen: »Ich habe ein Shellskript geschrieben und wenn ich es aufrufe, passiert gar nichts. Dabei gebe ich als allererstes eine Meldung auf die Standardfehlerausgabe aus! Linux ist Mist!« Bei einer peinlichen Befragung Ihrerseits kommt heraus, dass der Benutzer sein Skript `test` genannt hat. Was geht hier vor?



2.4 [3] (Trickreich.) Wie würden Sie dafür sorgen, dass ein Shellskript in einem Kindprozess ausgeführt wird und trotzdem das aktuelle Verzeichnis der aufrufenden Shell ändern kann?

2.3 Aufbau von Shellskripten

Shellskripte sind eigentlich nur Folgen von Shellkommandos, die in einer Textdatei abgespeichert sind. Die Shell kann ihre Eingabe wahlweise von der Tastatur (Standardeingabe) oder aus einer anderen Quelle, etwa einer Shellskript-Datei lesen – bei der Ausführung der Kommandos besteht da eigentlich kein Unterschied. Zeilenumbrüche dienen zum Beispiel als Kommandooperatoren, ganz wie auf der »echten« Kommandozeile.

Zeilenumbrüche

Einige Lesbarkeits-Tipps: Was Sie auf der Kommandozeile der Bequemlichkeit halber in der Form

```
⟨Kommando1⟩ ; ⟨Kommando2⟩
```

eingeben würden, sollten Sie im Skript der Übersichtlichkeit wegen untereinander schreiben:

```
⟨Kommando1⟩
⟨Kommando2⟩
```

Für die Ausführung bedeutet das keinen Unterschied.

Die Lesbarkeit eines Skripts können Sie ferner durch gezielte Verwendung von Leerzeilen steigern, die von den Shells ignoriert werden – ganz wie auf der Kommandozeile. Ebenfalls ignoriert wird alles, was einem Rautenzeichen (»#«) folgt.

Kommentare

Damit können Sie Ihre Skripte kommentieren:

```
# Zuerst kommt Kommando1
⟨Kommando1⟩
⟨Kommando2⟩ # Das ist Kommando2
```

Kommentarblock

Größere Skripte sollten mit einem Kommentarblock anfangen, der beschreibt, wie das Skript heißt, was es tun soll und wie es das macht, wie es aufgerufen werden muss und so weiter. Auch der Name des Autors und eine Versionsgeschichte könnten dort erscheinen.

Mit `chmod` als ausführbar gekennzeichnete Textdateien werden vom System als Skripte für die Shell `/bin/sh` angesehen – auf Linux-Systemen ist das oft (aber nicht immer) ein Link auf die Bash. Um sicherzugehen, sollten Shellskripte daher mit einer Zeile beginnen, die die Zeichen »#!« und den Namen der gewünschten Shell als absoluten Pfad enthält. Zum Beispiel:

```
$ cat skript
#!/bin/bash
<<<<<<
```

Dann wird zur Ausführung des Skripts die benannte Shell herangezogen, indem der Dateiname des Skripts als Parameter an den angegebenen Shellnamen angehängt wird. Unter dem Strich startet der Linux-Kern in unserem Beispiel also das Kommando

```
/bin/bash skript
```



Das Programm, das das Skript ausführt, muss keine Shell im engeren Sinne sein – jedes Programm, das im Binärcode vorliegt, kommt in Frage. Auf diese Weise können Sie zum Beispiel »awk-Skripte« schreiben (Kapitel 7).



Die »#!«-Zeile darf bei Linux maximal 127 Zeichen lang sein und außer dem Programmnamen auch Parameter enthalten; der Skriptname wird in jedem Fall ans Ende angehängt. Beachten Sie, dass proprietäre Unix-Systeme oft wesentlich engere Grenzen setzen: Zum Beispiel ist nur eine Gesamtlänge von 32 Zeichen mit höchstens einem Optionsargument erlaubt. Dies kann zu Schwierigkeiten führen, wenn Sie ein Linux-Shellskript auf einem proprietären Unix-System ausführen wollen.

Übungen



2.5 [1] Mit welcher Ausgabe rechnen Sie, wenn das ausführbare Skript `blubb` mit dem folgenden Inhalt

```
#!/bin/echo bla fasel
echo Hallo Welt
```

über das Kommando »./blubb« aufgerufen wird?



2.6 [2] Was ist als erste Zeile für Shellskripte besser – »#!/bin/sh« oder »#!/bin/bash«?

2.4 Shellskripte planen

Wer auch nur ein bisschen Programmiererfahrung hat, hat die leidvolle Erfahrung gemacht: Programme sind selten auf Anhieb korrekt. Das *Debuggen* größerer Programme kostet meist sehr viel Zeit und Mühe. Der bekannte Programmierer und Autor Brian W. Kernighan formuliert das so:

Jeder weiss, dass Fehlersuche zweimal so schwierig ist wie Programme erst mal hinzuschreiben. Wenn Sie also schon beim Programmieren so clever sind, wie Sie sein können – wie wollen Sie dann je die Fehler finden?

Es empfehlen sich also sorgfältige Planung, schrittweises Vorgehen und die Kenntnis der gängigsten Fehlerquellen.

In die Situation, ein Shellskript verfassen zu müssen, kommen Sie meist, weil Sie eine bestimmte Tätigkeit automatisieren wollen. In einem solchen Fall ist einerseits die Aufgabe des Skriptes klar umrissen, andererseits wissen Sie auch schon grob, welche Kommandos Sie in welcher Reihenfolge ausführen müssen. Zum Schreiben des Skriptes bietet sich hier eine »evolutionäre« Herangehensweise an. Was soviel bedeutet wie: Sie schreiben erst mal alle Kommandos in eine Datei, die Sie auch auf der Kommandozeile eingegeben hätten, und verbinden sie dann auf sinnvolle Weise. Zum Beispiel können Sie Alternativen oder Schleifen einführen, wenn diese das Skript übersichtlicher, fehlertoleranter oder universeller machen. Auch mehrfach benutzte Dateinamen (etwa für Protokolldateien) sollten Sie in Variable schreiben und dann nur noch die Variablennamen verwenden. Kommandozeilenparameter können angeschaut und verwendet werden, um von Aufruf zu Aufruf unterschiedliche Elemente einzusetzen; natürlich sollten Sie dann auch die Vollständigkeit und Plausibilität dieser Parameter prüfen und gegebenenfalls Warnungen oder Fehlermeldungen ausgeben.

Auch bei größeren Programmen können Sie die »evolutionäre« Herangehensweise wählen, was bedeuten würde, Sie fangen intuitiv mit dem an, was Ihnen als erstes einfällt und entwickeln darauf aufbauend das Skript. Der große Nachteil: Es passiert schnell, dass man einen Fehler im Konzept begeht, und die anschließenden Umschreibarbeiten machen einen Haufen unnötige Arbeit.

Sie sollten also auf jeden Fall einen groben Plan mit allen voraussichtlichen Einzelschritten aufstellen – dann erkennen Sie schneller, ob schon das Konzept

Einfache Skripte

Evolution

Größere Projekte

Plan

logische Fehler aufweist. Es reicht dabei vollkommen aus, wenn Sie sich die Einzelschritte in Form einer »natürlichsprachigen« Liste aufschreibt, mit den genauen Kommandos und ihrer Syntax können Sie sich dann später befassen.

Ein weiterer Vorteil dieser planenden Herangehensweise ist, dass Sie für die Einzelschritte im Vorhinein überlegen können, welche Kommandos sich am besten eignen, beispielsweise ob Sie sich mit einfachen Filterkommandos herumschlagen, nicht doch lieber `awk` oder `gar` gleich ein Perl-Skript nehmen wollen

...

Ein guter Plan hilft Ihnen allerdings wenig, wenn Sie nach einigen Monaten merken, dass am Skript etwas zu verbessern ist und Sie das Skript nicht mehr verstehen. Wenn Sie sich schon die Mühe machen, einen Plan zu erstellen, verewigen Sie ihn am besten im Programm selbst – nicht (nur) in Form von Kommandos, sondern in Form von Kommentaren. Dabei sollten Sie darauf verzichten, jedes einzelne Kommando zu kommentieren. Geben Sie lieber die grobe Struktur wieder und kommentieren Sie vor allem den Datenfluss und insbesondere das Format der Ein- und Ausgabe: Meistens ergibt die Verarbeitung der Daten sich relativ zwingend aus den Definitionen der Ein- und Ausgabe, während der Umkehrschluss sich bei weitem nicht so sehr aufdrängt. Auch ist es nicht falsch, bei umfangreicheren Programmen, externe Dokumentation (z. B. Handbuchseiten) zu erstellen.

Dokumentation

Einrückungen

Ein guter Plan hat eine Struktur. Es schadet nicht, diese Struktur auch im Programmtext zum Ausdruck zu bringen, indem Sie zum Beispiel mit Einrückungen arbeiten. Das bedeutet, dass Sie Kommandos, die logisch gesehen auf derselben Ebene liegen, den gleichen Abstand vom linken Textrand geben. Hierbei können Sie wahlweise Tabulator- oder Leerzeichen verwenden, sollten aber konsistent vorgehen.

Übungen



2.7 [!2] Sie möchten ein Shellskript erstellen, das für jeden »echten« Benutzer des Systems (also keine administrativen Konten wie `root` oder `bin`) den Zeitpunkt des letzten Einloggens und den gerade von dessen Heimatverzeichnis belegten Plattenplatz anzeigt. Wie würden Sie die folgenden Schritte anordnen, um einen vernünftigen »Plan« für ein Shellskript zu erhalten?

1. u , t und p ausgeben
2. Den Zeitpunkt t des letzten Einloggens von u bestimmen
3. Ende der Wiederholung
4. Den durch v belegten Plattenplatz p bestimmen
5. Eine Liste aller »echten« Benutzer aufstellen
6. Das Heimatverzeichnis v von u bestimmen
7. Wiederhole die folgenden Schritte für jeden Benutzer u in der Liste



2.8 [2] Entwerfen Sie einen Plan für die folgende Aufgabe: In einer großen Installation sind die Heimatverzeichnisse auf verschiedene Platten verteilt (stellen Sie sich zum Beispiel vor, dass die Heimatverzeichnisse heißen wie `/home/entwick/hugo` oder `/home/market/susi` für die Benutzer `hugo` aus der Entwicklungsabteilung und `susi` aus der Marketingabteilung). Sie möchten in periodischen Abständen prüfen, wie sehr die Platten mit den verschiedenen Heimatverzeichnissen ausgelastet sind; das Testskript soll Ihnen eine E-Mail-Nachricht schicken, wenn mindestens eine Platte zu mehr als 95% belegt ist. (Wir ignorieren hier die Existenz von LVM.)

2.5 Fehlertypen

Grundsätzlich können Sie zwei verschiedene Fehlertypen unterscheiden:

Konzeptuelle Fehler Hierbei handelt es sich um Fehler im logischen Ablauf des Programms. Es kann sehr aufwendig sein, solche Fehler zu erkennen und zu beheben. Sie sollten am Besten von vornherein vermieden werden – durch sorgfältige Planung.

Syntaktische Fehler Solche Fehler treten im Prinzip immer auf. Es reicht schon, einfache Tippfehler ins Programm einzubauen: Ein Zeichen vergessen und nichts läuft mehr. Viele syntaktische Fehler können Sie umschiffen, indem Sie sich beim Verfassen des Skriptes vom Einfachen zum Speziellen vortasten. So sollten Sie bei Klammersausdrücken erst *beide* Klammern eingeben und diese dann mit Inhalt füllen. Eine vergessene Klammer ist dann Schnee von gestern. Das gleiche gilt für Kontrollelemente: Schreiben Sie nie ein `if`, ohne direkt das `fi` eine Zeile dahinter zu setzen. Ein guter Editor, der Syntaxhervorhebung beherrscht, ist ein wichtiges Hilfsmittel zur frühzeitigen Vermeidung von solchen »strukturellen« Syntaxfehlern.

Natürlich können Sie sich immer noch bei Programmnamen und -optionen oder bei Verweisen auf Shellvariable vertippen. (Dabei hilft der Editor Ihnen auch nicht.) Die Shell hat hier einen eindeutigen Nachteil gegenüber »traditionellen« Programmiersprachen mit fester Syntax, die pingelig von einem Sprachübersetzer geprüft und gegebenenfalls angemockert wird. Da dieser Übersetzungsschritt und die damit verbundenen Prüfungen Ihres Programms entfallen, müssen Sie besonderes Augenmerk darauf legen, Ihr Skript systematisch zu testen, damit möglichst alle Skriptzeilen, Zweige von Fallunterscheidungen usw. tatsächlich durchlaufen werden. Dieser Umstand legt den Schluss nahe, dass Shellskripte jenseits einer bestimmten »kritischen Masse« von einigen hundert bis tausend Zeilen nicht mehr wirklich beherrschbar sind – hier sollten Sie dann doch zumindest zu einer Skriptsprache mit besserer Syntaxprüfung, etwa Perl oder Python, greifen

Der Rest des Kapitels beschäftigt sich hauptsächlich mit der Diskussion der häufigsten syntaktischen Fehler und deren Behebung.

2.6 Fehlererkennung

Fehler erkennen Sie, wie erwähnt, erst, wenn das Programm abläuft. Darum sollten Sie Ihre Skripte schon während der Entwicklung so häufig wie möglich testen. Testdurchlauf Speziell wenn das Skript bestehende Dateien verändert, sollten Sie dabei auf eine »Testumgebung« zurückgreifen.



Wenn Sie zum Beispiel automatisch Konfigurationsdateien in `/etc` editieren wollen, dann schreiben Sie Ihr Skript so, dass alle Verweise auf Dateien `/etc/...als $ROOT/etc/...` geschrieben werden. Zum Testen können Sie dann die Umgebungsvariable `ROOT` auf einen unverfänglichen Wert setzen und kommen dann vielleicht (hoffentlich!) sogar ohne Administratorrechte aus. Wenn Ihr Skript »myscript« heißt, können Sie es zum Testen aus einem »Testgerüst« aufrufen, das ungefähr so aussieht:

```
#!/bin/sh
# test-myscript -- Testskript für myscript
#
# Stelle die Testumgebung her
cd $HOME/myscript-dev
rm -rf test
cp -a files test # Frische Kopie der Testdateien
# Rufe das Skript auf
ROOT=$HOME/myscript-dev/test $HOME/bin/myscript-dev "$@"
```

Dabei stehen in `~/myscript-dev/files` die ursprünglichen Dateien, die vor jedem Testlauf nach `~/myscript-dev/test` kopiert werden. Nach dem Programmablauf könnten Sie den Inhalt von `test` auch automatisch (etwa mit `»diff -r«`) mit einem weiteren Verzeichnis vergleichen, das Muster für die korrekte Ausgabe enthält. `myscript` bekommt die Argumente von `test-myscript` übergeben; Sie könnten `test-myscript` also als Baustein in einer noch aufwendigeren Infrastruktur verwenden, die `myscript` mit verschiedenen vorgekochten Kommandoargumenten aufruft und jeweils automatisch testet, dass das Programm die erwarteten Ausgaben liefert.

Viele Shellskripte rufen externe Kommandos auf. In solchen Fällen können Sie bei der Fehlererkennung auf die eingebauten Fehlermeldungen des entsprechenden Programmes zurückgreifen – sofern es sich um syntaktische Fehler handelt.

Sollten die beschriebenen Fehlermeldungen nicht ausreichen, so können Sie unter anderem die Bash wesentlich geschwätziger machen. Häufig handelt es sich bei Syntaxfehlern nämlich um Fehler, die die Syntax der Shell betreffen – beispielsweise, wenn Substitutionen in einer nicht vorgesehenen Reihenfolge durchgeführt werden oder wenn Klammerfehler auftauchen.

Mit `»set -x«` können Sie sehen, was die Bash für Schritte unternimmt:

```
$ set -x
$ echo "My $HOME is my castle"
+ echo 'My /home/tux is my castle'
My /home/tux is my castle
```

Ablaufverfolgung Das Ganze nennt sich auch Ablaufverfolgung (*tracing*).

Der Nachteil an `»set -x«` ist, dass der Befehl trotzdem ausgeführt wird. Im Falle von Substitutionen kann es besser sein, wenn die Kommandos nur so angezeigt werden, wie sie ausgeführt worden wären. Dazu steht die Option `»-n«` zur Verfügung. Allerdings funktioniert diese Option nur in Shellskripten, nicht aber in interaktiven Shells (warum wohl?).

Eine weitere nützliche Option ist `»-v«`. Hiermit wird das Kommando zwar ausgeführt, zusätzlich zeigt die Shell aber alle Kommandos an. Das heißt für ein Shellskript, Sie bekommen nicht nur seine Ausgaben, sondern auch das, was drin steht.

Mit der Option `»-u«` wird ein Fehler gemeldet, wenn Sie auf den Wert einer Variable zugreifen wollen und diese Variable gerade undefiniert ist. Auch das kann dabei helfen, obskure Probleme aufzudecken.

Alle vier Optionen lassen sich wieder abschalten, wenn Sie im `set`-Kommando statt des `»-«` bei den Optionen ein `»+«` setzen.

In der Praxis der Shellprogrammierung funktioniert das Ganze auch anders. Während der Entwicklung des Skripts schreiben Sie einfach die entsprechende Option in die erste Zeile des Skripts:

```
#!/bin/bash -x
<<<<<<
```

Als weitere Regel zur Fehlererkennung gilt: Bevor Sie ein Kommando, das »zweifelhafte« Substitutionen enthält, ausführen lassen, setzen Sie erstmal ein `echo`. Dadurch wird das Kommando komplett (das heißt, inklusive aller Substitutionen und Expansionen) ausgegeben, ohne ausgeführt zu werden. Für den schnellen Test reicht das allemal.

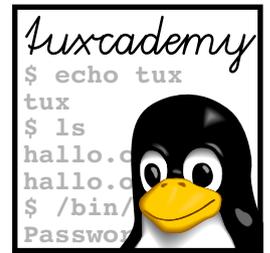
Kommandos in diesem Kapitel

chmod Setzt Rechte für Dateien und Verzeichnisse

chmod(1) 24

Zusammenfassung

- Shellskripte bieten eine einfache Möglichkeit zur Automatisierung von Kommandofolgen.
- Sie können Shellskripte entweder explizit einer Shell als Dateiargument übergeben, sie ausführbar machen und direkt starten oder über `source` unmittelbar in die aktuelle Shell einlesen.
- Shellskripte sind Textdateien, die Folgen von Shellkommandos enthalten.
- Die erste Zeile eines ausführbaren Skripts kann ein Programm benennen (außer der Shell auch andere), das das Skript ausführen soll.
- Sorgfältige Planung beim Programmieren reduziert späteres Kopfzerbrechen.
- Die Bash verfügt über verschiedene Eigenschaften zur Fehlersuche.



3

Die Shell als Programmiersprache

Inhalt

3.1	Variable	34
3.2	Arithmetische Ausdrücke	40
3.3	Bearbeitung von Kommandos	41
3.4	Kontrollstrukturen	42
3.4.1	Überblick.	42
3.4.2	Der Rückgabewert von Programmen als Steuergröße	42
3.4.3	Alternativen, Bedingungen und Fallunterscheidungen	44
3.4.4	Schleifen	49
3.4.5	Schleifenunterbrechung	51
3.5	Shellfunktionen	54
3.6	Das Kommando exec.	55

Lernziele

- Die Programmiersprachen-Eigenschaften der Shell (Variable, Kontrollstrukturen, Funktionen) kennen
- Einfache Shellskripte erstellen können, die diese Eigenschaften ausnutzen

Vorkenntnisse

- Vertrautheit mit der Kommandooberfläche von Linux
- Umgang mit Dateien und einem Texteditor
- Shell-Vorkenntnisse (etwa aus Kapitel 1 und Kapitel 2)

3.1 Variable

Umgebungsvariable

Grundlagen Variable können in Zusammenhang mit der Shell unterschiedliche Rollen spielen. Einerseits sind sie auf Shell-Ebene Hilfsmittel zum Beispiel bei der Programmierung von Shellskripten, andererseits wirken sich bestimmte Variable auf das Verhalten der Shell und (als **Umgebungsvariable**) ihrer Kindprozesse aus.

Variable in der Shell sind Paare aus einem Namen und einem (textuellen) Wert. Der Variablenname besteht aus einer Folge von Buchstaben, Ziffern und Unterstrichungen (»_«) und muss mit einem Buchstaben oder einer Unterstrichung anfangen. Variablennamen dürfen in der Bash für alle praktischen Zwecke beliebig lang sein. Der Wert einer Variablen ist eine beliebige Zeichenkette ebenfalls nahezu beliebiger Länge.



Moderne Shells wie die Bash kennen auch Variable mit numerischen Werten und Felder (*arrays*). Hierauf kommen wir später zurück.

In der Shell müssen Variable nicht deklariert werden wie in anderen Programmiersprachen; Sie können einer Variablen einfach einen Wert zuweisen, und ab diesem Zeitpunkt existiert sie dann, falls es sie vorher noch nicht gab:

```
$ farbe=blau
$ a123=?_/@xz
```

Achten Sie darauf, dass rund um das Gleichheitszeichen »=« keine Leerzeichen stehen dürfen! Variablenname, »=« und Wert müssen unmittelbar aufeinander folgen.

Variablensubstitution
echo

Variablensubstitution Bei der Bearbeitung von Kommandos ersetzt die Shell Bezüge auf Variable durch den jeweiligen Variablenwert. Auf eine Variable können Sie sich beziehen, indem Sie ein »\$« vor ihren Namen setzen. (Die Shell betrachtet die längstmögliche Folge von Buchstaben, Ziffern und Unterstrichungen nach dem »\$« als Variablennamen.) Wir sprechen auch von **Variablensubstitution**.

Sie können sich so zum Beispiel mit echo Variablenwerte anzeigen lassen:

```
$ echo $PAGER
less
```

zeigt Ihnen den Wert von PAGER an, einer Variablen, die bestimmt, welches Anzeigeprogramm u. a. das man-Kommando benutzen soll.



Beachten Sie, dass die Shell sich um die Expansion des Variablenbezugs \$PAGER kümmert – letztendlich ausgeführt wird nur »echo less«.

set



Alle gerade gesetzten Variablen (und auch Shell-Funktionen – siehe später) zeigt Ihnen das Bash-interne Kommando set (ohne Optionen und Argumente aufgerufen).

Variable und Anführungszeichen Variablensubstitution ist nützlich, aber nicht immer wünschenswert. Sie können Variablensubstitution auf zwei Arten verhindern:

1. Sie setzen vor das Dollarzeichen einen Rückstrich:

```
$ echo \farbe hat den Wert $farbe
farbe hat den Wert blau
```

2. Sie setzen den Variablenbezug komplett in einfache Anführungszeichen:

```
$ echo '$farbe' hat den Wert $farbe
farbe hat den Wert blau
```

Innerhalb von doppelten Anführungszeichen ("...") und verkehrten Anführungszeichen (`...`) findet Variablensubstitution statt!

Doppelte Anführungszeichen sind wichtig im Umgang mit Variablen, wenn deren Werte Leerzeichen enthalten können (also meistens bis immer). Betrachten Sie das folgende Beispiel:

```
$ mkdir "Meine Fotos"
$ d="Meine Fotos"
$ cd $d
bash: cd: Meine: Datei oder Verzeichnis nicht gefunden
```

In der dritten Zeile wird der Wert der Variablen `d` substituiert und erst dann die Kommandozeile in »Wörter« aufgeteilt. Entsprechend versucht `cd`, ins Verzeichnis »Meine« (statt »Meine Fotos«) zu wechseln. *Moral:* Setzen Sie doppelte Anführungszeichen, wo Sie nur können – »`cd "$d"`« wäre richtig gewesen.

Umgebungsvariable »Normale« Variable sind nur innerhalb der Shell sichtbar, in der sie definiert wurden. Damit Variable auch in Kindprozessen (»Sub-Shells« oder anderen aus der Shell gestarteten Programmen) zu sehen sind, müssen sie in die Prozessumgebung »exportiert« werden:

Prozessumgebung

```
$ farbe=blau
$ sh                               Starte eine Sub-Shell
$ echo $farbe                       In der Sub-Shell
                                     In der Sub-Shell ist die Variable nicht definiert
$ exit                               Zurück in die ursprüngliche Shell
$ export farbe                       Exportiere die Variable in die Umgebung
$ sh                                  Wieder eine Sub-Shell
$ echo $farbe                         In der Sub-Shell
blau
```

In einem `export`-Kommando können Sie mehrere Variable aufzählen:

```
$ export blau weiss rot
```

Außerdem können Sie Exportieren und (initiale) Zuweisung zusammen tun:

```
$ export form=oval geruch=muffig
```

Mit »`export -n`« wird das Exportieren rückgängig gemacht:

```
$ export -n blau weiss
```

Wenn die Shell einen Kindprozess startet, bekommt dieser eine Kopie der Shell-Umgebung zum Startzeitpunkt mit. Ab dann sind die beiden Umgebungen völlig unabhängig voneinander – Änderungen in der einen haben keine Auswirkung auf die andere. Insbesondere ist es (ähnlich wie beim aktuellen Verzeichnis) nicht möglich, aus dem Kindprozess heraus direkt die Umgebung des Elternprozesses zu verändern.



Das geht, wenn überhaupt, nur mit Tricks analog zu Übung 2.4 – ein Programm, das diese Methode benutzt, ist der `ssh-agent`.

Sie können Umgebungsvariable auch für einen einzelnen Kommandoaufruf setzen, ohne dass eine gleichnamige Shellvariable davon beeinflusst wird:

```
$ TZ=bla
$ TZ=Europe/Istanbul date           Türkische Zeit
Mon May 10 19:23:38 EEST 2004
$ echo $TZ
bla
```

Eine aktuelle Liste aller Umgebungsvariablen erhalten Sie mit den Kommandos `export` (ohne Argumente) oder `env`. Einige wichtige Umgebungsvariable sind

PATH Suchpfad mit den Verzeichnissen, in denen die Shell nach den auszuführenden Kommandos sucht

HOME Heimatverzeichnis; ist in der Shell das Verzeichnis, durch das »~« substituiert wird

TERM Typ des aktuellen Terminals (wichtig für bildschirm- bzw. fensterfüllende textorientierte Programme)

USER Aktueller Benutzername

UID Aktuelle Benutzer-ID; kann nicht verändert werden!

Wertzweisung: Tipps und Tricks Bei der Wertzweisung an Variable sind natürlich auch kompliziertere Ausdrücke möglich, wie in

```
$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:.
$ PATH=$PATH:$HOME/bin
$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:./home/tux/bin
```

Hier wird der Suchpfad für Programme in `PATH` neu aufgebaut aus dem alten Wert von `PATH`, einem »:« und dem Wert von `HOME` gefolgt von »/bin«.

Sie können mit »verkehrten Anführungszeichen«¹ sogar die Standardausgabe eines Programm-Aufrufs einer Variablen zuweisen, beispielsweise

```
$ dir=`pwd`
$ echo $dir
/home/tux
```

Das ist selbstverständlich nur eine Konsequenz der Tatsache, dass die Shell »...«-Ausdrücke *irgendwo* auf der Kommandozeile durch die Standardausgabe des betreffenden Kommandos ersetzt. Das »Kommando« darf übrigens eine komplette Pipeline sein, nicht nur ein einzelnes Kommando.



Dasselbe Ergebnis lässt sich mit »\$(...)« erreichen (»dir=\$(pwd)«). Diese Konstruktion funktioniert in moderneren Shells wie der Bash und macht es einfacher, solche Aufrufe zu verschachteln – aber macht Ihr Skript auch Bash-abhängig.

Maskierung von Sonderzeichen

Wollen Sie Leerzeichen oder andere Sonderzeichen in einer Variablen speichern, so müssen Sie diese durch einfache Anführungszeichen ('...') vor der Shell schützen, wie in

```
$ PS1='% '
% _
```

Dies verändert die Variable `PS1`, die das Aussehen Ihrer Eingabeaufforderung steuert.

Haben Sie von einer Variablen genug, so können Sie ihren *Inhalt* durch die Zuweisung der leeren Zeichenkette löschen. Noch drastischer gehen Sie mit dem Kommando `unset` vor: es entfernt die Variablen komplett und verwischt alle Spuren Ihrer Tat:

¹Auf einer deutschen Tastatur erzeugen Sie dieses Zeichen durch die Umschalttaste plus die Taste rechts von .

```

$ A='Mord im Dom'
$ set | grep A=
A='Mord im Dom'
$ A=''
$ set | grep A=
A=
$ unset A
$ set | grep A=
$ _

```

Spezielle Shellvariable Die bash kennt einige besondere Shellvariable, die vor allem innerhalb von Shell-Skripten von Interesse sind. Diese speziellen Variablen können nicht verändert, sondern nur gelesen werden. Zum Beispiel:²

- \$? Rückgabewert des letzten Kommandos
- \$\$ Prozessnummer (PID) der aktuellen Shell
- #! PID des zuletzt gestarteten Hintergrundprozesses
- \$0 Name des gerade ausgeführten Shell-Skripts (oder der Shell selbst, wenn sie interaktiv ist)
- ## Anzahl der dem Shell-Skript auf der Kommandozeile übergebenen Parameter
- * Gesamtheit aller übergebenen Parameter; »"\$*"« ist gleichbedeutend mit »"\$1 \$2 \$3 ..."«
- @ Gesamtheit aller übergebenen Parameter; »"\$@"« ist gleichbedeutend mit »"\$1" "\$2" "\$3" ..."«
- n n-ter Parameter des Shellskripts. Für $n > 9$ müssen Sie »\${n}« schreiben. Alternativ können Sie auch mit dem Kommando `shift` arbeiten (siehe dazu `help shift`).

Besonders die Variablen für den Parameterzugriff werden in Shellskripten häufig verwendet. Hier ein kleines Beispiel zur Verdeutlichung:

```

$ cat skript
#!/bin/sh
echo "Das Shellskript wurde mit \"$0\" aufgerufen"
echo "Insgesamt wurden $# Parameter übergeben"
echo "Alle Parameter zusammen: \"$*\" "
echo "Der erste Parameter ist \"$1\" "
echo "Der zweite Parameter ist \"$2\" "
echo "Der dritte Parameter ist \"$3\" "
$ ./skript Eat my shorts
Das Shellskript wurde mit "./skript" aufgerufen
Insgesamt wurden 3 Parameter übergeben
Alle Parameter zusammen: "Eat my shorts"
Der erste Parameter ist "Eat"
Der zweite Parameter ist "my"
Der dritte Parameter ist "shorts"

```

Insbesondere beim Aufruf der Variablen für die Positionsparameter ist es wichtig, Variablenbezüge in Anführungszeichen zu setzen, da Sie es als Programmierer nicht in der Hand haben, welche Parameter der aufrufende Benutzer übergibt. Eingestreute Leerzeichen können sonst den Skriptablauf gründlich durcheinander bringen.

²Hier und auch sonst werden wir von »der Variablen \$?« usw. reden, obgleich dies nicht ganz korrekt ist, denn »\$?« ist eigentlich der Wert der Variablen »?«. Da die speziellen Variablen ohne Tricks nicht geschrieben, sondern nur gelesen werden können, ist diese Ungenauigkeit unproblematisch.

Spezialformen der Variablensubstitution Bei einer Substitution über »\$name« oder »\${name}« wird die Variable einfach durch ihren Wert ersetzt. Die Shell kann allerdings einiges mehr:

Standardwert zuweisen Mit `${name}=(Standardwert)` bekommt die Variable `name` den `Standardwert` zugewiesen, wenn sie undefiniert oder ihr Wert die leere Zeichenkette ist. Anschließend wird der Wert der Variable in das aktuelle Kommando substituiert.

```
$ unset farbe                                     Variable ist undefiniert
$ echo Lieblingsfarbe: ${farbe:=gelb}
Lieblingsfarbe: gelb                               Standardwert tritt in Kraft
$ echo $farbe
gelb                                               Wurde im vorigen Kommando zugewiesen
$ farbe=rot                                       Variable ist definiert
$ echo Lieblingsfarbe: ${farbe:=gelb}
Lieblingsfarbe: rot                               Existierender Wert hat Vorrang
```

Sie können den Doppelpunkt auch weglassen (`>${farbe=gelb}<`). In diesem Fall erfolgt die Zuweisung nur, wenn die Variable `name` undefiniert ist; ein vorhandener, aber leerer Wert bleibt erhalten.

Standardwert benutzen Der Ausdruck `${name}-(Standardwert)` wird durch den Wert von `name` ersetzt, falls `name` einen von der leeren Zeichenkette verschiedenen Wert hat. Sonst wird er durch den `Standardwert` ersetzt. Der Unterschied zu `:=` ist, dass der tatsächliche Wert von `name` nicht verändert wird:

```
$ unset farbe
$ echo Lieblingsfarbe: ${farbe:-gelb}
Lieblingsfarbe: gelb
$ echo $farbe
Ausgabe: Leere Zeichenkette
```

Auch hier kann der Doppelpunkt wegfallen.

Fehlermeldung, wenn kein Wert Über `${name}?(Fehlermeldung)` wird geprüft, ob die Variable `name` einen Wert hat, der von der leeren Zeichenkette verschieden ist. Ist dies nicht der Fall – insbesondere wenn die Variable undefiniert ist –, wird die `Fehlermeldung` ausgegeben. Passiert das in einem Shellskript, endet dessen Ausführung an dieser Stelle.

```
$ cat skript
#!/bin/sh
echo "${1:?Da fehlt wohl was}"
echo "Und weiter geht's"
$ ./skript blah
blah
Und weiter geht's
$ ./skript
./skript: line 2: 1: Da fehlt wohl was
```

Der Doppelpunkt kann hier ebenfalls entfallen – die Fehlermeldung erscheint dann nur, wenn die Variable wirklich undefiniert ist.

Teilzeichenketten Ein Ausdruck der Form `${name:p:l}` wird durch bis zu `l` Zeichen des Werts der Variablen `name` ersetzt, beginnend ab Position `p`. Fehlt `l`, wird alles ab Position `p` zurückgegeben. Das erste Zeichen des Werts ist an Position 0:

```
$ abc=ABCDEFGH
$ echo ${abc:3:2}
DE
$ echo ${abc:3}
DEFG
```

Tatsächlich werden p und l als *arithmetische Ausdrücke* ausgewertet (Abschnitt 3.2). p darf negativ sein und zählt dann von hinten:

```
$ echo ${abc:2*2-1:5-2}
DEF
$ echo ${abc:0-2:2}
FG
```

Als spezielle Sonderfälle werden die Variable $*$ und $@$ betrachtet. Hier bekommen Sie l Positionsparameter des Skripts zurück, beginnend beim Parameter p , wobei (verwirrenderweise, aber doch irgendwie logisch, wenn man sich überlegt, dass $*$ im Grunde dasselbe ist wie »\$1 \$2 ...«) der erste Parameter an Position 1 ist:

```
$ set hund katze maus baum
$ echo ${*:2:2}
katze maus
```

Entfernen von variablem Text am Anfang des Werts In einem Ausdruck der Form $\${name#\langle Muster \rangle}$ wird $\langle Muster \rangle$ als Suchmuster interpretiert (mit »*«, »?« usw. wie bei Dateinamen). Der Ausdruck liefert den Wert der Variablen $name$, wobei an dessen Anfang alles entfernt wurde, worauf $\langle Muster \rangle$ passt:

```
$ vorseise=AALSUPPE
$ echo "${vorseise}"
AALSUPPE
$ echo "${vorseise#A}"
ALSUPPE
$ echo "${vorseise#A*L}"
SUPPE
```

Wenn es mehrere Möglichkeiten gibt, dann wird mit »#« immer so wenig Text wie möglich entfernt. Wenn Sie statt dessen »##« setzen, wird so viel Text wie möglich entfernt:

```
$ oldmacd=EIEIO
$ echo ${oldmacd#E*I}
EIO
$ echo ${oldmacd##E*I}
0
```

Die typische Anwendung ist das Entfernen des Pfadanteils von einem Dateinamen:

```
$ file=/var/log/apache/access.log
$ echo ${file##*/}
access.log
```



Hierfür könnten Sie natürlich auch das `basename`-Kommando verwenden, aber das bedingt einen externen Programmaufruf – der »##«-Ausdruck ist tendenziell viel effizienter.

Entfernen von variablem Text am Ende des Werts Ausdrücke der Form `${name%<Muster>}` und `${name%%<Muster>}` funktionieren genauso, aber wirken auf das Ende des Werts von `name` und nicht den Anfang:

```
$ kasper=TRULLALA
$ echo ${kasper%L*A}
TRULLA
$ echo ${kasper%%L*A}
TRU
```

Die Bash bietet auch noch einige weitere Substitutionsmöglichkeiten an, die Sie am besten dem Originalhandbuch entnehmen.

Solche Textverarbeitungstricks sind sehr nützlich, aber für weitergehende Operationen müssen Sie meist auf Werkzeuge wie `cut`, `sed` (Kapitel 6) und `awk` (Kapitel 7) zurückgreifen. Das ist natürlich mit der Ineffizienz eines weiteren Kindprozesses verbunden, womit man im Einzelfall sicher leben kann; für Textverarbeitung in großem Stil sind Sie aber mit einer Programmiersprache wie Perl, Python oder Tcl besser beraten, die wesentlich ausgefeiltere Operationen anbietet und diese auch effizienter verpackt.

Übungen



3.1 [!2] Was genau ist der Unterschied zwischen `$*` und `$@`? (Hinweis: Lesen Sie im Bash-Handbuch nach und vergleichen Sie die Ausgabe der Kommandos)

```
$ set a "b c" d
$ for i in $*; do echo $i; done
$ for i in $@; do echo $i; done
$ for i in "$*"; do echo $i; done
$ for i in "$@"; do echo $i; done
```

Siehe Abschnitt 3.4.4 für Informationen über `for`.)

3.2 Arithmetische Ausdrücke

Die Bash hat gewisse Rechenfähigkeiten, auch wenn Sie diese nicht überstrapazieren sollten: Sie kann zum Beispiel nur mit ganzen Zahlen umgehen und prüft nicht auf Überlauf. Die Rechenoperatoren entsprechen denen der Programmiersprache C, Sie können also zum Beispiel über die vier Grundrechenarten, die gängigen Vergleichs- und logischen Operatoren verfügen (Details siehe Bash-Dokumentation). Punktrechnung geht vor Strichrechnung und explizite Klammern werden immer zuerst betrachtet.

Arithmetische Expansion »Arithmetische Expansion« wird auf Ausdrücke angewendet, die in `$((...))` geklammert sind. Als Operanden sind neben Zahlen auch Bash-Variable zulässig. Der Ausdruck in den Klammern wird so behandelt, als stünde er in doppelten Anführungszeichen; Sie können also auch auf Kommandoexpansion und die »Spezialformen der Variablensubstitution« aus dem vorigen Abschnitt zurückgreifen.

```
$ echo $((1+2*3))
7
$ echo $(((1+2)*3))
9
$ a=123
$ echo $((3+4*a))
495
```

Auf Shellvariable können Sie in arithmetischen Ausdrücken Bezug nehmen, ohne dass Sie ein »\$« davorsetzen müssen.



Manchmal werden Sie auch noch die alte Schreibweise »\$[...]« antreffen. Diese ist inzwischen verpönt und wird aus künftigen Bash-Versionen entfernt werden.

Übungen



3.2 [!1] Wie geht die Bash mit Division um? Prüfen Sie das Ergebnis verschiedener Divisionsaufgaben mit positiven und negativen Dividenden und Divisoren. Können Sie Regeln aufstellen?



3.3 [2] (Beim zweiten Durcharbeiten.) Was ist die größte Zahl, die in Bash-Arithmetik benutzt werden kann? Wie können Sie das herausfinden?

3.3 Bearbeitung von Kommandos

Die Shell hält bei der Bearbeitung von Kommandos eine genau definierte Reihenfolge von Schritten ein:

1. Die Kommandozeile wird in **Wörter** aufgeteilt (siehe auch unten). Dabei gilt jedes Zeichen in IFS, das außerhalb von Anführungszeichen steht, als Trennzeichen. Der Standardwert von IFS besteht aus dem Leerzeichen, dem Tabulator und dem Zeilentrenner. Wörter
2. Dann werden Mengenklammern expandiert, aus »a{b,c}d« wird »abd acd«. Alle anderen Zeichen bleiben unberührt und werden gegebenenfalls ins Ergebnis übernommen.
3. Als nächstes wird eine Tilde (~) am Anfang eines Worts durch den Wert der Variablen HOME ersetzt; wenn der Tilde unmittelbar ein Benutzername folgt, wird dieser (mitsamt der Tilde) durch den Namen des Heimatverzeichnis des betreffenden Benutzers ersetzt. (Es gibt noch einige weitere Regeln, die in der Bash-Dokumentation beschrieben sind.)



Tilden-Expansion findet auch in Variablenzuweisungen statt, wenn eine Tilde unmittelbar hinter einem »=« oder »:« steht. Das heißt, dass auch für PATH & Co. das Richtige passiert.

4. Anschließend werden die folgenden Ersetzungen parallel von links nach rechts vorgenommen:
 - Variablensubstitution
 - Kommandosubstitution
 - Arithmetische Expansion

(anders gesagt, alles, was mit \$ anfängt – wenn wir mal der modernen Form der Kommandosubstitution mit »\$(...)« den Vorrang geben).

5. Falls in den vorhergehenden Schritten eine Ersetzung stattfand, wird die Kommandozeile nochmals anhand der Variablen IFS in Wörter aufgeteilt. »Explizite« leere Wörter (" und ') bleiben erhalten, »implizite« leere Wörter, die zum Beispiel durch die Expansion von Variablen ohne Wert entstehen, werden entfernt.
6. Am Ende der Bearbeitung werden Wörter, die Suchmusterzeichen wie »*« oder »?« enthalten, als Suchmuster interpretiert; die Shell versucht, sie durch die passenden Dateinamen zu ersetzen. Wenn das nicht möglich ist, bleibt das Suchmuster stehen (normalerweise). Suchmuster

Wert	Bedeutung
0	Erfolg
1	Allgemeine Fehler
2	Missbrauch eingebauter Shell-Funktionen (selten benutzt)
126	Aufgerufenes Kommando war nicht ausführbar (kein Ausführungsrecht oder kein Binärprogramm)
127	Aufgerufenes Kommando wurde nicht gefunden
128	Ungültiges Argument bei <code>exit</code> – etwa » <code>exit 1.5</code> «
129–165	Programmabbruch durch Signal $\langle \text{Wert} \rangle$ – 128

Tabelle 3.1: Reservierte Rückgabewerte bei der Bash

- Zum Schluss werden alle nicht maskierten Anführungszeichen und Rückstriche entfernt (sie werden nicht mehr gebraucht, da alle Ersetzungen getan sind und die Wortaufteilung sich nicht mehr ändert).

Die einzigen Bearbeitungsschritte, die die Anzahl der Wörter auf der Kommandozeile ändern können, sind die Mengenklammern-Expansion, die Wortaufteilung und die Suchmusterexpansion. Alle anderen Bearbeitungsschritte ersetzen ein einziges Wort durch ein einziges Wort, mit der Ausnahme der Expansion von »"\$@"«.



Ganz am Anfang (vor unserem Schritt 1 oben) findet noch ein weiterer Schritt statt: Kommandos können auch »zusammengesetzt« sein, etwa ist es möglich, mehrere Kommandos durch »;« getrennt auf dieselbe (logische) Zeile zu schreiben. Die Aufteilung solcher »Monsterkommandos« in einzelne in sich abgeschlossene »Shell-Kommandozeilen« wird vor der Wortaufteilung vorgenommen.

3.4 Kontrollstrukturen

3.4.1 Überblick

Jede ernstzunehmende (d. h. Turing-vollständige) Programmiersprache braucht Kontrollstrukturen wie Verzweigungen, Schleifen und Alternativen³, und darum dürfen diese auch in der Shell nicht fehlen. Wie üblich bei der Shellprogrammierung ist alles etwas »hintenrum« und für Kenner »echter« Programmiersprachen möglicherweise verwirrend, aber folgt irgendwie auch seiner eigenen perversen Logik.



Stephen L. Bourne, der Autor der ursprünglichen Bourne-Shell, war ein großer Anhänger der Programmiersprache Algol [WMPK69], und das macht sich auch in der Bourne-Shell-Syntax (und damit der ihrer Epigonen) bemerkbar. Die Idee, am Ende einer Kontrollstruktur das Anfangs-Schlüsselwort rückwärts hinzuschreiben – bei der Bourne-Shell nicht konsequent verwirklicht –, war zum Beispiel in Algol-Kreisen *en vogue*.

3.4.2 Der Rückgabewert von Programmen als Steuergröße

Hier die erste Merkwürdigkeit der Shell-Kontrollstrukturen: In vielen Programmiersprachen wird als Steuergröße für Verzweigungen oder Schleifen ein logischer Wert (»wahr« oder »falsch«) genommen. Nicht so in der Bash – hier dient als Steuergröße der **Rückgabewert** eines Programms.

³Man kann auch nur mit der `while`-Schleife auskommen, zumindest wenn man Edsger Dijkstra heißt, aber ein bisschen mehr Auswahl schadet hier nicht.

Unter Linux teilt jeder Prozess am Ende seiner Ausführung seinem Elternprozess mit, ob er »erfolgreich« ausgeführt wurde oder nicht. Der Rückgabewert in der Bash ist eine ganze Zahl zwischen 0 und 255; der Rückgabewert 0 bedeutet immer Erfolg, jeder andere Wert (bis einschließlich 255) steht für Misserfolg. Dadurch kann ein Prozess genauer mitteilen, was schiefgelaufen ist.



Wie er das im Detail macht, ist nicht festgelegt. Bei `grep` zum Beispiel bedeutet der Rückgabewert 0 soviel wie »es wurden passende Zeilen gefunden«, 1 steht für »es wurden keine passenden Zeilen gefunden, aber sonst war alles OK« und 2 für »ein Fehler ist aufgetreten«.



Man könnte jetzt eine lange ontologische Diskussion darüber anfangen, ob »es wurden keine passenden Zeilen gefunden« eine Fehlersituation ist oder eigentlich auch irgendwie eine Art von Erfolg. Fakt ist, dass es nützlich ist, die Fälle »Zeilen wurden gefunden« und »Zeilen wurden nicht gefunden« am Rückgabewert von `grep` auseinanderhalten zu können. Fakt ist ebenfalls, dass Unix nur eine Sorte Erfolg kennt, nämlich den Rückgabewert 0. Wem das nicht passt, der muss sich ein anderes Betriebssystem suchen (etwa VMS, das jeden geraden Rückgabewert (inklusive 0) als Misserfolg und jeden ungeraden als Erfolg ansieht).



C-Programmierer, die gewöhnt sind, dass 0 für »falsch« und »nein« und »alles andere« für »wahr«, »ja«, »Erfolg« steht, müssen an dieser Stelle umdenken.



Es gibt zumindest für die Bash gewisse Konventionen über Rückgabewerte (siehe Tabelle 3.1). Der Systemaufruf `exit()` akzeptiert zwar Werte bis einschließlich 255 (größere Werte werden »modulo 255« weitergegeben), aber die Bash verwendet Rückgabewerte über 128, um zu signalisieren, dass der Kindprozess durch ein Signal beendet wurde. (Welches Signal das genau war, können Sie herausfinden, indem Sie 128 vom Rückgabewert subtrahieren; die Signalnummern erhalten Sie zum Beispiel durch »`kill -l`«.)



Sie können in Ihren Skripten natürlich mit `exit` Rückgabewerte über 128 produzieren, aber das ist im Allgemeinen keine gute Idee, da es Verwirrung stiften könnte, wenn Ihr Skript von einem anderen aufgerufen wird, das diese Rückgabewerte als Prozessabbrüche qua Signal interpretiert.

In der Shell steht der Rückgabewert des letzten Kommandos in der speziellen Variablen `$?`:

```
$ ls /root
ls: /root: Permission denied
$ echo $?
1
$ echo $?
0
```

Misserfolg von ls

Erfolg des ersten echo

Der Rückgabewert hat nichts mit Fehlermeldungen zu tun, die auf der Standard-Fehlerausgabe erscheinen.



Ein wenig bekannter Trick ist, dass Sie vor ein Kommando ein Ausrufungszeichen (!) schreiben können. Der Rückgabewert des Kommandos wird dann »logisch umgekehrt« – aus Erfolg wird Misserfolg, aus Misserfolg Erfolg:

```
$ true; echo $?
0
$ ! true; echo $?
1
$ ! false; echo $?
0
```

Allerdings geht dabei Information verloren: Wie erwähnt kennt die Bash 255 Arten von Misserfolg, aber nur eine Art von Erfolg.



C-Programmierer oder Kenner C-ähnlicher Sprachen wie Perl, Tcl und PHP erinnern sich hier natürlich an den logischen Nicht-Operator »!<«.

Natürlich können Sie in der Bash wie in anderen Programmiersprachen auch auf logische Ausdrücke zurückgreifen, um Verzweigungen oder Schleifen zu steuern. Sie müssen dafür nur ein Kommando aufrufen, das die logischen Ausdrücke auswertet und einen passenden Rückgabewert liefert. Ein solches Kommando ist `test`.

`test` wird zum Vergleich von Zahlen oder Zeichenketten und zum Überprüfen von Dateieigenschaften verwendet. Hierzu einige Beispiele:

test "\$x" Bei nur einem Argument prüft `test`, ob dieses Argument *nichtleer* ist, also aus einem oder mehreren Zeichen besteht – hier also, ob die Variable `x` »etwas« enthält. Auch wenn Sie es immer wieder sehen: Verkneifen Sie sich das scheinbar kürzere »`test $x`« (ohne Anführungszeichen) – es funktioniert nicht, wie Sie leicht mit »`x='3 -gt 7'`; `test $x`« durchspielen können.

test \$x -gt 7 Hierbei muss `x` eine Zahl darstellen. `test` überprüft dann, ob der *numerische* Wert von `x` größer als 7 ist. `-gt` steht für "*greater than*"; es gibt auch die analogen Operatoren `-lt` (kleiner als), `-ge` (größer oder gleich), `-le`, `-eq` (gleich) und `-ne` (ungleich).

lexikographische Ordnung

test "\$x" \> 10 Testet, ob die erste Zeichenkette *in einem Wörterbuch* hinter der zweiten stünde (sog. **lexikographische Ordnung**). Somit meldet `test 7 \> 10` Erfolg; `test 7 -gt 10` hingegen Misserfolg. Beachten Sie die Schreibweise: »><« ist ein Sonderzeichen der Shell; damit es von der Shell nicht interpretiert wird, müssen Sie es vor ihr verstecken. Statt »><« können Sie auch »>'<« schreiben.

test -r "\$x" Prüft, ob die Datei, deren Name in `x` gespeichert ist, existiert und lesbar ist. Es gibt diverse weitere Dateitestoperatoren.

Die komplette Liste der von `test` unterstützten Operatoren können Sie der Dokumentation zu `test` entnehmen.



Ähnlich wie `echo` ist `test` nicht nur als Programm vorhanden, sondern wird aus Effizienzgründen auch von der Bash als internes Kommando angeboten (die traditionelle Bourne-Shell macht das nicht). Mit »`help test`« bekommen Sie die Dokumentation zum Bash-internen `test`, mit »`man test`« die des externen Kommandos.



Außer der beschriebenen Langform kennt `test` auch noch eine Kurzform, bei der die zu überprüfende Bedingung in eckige Klammern gesetzt wird (mit Leerzeichen innerhalb der Klammern). So heißt das Langform-Beispiel »`test "$x"`« in der Kurzform »`["$x"]`«.

3.4.3 Alternativen, Bedingungen und Fallunterscheidungen

Verzweigungen können Sie mit `if` und mit `case` realisieren. `if` ist eher für einfache Fallunterscheidungen und Ketten von Fallunterscheidungen gedacht, während `case` eine Mehrfachverzweigung anhand eines einzigen Werts ermöglicht.

Bedingte Ausführung Für die gängigen Fälle »Tue *A* nur, wenn *B* geklappt hat« oder »Tue *A* nur, wenn *B* nicht geklappt hat« bietet die Shell bequeme Kurzschreibweisen an. Ein sehr typisches Idiom in Shellskripten ist etwas wie

```
test -d "$HOME/.mydir" || mkdir "$HOME/.mydir"
```

Sie finden so etwas zum Beispiel in Skripten, die ein benutzerspezifisches Verzeichnis zum Ablegen von Zwischenergebnissen oder Konfigurationsdateien verwenden wollen. Die Kommandofolge stellt sicher, dass das Verzeichnis `$HOME/.mydir` existiert, indem es zunächst die Existenz des Verzeichnisses prüft. Meldet das `test`-Kommando Erfolg, dann wird das `mkdir` nicht mehr ausgeführt. Schlägt `test` jedoch fehl – das Verzeichnis existiert nicht –, dann wird das Verzeichnis mit `mkdir` angelegt. Der `||`-Operator sorgt also dafür, dass das folgende Kommando nur ausgeführt wird, wenn das Kommando davor Misserfolg signalisiert hat.



C-Programmierer kennen `||` als »logisches Oder«, dessen Ergebnis dann »wahr« ist, wenn die linke Seite *oder* die rechte Seite »wahr« ist (oder beide). Eine der geschickteren Eigenschaften von C ist, dass die Sprache garantiert, dass zuerst die linke Seite angeschaut wird. Ist diese schon »wahr«, dann steht das Gesamtergebnis fest, und die rechte Seite wird ignoriert. Die rechte Seite wird nur dann angeschaut, wenn die linke »falsch« ist. – Der `||`-Operator in der Shell funktioniert im Grunde genauso: In »`a || b`« wollen wir das Kommando *a* *oder* das Kommando *b* erfolgreich ausführen. Meldet *a* bereits Erfolg, sind wir am Ziel unserer Wünsche und können *b* ignorieren; *b* kommt nur zum Zug, wenn *a* nicht erfolgreich ausgeführt werden konnte.

In Analogie dazu führt der `&&`-Operator das folgende Kommando nur dann aus, wenn das davorstehende Kommando *Erfolg* signalisiert hat. Auch hierfür ein Beispiel aus dem wirklichen Leben: Etwas wie

```
test -e /etc/default/myprog && source /etc/default/myprog
```

finden Sie beispielsweise in den Init-Skripten vieler Distributionen. Hier wird geprüft, ob eine Datei `/etc/default/myprog` existiert, die (mutmaßlich) Voreinstellungen enthält, auf die später im Skript Bezug genommen werden soll (als Shell-Zuweisungen). Wenn diese Datei existiert, dann wird sie per `source` eingelesen, ansonsten passiert nichts.



Hier lässt sich eine ähnliche Analogie zum C-Operator `&&` (logisches Und) ziehen wie vorher bei `||`.

Die Kurzschreibweisen `||` und `&&` sind sehr nützlich und verhalten sich auch in Kombination wie erwartet (probieren Sie mal etwas wie

```
true && echo Wahr || echo Falsch
false && echo Wahr || echo Falsch
```

aus). Sie sollten sie jedoch nicht überstrapazieren – oftmals sind explizite `if`-Konstruktionen, wie gleich erklärt, letzten Endes klarer.

Einfache Verzweigung Das `if`-Kommando führt ein Kommando aus (oft `test`) `if` und entscheidet anhand dessen Rückgabewert über das weitere Vorgehen. Die Syntax ist:

```
if <Testkommando>
then
  <Kommandos bei »Erfolg«>
[else
  <Kommandos bei »Misserfolg«>]
fi
```

War das Test-Kommando erfolgreich, so werden die Kommandos, die `then` folgen, ausgeführt. War es nicht erfolgreich, so werden die Kommandos nach dem `else` ausgeführt (dieser Zweig ist optional). In beiden Fällen geht es danach hinter dem `fi` weiter.

Dies wird am besten anhand eines Beispiels klar. Als Demonstrationsprogramm schreiben wir das Programm `ifdemo`, dem Sie als ersten Positionsparameter Ihren Benutzernamen übergeben können. (Ihren »echten« Benutzernamen hat das `login`-Programm freundlicherweise in der Umgebungsvariablen `LOGNAME` hinterlegt.) Wird der Benutzername richtig eingegeben, dann wird eine Meldung der Form »Das ist korrekt« angezeigt. Wird ein anderer Wert eingegeben, kommt eine andere Meldung zurück:

```
#!/bin/bash
if test "$1" = "$LOGNAME"
then
    echo "Der übergebene Name ist tatsächlich Ihr Benutzername!"
else
    echo "Der übergebene Name ist nicht Ihr Benutzername!"
fi
echo "Programmende"
```

Natürlich sind Sie nicht gezwungen, `test` als Test-Kommando aufzurufen – jedes beliebige Programm, das die Konventionen über den Rückgabewert einhält, kommt in Frage. Zum Beispiel `egrep`:

```
#!/bin/bash
if df | egrep '(9[0-9]%|100%)' > /dev/null 2>&1
then
    echo "Eine Partition läuft über" | mail -s "Warnung" root
fi
```

Dieses Skript der Kategorie »quick and dirty« testet ob ein eingehängtes Dateisystem zu 90% oder mehr gefüllt ist. Wenn ja, bekommt `root` eine Mail, in der auf diesen Sachverhalt aufmerksam gemacht wird.

Um tiefe Verschachtelungen der Form

```
if bla
then
    ...
else
    if fasel
    then
        ...
    else
        if blubb
        then
            ...
        else
            ...
        fi
    fi
fi
```

zu vermeiden, ist es erlaubt, abhängige Fallunterscheidungen mit `elif` zu kaskadieren. Äquivalent zum vorstehenden Beispiel wäre etwa

```
if bla
then
    ...
elif fasel
then
    ...
elif blubb
```

```
then
  ...
else
  ...
fi
```

In jedem Fall bleibt der else-Zweig optional. Die jeweils mit if und den elifs geprüften Bedingungen müssen selber nichts miteinander zu tun haben, aber Bedingungen in »späteren« elifs werden natürlich nur dann angeschaut, wenn die vorherigen Bedingungen jeweils Werte ungleich 0 geliefert haben.

Mehrfachalternative Das Kommando case erlaubt im Gegensatz zu if auf einfache Art und Weise Verzweigungen mit mehreren Alternativen. Seine Syntax ist wie folgt:

```
case <Wert> in
  <Muster1>
    ...
    ;;
  <Muster2>
    ...
    ;;
...
esac
```

case vergleicht den <Wert> (der sich aus einer Variable, einem Programmaufruf, ... ergeben kann) der Reihe nach mit den angebotenen Mustern. Passt ein Muster, wird die entsprechende Kommandofolge ausgeführt, bis zum ;;. Danach wird case beendet, weitere Treffer werden nicht beachtet. Der Rückgabewert ist der des letzten Kommandos der ausgeführten Kommandofolge; passt keines der Muster, wird ein Rückgabewert von »null« erzeugt.

In den case-Mustern können Sie Suchmuster verwenden, wie sie auch für die Dateinamenexpansion benutzt werden (»*«, »?«, ...). Sie können also als letztes Muster »*« einsetzen und damit zum Beispiel eine Fehlermeldung ausgeben. Außerdem erlaubt ist die Angabe von Alternativen in der Form

```
[Jj]a|[Yy]es|[0o]ui)
```

(um zum Beispiel die Werte »Ja«, »ja«, »Yes«, »yes«, »oui« oder »oui« zu erkennen).

Am besten sehen Sie die Funktion von case anhand eines häufigen Beispiels demonstriert – eines Init-Skripts. Zur Erinnerung, Init-Skripte sind für das Starten und Stoppen von Hintergrund-Diensten zuständig. In aller Regel funktionieren sie wie

```
<Init-Skript> start
```

bzw.

```
<Init-Skript> stop
```

Dabei wird als erster Positionsparameter ein »start«, »stop«, »status«, ... eingegeben – ein gefundenes Fressen für case. Bild 3.1 zeigt ein simples Init-Skript, das das Programm tcpdump (einen Paketsniffer) so startet, dass alle empfangenen Pakete in einer Datei gespeichert werden (zur späteren Analyse). Achtung: Die bei Distributionen mitgelieferten Init-Skripte sind meist wesentlich komplexer gestrickt.

```
#!/bin/sh

DIENST=/usr/sbin/tcpdump
DIENSTOPTS="-w"
DUMPFILER="/tmp/tcpdump.`date +%Y-%m-%d_%H:%M`"
INTERFACE=eth0
PIDFILE=/var/run/tcpdump.pid

case $1 in
  start)
    echo "Starte $DIENST"
    nohup "$DIENST" "${DIENSTOPTS}" "$DUMPFILER" > /dev/null 2>&1 &
    echo "$!" > "$PIDFILE"
    ;;
  stop)
    echo "Stoppe $DIENST"
    kill -15 `cat "$PIDFILE`
    rm -f "$PIDFILE"
    ;;
  status)
    if [ -f $PIDFILE ]
    then
      if ps `cat $PIDFILE` > /dev/null 2>&1
      then echo "Dienst $DIENST läuft"
      fi
    else echo "Dienst $DIENST läuft NICHT"
    fi
    ;;
  clean)
    echo "Welche Dumpfiles wollen Sie löschen?"
    rm -i ${DUMPFILER%.`date +%Y`}*
    ;;
  *)
    echo "Fehler: Bitte verwenden Sie einen der Parameter (start|stop|status)!"
    ;;
esac
```

Bild 3.1: Ein einfaches Init-Skript

Übungen



3.4 [!] Geben Sie ein Shellskript an, das prüft, ob sein (einziger) Positionsparameter nur aus Vokalen besteht, und einen entsprechenden Rückgabewert liefert (0 sei »ja«).



3.5 [!] Geben Sie ein Shellskript an, das prüft, ob sein (einziger) Positionsparameter ein absoluter oder relativer Pfadname ist, und entsprechend »absolut« oder »relativ« ausgibt.



3.6 [2] Wie würden Sie mit möglichst wenig Aufwand im Init-Skript aus Bild 3.1 eine »restart«-Aktion implementieren, die den Dienst anhält und gleich wieder startet?

3.4.4 Schleifen

Oftmals ist es nützlich, eine Folge von Kommandos mehrmals hintereinander ausführen zu können, vor allem dann, wenn die Anzahl der Wiederholungen nicht bekannt ist, wenn Sie das Skript schreiben, sondern sich erst beim Aufruf ergibt. Die Shell unterstützt zwei verschiedene Ansätze zur Schleifenbildung:

- Die **Iteration** über eine vorgegebene Liste von Werten, etwa alle Positionsparameter oder alle Dateinamen in einem Verzeichnis. Hierbei steht die Anzahl der Durchläufe in dem Moment fest, wo die Schleife begonnen wird. Iteration
- Eine **Testschleife**, bei der am Anfang jedes Durchlaufs ein Kommando ausgeführt wird. Das Resultat des Kommandos entscheidet darüber, ob die Schleife wiederholt oder beendet wird. Testschleife

Iteration mit for Zur Iteration über eine gegebene Liste von Werten dient das `for`-Kommando. Eine Variable nimmt in jedem Durchlauf einen neuen Wert aus der Liste an:

```
for <Variable> [in <Liste>]
do
  <Kommandos>
done
```

Hier ein kleines Beispiel dafür:

```
#!/bin/bash
# Name: fordemo
for i in eins Zwei DREI
do
  echo $i
done
```

Die Ausgabe sieht so aus:

```
$ fordemo
eins
Zwei
DREI
```

Die Liste, über die `for` iteriert, muss nicht fest im Skript stehen, sondern kann sich auch erst während der Skriptausführung ergeben, etwa durch Dateinamenexpansion:

```
#!/bin/bash
for f in *
do
    mv "$f" "$f.txt"
done
```

Die Variable *f* läuft über alle Dateinamen im aktuellen Arbeitsverzeichnis. Dadurch werden alle Dateien umbenannt. Beachten Sie, dass die Liste genau einmal aufgebaut wird, nämlich wenn die Shell auf das `for`-Kommando stößt. Dass die Dateinamen im aktuellen Verzeichnis sich durch die Ausführung der Schleife ändern, hat für die Liste, über die iteriert wird, keine Auswirkung.

Wenn Sie die *⟨Liste⟩* ganz weglassen, so iteriert die Schleife über die Positionsparameter des Skripts, die Schleifenvariable nimmt also nacheinander die Werte *\$1*, *\$2*, ... an. Damit ist ein bloßes »`for i`« äquivalent zu »`for i in "$@"`«.



Eine Variante der `for`-Schleife ist an die Programmiersprache C angelehnt: Bei

```
for (( i=0 ; $i<10; i=i+1 ))
do
    echo $i
done
```

nimmt *i* nacheinander die Werte 0, 1, ..., 9 an. Genaugenommen dient der erste arithmetische Ausdruck im Klammerpaar zur Initialisierung, der zweite wird immer am Schleifenanfang überprüft und der dritte wird am Schleifenende ausgeführt, bevor an den Schleifenanfang zurückgesprungen (und der zweite Ausdruck als Test ausgeführt) wird. Hierbei handelt es sich also nicht um eine Iteration wie beim eben gezeigten Listen-`for`, sondern um eine Testschleife wie im nächsten Abschnitt – dieses `for` ist ein enger Verwandter von `while`.

Testschleifen mit `while` und `until` Die Kommandos `while` und `until` dienen dazu, Schleifen zu programmieren, bei denen die Anzahl der Durchläufe sich erst während der Ausführung der Schleife ergibt (und nicht schon am Anfang feststeht wie bei `for`). Dazu wird ein Kommando angegeben, dessen Rückgabewert darüber entscheidet, ob der Schleifeninhalt (und das Testkommando dann erneut) ausgeführt werden soll oder ob die Schleife beendet wird:

```
while <Test-Kommando>
do
    <Kommandos>
done
```

Das folgende Beispiel gibt beim Aufruf die Zahlen von 1 bis 5 aus:

```
#!/bin/bash
# Name: whiledemo
i=1
while test $i -le 5
do
    echo $i
    i=$((i+1))
done
```

Die `$(...)`-Konstruktion berechnet den numerischen Wert des Klammerinhalts, hier wird also der Wert der Variablen *i* um 1 erhöht.

Auch bei `while` sind Sie nicht an `test` als Test-Kommando gebunden, was sich besonders nützlich im folgenden Beispiel niederschlägt:

```
#!/bin/bash
# Name: readline
while read LINE
do
    echo "--$LINE--"
done < /etc/passwd
```

Hierbei dient `read` als Test-Kommando. Es liest bei jedem Aufruf eine Zeile von der Standard-Eingabe und legt sie in der Variable `LINE` ab. Kann `read` nichts lesen, beispielsweise weil das Dateiende erreicht ist, so ist sein Rückgabewert von 0 verschieden. Damit läuft die `while`-Schleife so lange, bis die ganze Datei gelesen ist. Da hier die Standard-Eingabe der Schleife umgelenkt wurde, wird also `/etc/passwd` zeilenweise gelesen und bearbeitet.



Auch auf die Gefahr hin, einen Kandidaten für den *useless use of cat award* zu produzieren, halten wir die Konstruktion

```
cat /etc/passwd | while read LINE
do
    ...
done
```

für klarer. Merken Sie sich auf jeden Fall, dass auch Schleifen und Fallunterscheidungen eine Standardeingabe und eine Standardausgabe haben und demnach in der Mitte einer Pipeline stehen können.

`until` verhält sich wie `while`, bis darauf, dass bei `until` die Schleife wiederholt wird, solange das Testkommando »Misserfolg« signalisiert, also einen von 0 verschiedenen Rückgabewert liefert. Das Zählbeispiel könnten Sie also auch so formulieren:

```
#!/bin/bash
# Name: untildemo
i=1
until test $i -gt 5
do
    echo $i
    i=$((i+1))
done
```

Übungen



3.7 [!1] Schreiben Sie ein Shellskript, das alle Vielfachen von 3 bis zu einem als Positionsparameter gegebenen Maximalwert ausgibt.



3.8 [3] Schreiben Sie ein Shellskript, das alle Primzahlen bis zu einer gegebenen Obergrenze ausgibt. (*Tipp*: Verwenden Sie den Modulo-Operator `%`, um zu prüfen, ob eine Zahl ohne Rest durch eine andere teilbar ist.)

3.4.5 Schleifenunterbrechung

Ab und zu kommt es vor, dass es – beispielsweise beim Auftreten eines Fehlers – nötig ist, eine Schleife vorzeitig abubrechen. Oder es stellt sich heraus, dass ein Schleifendurchlauf nicht bis zu Ende geführt werden muss, sondern gleich der nächste begonnen werden kann. Die Bash unterstützt das durch die Kommandos `break` und `continue`.

Schleifen abbrechen mit break Das `break`-Kommando bricht die Ausführung der aktuellen Schleife ab und sorgt dafür, dass hinter dem zugehörigen `done` fortgefahren wird. Betrachten Sie zum Beispiel das folgende Skript:

```
#!/bin/bash
# Name: breakdemo
for f
do
    [ -f $f ] || break
    echo $f
done
```

Wenn Sie dieses Skript mit einer Reihe von Argumenten aufrufen, prüft es für jedes Argument, ob eine gleichnamige Datei existiert. Ist das für ein Argument nicht der Fall, wird die Schleife sofort abgebrochen:

```
$ touch a c
$ ./breakdemo a b c
a
$_
```



Mit `break` können Sie auch aus verschachtelten Schleifen »ausbrechen«: Geben Sie als Argument die Anzahl der »inneren Schleifen« an, die Sie beenden wollen. – Probieren Sie einmal das folgende Skript aus:

```
#!/bin/bash
# Name: breakdemo2
for i in a b c
do
    for j in p q r
    do
        for k in x y z
        do
            break $1
        done
        echo Hinter der inneren Schleife
    done
    echo Hinter der mittleren Schleife
done
echo Hinter der äußeren Schleife
```

Hier können Sie über einen Kommandozeilenparameter angeben, wie viele Schleifen (von innen gesehen) abgebrochen werden sollen.

Schleifendurchläufe abbrechen mit continue Das Kommando `continue` beendet nicht die komplette Schleife, sondern nur den aktuellen Durchlauf. Danach wird entweder sofort der nächste Durchlauf gestartet (bei `for`) oder das Testkommando ausgeführt und geprüft, ob noch ein Durchlauf nötig ist (bei `while` und `until`).

Das folgende Skript zeigt eine etwas umständliche Methode dafür, Dateien nur dann zu kopieren, wenn sie bestimmte Zeichenketten enthalten:

```
#!/bin/bash
# Name: continuedemo
pattern=$1
shift
for f
do
    fgrep -q $pattern $f
```

```

if [ $? = 1 ]
then
    continue
fi
cp $f $HOME/backups
done

```

(Siehe hierzu auch Übung 3.9.)

Genau wie bei `break` können Sie auch bei `continue` ein numerisches Argument angeben, das bestimmt, bei der wievielten Schleife (von innen) ein neuer Durchgang angefangen werden soll.

Übungen



3.9 [!2] Wie würden Sie das Skript aus dem `continue`-Beispiel sinnvoll vereinfachen?



3.10 [!1] Betrachten Sie das Skript `breakdemo2` auf Seite 52 und ändern Sie es so, dass Sie mit dem neuen Programm ausprobieren können, wie das `continue`-Kommando sich mit verschiedenen numerischen Argumenten verhält.

Ausnahmebehandlung In Bash-Skripten können Sie auf eingehende Signale oder andere ungewöhnliche Vorkommnisse reagieren. Dazu dient das Kommando `trap`, das Sie zum Beispiel so aufrufen können:

```
trap "rm /tmp/script.$$; exit" TERM
```

Wenn Sie dieses Kommando in einem Skript ausführen, wird das Kommando »`rm /tmp/script.$$; exit`« gespeichert. Falls der Shellprozess ein `SIGTERM` geschickt bekommt, wird das gespeicherte Kommando ausgeführt. In diesem Beispiel würde etwa eine temporäre Datei des Skripts entfernt – eine typische Form von Aufräumarbeiten.

Betrachten Sie das Skript `traptest`:

```
#!/bin/sh
trap "echo Signal empfangen ; exit" TERM HUP
sleep 60
```

Wir können dieses Skript im Hintergrund ausführen und dem Prozess anschließend zum Beispiel ein `SIGTERM` schicken:

```
$ sh traptest &
[2] 11086
$ kill -TERM %2
Signal empfangen

[2]+ Exit 143      sh traptest
```

Die Signale `SIGSTOP` und `SIGKILL` können Sie natürlich nicht abfangen. Das genaue Verhalten der Bash im Zusammenhang mit Signalen ist im Abschnitt *Signals* der Bash-Dokumentation erklärt.



Sie können am Rückgabewert eines Prozesses sehen, ob er mit einem Signal beendet wurde: Falls das so ist, ist der Rückgabewert 128 plus die Signallnummer, bei `SIGTERM` 15.

Mit `trap` können Sie nicht nur auf (externe) Signale reagieren, sondern auch auf andere Ereignisse. Für das Ereignis `EXIT` registrierte Kommandos werden zum Beispiel ausgeführt, wenn der Prozess wie auch immer zu Ende kommt (durch ein Signal oder einfach nur durch `exit` oder dadurch, dass das Ende des Skripts erreicht

wurde). Mit dem Ereignis ERR können Sie ein Kommando ausführen, wenn ein einfaches Shellkommando einen Rückgabewert ungleich 0 liefert (funktioniert nicht, wenn das Kommando Teil einer `while`- oder `until`-Schleife, eines `if`-Kommandos oder einer Kommandofolge mit `&&` oder `||` ist oder unter `!` aufgerufen wird).

Übungen

 **3.11** [!1] Überzeugen Sie sich, dass das `trap`-Kommando für gängige Ereignisse wie `SIGTERM` oder `EXIT` so funktioniert wie angegeben.

 **3.12** [2] Schreiben Sie ein Shellskript, das in einem Textterminal eine Digitaluhr anzeigt. Wenn der Anwender das Skript mit `Strg+C` abbricht, soll der Bildschirm gelöscht und das Programm beendet werden. (Wenn Ihr System über das `SysV-banner`-Programm verfügt, macht diese Aufgabe besonders Spaß.)

3.5 Shellfunktionen

Häufig gebrauchte Codesequenzen lassen sich prinzipiell als »Unter-Shellskripte« realisieren, die Sie aus einem Shellskript heraus aufrufen können. Moderne Shells wie die `Bash` erlauben aber auch die Definition von »Shellfunktionen« im selben Skript:

```
#!/bin/bash

function sort-num-rev () {
    sort -n -r
}

ls -l | sort-num-rev
```

Shell-Funktionen benehmen sich aus der Sicht des aufrufenden Codes wie »normale« Kommandos – sie haben eine Standard-Ein- und -Ausgabe, können Kommandoargumente übernehmen und so weiter.

 Sie dürfen entweder auf das Wort `function` oder auf die runden Klammern verzichten (nicht auf beides). Wir empfehlen jedoch, alles hinzuschreiben, einfach damit klar ist, was passiert.

 Hinter den runden Klammern sind ganz verschiedene Sachen erlaubt (schlagen Sie in der `Bash`-Dokumentation unter *Compound Commands* nach); wir empfehlen Ihnen aber, sich auf Listen von Shell-Kommandos in geschweiften Klammern zu beschränken.

Positionsparameter Innerhalb von Shellfunktionen entsprechen die Positionsparameter `$1`, `$2`, ... den Argumenten der Shellfunktion, nicht denen des eigentlichen Shellprozesses. Entsprechend gibt auch `$#` die Anzahl der Positionsparameter der Shellfunktion wieder, `$*` ist die Gesamtheit der Positionsparameter und so weiter (nach dem Ende der Shellfunktion ist wieder alles so, wie es war). Ansonsten haben Sie auch aus einer Shellfunktion heraus Zugriff auf die Shell- und damit auch die Umgebungsvariablen des kompletten Prozesses. Ein Beispiel:

```
#!/bin/bash

function panic () {
    exitcode=$1
    shift
    echo >&2 "$0: PANIC: $*"
}
```

```

    exit $exitcode
}
<<<<<
[ -f file.txt ] || panic 3 file.txt ist nicht vorhanden
<<<<<

```

Hier dient der erste Positionsparameter als Rückgabewert des Prozesses, die restlichen Positionsparameter als Fehlermeldung.



Auch in Shellfunktionen ist \$0 der Name des Shellskripts, nicht der der Funktion. Wenn Sie den Namen der Shellfunktion wissen wollen: Der steht in der Variablen FUNCNAME.

Den Rückgabewert einer Shellfunktion können Sie mit dem return-Kommando festlegen: Rückgabewert

```

function sonntag () {
    if [ $(date +%u) = "7" ]
    then
        return 1
    else
        return 0
    fi
}

```

Hat das return-Kommando keinen Parameter oder kommt in der Funktion gar kein return vor, dann ist der Rückgabewert der Funktion der Rückgabewert des letzten in der Funktion ausgeführten Kommandos. Statt der etwas umständlichen Konstruktion im gerade gezeigten Beispiel genügt also auch ein einfaches

```

function sonntag () {
    test $(date +%u) = "7"
}

```

Sie können die Namen und Definitionen der in Ihrer aktuellen Shell vorhandenen Funktionen mit dem Kommando »typeset -f« abrufen. »typeset -F« liefert Ihnen nur die Funktionsnamen. Shellfunktionen finden



Sie können bequem »Funktionsbibliotheken« realisieren, indem Sie die gewünschten Shellfunktionen in eine Datei schreiben, die Sie in jedem Shellskript, das sie benutzen soll, mit »source« einlesen. Funktionsbibliotheken

Übungen



3.13 [!1] Definieren Sie eine Shellfunktion toupper, die ihre Positionsparameter in Großbuchstaben umsetzt und auf der Standardausgabe ausgibt.

3.6 Das Kommando exec

Normalerweise wartet die Shell darauf, dass ein externes Kommando sich beendet, und liest dann das nächste Kommando ein. Mit dem Kommando exec können Sie ein externes Kommando so starten, dass es die Shell *ersetzt*. Zum Beispiel können Sie, wenn Sie statt der Bash lieber die C-Shell verwenden möchten, mit

```

$ exec /bin/csh
% _

```

Hier ist die C-Shell!

in Ihrer Sitzung eine C-Shell starten, ohne eine unbenutzte Bash herumliegen zu haben, die Sie beim Abmelden auch noch zusätzlich beenden müssen.



exec wird vor allem in Shellskripten verwendet und selbst da nicht oft. Für die C-Shell-statt-Bash-Geschichte gibt es bequemere Methoden.

Übungen



3.14 [!2] Die Datei test1 bestehe aus den Zeilen

```
echo Hallo
exec bash test2
echo Auf Wiedersehen
```

und die Datei test2 aus der Zeile

```
echo Huhu
```

Was gibt das Kommando »bash test1« aus?

Kommandos in diesem Kapitel

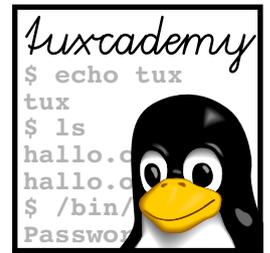
env	Gibt die Prozessumgebung aus oder startet Programme mit veränderter Umgebung	env(1)	35
exec	Startet ein neues Programm im aktuellen Shell-Prozess	bash(1)	55
export	Definiert und verwaltet Umgebungsvariable	bash(1)	35
for	Shell-Kommando für Schleifen über eine Liste von Elementen	bash(1)	49
set	Verwaltet Shellvariable	bash(1)	34
test	Wertet logische Ausdrücke auf der Kommandozeile aus	test(1), bash(1)	44
unset	Löscht Shell- oder Umgebungsvariable	bash(1)	36

Zusammenfassung

- Variable dienen zur Zwischenspeicherung von Resultaten, zur Steuerung der Shell und (als Umgebungsvariable) zur Kommunikation mit Kindprozessen.
- Die Shell definiert diverse spezielle Variable zum Beispiel zum Zugriff auf Positionsparameter der Shell.
- Es gibt einige Spezialformen der Variablensubstitution zum Einsetzen von Standardwerten oder zum Bearbeiten der Variableninhalte.
- Die Bash folgt bei der Bearbeitung von Kommandozeilen einer Reihe von definierten Ersetzungsschritten.
- Die Bash unterstützt die gängigen Kontrollstrukturen von Programmiersprachen.
- Als Steuergröße in Kontrollstrukturen verwendet die Shell den Rückgabewert von Unterprozessen; ein Rückgabewert von 0 steht für »wahr«, alles andere für »falsch«.
- Fallunterscheidungen sind über die &&- und ||-Operatoren und das if- und das case-Kommando möglich.
- Schleifen lassen sich mit while, until und for definieren und mit break und continue steuern.
- Mit trap können Skripte auf Signale und andere Ereignisse reagieren.
- Shellfunktionen erlauben es, öfter benötigte Kommandofolgen im selben Skript zur Verfügung zu haben.

Literaturverzeichnis

- WMPK69** A. van Wijngarden, B. J. Mailloux, J. E. L. Peck, et al. »Report on the Algorithmic Language ALGOL 68«. *Numerische Mathematik*, 1969. 14:79–218. Dieser Artikel ist ein Klassiker – der erste Versuch, die Semantik einer Programmiersprache in halbwegs formeller Weise niederzulegen. Die Sprache selber hatte leider niemals praktische Bedeutung.



4

Praktische Shellskripte

Inhalt

4.1	Shellprogrammierung in der Praxis	60
4.2	Rund um die Benutzerdatenbank	60
4.3	Dateioperationen	65
4.4	Protokolldateien	67
4.5	Systemadministration	73

Lernziele

- Einige Beispiele für Shellskripte analysieren und verstehen
- Grundlegende Techniken der praktischen Shellprogrammierung kennen und anwenden können

Vorkenntnisse

- Vertrautheit mit der Kommandooberfläche von Linux
- Umgang mit Dateien und einem Texteditor
- Shell-Vorkenntnisse (etwa aus Kapitel 3 und den vorigen Kapiteln)

4.1 Shellprogrammierung in der Praxis

Die vorigen Kapitel haben große Teile der Shell-Syntax vorgestellt. Vermutlich wundern Sie sich inzwischen, was das alles soll: Erklären wir Ihnen das komplette Inventar der Küche und Speisekammer und erwarten jetzt, dass Sie ein sternverdächtiges Fünfgang-Menü kochen? Keine Angst. Programmieren lernt man nicht aus einer Syntaxbeschreibung, sondern durch das Studium vorbildlicher Programme und vor allem durch eigene Experimente. In diesem Kapitel haben wir darum einige Shellskripte für Sie zusammengetragen, an denen Sie viele gängige Techniken kennenlernen und schon Gesehenes vertiefen können; vor allem sollen diese Skripte Sie aber dazu inspirieren, selbst Hand anzulegen und die Möglichkeiten der Shell in der Praxis kennenzulernen.

4.2 Rund um die Benutzerdatenbank

Auf »normalen« Linux-Systemen sind die Informationen über Benutzer – Benutzernamen und -IDs, primäre Gruppe, bürgerlicher Name, Heimatverzeichnis und so weiter – in der Datei `/etc/passwd` abgelegt.¹ Hier ein Beispiel zur Erinnerung:

Benutzerdatei

```
tux:x:1000:100:Tux der Pinguin:/home/tux:/bin/bash
```

Der Benutzer `tux` hat also die UID 1000 und als primäre Gruppe die mit der GID 100. Im wirklichen Leben heißt er »Tux der Pinguin«, sein Heimatverzeichnis ist `/home/tux` und seine Login-Shell ist die Bash.

Gruppendatei

Die Gruppendatei `/etc/group` enthält für jede Gruppe den Namen, ein optionales Kennwort, die GID und eine Mitgliederliste, die üblicherweise nur diejenigen Benutzer aufführt, die die betreffende Gruppe als zusätzliche Gruppe haben:

```
users:x:100:
pinguine:x:101:tux
```

Wer ist in dieser Gruppe? Unser erstes Skript soll zu einem Gruppennamen alle Benutzer aufzählen, die diese Gruppe als primäre Gruppe führen. Die Herausforderung besteht darin, dass in `/etc/passwd` nur die GID der primären Gruppe steht und wir uns diese also zuerst aus `/etc/group` anhand des Gruppennamens holen müssen. Unser Skript übernimmt die gefragte Gruppe als Kommando-parameter.

```
#!/bin/bash
# pgroup -- erste Version

# Hole die GID aus /etc/group
gid=$(grep "^$1:" /etc/group | cut -d: -f3)

# Suche nach Benutzern mit der betreffenden GID in /etc/passwd
grep "^[^:]*:[^:]*:[^:]*:$gid:" /etc/passwd | cut -d: -f1
```

Beachten Sie die Verwendung von `grep` zum Finden der richtigen Zeile und von `cut` zum Extrahieren der passenden Spalte in dieser Zeile. Im zweiten `grep` ist der Suchausdruck etwas umständlich, aber wir müssen aufpassen, dass wir nicht irrtümlich eine UID selektieren, die aussieht wie unsere gesuchte GID – darum achten wir darauf, erst mal drei Doppelpunkte von links abzuzählen und dann nach der GID zu schauen.

Bei der Programmierung von Shellskripten gilt das Gleiche wie bei der Programmierung überhaupt: Der meiste Aufwand geht in das Abfangen von Benutzer-

Fehler abfangen

¹Unter nicht ganz normalen Linux-Systemen greifen Methoden wie LDAP zur Speicherung für Benutzerdaten um sich. Sollten Sie ein solches System vor sich haben, können Sie sich in der Regel mit Kommandos wie »`getent passwd >$HOME/passwd`« und »`getent group >$HOME/group`« die Dateien besorgen, die für die folgenden Experimente nötig sind.

und anderen Fehlern. Unser Skript kümmert sich zum Beispiel weder um Probleme beim Aufrufen – Weglassen des Gruppennamens oder das Angeben zusätzlicher, überflüssiger Parameter – noch über Probleme bei der Ausführung. Es ist relativ unwahrscheinlich, dass `/etc/passwd` nicht zur Verfügung steht (das wäre Ihnen dann wohl schon anderweitig aufgefallen), aber es kann durchaus sein, dass der Benutzer des Skripts einen Gruppennamen angibt, den es nicht wirklich gibt. Aber fangen wir vorne an.

Als allererstes sollten wir uns davon überzeugen, dass das Skript mit der richtigen Anzahl Parameter aufgerufen wurde, nämlich einem. Prüfen können Sie das zum Beispiel, indem Sie den Wert von `$#` inspizieren:

Syntaxprüfung

```
if [ $# -ne 1 ]
then
    echo >&2 "usage: $0 GROUP"
    exit 1
fi
```

Dieses Stückchen Shell-Code illustriert gleich einige wichtige Techniken. Ist die Anzahl der Parameter (in `$#`) nicht 1, wollen wir das Skript mit einer Fehlermeldung beenden. Die Fehlermeldung schreibt `echo` , wobei wir darauf achten, dass sie schön ordentlich nicht auf der Standard-Ausgabe, sondern auf der Standard-Fehlerausgabe erscheint (das `>&2 - 2` ist ja der Standard-Fehlerausgabe-Kanal). `$0` ist der Name, unter dem das Skript aufgerufen wurde; es ist üblich, diesen in der Fehlermeldung anzugeben, und so stimmt er immer, auch wenn die Skript-Datei umbenannt wurde.

Fehlermeldung



`$0` liefert den Namen des Skripts so, wie der Aufrufer ihn angegeben hat, also möglicherweise mitsamt Überflüssigem wie einer Pfadangabe. Für Fehlermeldungen ist das nicht notwendigerweise schön; Sie können etwas verwenden wie

```
myself=${0##*/}
<<<<<<
echo >&2 "usage: $myself GROUP"
```

und so die Ausgabe auf den tatsächlichen Skriptnamen beschränken.

Mit `exit` wird das Skript (vorzeitig) beendet, wobei wir als Rückgabewert eine 1 liefern, für »generischen Misserfolg«.

Rückgabewert



Wenn Sie `exit` ohne Argument aufrufen oder einfach am Ende eines Shell-Skripts ankommen, beendet sich die Shell auch. In diesem Fall ist der Rückgabewert der Shell der Rückgabewert des letzten ausgeführten Kommandos (`exit` zählt nicht mit). Vergleichen Sie

```
$ sh -c "true; exit"; echo $?
0
$ sh -c "false; exit"; echo $?
1
```

Dann müssen wir uns natürlich noch um den Fall der nicht existierenden Gruppe kümmern. Im Abschnitt 3.4.2 haben Sie gehört, wie `grep` mit seinen Rückgabewerten umgeht: 0 bedeutet »etwas Passendes wurde gefunden«, 1 steht für »alles im Grunde OK, aber keine passende Zeile gefunden« und 2 für »irgendein Fehler ist aufgetreten« (möglicherweise war der reguläre Ausdruck nicht in Ordnung, oder beim Lesen der Eingabe ist irgendetwas schief gegangen). Das wäre schon sehr nützlich; wir könnten prüfen, ob der Rückgabewert von `grep` 1 oder 2 ist und das Skript gegebenenfalls mit einer Fehlermeldung beenden. Beim kritischen Kommando

```
#!/bin/bash
# pgroup -- verbesserte Version

# Prüfe die Parameter
if [ $# -ne 1 ]
then
    echo >&2 "usage: $0 GROUP"
    exit 1
fi

# Hole die GID aus /etc/group
gid=$(grep "^$1:" /etc/group | cut -d: -f3)
if [ -z "$gid" ]
then
    echo >&2 "$0: group $1 does not exist"
    exit 1
fi

# Suche nach Benutzern mit der betreffenden GID in /etc/passwd
grep "^[^:]*:[^:]*:[^:]*:$gid:" /etc/passwd | cut -d: -f1
```

Bild 4.1: Welche Benutzer haben eine bestimmte primäre Gruppe? (Verbesserte Version)

```
gid=$(grep "^$1:" /etc/group | cut -d: -f3)
```

Rückgabewert einer Pipeline gibt es da nur leider ein kleines Problem: Der Rückgabewert einer Pipeline ist der Rückgabewert des *letzten* Kommandos, und das cut klappt grundsätzlich eigentlich immer, auch wenn das grep ihm eine leere Eingabe liefert (die cut-Argumente sind ja fundamental in Ordnung). Wir müssen uns also etwas Anderes einfallen lassen.

Was passiert aber, wenn grep keine passende Zeile findet? Richtig, die Ausgabe von cut ist leer, ganz im Gegensatz zu dem Fall, dass grep die Gruppe finden kann (wenn wir eine syntaktisch korrekte /etc/group-Datei voraussetzen, dann hat die gefundene Zeile auch eine dritte Spalte mit einer GID). Wir müssen also nur prüfen, ob unsere Variable gid einen »echten« Wert hat:

```
if [ -z "$gid" ]
then
    echo >&2 "$0: group $1 does not exist"
    exit 1
fi
```

(Bemerkten Sie auch hier \$0 als Teil der Fehlermeldung.)

Damit ergibt sich als »vorläufig endgültige« Version unseres Skripts der Inhalt von Bild 4.1.

In welchen Gruppen ist ein Benutzer Mitglied? Unser nächstes Beispiel ist ein Skript, das die Gruppen ausgibt, in denen ein Benutzer Mitglied ist – ähnlich dem Kommando groups. Hierbei müssen wir beachten, dass es nicht ausreicht, nur /etc/group zu betrachten, denn normalerweise sind Benutzer bei ihrer primären Gruppe nicht in /etc/group eingetragen. Wir verfolgen also den folgenden Plan:

1. Gib den Namen der primären Gruppe des Benutzers aus
2. Gib die Namen der zusätzlichen Gruppen des Benutzers aus

Der erste Teil sollte uns leicht fallen – er ist im Prinzip das vorige Skript »andersherum«:

```
# Primäre Gruppe
gid=$(grep "^$1:" /etc/passwd | cut -d: -f4)
grep "^[^:]*:[^:]*:$gid:" /etc/group | cut -d: -f1
```

Der zweite Teil scheint eher noch einfacher zu sein: Ein simples

```
grep $1 /etc/group
```

bringt uns schon in die Nähe des Nirwana. Oder nicht? Überlegen Sie, was dieses `grep` alles finden könnte:

- Zunächst mal den Benutzernamen in der Mitgliederliste einer Gruppe. Das ist das, was wir wollen.
- Außerdem Benutzernamen in der Mitgliederliste, die den gesuchten Benutzernamen als Teilzeichenkette enthalten. Wenn wir nach `john` suchen, bekommen wir auch alle Zeilen von Gruppen, in denen der Benutzer `johnboy` Mitglied ist, `john` aber nicht. Schon falsch.
- Dasselbe Problem gilt für Benutzernamen, die Teilzeichenketten von Gruppennamen sind. Eine Gruppe `staff` hat mit einem Benutzer `taf` erst mal nichts zu tun, passt aber trotzdem.
- Ziemlich absurd, aber möglich: Der gesuchte Benutzername könnte als Teilzeichenkette eines verschlüsselten Gruppenkennworts auftreten, das nicht in `/etc/gshadow`, sondern in `/etc/group` steht (absolut erlaubt).

Auch hier ist also Sorgfalt angesagt, wie immer bei `grep` – Sie sollten sich angewöhnen, beim Erstellen von regulären Ausdrücken so böseartig und negativ zu denken, wie Sie irgend können. Dann kommen Sie bei etwas an wie

```
grep "^[^:]*:[^:]*:[^:]*.*\<$1\>" /etc/group | cut -d: -f1
```

Das heißt, wir suchen den Benutzernamen nur im vierten Feld von `/etc/group`. Die »Wortklammern« `\<...\>` (eine Spezialität von GNU-`grep`) helfen uns gegen den `johnboy`-Fehler. Insgesamt sind wir dann bei

```
#!/bin/bash
# lsgroups -- erste Version

# Primäre Gruppe
gid=$(grep "^$1:" /etc/passwd | cut -d: -f4)
grep "^[^:]*:[^:]*:$gid:" /etc/group | cut -d: -f1

# Zusätzliche Gruppen
grep "^[^:]*:[^:]*:[^:]*.*\<$1\>" /etc/group | cut -d: -f1
```

Probieren wir das Skript mal auf einem Debian-GNU/Linux-System aus:

```
$ ./lsgroups tux
tux
dialout
fax
voice
cdrom
floppy
<<<<<<
```

```
#!/bin/bash
# lsgroups -- endgültige Version

# Primäre Gruppe
( gid=$(grep "^$1:" /etc/passwd | cut -d: -f4)
  grep "^[^:]*:[^:]*:$gid:" /etc/group | cut -d: -f1

# Zusätzliche Gruppen
grep "^[^:]*:[^:]*:[^:]*:.*\<$1\>" /etc/group \
  | cut -d: -f1 ) | sort -u
```

Bild 4.2: In welchen Gruppen ist Benutzer *x*?

```
src
tux
scanner
```

Hierbei sollten uns zwei Dinge auffallen. Zunächst ist die Liste unsortiert, was un schön ist; ferner taucht die Gruppe *tux* in der Liste zweimal auf. (Debian GNU/Linux ist eine der Distributionen, die standardmäßig jeden Benutzer in eine eigene gleichnamige Gruppe tun.) Letzteres rührt daher, dass */etc/group* eine Zeile der Form

```
tux:x:123:tux
```

enthält – ungewöhnlich, aber absolut erlaubt.

Wir sollten die Ausgabe also noch sortieren und dabei Dubletten entfernen (»`sort -u`«). Die Frage ist: Wie? Ein explizites »`lsgroups | sort -u`« liefert uns die gewünschte Lösung, ist aber unbequem; das Sortieren sollte Teil des Skripts sein. Dabei stören wiederum die beiden logisch getrennten Pipelines. Eine Möglichkeit, das zu beheben, wäre die Verwendung einer Zwischendatei:

```
grep ... >/tmp/lsgroups.$$
grep ... >>/tmp/lsgroups.$$
sort -u /tmp/lsgroups.$$
```

(das `$$` wird durch die PID der Shell ersetzt und macht den Namen der Zwischendatei eindeutig). Dieser Ansatz ist unappetitlich, da er potentiell Schrott liegen lässt, wenn aufgrund eines Fehlers die Zwischendatei am Ende des Skripts nicht gelöscht werden kann (es gibt Mittel und Wege, das zu verhindern). Außerdem stellt das Anlegen von Zwischendateien mit solch simplen Namen eine mögliche Sicherheitslücke dar. Viel bequemer ist es, die beiden Pipelines in einer expliziten gemeinsamen Subshell auszuführen und deren Ausgabe dann nach `sort` zu leiten:

```
( grep ...
  grep ... ) | sort -u
```

Auf diese Weise landen wir bei unserer endgültigen Version (Bild 4.2).

Übungen



4.1 [1] Ändern Sie das Skript `pgroup` so ab, dass es zwischen den Fehlersituationen »Syntaxfehler in der Eingabe« und »Gruppe existiert nicht« differenziert, indem es verschiedene Rückgabewerte liefert.

4.3 Dateioperationen

Das Automatisieren von Dateioptionen ist ein dankbares Einsatzgebiet für Shellskripte – das Verschieben, Umbenennen, Sichern von Dateien in Abhängigkeit verschiedenster Kriterien ist oft vielschichtiger, als sich das in einfachen Kommandos ausdrücken lässt. Shellskripte sind also eine bequeme Möglichkeit für Sie, Ihre eigenen Kommandos zu definieren, die genau das tun, was Sie brauchen.

Umbenennen mehrerer Dateien Das `mv`-Kommando ist nützlich, um eine Datei umzubenennen oder mehrere Dateien in ein anderes Verzeichnis zu verschieben. Was damit nicht geht, ist das Umbenennen von mehreren Dateien auf einmal, so ähnlich wie Sie das vielleicht von MS-DOS kennen:

```
C:\> REN *.TXT *.BAK
```

Das funktioniert, weil bei DOS das `REN`-Kommando selbst sich um die Bearbeitung der Suchmuster kümmert – bei Linux ist die Bearbeitung der Suchmuster dagegen Sache der Shell, die nicht weiß, was `mv` mit den Namen anschließend vor hat.

Den allgemeinen Fall der Mehrfachumbenennung per Shellskript zu lösen ist möglich, aber wir beschäftigen uns an dieser Stelle mit einer Einschränkung, nämlich dem Ändern der »Endung« einer Datei. Genauer gesagt möchten wir ein Shellskript `chext` aufstellen, das per

```
$ chext bak *.txt
```

alle aufgezählten Dateien so umbenennt, dass sie die als ersten Parameter angegebene Endung bekommen. In unserem Beispiel würden also sämtliche Dateien, deren Namen auf `».txt«` enden, so umbenannt, dass die Namen auf `».bak«` enden.

Die Hauptaufgabe des `chext`-Skripts ist es offenbar, die passenden Argumente für `mv` zu konstruieren. Wir müssen in der Lage sein, die Endung eines Dateinamens zu entfernen und eine andere Endung anzuhängen – und wie Sie das in der Bash realisieren können, haben Sie schon gelernt: Erinnern Sie sich an die `${...%...}`-Konstruktion bei der Variablensubstitution und betrachten Sie etwas wie

```
$ f=./a/b/c.txt; echo ${f%.*}
./a/b/c
```

Alles ab dem letzten Punkt wird entfernt.

Und daraus ergibt sich der erste Ansatz für unser `chext`-Skript:

```
#!/bin/bash
# chext -- Dateiondung ändern, erste Version

suffix="$1"
shift

for f
do
    mv "$f" "${f%.*}.$suffix"
done
```

Beachten Sie zunächst den Einsatz von Anführungszeichen, um Probleme mit Leerzeichen in Dateinamen zu vermeiden. Ebenfalls interessant ist der Umgang mit der Kommandozeile: Das erste Argument – gleich hinter dem Skriptnamen – ist die gewünschte neue Endung. Diese legen wir in der Variablen `suffix` ab und rufen anschließend das Kommando `shift` auf. `shift` sorgt dafür, dass alle Positionsparameter einen Schritt »nach links« machen: aus `$2` wird `$1`, aus `$3` wird `$2`

und so weiter. Das alte \$1 wird verworfen. Nach unserem `shift` besteht die Kommandozeile also nur noch aus den Namen der umzubenennenden Dateien (die die Shell freundlicherweise aus allfälligen Suchmustern für uns aufgestellt hat), so dass wir bequem mit `»for f«` darüber iterieren können.

Das `mv`-Kommando sieht möglicherweise etwas ehrfurchtgebietend aus, aber es ist eigentlich nicht schwer zu verstehen. `f` ist einer unserer zu ändernden Dateinamen, und mit

```
${f%.*}.$suffix
```

wird, wie schon oben angedeutet, das alte Suffix entfernt und das neue – das in der Shellvariable `suffix` steht – textuell angehängt.



Die Sache ist natürlich nicht ganz ungefährlich, wie Sie sich leicht klar machen können, wenn Sie sich Dateinamen wie `../a.b/c` vorstellen, wo der letzte Punkt nicht in der letzten Pfadkomponente steht. Es gibt mehrere Möglichkeiten, dieses Problem zu lösen. Eine davon involviert den *stream editor*, `sed`, den wir in Kapitel 6 kennen lernen werden, eine andere benutzt die Kommandos `basename` und `dirname`, die auch in manch anderem Zusammenhang nützlich werden können. Sie dienen dazu, einen Dateinamen in einen Verzeichnis- und einen Dateianteil zu zerlegen:

```
$ dirname ../a/b.c/d.txt
../a/b.c
$ basename ../a/b.c/d.txt
d.txt
```

Sie können den Bash-Operator `%` also »entschärfen«, indem sie ihm nur den Dateianteil des zu verändernden Namens vorlegen. Das `mv`-Kommando wird dann zu etwas wie

```
d=$(dirname "$f")
b=$(basename "$f")
mv "$f" "$d/${b%.*}.$suffix"
```

(in der `${...%...}`-Konstruktion sind leider nur Variablennamen erlaubt, keine Kommandosubstitutionen).

Außerdem gehört zu einem anständigen Shellskript natürlich auch die Syntaxprüfung der Kommandozeile. Unser Skript braucht mindestens zwei Parameter – das neue Suffix und einen Dateinamen –, nach oben sind keine Grenzen gesetzt (naja fast). Bild 4.3 zeigt die fertige Version.

Übungen



4.2 [!2] Schreiben Sie ein Shellskript, das einen Dateinamen übernimmt und als Ausgabe die Namen der übergeordneten Verzeichnisse liefert, etwa so:

```
$ hierarchy /a/b/c/d.e
/a/b/c/d.e
/a/b/c
/a/b
/a
/
```



4.3 [2] Benutzen Sie das in der vorherigen Übung erstellte Skript, um ein Skript zu schreiben, das dasselbe tut wie `»mkdir -p«` – das Skript bekommt den Namen eines anzulegenden Verzeichnisses übergeben und soll außer diesem Verzeichnis auch alle möglicherweise nicht existierenden Verzeichnisse weiter oben im Dateibaum erzeugen.

```
#!/bin/bash
# chext -- Dateiendung ändern, verbesserte Version

if [ $# -lt 2 ]
then
    echo >&2 "usage: $0 SUFFIX NAME ..."
    exit 1
fi

suffix="$1"
shift

for f
do
    mv "$f" "${f%.*}.${suffix}"
done
```

Bild 4.3: Massen-Suffixänderung für Dateinamen

4.4 Protokolldateien

Größenkontrolle von Protokolldateien Ein Linux-System erzeugt im laufenden Betrieb diverse Protokolldaten, die zum Beispiel über den Syslog-Daemon in Dateien geschrieben werden. Typische Protokolldateien können schnell wachsen und im Laufe der Zeit beachtliche Größen annehmen. Eine gängige Aufgabe bei der Systemadministration ist darum die Größenkontrolle und gegebenenfalls das Kürzen bzw. Neuansetzen von Protokolldateien. – Die meisten Linux-Distributionen benutzen dafür heuer ein standardisiertes Werkzeug namens `logrotate`.

Als nächstes wollen wir ein Shellskript `checklog` entwickeln, das prüft, ob eine Protokolldatei eine gewisse Länge erreicht oder überschritten hat, und sie gegebenenfalls umbenennt und unter ihrem alten Namen neu anlegt. Als Grundgerüst wäre etwas denkbar wie

```
#!/bin/bash
# checklog -- Prüfe eine Protokolldatei
#             und erneuere sie, falls nötig

if [ $# -ne 2 ]
then
    echo >&2 "usage: $0 FILE SIZE"
    exit 1
fi

if [ $(ls -l "$1" | cut -d' ' -f5) -ge $(( 1024*$2 )) ]
then
    mv "$1" "$1.old"
    > "$1"
fi
```

Die interessante Zeile in diesem Skript ist die mit dem Ausdruck

```
$(ls -l "$1" | cut -d' ' -f5) -ge $(( 1024*$2 ))
```

Hier bestimmen wir die Länge der als Parameter angegebenen Datei (fünfte Spalte der Ausgabe von `»ls -l«`) und vergleichen diese mit der ebenfalls als Parameter angegebenen Maximallänge. Den Längenparameter interpretieren wir dabei als Anzahl von Kibibytes.

Ist die Datei so lang wie die Maximallänge oder länger, wird sie umbenannt und unter dem alten Namen eine leere Datei erzeugt. Im wirklichen Leben ist das nur die halbe Miete: Ein Programm wie der `syslogd` öffnet die Protokolldatei beim Start und schreibt dann fröhlich hinein, egal wie die Datei später heißt – unser Skript kann sie zwar umbenennen, aber das bedeutet noch lange nicht, dass der `syslogd` dann in die neue Datei schreibt. Dazu muss er erst ein `SIGHUP` geschickt bekommen. Eine Möglichkeit, dies zu realisieren, ist über einen (optionalen) dritten Parameter:

```
<<<<<<
> "$1"
[ -n "$3" ] && killall -HUP "$3"
<<<<<<
```

Unser Skript könnte dann etwa so aufgerufen werden:

```
checklog /var/log/messages 1000 syslogd
```

Mehrere Protokolldateien überwachen Unser Skript aus dem vorigen Abschnitt ist vielleicht ganz nett, aber im wirklichen Leben haben Sie es normalerweise mit mehr als einer Protokolldatei zu tun. Sie können `checklog` natürlich x -mal mit verschiedenen Argumenten aufrufen, aber wäre es nicht möglich, dass das Programm sich um mehrere Dateien auf einmal kümmert? Im Idealfall könnten wir eine Konfigurationsdatei haben, die die zu erledigende Arbeit beschreibt und ungefähr so aussehen könnte:

Konfigurationsdatei

```
SERVICES="apache syslogd"
FILES_apache="/var/log/apache/access.log /var/log/apache/error.log"
FILES_syslogd="/var/log/messages /var/log/mail.log"
MAXSIZE=100
MAXSIZE_syslogd=500
NOTIFY_apache="apachectl graceful"
```

Im Klartext: Das Programm soll sich um die »Dienste« `apache` und `syslogd` kümmern. Zu jedem dieser Dienste gibt es eine Konfigurationsvariable, die mit »FILES_« anfängt und eine Liste der interessanten Protokolldateien enthält, und optional eine, die mit »MAXSIZE_« anfängt und die gewünschte Maximalgröße angibt (die Variable `MAXSIZE` gibt einen Standardwert vor, falls für einen Dienst keine eigene `MAXSIZE_`-Variable definiert ist). Ebenfalls optional ist die »NOTIFY_«-Variable, die ein Kommando angibt, mit dem der Dienst über eine neue Protokolldatei benachrichtigt wird (ersatzweise wird wie oben »killall -HUP *<Dienst>*« ausgeführt).

Damit haben wir die Arbeitsweise unseres neuen Skripts – nennen wir es `multichecklog` – schon fast festgelegt:

1. Die Konfigurationsdatei einlesen
2. Für jeden Dienst in der Konfigurationsdatei (`SERVICES`):
3. Bestimme die gewünschte Maximalgröße (`MAXSIZE`, `MAXSIZE_*`)
4. Prüfe jede Protokolldatei gegen die Maximalgröße
5. Benachrichtige gegebenenfalls den Dienst

Der Anfang von `multichecklog` könnte zum Beispiel so aussehen:

```
#!/bin/bash
# multichecklog -- Prüfe mehrere Protokolldateien

conffile=/etc/multichecklog.conf
[ -e $conffile ] || exit 1
. $conffile
```

Wir prüfen, ob unsere Konfigurationsdatei – hier `/etc/multichecklog.conf` – existiert; wenn nein, gibt es nichts für uns zu tun. Wenn sie existiert, lesen wir sie einfach ein und definieren dadurch *in der aktuellen Shell* die Variablen `SERVICES` usw. (wir haben die Syntax der Konfigurationsdatei trickreich so festgelegt, dass das geht).

Anschließend betrachten wir die einzelnen Dienste:

```
for s in $SERVICES
do
  maxsizevar=MAXSIZE_$s
  maxsize=${!maxsizevar:-${MAXSIZE:-100}}
  filesvar=FILES_$s
  for f in ${!filesvar}
  do
    checklonger "$f" "$maxsize" && rotate "$f"
  done
done
```

Wenn Sie aufmerksam mitgelesen haben, fällt Ihnen sicherlich die etwas verquere Konstruktion

```
maxsizevar=MAXSIZE_$s
maxsize=${!maxsizevar:-${MAXSIZE:-100}}
```

auf. Es geht uns darum, den Wert von `maxsize` wie folgt zu bestimmen: Zunächst wollen wir prüfen, ob »`MAXSIZE_⟨Dienst⟩`« existiert und nichtleer ist; wenn ja, wird der Wert dieser Variablen übernommen. Wenn nein, prüfen wir, ob `MAXSIZE` existiert; wenn ja, wird der Wert dieser Variablen übernommen, sonst 100. Das Problem bei der Sache ist der tatsächliche Name von »`MAXSIZE_⟨Dienst⟩`«, der sich erst in der Schleife ergibt. Dafür verwenden wir eine bisher nicht erklärte Eigenschaft der Variablensubstitution: In einem Variablenbezug der Form »`${!⟨Name⟩}`« wird der Wert von `⟨Name⟩` als Name der Variablen interpretiert, deren Wert schließlich eingesetzt wird, etwa so:

Indirekte Substitution

```
$ DE=Hallo
$ CH=Grüezi
$ AT=Servus
$ land=CH
$ echo ${!land} Welt
Grüezi Welt
```

Der `⟨Name⟩` muss trotzdem ein gültiger Variablenname sein – in unserem Skript würden wir gerne etwas sagen wie

```
maxsize=${!MAXSIZE_$s:-${MAXSIZE:-100}}
```

aber das ist nicht erlaubt, also die Extraumleitung mit `maxsizevar`. (Derselbe Trick ist auch für »`FILES_⟨Dienst⟩`« nötig.)

Bemerken Sie ferner, dass wir den Größentest und das Umbenennen nicht in der inneren Schleife ausformuliert haben. Damit unser Skript übersichtlicher wird, realisieren wir diese beiden Aktionen als Shellfunktionen:

Übersichtlichkeit

```
# checklonger FILE SIZE
function checklonger () {
  test $(ls -l "$1" | cut -d' ' -f5) -ge $(( 1024*$2 ))
}

# rotate FILE
function rotate () {
  mv "$1" "$1.old"
```

```
#!/bin/bash
# multichcklog -- Prüfe mehrere Protokolldateien

conffile=/etc/multichcklog.conf
[ -e $conffile ] || exit 1
. $conffile

# checklonger FILE SIZE
function checklonger () {
    test $(ls -l "$1" | cut -d' ' -f5) -ge=$(( 1024*$2 ))
}

# rotate FILE
function rotate () {
    mv "$1" "$1.old"
    > "$1"
}

for s in $SERVICES
do
    maxsizevar=MAXSIZE_$s
    maxsize=${!maxsizevar:-${MAXSIZE:-100}}
    filesvar=FILES_$s
    notify=0
    for f in ${!filesvar}
    do
        checklonger "$f" "$maxsize" && rotate "$f" && notify=1
    done
    notifyvar=NOTIFY_$s
    [ $notify -eq 1 ] && ${!notifyvar:-killall -HUP \}$s
done
```

Bild 4.4: Mehrere Protokolldateien überwachen

```
    > "$1"
}
```

Als letztes fehlt uns nur noch die Benachrichtigung des Dienstes (wir haben sie in der ersten Version unserer Schleife unterschlagen). Wir müssen den Dienst nur einmal benachrichtigen, egal wie viele seiner Protokolldateien wir »rotiert« haben. Am geschicktesten geht das so:

```
notify=0
for f in ${!filesvar}
do
    checklonger "$f" "$maxsize" && rotate "$f" \
        && notify=1
done
notifyvar=NOTIFY_$s
[ $notify -eq 1 ] && ${!notifyvar:-killall -HUP \}$s
```

Auch hier kommt wieder der Trick mit der indirekten Substitution zur Anwendung. Die Variable `notify` hat genau dann den Wert 1, wenn eine Protokolldatei rotiert werden musste.

Insgesamt sieht unser Skript jetzt so aus wie in Bild 4.4. – Die in diesem Skript verwendete Technik, eine »Konfigurationsdatei« zu lesen, die Zuweisungen an

Shellvariable enthält, ist sehr verbreitet. Linux-Distributionen verwenden sie gerne, die SUSE-Distributionen zum Beispiel für die Dateien in `/etc/sysconfig` und Debian GNU/Linux für die Dateien in `/etc/default`. Grundsätzlich kann in diesen Konfigurationsdateien alles auftreten, was in der Shell möglich ist; Sie tun jedoch gut daran, sich auf Variablenzuweisungen zu beschränken, die Sie möglicherweise durch Kommentarzeilen erklären können (einer der Hauptvorteile des Ansatzes).

Übungen



4.4 [1] Ändern Sie das Skript `multichecklog` so ab, dass der Name der Konfigurationsdatei wahlweise auch in der Umgebungsvariablen `MULTICHECKLOG_CONF` stehen kann. Vergewissern Sie sich, dass Ihre Änderung das tut, was sie soll.



4.5 [2] Wie würden Sie dafür sorgen, dass die Maximallänge der Protokolldateien bequem in der Form »12345« (Bytes), »12345k« (Kilobytes), »12345M« (Megabytes) angegeben werden kann? (*Tip*: `case`)



4.6 [3] Bisher benennt die `rotate`-Funktion die aktuelle Protokolldatei `$f` um in `$f.old`. Definieren Sie eine alternative `rotate`-Funktion, die beispielsweise 10 alte Versionen der Datei wie folgt verwaltet: Beim Rotieren wird `$f` umbenannt zu `$f.0`, eine etwa schon vorhandene Datei `$f.0` wird umbenannt in `$f.1` usw.; eine etwa vorhandene Datei `$f.9` wird gelöscht. (*Tip*: Das Kommando `seq` erzeugt Folgen von Zahlen.) Wenn Sie besonders gründlich sein wollen, machen Sie die Anzahl der alten Versionen konfigurierbar.



4.7 [2] Bei vielen Protokolldateien sind der Eigentümer, die Gruppe und der Zugriffsmodus wichtig. Erweitern Sie die `rotate`-Funktion so, dass die neu angelegte leere Protokolldatei für Eigentümer, Gruppe und Zugriffsmodus dieselben Einstellungen hat wie die alte.

Wichtige Ereignisse Ab und zu landen Meldungen im Protokoll, über die Sie als Systemverwalter möglichst bald informiert werden wollen. Natürlich haben Sie Dringenderes zu tun, als ständig das Protokoll zu beobachten – was läge also näher, als ein Shellskript damit zu beauftragen? Natürlich ist es unklug, einfach `/var/log/messages` periodisch mit `grep` zu durchsuchen, weil Sie dann mitunter mehrmals durch dasselbe Ereignis alarmiert werden. Sie können eher von einer Eigenschaft des Linux-`syslogd` profitieren, nämlich dass dieser auf Wunsch in eine *named pipe* schreibt, wenn Sie vor deren Namen einen vertikalen Balken setzen:

```
# syslog.conf
<<<<<<
*.*;mail.none;news.none |/tmp/logwatch
```

Die *named pipe* legen Sie natürlich vorher mit `mkfifo` an.

Ein einfaches Protokoll-Mitleseskript könnte also so aussehen:

```
#!/bin/bash

fifo=/tmp/logwatch
[ -p $fifo ] || ( rm -f $fifo; mkfifo -m 600 $fifo )

grep --line-buffered ALARM $fifo | while read LINE
do
    echo "$LINE" | mail -s ALARM root
done
```

Wir erzeugen zuerst die *named pipe*, falls sie nicht existiert. Anschließend wartet das Skript darauf, dass im Strom der Protokollzeilen eine auftaucht, die die

Zeichenkette »ALARM« enthält. Diese Zeile wird dann an den Systemverwalter geschickt.



Statt per Mail könnten Sie die Nachricht natürlich auch per SMS, Piepser, ... verschicken, je nachdem, wie dringend sie ist.

Kritisch dafür, dass das Skript wirklich funktioniert, ist die GNU-grep-Erweiterung `--line-buffered`. Sie sorgt dafür, dass `grep` seine Ausgabe zeilenweise schreibt, statt größere Mengen Ausgabe zu puffern, wie er das sonst aus Effizienzgründen beim Schreiben in Pipes tut. Eine Zeile könnte sonst eine halbe Ewigkeit brauchen, bis das `read` sie tatsächlich zu sehen bekommt.



Wenn Sie das `expect`-Paket von Don Libes installiert haben, verfügen Sie über ein Programm namens `unbuffer`, das die Ausgabe beliebiger Programme »entpuffert«. Sie könnten dann etwas schreiben wie

```
unbuffer grep ALARM $fifo | while read LINE
```

auch wenn `grep` die `--line-buffered`-Option nicht unterstützte.

Warum benutzen wir nicht einfach etwas wie

```
grep --line-buffered ALARM $fifo | mail -s ALARM root
```

? Ganz klar: Wir wollen ja, dass die Benachrichtigung möglichst umgehend erfolgt. Bei der einfachen Pipeline würde `mail` aber darauf warten, dass `grep` ihm das Dateiende signalisiert, um sicherzugehen, dass es alle Mailenswerte bekommen hat, bevor es die Mail tatsächlich verschickt. Die umständlichere »while-read-echo«-Konstruktion dient zur »Vereinzelung« der Nachrichten.

Statt dass Sie sich mühevoll einen regulären Ausdruck überlegen, der alle Ihre interessanten Protokollzeilen erfasst, können Sie übrigens auch die `-f`-Option von `grep` verwenden. Damit liest `grep` eine Reihe von regulären Ausdrücken aus einer Datei (einen pro Zeile) und sucht nach allen gleichzeitig:

```
grep --line-buffered -f /etc/logwatch.conf $fifo | ...
```

entnimmt die Suchmuster der Datei `/etc/logwatch.conf`. Mit einem Trick können Sie die Suchmuster aber auch in die `logwatch`-Datei selber aufnehmen:

```
grep <<ENDE --line-buffered -f - $fifo | while read LINE
ALARM
WARNUNG
KATASTROPHE
ENDE
do
    echo ...
done
```

Hier stehen die Muster in einem Hier-Dokument, das dem `grep`-Prozess auf dessen Standardeingabe zur Verfügung steht; der spezielle Dateiname »-« bringt die `-f`-Option dazu, die Musterdatei von der Standardeingabe zu lesen.

Übungen



4.8 [2] Welche andere Möglichkeit gibt es, `grep` nach mehreren regulären Ausdrücken gleichzeitig suchen zu lassen?

4.5 Systemadministration

Shellskripte sind ein wichtiges Werkzeug für die Systemadministration – sie erlauben es, stumpfsinnig wiederholte Vorgänge zu automatisieren oder selten Gebrautes in bequemer Form zugänglich zu machen, damit Sie es sich nicht immer von neuem überlegen müssen. Außerdem ist es möglich, sich zusätzliche Funktionalität zusammenzubauen, die das System nicht von sich aus mitbringt.

df mit Nachbrenner Das `df`-Kommando gibt aus, wieviel Platz auf den Dateisystemen noch frei ist. Leider ist die Ausgabe auf den ersten Blick recht unübersichtlich – es wäre oft schön, zum Beispiel die prozentuale Auslastung der Dateisysteme etwa als »Balkengrafik« visualisieren zu können. Nichts leichter als das: Hier ist die Ausgabe von `df` auf einem typischen System:

```
$ df
Filesystem      1K-blocks    Used Available Use% Mounted on
/dev/hda3       3842408    3293172   354048   91% /
tmpfs           193308      0    193308    0% /dev/shm
/dev/hda6       10886900   7968868   2364996   78% /home
/dev/hdc        714808     714808      0 100% /cdrom
```

»Grafisch« etwas aufbereitet könnte das so aussehen:

```
$ gdf
Mounted      Use%
/             91% #####-----
/dev/shm     0% -----
/home        78% #####-----
/cdrom       100% #####
```

Wir müssen uns also die 5. und 6. Spalte der `df`-Ausgabe holen und sie als 1. und 2. Spalte in die Ausgabe unseres `gdf`-Skripts schreiben. Der einzige Haken an der Sache ist, dass `cut` sich im Feld-Ausschneidemodus nicht am Leerzeichen als Feldtrenner orientieren kann: Zwei aneinandergrenzende Leerzeichen führen zu einem leeren Feld in der Zählung (versuchen Sie mal »`df | cut -d ' ' -f5,6`«). Statt mühselig die Zeichen in der Zeile zu zählen, um den Spalten-Ausschneidemodus von `cut` benutzen zu können, machen wir es uns einfach und ersetzen mit `tr` jede Folge von Leerzeichen durch ein Tabulatorzeichen. Unser `gdf`-Skript sieht dann – noch ohne die grafische Ausgabe – erst einmal so aus:

```
#!/bin/bash
# gdf -- "Grafische" Ausgabe von df (Vorversion)

df | tr -s ' ' '\t' | cut -f5,6 | while read pct fs
do
    printf "%-12s %4s " $fs $pct
done
```

Neu sind hier nur zwei Dinge: Die `read`-Anweisung liest das erste von `cut` ausgeschnittene Feld in die Variable `pct` und das zweite in die Variable `fs` (mehr über `read` erfahren Sie in Abschnitt 5.2). Und das `printf`-Kommando erlaubt die Ausgabe von Zeichenketten und Zahlen gemäß der Angaben in einem »Format« – hier »`%-12s %4s`« oder »eine linksbündige Zeichenkette in einem Feld von genau 12 Zeichen Breite (gegebenenfalls abgeschnitten oder mit Leerzeichen aufgefüllt) gefolgt von einem Leerzeichen und einer rechtsbündigen Zeichenkette in einem Feld von genau 4 Zeichen Breite (dito)«.

```
#!/bin/bash
# gdf -- "Grafische" Ausgabe von df (Fertige Version)

hash="#####"
dash="-----"

df "$@" | tr -s ' ' '\t' | cut -f5,6 | while read pct fs
do
    printf "%-12s %4s " $fs $pct
    if [ "$pct" != "Use%" ]
    then
        usedc=$(( ${#hash} * ${pct%}/100 )
        echo "${hash:0:$usedc}${dash:$usedc}"
    else
        echo ""
    fi
done
```

Bild 4.5: df mit Balkengrafik für die Plattenauslastung



printf steht als externes Programm zur Verfügung, ist aber (in leicht erweiterter Form) auch fest in die Bash eingebaut. Es gibt Dokumentation entweder als Handbuchseite (printf(1)), als Info-Dokument oder in der Bash-Anleitung; Details über die erlaubten Formate müssen Sie allerdings in der Dokumentation der C-Bibliotheksfunktion printf() nachschlagen (in printf(3)). Von printf wird offenbar angenommen, dass es so tief im kollektiven Unterbewusstsein der Unix-Benutzer verankert ist, dass es nicht mehr groß erklärt werden muss ...

Zur Lösung der Aufgabe fehlen uns nur noch die grafischen Balken, die sich natürlich aus der prozentualen Auslastung der Dateisysteme gemäß Spalte 5 (vulgo pct) ergeben. Die Balken selbst schneiden wir der Einfachheit halber aus vorgegebenen Zeichenketten der gewünschten Länge aus; wir müssen nur aufpassen, dass wir uns nicht an der Titelzeile (»Use%«) verschlucken. Nach dem printf steht also etwas wie

```
if [ "$pct" != "Use%" ]
then
    usedc=$(( ${#hash} * ${pct%}/100 )
    echo "${hash:0:$usedc}${dash:$usedc}"
else
    echo ""
fi
```

Dabei sind hash eine lange Kette von »#«-Zeichen und dash eine ebenso lange Kette von Strichen. usedc ergibt sich aus der Länge von hash – erhältlich über die spezielle Expansion \$#hash – multipliziert mit dem Prozentsatz der Auslastung geteilt durch 100, gibt also die Anzahl von »#«-Zeichen an, die im Balken angezeigt werden sollen. Den Balken selbst bekommen wir, indem wir \$usedc viele Zeichen aus hash ausgeben und gerade genug Zeichen aus dash anhängen, dass der Balken genauso lang ist wie hash. Das gesamte Skript steht in Bild 4.5; hash und dash sind je 60 Zeichen lang, was bei einer Standard-Terminalfensterbreite von 80 Zeichen zusammen mit dem Format den Platz gerade gut ausnutzt.

Übungen



4.9 [1] Warum steht in Bild 4.5 »df "\$@"«?



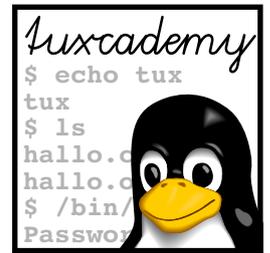
4.10 [3] Schreiben Sie eine Version von `gdf`, die die Länge des Balkens abhängig von der Größe des Dateisystems macht. Das größte Dateisystem soll über die ganze ursprüngliche Breite gehen und die anderen Dateisysteme einen proportional kürzeren Balken haben.

Kommandos in diesem Kapitel

<code>logrotate</code>	Verwaltet, kürzt und „rotiert“ Protokolldateien	<code>logrotate(8)</code>	66
<code>mkfifo</code>	Legt FIFOs (benannte Pipes) an	<code>mkfifo(1)</code>	71
<code>printf</code>	Gibt Zahlen und Zeichenketten formatiert aus	<code>printf(1)</code> , <code>bash(1)</code>	73
<code>seq</code>	Erzeugt Folgen von Zahlen auf der Standardausgabe	<code>seq(1)</code>	71
<code>tr</code>	Tauscht Zeichen in der Standardeingabe gegen andere aus oder löscht sie	<code>tr(1)</code>	73
<code>unbuffer</code>	Unterdrückt Ausgabepufferung eines Programms (Bestandteil des <code>expect</code> -Pakets)	<code>unbuffer(1)</code>	72

Zusammenfassung

- `grep` und `cut` sind nützlich, um bestimmte Zeilen und Spalten aus Dateien zu extrahieren.
- Sie sollten Fehlersituationen sorgfältig abfangen – sowohl beim Aufruf Ihres Skripts als auch während des Ablaufs.
- Ein- und Ausgabeumleitung für Folgen von Kommandos ist über explizite Subshells möglich.
- Die Kommandos `dirname` und `basename` erlauben Dateinamen-Manipulationen.
- Dateien mit Zuweisungen an Shellvariable können als bequeme »Konfigurationsdateien« für Shellskripte fungieren.
- Protokollnachrichten lassen sich einzeln verarbeiten, indem Sie sie mit `read` aus einer *named pipe* lesen.
- Das `printf`-Kommando erlaubt die formatierte Ausgabe von textuellen oder numerischen Daten.



5

Interaktive Shellskripte

Inhalt

5.1	Einleitung	78
5.2	Das Kommando read.	78
5.3	Menüauswahl mit select	80
5.4	»Grafische« Oberflächen mit dialog	84

Lernziele

- Techniken zur Steuerung von Shellskripten erlernen
- Die Bash-Methoden zur Benutzerinteraktion kennen
- Mit dem dialog-Paket umgehen können

Vorkenntnisse

- Kenntnisse über Shellprogrammierung (aus den vorigen Kapiteln)

5.1 Einleitung

In den vorigen Kapiteln haben Sie gelernt, wie Sie Shellskripte schreiben können, die Dateinamen und andere Informationen über die Kommandozeile erhalten. Das ist in Ordnung für Personen, die an die Kommandozeile gewöhnt sind – »normale« Benutzer schätzen aber oft einen »interaktiveren« Stil, bei denen ein Skript Fragen stellt oder eine Menüoberfläche anbietet. In diesem Kapitel lernen Sie, wie Sie dies mit der Bash realisieren können.

5.2 Das Kommando read

Das Shell-Kommando `read` haben wir ja schon im Vorübergehen kennengelernt: Es liest Zeilen von seiner Standardeingabe und weist sie an Shellvariable zu, in Konstruktionen wie

```
grep ... | while read line
do
    ...
done
```

Sie können (wie gesehen) mehrere Variable angeben. Die Eingabe wird dann in »Wörter« aufgeteilt, und die erste Variable bekommt das erste Wort als Wert, die zweite das zweite und so weiter:

```
$ read h w
Hallo Welt
$ echo $w $h
Welt Hallo
```

Überzählige Variable bleiben leer:

```
$ read a b c
1 2
$ echo $c
$ _
```

Nichts

Wenn mehr Wörter existieren als Variable, bekommt die letzte Variable den ganzen Rest:

```
$ read a b
1 2 3
$ echo $b
2 3
$ _
```

(Das ist natürlich das Geheimnis hinter dem Erfolg von »while read line«.)



read verträgt sich nicht gut mit Pipelines: Probieren Sie mal

```
$ echo Hallo Welt | read h w
$ echo $h $w
```

Nichts!?

Der Grund dafür: Die einzelnen Kommandos einer Pipeline führt die Shell in Unterprozessen aus, und auf Shellvariable aus einem Unterprozess kann in der »Eltershell« nicht zugegriffen werden (Sie erinnern sich!). Schleifenkonstruktionen wie `while` und `for` genießen einen speziellen Dispens.

Was ist ein »Wort«? Die Shell verwendet auch hier als mögliche Trennzeichen den Inhalt der Variablen IFS (kurz für *internal field separator*), zeichenweise betrachtet. Der Standardwert von IFS besteht aus dem Leerzeichen, dem Tabulatorzeichen und dem Zeilentrenner, aber Sie können sich oft das Leben erleichtern, indem Sie einen anderen Wert vergeben:

```
IFS=":"
cat /etc/passwd | while read login pwd uid gid gecost dir shell
do
    echo $login: $gecost
done
```

spart Ihnen das Jonglieren mit cut.

In der Bash können Sie das Lesen von der Tastatur auch mit einer Eingabeaufforderung kombinieren, wie im folgenden Skript zum Anlegen eines neuen Benutzers:

Eingabeaufforderung

```
#!/bin/bash
# newuser -- neuen Benutzer anlegen

read -p "Benutzername: " login
read -p "Bürgerlicher Name: " gecost

useradd -c "$gecost" $login
```

Gerade bei Verwendung von read ist es wichtig, die »eingeleseenen« Variablen in Anführungszeichen zu setzen, um Probleme mit Leerzeichen zu vermeiden.

Was passiert, wenn der Aufrufer des newuser-Skripts keine gültige Eingabe macht? Wir könnten zum Beispiel fordern, dass der Benutzername des neuen Benutzers nur aus Kleinbuchstaben und Ziffern besteht (eine gängige Konvention). Umsetzen könnten wir das wie folgt:

Plausibilitätskontrolle

```
read -p "Benutzername: " login

test=$(echo "$login" | tr -cd 'a-z0-9')
if [ -z "$login" -o "$login" != "$test" ]
then
    echo >&2 "Ungültiger Benutzername \"$login\""
    exit 1
fi
```

Hier betrachten wir, was übrig bleibt, wenn wir aus dem vorgeschlagenen Benutzernamen alle ungültigen Zeichen entfernen. Ist das Ergebnis ungleich dem ursprünglichen Benutzernamen, dann enthält letzterer »verbotene« Zeichen. Leer darf er auch nicht sein, was wir mit der -z-Bedingung von test prüfen.

Bei durchgreifenden Operationen wie dem Anlegen neuer Benutzer empfiehlt sich eine Sicherheitsabfrage am Schluss, damit der Aufrufer das Skript abbrechen kann, falls er »kalte Füße« bekommen hat (etwa wegen einer Fehleingabe weiter vorne). Eine bequeme Möglichkeit dazu ist über eine Shellfunktion der folgenden Form:

Sicherheitsabfrage

```
function confirm () {
    while read -p "Bitte bestätigen (j/n): " answer
    do
        case $answer in
            [Jj]*) result=0; break ;;
            [Nn]*) result=1; break ;;
            *) echo "Bitte antworten Sie mit 'ja' oder 'nein'" ;;
        esac
    done
}
```

```
done
return $result
}
```

Die Sicherheitsabfrage im Skript wäre dann etwas wie

```
confirm && useradd ...
```

Übungen

 **5.1** [!1] Ändern Sie das `newuser`-Skript so, dass es prüft, ob der bürgerliche Name des neuen Benutzers einen Doppelpunkt enthält, und gegebenenfalls eine Fehlermeldung ausgibt und abbricht.

 **5.2** [2] Erweitern Sie das `newuser`-Skript so, dass der Aufrufer die Login-Shell des neuen Benutzers wählen kann. Sorgen Sie dafür, dass nur Shells aus `/etc/shells` akzeptiert werden.

 **5.3** [2] Erweitern Sie die Shellfunktion `confirm` so, dass sie die Eingabeaufforderung als Parameter übergeben bekommt. Wenn kein Parameter übergeben wurde, soll die Standardaufforderung »Bitte bestätigen« ausgegeben werden.

 **5.4** [3] Schreiben Sie ein einfaches Ratespiel: Der Computer wählt eine Zufallszahl zwischen (zum Beispiel) 1 und 100 (die Shellvariable `RANDOM` liefert Zufallszahlen). Der Benutzer darf eine Zahl eingeben und der Computer antwortet mit »Zu groß« oder »Zu klein«. Dies wird wiederholt, bis der Benutzer die richtige Zahl gefunden hat.

5.3 Menüauswahl mit `select`

Die Bash hat ein sehr mächtiges Kommando zur Auswahl von Optionen aus einer Liste, `select`, mit einer Syntax ähnlich der von `for`:

```
select <Variable> [in <Liste>]
do
    <Kommandos>
done
```

Diese Konstruktion beschreibt eine Schleife, bei der der Wert von `<Variable>` sich durch eine Benutzerauswahl aus `<Liste>` ergibt. Die Schleife wird immer wieder durchlaufen, bis das Dateiende auf der Standardeingabe erreicht ist. Sie können sich das etwa so vorstellen:

```
$ select typ in Hamburger Cheeseburger Fischburger
> do
>     echo $typ kommt
> done
1) Hamburger
2) Cheeseburger
3) Fischburger
#? 2
Cheeseburger kommt
#? 3
Fischburger kommt
#?  + 
$ _
```

Das heißt, die Bash präsentiert die Einträge der *<Liste>* mit vorgestellten Nummern, und der Benutzer kann über eine der Nummern eine Auswahl treffen.



select benimmt sich sehr wie for: Wird die *<Liste>* weggelassen, präsentiert es die Positionsparameter der Bash zur Auswahl. Die select-Schleife kann wie alle anderen Shell-Schleifen auch mit break oder continue abgebrochen werden.

newuser neu aufgegossen Wir können select verwenden, um unser newuser-Skript noch weiter zu verfeinern. Beispielsweise könnten Sie verschiedene Typen von Benutzern unterstützen wollen, an einer Universität etwa die Professoren, die Assistenten und die Studenten. In diesem Fall wäre es nützlich, wenn das newuser-Skript eine Auswahl der verschiedenen Benutzertypen anbieten würde. Aus der Gruppenwahl ergeben sich dann jeweils andere Voreinstellungen für die Benutzer, etwa die Unix-Gruppenzuordnung, das Heimatverzeichnis oder die Grundausstattung an Dateien im Heimatverzeichnis. Bei dieser Gelegenheit können Sie auch gleich noch eine weitere Möglichkeit dafür kennenlernen, Konfigurationsdaten für Shellskripte zu speichern.

Wir gehen davon aus, dass sich im Verzeichnis /etc/newuser für jeden Typ von Benutzern *t* eine Datei namens *t* befindet, für Professoren zum Beispiel /etc/newuser/Professor und für Studenten /etc/newuser/Student. Der Inhalt von /etc/newuser/Professor könnte zum Beispiel so aussehen:

```
# /etc/newuser/Professor
GROUP=profs
EXTRAGROUP=dekanat
HOMEDIR=/home/$GROUP
SKELDIR=/etc/skel-$GROUP
```

(Professoren werden in die Linux-Gruppe profs als primäre Gruppe gesteckt und bekommen außerdem dekanat als zusätzliche Gruppe). Das ist natürlich unser alter Trick »Konfigurationsdatei mit Shellvariablenzuweisungen«, mit dem Unterschied, dass es jetzt für jeden Benutzertyp eine eigene Konfigurationsdatei gibt. Das Anlegen eines neuen Benutzers, wenn wir den Benutzertyp wissen, geht dann ungefähr wie

```
confirm || exit 0

./etc/newuser/$type
useradd -c "$gecos" -g $GROUP -G $EXTRAGROUP \
-m -d $HOMEDIR/$login -k $SKELDIR $login
```

Uns bleibt noch die Auswahl des passenden Benutzertyps. Die Auswahlliste wollen wir natürlich nicht hart im newuser-Skript codieren, sondern vom Inhalt von /etc/newuser abhängig machen:

```
echo "Die folgenden Benutzertypen sind verfügbar:"
PS3="Benutzertyp: "
select type in $(ls /etc/newuser) '[Abbrechen]';
do
    [ "$type" = "[Abbrechen]" ] && exit 0
    [ -n "$type" ] && break
done
```

Die Shellvariable PS3 gibt die Eingabeaufforderung an, die von select ausgegeben wird.

Übungen



5.5 [!1] Was passiert, wenn Sie bei select etwas eingeben, das nicht mit der Nummer eines Menüeintrags korrespondiert?

Wer wird ... Unser nächstes Skript ist lose an eine populäre Ratesendung im Fernsehen angenähert: Gegeben ist eine Datei `wmm.txt` mit Fragen und Antworten in der Form

```
0?:Was hat die Morgenstund im Sprichwort?
0:-:Blei im Hintern
0:-:Eisen im Bauch
0+:Gold im Mund
0:-:Silber im Kopf
0>:50
50?:Wovor sollten Sie sich beim Badeurlaub in Acht nehmen?
50:-:Hallowal
50+:Haifisch
50:-:Huhurobbe
50:-:Grüezischildkröte
50>:100
```

Das Skript – nennen wir es `wmm` – soll, beginnend beim Punktwert 0, die Fragen und Antworten präsentieren. Bei einer falschen Antwort endet das Programm, bei einer richtigen Antwort geht es mit der nächsten Frage weiter (deren Punktzahl sich aus der »>:«-Zeile ergibt).

Ein wichtiger Grundgedanke in komplizierteren Programmierprojekten ist **Abstraktion**. In unserem Fall versuchen wir, das konkrete Format der Fragendatei in einer Funktion zu »verstecken«, die die jeweils richtigen Elemente heraussucht. Theoretisch könnten wir dann später die Fragen statt einer einfachen Textdatei zum Beispiel einer Datenbank entnehmen oder die Datenhaltung auf andere Weise ändern (etwa indem wir pro Punktstufe zufällig eine Frage aus einer Mehrzahl auswählen). Eine mögliche, wenn auch nicht übermäßig effiziente Methode zum Zugriff auf die Fragendaten geht so:

```
qfile=wmm.txt

function question () {
  if [ "$1" = "get" ]
  then
    echo "$2"
    return
  fi
  case "$2" in
    display) re='?' ;;
    answers) re='[-+]' ;;
    correct) re='+' ;;
    next)    re='>' ;;
    *)      echo >&2 "$0: get: ungültiger Feldtyp $2"; exit 1 ;;
  esac
  grep "^$1:$re:" $qfile | cut -d: -f3
}
```

Die Funktion `question` muss zunächst in der Form

```
q=$(question get <Punktzahl>)
```

aufgerufen werden. Sie liefert dann als Ausgabe die eindeutige Bezeichnung einer Frage mit der angegebenen Punktzahl (bei uns `simpel` – es gibt pro Punktzahl nur

eine Frage in der Datei, so dass wir einfach die Punktzahl als »Fragennummer« zurückgeben). Diese Bezeichnung merken wir uns in einer Shellvariablen (hier q). Anschließend stehen uns die folgenden Aufrufe zur Verfügung:

question \$q display	<i>liefert den Fragentext</i>
question \$q answers	<i>liefert alle Antworten, eine pro Zeile</i>
question \$q correct	<i>liefert die richtige Antwort</i>
question \$q next	<i>liefert die Punktzahl bei richtiger Antwort</i>

Alle Daten stehen auf der Standardausgabe zur Verfügung.



Für Kenner: Dies sind natürlich die Grundzüge eines »objektbasierten« Ansatzes – »question get« liefert ein »Fragenobjekt«, das dann die verschiedenen Methoden display usw. unterstützt.

Als nächstes brauchen wir eine Funktion, die eine Frage anzeigt und die Antwort abholt. Diese Funktion baut natürlich auf unserer eben gezeigten question-Funktion auf:

```
function present () {
  # Finde und zeige die Frage
  question $1 display
  # Finde die richtige Antwort
  rightanswer=$(question $1 correct)
  # Zeige die Antworten
  PS3="Ihre Antwort: "
  IFS=$'\n'
  select answer in $(question $1 answers)
  do
    if [ -z "$answer" ]
    then
      echo "Bitte geben Sie etwas Vernünftiges ein."
    else
      test "$answer" = "$rightanswer"
      return
    fi
  done
}
```

Die Antworten präsentieren wir natürlich mit select. Dabei müssen wir darauf achten, dass select die IFS-Variable benutzt, um beim Aufstellen des Menüs die einzelnen Menüpunkte voneinander zu trennen – mit dem Standardwert von IFS würde select jedes einzelne Wort aller Antworten als Menüpunkt anzeigen (!). Probieren Sie es aus! Für den Rückgabewert der Funktion nutzen wir die Tatsache aus, dass der Rückgabewert des letzten »echten« Kommandos (return zählt nicht) als Rückgabewert der Funktion gilt. Statt eines umständlichen

```
if [ "$answer" = "$rightanswer" ]
then
  return 0
else
  return 1
fi
```

verwenden wir also die oben gezeigte Konstruktion mit einem return nach einem test.

Als letztes bleibt uns noch der »Rahmen«, der die beiden getrennten Programmteile »Fragenverwaltung« und »Benutzungsoberfläche« zusammenbringt. Der könnte ungefähr so aussehen:

```

score=0
while [ $score -ge 0 -a $score -lt 1000000 ]
do
    q=$(question get $score)
    if present $q
    then
        score=$(question $q next)
    else
        score=-1
    fi
done

```

Das Rahmenprogramm kümmert sich darum, eine Frage auszusuchen (mit »question get«), die zur aktuellen Punktzahl des Teilnehmers passt. Diese Frage wird angezeigt (mit present) und in Abhängigkeit vom »Erfolg« der Anzeige (also der Richtigkeit der Antwort) wird die Punktzahl entweder auf die nächste Stufe erhöht, oder das Spiel ist zu Ende. Zum Schluss nur noch die warmen Abschiedsworte des (computerisierten) Showmasters:

```

if [ $score -lt 0 ]
then
    echo "Das war wohl nicht so prall, leider verloren"
else
    echo "Herzlichen Glückwunsch, Sie haben gewonnen"
fi

```

Übungen

-  **5.6** [!1] Erweitern Sie das wmm-Skript so, dass es den »Punktwert« der Frage anzeigt (also die Punktzahl, die der Teilnehmer haben wird, wenn er die Frage richtig beantworten kann).
-  **5.7** [!3] Überlegen Sie sich eine interessante Erweiterung von wmm und implementieren Sie diese (oder zwei oder drei).
-  **5.8** [3] Überarbeiten Sie die question-Funktion von wmm so, dass sie mit weniger grep-Aufrufen auskommt.

5.4 »Grafische« Oberflächen mit dialog

Statt langweiliger Textmenüs und Fernschreiber-Dialogen können Sie in Ihren Skripten auch auf eine fast »grafische« Oberfläche zurückgreifen. Hierzu dient das Programm dialog, das Sie allerdings unter Umständen extra installieren müssen, wenn Ihre Linux-Distribution das nicht für Sie tut. dialog nutzt die Fähigkeiten moderner Terminals (oder Terminal-Emulationsprogramme), um Menüs, Auswahllisten, Texteingabefelder und ähnliches bildschirmfüllend und farbig zu präsentieren.

dialog kennt eine ganze Reihe von Interaktions-Elementen (Tabelle 5.1). Die Details der Konfiguration sind durchaus komplex und Sie sollten sie in der Dokumentation zu dialog nachlesen; wir beschränken uns hier auf das Allernötigste.

Wir können zum Beispiel unser wmm-Programm so ändern, dass die Fragen und die Abschlussauswertung mit dialog angezeigt werden. Für das Anzeigen unserer Fragen und Antworten ist das menu-Element am besten geeignet. Ein dialog-Aufruf für ein Menü sieht ungefähr so aus:

Tabelle 5.1: Interaktions-Elemente von dialog

	Beschreibung
calendar	Zeigt Tag, Monat und Jahr in getrennten Fenstern; der Benutzer kann editieren. Liefert den eingestellten Wert in der Form »Tag/Monat/Jahr«
checkboxlist	Zeigt eine Liste von Einträgen, die individuell ein- und ausgeschaltet werden können. Liefert eine Liste der »eingeschalteten« Einträge
form	Stellt ein Formular dar. Liefert die eingetragenen Werte, einen pro Zeile
fselect	Zeigt einen Dateiauswahldialog. Liefert den gewählten Dateinamen
gauge	Zeigt einen Fortschrittsbalken
inbox	Gibt eine Nachricht aus (ohne den Bildschirm zu löschen)
inputbox	Erlaubt die Eingabe einer Zeichenkette, liefert diese
inputmenu	Zeigt ein Menü, bei dem der Anwender die Einträge umbenennen kann
menu	Zeigt ein Auswahlmenü
msgbox	Gibt eine Nachricht aus, wartet auf Bestätigung
passwordbox	inputbox, die die Eingabe nicht anzeigt
radiolist	Zeigt eine Liste von Einträgen, von denen genau einer ausgewählt sein kann; liefert den ausgewählten Eintrag
tailbox	Zeigt den Inhalt einer Datei, à la »tail -f«
tailboxbg	Wie tailbox, Datei wird im Hintergrund gelesen
textbox	Zeigt den Inhalt einer Textdatei
timebox	Zeigt Stunde, Minute und Sekunde mit Editiermöglichkeit; liefert Zeit im Format »Stunde:Minute:Sekunde«
yesno	Gibt eine Nachricht aus und erlaubt eine Antwort mit »Ja« oder »Nein«

**Bild 5.1:** Ein Menü mit dialog

```
$ dialog --clear --title "Menüauswahl" \
--menu "Was darf es sein?" 12 40 4 \
  "H" "Hamburger" \
  "C" "Cheeseburger" \
  "F" "Fischburger" \
  "V" "Veggieburger"
```

Das Resultat sehen Sie in Bild 5.1. Die wichtigeren Optionen sind `--clear` (löscht den Bildschirm vor der Anzeige des Menüs) und `--title` (gibt einen Titel für das Menü an). Die Option `--menu` legt fest, dass dieser Dialog ein Auswahlmenü sein soll; danach folgen der erklärende Text für das Menü und die drei magischen Zahlen »Höhe des Menüs in Zeilen«, »Breite des Menüs in Zeichen« und »Anzahl von gleichzeitig angezeigten Einträgen«. Zum Schluss kommen die einzelnen Einträge, und zwar immer mit einem »Kurznamen« und dem eigentlichen Eintragsinhalt. Im angezeigten Menü können Sie über die Pfeiltasten navigieren, über die Zifferntasten `0` bis `9` und die Anfangsbuchstaben der Kurznamen; in unserem Beispiel würden etwa die Eingaben `3` und `f` zum »Fischburger« führen.

Ein weiterer Umstand ist wichtig im Umgang mit `dialog`: Das Programm liefert seine Ausgabewerte in der Regel auf der Standardfehlerausgabe. Das heißt, wenn Sie, was wahrscheinlich ist, die `dialog`-Ergebnisse auffangen und weiterverarbeiten wollen, müssen Sie die Standardfehlerausgabe von `dialog` umleiten, nicht die Standardausgabe. Das ergibt sich daraus, dass `dialog` auf seine Standardausgabe schreibt, um das Terminal anzusteuern; wenn Sie also die Standardausgabe umlenken, sehen Sie nichts mehr auf dem Bildschirm, und die eigentlichen Ausgabewerte von `dialog` würden unter diversen Terminalsteuerzeichen untergehen. Es gibt verschiedene Möglichkeiten, damit umzugehen: Sie können `dialog` über `2>` in eine temporäre Datei schreiben lassen, aber es ist mühselig, sich darum zu kümmern, dass diese temporären Dateien am Ende des Skripts wieder weggeräumt werden (Stichwort: `trap`). Alternativ dazu können Sie auch mit Ausgabeumlenkung kreativ werden, etwa so:

```
result=$(dialog ... 2>&1 1>/dev/tty)
```

Dies verbindet die Standardfehlerausgabe (Dateideskriptor 2) mit dem Ziel der aktuellen Standardausgabe (die Variable `result`) und danach die Standardausgabe mit dem Terminal (`/dev/tty`).



Das funktioniert natürlich nur, weil die Standardausgabe auf anderen Geräten als Terminals nicht zu gebrauchen ist – wenn wir die Standardausgabe und die Standardfehlerausgabe eines Skripts vertauschen wollen, brauchen wir eine »Ringtauschkonstruktion« etwa der folgenden Form:

```
( programm 3>&1 1>output 2>&3 ) | ...
```

Diese Kommandozeile leitet die Standardausgabe von `programm` in die Datei `output` und die Standardfehlerausgabe in die Pipeline. Sie ist damit quasi das Gegenteil zum gängigeren

```
programm 2>output | ...
```

Wer wird ... mit `dialog` Schauen wir uns jetzt eine `dialog`-basierte Version des `wm`-Skripts an. Hier lohnt sich die Mühe, die wir vorher in Sachen »Abstraktion« betrieben haben; die Änderungen beschränken sich im Wesentlichen auf die Funktion `present`.

Die Haupthürde, die wir überspringen müssen, um `wm dialog`-fähig zu machen, liegt darin begründet, dass in der Menüsyntax

```
dialog ... --menu t h w n k_0 e_0 k_1 e_1 ...
```

das dialog-Programm darauf besteht, die Menü-Kurznamen und -Einträge k_i und e_i als einzelne Wörter übergeben zu bekommen. Wir können zwar mit einer Schleife wie

```
items=''
i=1
question $q answers | while read line
do
    items="$items $i '$line'"
    i=$((i+1))
done
```

in der Variablen items lediglich einfach eine Liste von Kurznamen und Antworten der Form

```
1 'Blei im Hintern' 2 'Eisen im Bauch' ...
```

konstruieren, aber einen Aufruf der Form

```
dialog ... --menu 10 60 4 $items
```

mag dialog gar nicht gerne. Eine Lösung liegt in der Verwendung von **Feldern** (engl. *arrays*), die die Bash zumindest in eingeschränkter Form unterstützt.

Felder Ein Feld ist eine Variable, die eine Folge von Werten enthalten kann. Diese Werte werden durch numerische Indizes angesprochen. Sie müssen ein Feld nicht gesondert vereinbaren, sondern es reicht, wenn Sie eine Variable »indiziert« ansprechen:

```
$ gang[0]=Aperitif
$ gang[1]=Suppe
$ gang[2]=Fisch
$ gang[4]=Nachtisch
```

Zugreifen können Sie auf diese Variablen entweder einzeln, wie in

```
$ echo ${gang[1]}
Suppe
```

(die geschweiften Klammern sind nötig, damit keine Verwirrung mit Dateisuchmustern aufkommen kann) oder insgesamt, wie in

```
$ echo ${gang[*]}
Aperitif Suppe Fisch Nachtisch
```

(Dass nicht alle Indizes fortlaufend benutzt sind, ist übrigens egal.) Die Expansionen » $\${Name}[*]$ « und » $\${Name}[@]$ « sind analog zu $*$ und $@$, wie das folgende Beispiel illustriert:

```
$ gang[3]="Wiener Schnitzel"
$ for i in ${gang[*]}; do echo $i; done
Aperitif
Suppe
Fisch
Wiener
```

```
Schnitzel
Nachtisch
$ for i in "${gang[*]"; do echo $i; done
Aperitif Suppe Fisch Wiener Schnitzel Nachtisch
$ for i in "${gang[@]"; do echo $i; done
Aperitif
Suppe
Fisch
Wiener Schnitzel
Nachtisch
```

Letzteres ist das, was wir brauchen.



Sie können einem Feld auch im ganzen einen Wert zuweisen, etwa so:

```
$ gang=(Aperitif Gazpacho Lachs "Boeuf Stroganoff" "Crepes Suzette")
$ for i in "${gang[@]"; do echo $i; done
Aperitif
Gazpacho
Lachs
Boeuf Stroganoff
Crepes Suzette
$ gang=([4]=Pudding [2]=Hecht [1]=Brühe [3]=Frikassee [0]=Aperitif)
$ for i in "${gang[@]"; do echo $i; done
Aperitif
Brühe
Hecht
Frikassee
Pudding
```



Wenn Sie gründlich sein wollen, können Sie eine Variable mit

```
declare -a <Name>
```

offiziell zu einem Feld erklären. Nötig ist das normalerweise nicht.

Angewandte Felder Unsere neue present-Routine muss also in einem Feld die Liste von Menü-Kurznamen und Antworten aufbauen, die später an dialog übergeben wird. Aussehen könnte das so:

```
declare -a answers
i=0
rightanswer=$(question $1 correct)
IFS=$'\n'
for a in $(question $1 answers)
do
  answers[$((2*i))]=$((i+1))
  answers[$((2*i+1))="$a"
  [ "$a" = "$rightanswer" ] && rightshort=$((i+1))
  i=$((i+1))
done
```

Wir verwenden *i* als Index in das Feld `answers`. Für jede Antwort wird *i* um 1 erhöht, und die Plätze für den Kurznamen und die eigentliche Antwort ergeben sich aus einer Indextransformation: Beispielsweise landet für *i* = 1 der Kurzname in `answers[2]` und der Antworttext in `answers[3]`; für *i* = 3 der Kurzname in `answers[6]` und der Antworttext in `answers[7]`. Als Kurznamen verwenden wir allerdings die Zahlen 1, ..., 4 statt 0, ..., 3. Ferner merken wir uns den Kurznamen der richtigen

```
#!/bin/bash
# dwm -- wwm mit dialog

# Die question-Funktion ist genau wie in wwm
<<<<<<

function present () {
    declare -a answers
    i=0
    rightanswer=$(question $1 correct)
    IFS=$'\n'
    for a in $(question $1 answers)
    do
        answers[$((2*i))]=$((i+1))
        answers[$((2*i+1))="$a"
        [ "$a" = "$rightanswer" ] && rightshort=$((i+1))
        i=$((i+1))
    done

    # Zeige die Frage
    sel=$(dialog --clear --title "Für $(question $1 next) Punkte" \
        --no-cancel --menu "$(question $1 display)" 10 60 4 \
            ${answers[@]} 2>&1 1>/dev/tty)
    test "$sel" = "$rightshort"
}

# Das Hauptprogramm ist genau wie in wwm
<<<<<<

if [ $score -lt 0 ]
then
    msg="Das war wohl nicht so prall, leider verloren"
else
    msg="Gratuliere, Sie haben gewonnen"
fi
dialog --title "End-Ergebnis" --msgbox "$msg" 10 60
```

Bild 5.2: Eine Version von wwm mit dialog

Antwort in rightshort; das ist wichtig, da dialog uns nur den Kurznamen des gewählten Menüeintrags zurückliefert und nicht den vollen Eintrag, wie select das macht.

Mit unserem answers-Feld können wir jetzt dialog aufrufen:

```
# Zeige die Frage
sel=$(dialog --clear --title "Für $(question $1 next) Punkte" \
    --no-cancel --menu "$(question $1 display)" 10 60 4 \
        ${answers[@]} 2>&1 1>/dev/tty)
test "$sel" = "$rightshort"
```

Hier holen wir uns den Fragentext wieder über die display-Methode von question; die Option --no-cancel unterdrückt den »Abbrechen«-Knopf im Menü.

Zu guter Letzt können wir dialog auch benutzen, um das Gesamtergebnis bekanntzugeben:

```
if [ $score -lt 0 ]
```

```

then
    msg="Das war wohl nicht so prall, leider verloren"
else
    msg="Gratuliere, Sie haben gewonnen"
fi
dialog --title "End-Ergebnis" --msgbox "$msg" 10 60

```

Dazu dient die wesentlich einfachere `--msgbox`-Option. Die wesentlichen geänderten Stellen für das `dialog`-basierte Skript (es heißt entsprechend `dwwm`) sind in Bild 5.2 übersichtlich gezeigt.

Ausblick `dialog` ist nützlich, aber Sie sollten es damit wahrscheinlich nicht übertreiben. Haupteinsatzzweck für `dialog`-basierte Programme ist vermutlich die Grauzone, wo etwas Angenehmeres gewünscht ist als rohe Textterminal-Interaktion, aber eine »echte« Grafikoberfläche nicht oder nicht notwendigerweise zur Verfügung steht. Beispielsweise verwendet die Installationsroutine von Debian GNU/Linux `dialog` – für eine komplette GUI-Umgebung ist auf den Bootdisketten nicht wirklich Platz. Komplexere grafische Oberflächen liegen jenseits von dem, was auch mit `dialog` möglich wäre, und sind die Domäne von Umgebungen wie Tcl/Tk oder Programmen auf der Basis von C oder C++ (etwa mit Qt oder Gtk+, KDE oder GNOME).



Zur Darstellung einfacher Dialogboxen (mit Knöpfen zum Anklicken) unter X11 dient das Programm `xmessage`. Hiermit können Sie zum Beispiel Nachrichten für Benutzer anzeigen (siehe Übung 5.10). `xmessage` ist ein Standardbestandteil von X11 und sollte daher auf praktisch jedem Linux-System mit einer Grafikoberfläche zur Verfügung stehen, auch wenn sein Aussehen wahrscheinlich nicht modernen ästhetischen Ansprüchen genügt.



KDE bietet ein vage an `dialog` angelehntes Programm namens `kdiallog`, das KDE-artige GUIs aus Shellskripten erlaubt. Wir würden Ihnen für alles, was über ganz elementare Dinge hinausgeht, allerdings dringend zu einer »vernünftigen« Basis für Ihre GUI-Programme raten, etwa Tcl/Tk oder PyKDE.

Übungen



5.9 [!3] Schreiben Sie ein Shellskript `seluser`, das Ihnen ein Auswahlmü mit allen Benutzern aus `/etc/passwd` anbietet (verwenden Sie den Benutzer-namen als Kurznamen und den Inhalt des GECOS-Feldes – den bürgerlichen Namen – als Eintrag). Das Skript sollte den gewählten Benutzernamen auf der Standardausgabe liefern. (Stellen Sie sich vor, das Skript wäre ein Bestandteil eines umfassenderen Werkzeugs zur Benutzerverwaltung.)



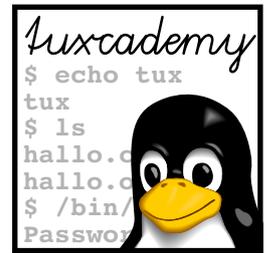
5.10 [3] Schreiben Sie ein Shellskript `show-motd`, das `xmessage` verwendet, um den Inhalt von `/etc/motd` anzuzeigen. Sorgen Sie dafür, dass dieses Shellskript gestartet wird, wenn ein Benutzer sich anmeldet (*Tipp*: `xsession`).

Kommandos in diesem Kapitel

dialog	Erlaubt „grafische“ Interaktionselemente auf einem Textbildschirm	<code>dialog(1)</code>	84
kdiallog	Erlaubt Benutzung von KDE-Interaktionselementen von Shellskripten aus	<code>kdiallog(1)</code>	90
xmessage	Zeigt eine Nachricht oder Anfrage in einem X11-Fenster an	<code>xmessage(1)</code>	90

Zusammenfassung

- Mit `read` können Sie Daten von Dateien, Pipelines oder der Tastatur in Shell-variable einlesen.
- Das `select`-Kommando erlaubt eine komfortable wiederholte Auswahl aus einer nummerierten Liste von Alternativen.
- Das Programm `dialog` macht es möglich, Shellskripte auf Textterminals mit an GUIs angelehnten Interaktions-Elementen zu versehen.



6

Der Stromeditor sed

Inhalt

6.1	Einsatzgebiete	94
6.2	Adressierung	94
6.3	sed-Anweisungen	96
6.3.1	Ausgeben und Löschen von Zeilen	96
6.3.2	Einfügen und Verändern	97
6.3.3	Zeichen-Transformationen	98
6.3.4	Suchen und Ersetzen	98
6.4	sed in der Praxis	99

Lernziele

- Funktion und Einsatzgebiete von sed kennen
- Einfache sed-Skripte formulieren können
- sed in Shellskripten verwenden können

Vorkenntnisse

- Kenntnisse der Shellprogrammierung (etwa aus den vorigen Kapiteln)
- Reguläre Ausdrücke

6.1 Einsatzgebiete

Linux verfügt über ein reiches Angebot an simplen Textbearbeitungswerkzeugen – von `cut` und `grep` über `tr`, `sort` bis zu `join` und `paste`. Werden die Aufgaben der Text-Manipulation jedoch anspruchsvoller, so reichen die klassischen Werkzeuge wie `cut` und `tr` nicht mehr aus. Beispielsweise kann `tr` Ihnen ein »X« für ein »U« vormachen, aber am alten Alchimisten-Traum, »Blei« in »Gold« zu verwandeln, scheitert es wie diese.

Normalerweise würden Sie jetzt einen hinreichend komfortablen Editor starten, um jedes Vorkommen von »Blei« in Ihrer Datei in »Gold« zu transformieren. Es gibt jedoch zwei Gründe, die es nahelegen könnten, einen anderen Weg einzuschlagen.

Erstens könnte es sein, dass Sie die Datei unbeaufsichtigt automatisch verändern wollen, beispielsweise in einem Shell-Skript. Zweitens gibt es Fälle, wo Sie gar keine *Datei* bearbeiten, sondern einen (potenziell unendlichen) Textstrom, etwa in der Mitte einer Pipeline.

sed Hier bietet sich `sed` (für engl. *stream editor*) an, der es ermöglicht, einen Textstrom nach vorgegebenen Anweisungen zu bearbeiten. In Fällen, wo auch `sed` nicht ausreicht, können Sie weiter aufrüsten und `awk` nehmen (Kapitel 7), oder Sie steigen gleich auf eine Skript-Sprache wie Perl um.

Programmiermodell Das Programmiermodell von `sed` ist einfach: Das Programm bekommt eine Menge von Anweisungen übergeben, liest seine Eingabe zeilenweise und wendet die Anweisungen da, wo angebracht, auf die Eingabezeilen an. Anschließend werden die Zeilen in gegebenenfalls manipulierter Form ausgegeben (oder auch nicht). Wichtig ist, dass `sed` seine Eingabedatei (falls es eine gibt) nur liest und in keinem Fall verändert; allenfalls werden veränderte *Daten* ausgegeben.

Anweisungen `sed` akzeptiert Anweisungen entweder eine nach der anderen mit der mehrmals auf der Kommandozeile zulässigen `-e`-Option oder gebündelt in einer Datei, deren Name mit der `-f`-Option übergeben wird. Wenn Sie nur eine einzige Anweisung auf der Kommandozeile übergeben wollen, können Sie sich das `-e` auch sparen.



Innerhalb einer `-e`-Option können Sie mehrere `sed`-Kommandos angeben, wenn Sie sie mit Semikolons trennen.



Grundsätzlich spricht natürlich nichts gegen ausführbare »sed-Skripte« der Form

```
#!/bin/sed -f
<sed-Kommandos>
```

Zeilenadresse Jede `sed`-Anweisung besteht aus einer Zeilenadresse, die die Zeilen bestimmt, auf die die Anweisung sich bezieht, und der eigentlichen Anweisung, zum Beispiel

```
$ sed -e '1,15y/uU/xX/'
```

`sed`-Anweisungen sind genau ein Zeichen lang, können aber weitere Parameter nach sich ziehen (je nach Anweisung).

6.2 Adressierung

Es gibt verschiedene Möglichkeiten, Zeilen in `sed` zu »adressieren«. Manche Kommandos wirken auf eine Zeile, manche auch auf Bereiche von Zeilen. Einzelne Zeilen können Sie wie folgt auswählen:

Zeilennummern Eine Zahl als Zeilenadresse wird als Zeilennummer interpretiert. Die erste Zeile der Eingabe hat die Nummer 1, und dann wird fortlaufend weitergezählt (auch wenn die Eingabe aus mehreren Dateien besteht).



Mit der `-s`-Option können Sie `sed` dazu bringen, jede Eingabedatei getrennt zu betrachten. In diesem Fall beginnt jede Eingabedatei mit einer Zeile Nr. 1.

Eine Zeilenspezifikation der Form `i-j` steht für »beginnend bei Zeile *i* jede *j*-te Zeile«. Mit »2-2« würden Sie also jede gerade nummerierte Zeile auswählen.

Reguläre Ausdrücke Ein regulärer Ausdruck der Form `»/⟨Ausdruck⟩/«` selektiert alle Zeilen, die auf den Ausdruck passen. `»/a.*a.*a/«` wählt zum Beispiel alle Zeilen aus, die mindestens drei »a« enthalten.

Letzte Zeile Das Dollarzeichen (`»$«`) steht für die letzte Zeile der letzten Eingabedatei (auch hier betrachtet `-s` jede Eingabedatei separat).

Bereiche von Zeilen können Sie adressieren, indem Sie die Einzeladressen beliebig kombinieren, mit einem Komma dazwischen. Der Bereich beginnt mit einer Zeile, die auf die erste Adresse passt, und erstreckt sich von da bis zur ersten Zeile, auf die die zweite Adresse passt. Bereiche können auch in einer Datei anfangen und in einer späteren enden, es sei denn, die Option `-s` wurde angegeben. Hier sind einige Beispiele für Bereichsadressierung:

`1,10` Dies selektiert die ersten zehn Zeilen der Eingabe

`1,/^\$/` Hiermit werden alle Zeilen bis zur ersten Leerzeile ausgewählt. Dieses Idiom ist zum Beispiel nützlich, um den »Kopf« einer E-Mail-Nachricht zu extrahieren (der per Definition immer mit einer Leerzeile aufhört, aber keine Leerzeilen enthalten darf)

`1,$` Dies beschreibt »alle Zeilen der Eingabe«, kann aber meistens wegfallen

`/^BEGIN/,/^END/` Dies beschreibt alle Zeilen von einer, die mit »BEGIN« anfängt, bis zu einer, die mit »END« anfängt (einschließlich).



Ist die zweite Adresse ein regulärer Ausdruck, dann wird nach ihm erst ab der Zeile *unmittelbar nach* der Zeile gesucht wird, bei der der Bereich beginnt. Wenn auch die erste Adresse ein regulärer Ausdruck ist, dann wird, nachdem eine für den zweiten Ausdruck passende Zeile gefunden wurde, wieder nach einer Zeile Ausschau gehalten, die auf den ersten Ausdruck passt – es könnte ja noch eine passende Folge von Zeilen geben.



Wenn die zweite Adresse eine Zahl ist, die eine Zeile *vor* derjenigen Zeile beschreibt, auf die die erste Adresse passt, wird nur die erste passende Zeile ausgegeben.

Sie können alle Zeilen auswählen, die von einer Adresse *nicht* beschrieben werden, indem Sie an die Adresse ein `»!«` anhängen:

`5!` adressiert alle Zeilen der Eingabe außer der fünften.

`/^BEGIN/,/^END/!` adressiert alle Zeilen der Eingabe, die *nicht* Teil eines BEGIN-END-Blocks sind.

Übungen



6.1 [!1] Betrachten Sie die folgende Datei:

```
ABCDEF
123 ABC
EFG
456 ABC
123 EFG
789
```

Tabelle 6.1: Von sed unterstützte reguläre Ausdrücke und ihre Bedeutung (Auswahl)

Regulärer Ausdruck	Bedeutung
[a-d]	ein Zeichen aus der Menge a, b, c oder d
[^abc]	kein Zeichen aus der Menge a, b oder c
.	genau ein beliebiges Zeichen (auch Leerzeichen oder Zeilenumbruch)
*	Platzhalter für das vorherige Zeichen; mögliche Anzahl 0 bis ∞
?	Platzhalter für das vorherige Zeichen; mögliche Anzahl 0 bis 1
^	Anfang einer Zeichenkette
\$	Ende einer Zeichenkette
\<	Anfang eines Wortes
\>	Ende eines Wortes

Welche Zeilen beschreiben die folgenden Adressen? (a) 4; (b) 2,/ABC/; (c) /ABC/,/EFG/; (d) \$; (e) /^EFG/,2; (f) /ABC/!



6.2 [2] Lesen Sie die GNU-sed-Dokumentation und erklären Sie die Bedeutung der (GNU-spezifischen) Adresse »0,/⟨Ausdruck⟩/«.

6.3 sed-Anweisungen

6.3.1 Ausgeben und Löschen von Zeilen

Normalerweise gibt sed jede eingelesene Zeile auf seiner Standardausgabe aus. Mit der Anweisung *d* (engl. *delete*, »löschen«) können Sie Zeilen unterdrücken, so dass sie nicht ausgegeben werden. Beispielsweise wäre

```
sed -e '11,$d'
```

äquivalent zum Kommando *head*: Nur die ersten 10 Zeilen der Eingabe werden durchgelassen.

Mit der Option *-n* wird das automatische Ausgeben unterdrückt. sed gibt dann nur noch Zeilen aus, wenn Sie die Anweisung *p* (engl. *print*, »drucken«) angeben. Sie können *head* also auch durch

```
sed -ne '1,10p'
```

nachbilden. Mit einem regulären Ausdruck statt des Zeilenbereichs so können wir *grep* nachbauen: Dem *grep*-Aufruf

```
grep '^[^#]' /etc/ssh/sshd_config
```

entspricht

```
sed -ne '/^[^#]/p' /etc/ssh/sshd_config
```

Nochmal zurück zu *head*: Unsere bisherigen Alternativen haben beide das Problem, dass sie die Eingabe von vorne bis hinten lesen. Wenn man überlegt, dass wir eigentlich nach der zehnten Zeile fertig sind, ist es natürlich irgendwo zwecklos, noch hunderttausend weitere Zeilen zu lesen, nur um sie entweder zu löschen oder nicht auszugeben. Die effizienteste *head*-Simulation ist tatsächlich

```
sed -e 10q
```

– mit der q-Anweisung (engl. *quit*) wird die sed-Ausführung einfach komplett abgebrochen.

Übungen



6.3 [!1] Wie würden Sie alle Leerzeilen aus der sed-Eingabe löschen?



6.4 [!1] Der populäre Web-Server Apache verwendet zur Konfiguration des Zugriffs auf bestimmte Verzeichnisse in der Datei `httpd.conf` (Verzeichnis kann von Distribution zu Distribution wechseln) Blöcke der Form

```
<Directory /var/www/htdocs>
...
</Directory>
```

Geben Sie ein sed-Kommando an, das alle diese Blöcke aus der Datei `httpd.conf` extrahiert.



6.5 [2] head haben Sie ja nun gesehen. Wie können Sie tail mit sed nachbilden?

6.3.2 Einfügen und Verändern

Wirklich seine Muskeln spielen lässt sed aber erst, wenn Sie ihm erlauben, den Textstrom nicht nur zu filtern, sondern auch zu verändern. Um dies zeilenweise zu erledigen, stehen Ihnen die drei Anweisungen »a« (engl. *append*), »i« (engl. *insert*) und »c« (engl. *change*) für das Anhängen nach einer Zeile, das Einfügen vor einer Zeile bzw. den Austausch einer Zeile durch eine andere zur Verfügung:

```
$ fortune | sed -e '1 i >>>' -e '$ a <<<'
>>>
"So here's a picture of reality: (picture of circle with ▷
< lots of squiggles in it) As we all know, reality is a mess."

-- Larry Wall (Open Sources, 1999 O'Reilly and Associates)
<<<
```

Dabei erlaubt Ihnen der bei Linux übliche GNU-sed das Schreiben in einer Zeile. Die traditionelle Form ist etwas umständlicher und besser für sed-Skripte geeignet. Im folgenden Beispiel sind die »\$« nicht Teil der Datei, sondern werden von cat an das Zeilenende angefügt, um dieses kenntlich zu machen.

```
$ cat -E sed-skript
1i\$
Erste eingefügte Zeile\$
Zweite eingefügte Zeile\$
```

sed erwartet (bei der traditionellen Form) nach dem Kommando a, i oder c einen Rückstrich *direkt vor dem Zeilenende*. Sollen mehr als eine Zeile eingefügt werden, so müssen Sie jede Zeile *bis auf die letzte* ebenfalls mit einem Rückstrich direkt vor dem Zeilenende beenden. Aufgerufen wird dieses Skript durch:

```
$ sed -f sed-skript datei
```

Die Kommandos `a` und `i` sind »Ein-Adress-Kommandos«: Sie erlauben nur Adressen, die auf eine Zeile passen – keine Bereichsangaben (wobei es durchaus mehrere Zeilen in der Eingabe geben kann, die auf die eine Adresse von `a` oder `i` passen und die dann auch alle bearbeitet werden). Diese eine Adresse darf aber alle oben angegebenen Tricks inklusive regulärer Ausdrücke usw. enthalten. Bei `c` bewirkt eine Bereichsangabe, dass der Bereich als Ganzes ersetzt wird.

Übungen



6.6 [!2] Geben Sie ein `sed`-Kommando an, das nach jeder Zeile der Eingabe, die nur aus Großbuchstaben und Leerzeichen besteht, eine Leerzeile einfügt. (Stellen Sie sich vor, Sie wollen Überschriften hervorheben.)

6.3.3 Zeichen-Transformationen

Durch das Kommando `y` können Sie `sed` dazu bringen, einzelne Zeichen durch andere zu ersetzen. So werden durch

```
$ echo 'unschöner Dateiname' | sed -e 'y/ ö/_?/'
unsch?ner_Dateiname
```

Dateinamen mit Leer- und Sonderzeichen »repariert«. Leider erlaubt `y` keine Bereichsangaben der Form `a-z`, so dass es nur ein schwacher Ersatz für `tr` ist.

Übungen



6.7 [1] Geben Sie ein `sed`-Kommando an, das in jeder »ungeraden« Zeile alle Kleinbuchstaben zu Großbuchstaben umwandelt.

6.3.4 Suchen und Ersetzen

Mit dem Kommando `s` (engl. *substitute*) haben Sie die mächtigsten Möglichkeiten. Es erlaubt die Ersetzung eines regulären Ausdrucks durch eine Zeichenkette, deren Zusammensetzung dynamisch verändert werden kann.

Dabei wird der zu ersetzende reguläre Ausdruck wie bei der Zeilenspezifikation in »/.../« eingeschlossen, gefolgt vom Ersetzungstext und »/«, also beispielsweise

```
$ sed -e 's/\<Blei\>/Gold/'
```

Dabei verhindert das Wortendezeichen die versehentliche Erfindung von Wörtern wie »Goldbe«, »Goldchmittel« oder »Goldvergiftung«.

Beachten Sie, dass »\<«, »\>«, «^» und »\$« für *leere Zeichenketten* stehen. Insbesondere ist »\$« *nicht* das Zeilenende, sondern die leere Zeichenkette »« direkt vor dem Zeilenende; durch »s/\$/|/« wird daher am Zeilenende ein »|« eingefügt und nicht etwa das Zeilenende dadurch ersetzt.

Der Ersatztext kann auch vom zu ersetzenden Text abhängen: Neben dem Rückbezug auf geklammerte Teilstücke durch »\1« usw. steht auch »&« zur Verfügung, das für den ganzen Text steht, der vom Suchmuster erfasst wird. Beispiele:

```
$ echo Jedes Wort in Anführungszeichen. | sed -e 's/\([A-Za-z]\+\)/"\1"/g'
"Jedes" "Wort" "in" "Anführungszeichen".
$ echo Jedes Wort in Anführungszeichen. | sed -e 's/[A-Za-z]\+/"&"/g'
"Jedes" "Wort" "in" "Anführungszeichen".
```

Normalerweise ersetzt `s` nur den ersten Treffer in jeder Zeile. Wird ans Ende der `s`-Anweisung noch ein »g« (wie »global«) angehängt – so wie hier –, wirkt sie auf alle Treffer in der Zeile. Ein anderer nützlicher Modifikator ist »p« (*print*), der die Zeile nach dem Ersetzen ausgibt (analog zum »p«-Kommando).

Wenn Sie eine Zahl n anhängen, wird nur der n -te Treffer ersetzt: Eine Eingabe der Form

```
Spalte 1   Spalte 2   Spalte 3   Spalte 4
```

mit Tabulatoren zwischen den Spalten wird durch die sed-Anweisung

```
s/[Tab]/\
/2
```

(wobei `[Tab]` in Wirklichkeit ein »echtes« Tabulatorzeichen¹ sein muss; der Rückstrich am Zeilenende versteckt einen Zeilentrenner) zu

```
Spalte 1   Spalte 2
Spalte 3   Spalte 4
```



Manchmal ist der Schrägstrich als Trennzeichen von Such- und Ersatztext eher lästig, etwa wenn es um Dateinamen geht. Tatsächlich können Sie sich das Trennzeichen praktisch frei aussuchen; Sie müssen nur konsequent sein und dreimal dasselbe benutzen. Nur Leerzeichen und Zeilentrenner sind als Trennzeichen nicht erlaubt.

```
sed 's,/var/spool/mail,/var/mail,'
```

(Bei regulären Ausdrücken als Adressen sind die Schrägstriche leider vorgeschrieben.)

Übungen



6.8 [!1] Geben Sie ein sed-Kommando an, das in der kompletten Eingabe das Wort gelb durch das Wort blau ersetzt.



6.9 [!2] Geben Sie ein sed-Kommando an, das in allen Zeilen, die mit »A« anfangen, das erste Wort entfernt. (Ein Wort ist für die Zwecke dieser Aufgabe eine Folge aus Groß- und Kleinbuchstaben.)

6.4 sed in der Praxis

Hier noch einige Beispiele dafür, wie Sie sed in komplexeren Shellskripten einsetzen können:

Dateien umbenennen In Abschnitt 4.3 hatten wir uns mit dem Ändern von Dateiendungen bei »vielen« Dateien beschäftigt. Mit sed verfügen wir über ein Werkzeug, das uns das beliebige Umbenennen von vielen Dateien ermöglicht. Ein entsprechendes Kommando könnte etwa so aussehen:

```
$ multi-mv 's/pqr/xyz/' abc-*.txt
```

Unser (einstweilen hypothetisches) multi-mv-Kommando bekommt also als erstes Argument ein sed-Kommando übergeben, das dann auf jeden der folgenden Dateinamen angewendet wird.

Als Shellskript formuliert könnte multi-mv ungefähr so aussehen:

¹In der Bash schwierig einzutippen; alle Steuerzeichen können Sie eingeben, indem Sie davor `[Strg]` + `[q]` oder `[Strg]` + `[v]` tippen.

```
#!/bin/bash
# multi-mv -- benennt mehrere Dateien um

sedcmd="$1"
shift

for f
do
    mv "$f" "$(echo \"$f\" | sed \"$sedcmd\")"
done
```

Dateien überschreiben Wenn Sie beim Thema »Ein- und Ausgabeumlenkung der Shell« aufgepasst haben, wissen Sie längst, dass Konstruktionen wie

```
$ sed ... datei.txt >datei.txt
```

nicht das tun, was man naiverweise erwarten könnte: Die Datei `datei.txt` wird gründlich ruiniert. Wenn Sie also eine Datei mit `sed` editieren und das Ergebnis wieder unter dem ursprünglichen Dateinamen abspeichern möchten, müssen Sie zusätzlich arbeiten: Schreiben Sie das Ergebnis zuerst in eine temporäre Datei und benennen Sie diese dann anschließend um, etwa so:

```
$ sed ... datei.txt >datei.tmp
$ mv datei.tmp datei.txt
```

Diese recht umständliche Vorgehensweise bietet sich dafür an, über ein Shellskript automatisiert zu werden:

```
$ oversed ... datei.txt
```

Dabei beschränken wir uns zunächst auf den einfachen Fall, dass das erste Argument von `oversed` die kompletten Instruktionen für `sed` enthält. Außerdem gehen wir davon aus, dass in einem Aufruf wie

```
$ oversed ... datei1.txt datei2.txt datei3.txt
```

die benannten Dateien jeweils einzeln betrachtet und mit ihrem neuen Inhalt überschrieben werden sollen (alles andere ergibt eigentlich keinen Sinn).

Das Skript könnte ungefähr so aussehen:

```
#!/bin/bash
# oversed -- Dateien mit sed "am Platz" editieren

out=/tmp/oversed.$$
sedcmd="$1"
shift

for f
do
    sed "$sedcmd" "$f" >$out
    mv $out "$f"
done
```

Das einzige Bemerkenswerte an diesem Skript ist vielleicht die Bestimmung des Namens für die temporäre Datei (in `out`). Wir achten darauf, dass es keine Kollision mit einem etwaigen anderen, gleichzeitigen `oversed`-Aufruf geben kann, indem wir die aktuelle Prozessnummer (in `$$`) in den Dateinamen einbauen.



Wenn Sicherheit Ihnen wichtig ist, sollten Sie von der »/tmp/oversed.\$\$«-Methode Abstand nehmen, da der von der Prozessnummer abgeleitete Dateiname vorhersehbar ist. Ein Angreifer könnte das ausnutzen, um Ihrem Programm eine Datei unterzuschleiben, die es dann überschreibt. Um sicher zu gehen, könnten Sie zum Beispiel das Programm `mktemp` verwenden, das statt der Prozessnummer einen garantiert nicht existierenden Dateinamen aus zufälligen Zeichen konstruiert. Es legt die Datei dann auch gleich mit restriktiven Rechten an.

```
$ TMP=$(mktemp -t oversed.XXXXXX)
                                     Erzeugt zum Beispiel /tmp/oversed.z19516
$ ls -l $TMP
-rw----- 1 anselm anselm 0 2006-12-21 17:42 /tmp/oversed.z19516
```



Die nützliche Option »-t« von `mktemp` tut die Datei in ein »temporäres Verzeichnis«, namentlich zuerst das durch die Umgebungsvariable `TMPDIR` benannte Verzeichnis, ersatzweise ein Verzeichnis, das Sie mit der Option »-p« benennen können, ersatzweise `/tmp`.

Wir können das Skript auch noch weiter verfeinern. Zum Beispiel könnten wir prüfen, ob `sed` überhaupt etwas an der Datei geändert hat, ob also die Ausgabe- und die Eingabedatei gleich sind. In diesem Fall können wir uns das Umbenennen sparen. Auch wenn die Ausgabedatei leer ist, ist vermutlich etwas schief gegangen, und wir sollten die Eingabedatei nicht überschreiben. In diesem Fall könnte die Schleife aussehen wie

```
for f
do
  sed "$sedcmd" "$f" >$out
  if [ test -s $out ]
  then
    if cmp -s "$f" $out
    then
      echo >&2 "$0: file $f not changed, not overwriting"
    else
      mv "$f" $out
    fi
  else
    echo >&2 "$0: file $f's output empty, not overwriting"
  fi
done
rm -f $out
```

Hier verwenden wir den Dateitestoperator `-s`, der Erfolg meldet, wenn die angegebene Datei existiert und länger als 0 Bytes ist. Das Kommando `cmp` vergleicht zwei Dateien Byte für Byte und liefert Erfolg zurück, wenn die beiden Dateien identisch sind; die Option `-s` unterdrückt die ansonsten unter Umständen produzierte Ausgabe der ersten unterschiedlichen Stelle (die wir hier nicht gebrauchen können).



Seien Sie ein bisschen vorsichtig mit `oversed` – Sie sollten sicher sein, dass Ihre `sed`-Kommandos das Richtige tun, bevor Sie sie auf wichtige Dateien loslassen. Beachten Sie auch Übung 6.10.

Im vorigen Beispiel hatten wir angenommen, dass nur das erste Argument des Skripts alles enthält, was Sie `sed` sagen wollen. Was können Sie tun, damit Sie `sed` mehrere `-e`-Optionen oder andere Optionen wie `-n` oder `-r` übergeben wollen? Hierfür müssen Sie die Kommandozeile genauer anschauen:

```
sedargs=""
while [ "${1:0:1}" = "-" ]
do
  case "$1" in
    *[ef]) sedargs="$sedargs $1 $2"
           shift 2 ;;
    *) sedargs="$sedargs $1"
       shift ;;
    *) break ;;
  esac
done
```

Diese Schleife prüft die Kommandozeilenargumente: Alles, was mit »-« anfängt und auf »e« oder »f« endet, ist mutmaßlich eine Option der Form »-f datei« oder »-ne '...'«. Die Option nebst dem folgenden Dateinamen oder sed-Kommandoargument wird in sedargs gespeichert und aus der Kommandozeile entfernt (»shift 2«). Entsprechend wird alles, was dann noch übrig ist und mit »-« anfängt, als »gewöhnliche« Option ohne nachfolgendes Argument betrachtet und ebenfalls in sedargs untergebracht. Der sed-Aufruf in der Schleife wird also zu

```
sed $sedargs "$f"
```

Auch dies ist leider noch nicht perfekt (siehe Übung 6.11).



Der unter Linux übliche GNU-sed unterstützt in neueren Versionen als Erweiterung eine Option »-i«, die in etwa die Arbeit von oversed tut. Sie können auch etwas angeben wie »-i.bak«; in diesem Fall wird alles hinter dem »-i« als neue Endung für die Originaldatei angesehen. Bei

```
$ sed -i- ... bla
```

heißt die Eingabedatei später bla- und die Ausgabe steht in bla.) Sie sollten auf die Anwendung dieser Option allerdings verzichten, wenn das Risiko besteht, dass Ihre Skripte auch auf anderen Unixen laufen sollen als Linux.

Übungen



6.10 [!2] Sorgen Sie dafür, dass in oversed vor dem Umbenennen der sed-Ausgabedatei die ursprüngliche Eingabedatei unter einem geeigneten Namen gesichert wird (Sie könnten zum Beispiel die zusätzliche Endung ».bak« an den Namen anhängen).



6.11 [3] sed erlaubt noch einige weitere Kommandozeilenargumente. Beispielsweise steht »-« (wie auch anderswo) für »Ende der Optionen – was jetzt kommt, ist ein Dateiname, selbst wenn es aussieht wie eine Option«, und es gibt »lange« Optionen der Form »-expression=...« (als Synonym für -e). Passen Sie oversed so an, dass es auch für diese Argumente das »Richtige« tut.



6.12 [3] Mit Ihren neuerworbenen Kenntnissen über sed können Sie das wwm-Skript aus Kapitel 5 etwas weniger umständlich machen, was das Format der Fragendatei angeht. Geben Sie eine Implementierung der question-Funktion an, die eine Fragendatei im Format

```
question 0
?Was hat die Morgenstund im Sprichwort?
-Blei im Hintern
-Eisen im Bauch
```

```
+Gold im Mund
-Silber im Kopf
>50
end
question 50
?Wovor sollten Sie sich beim Badeurlaub in Acht nehmen?
-Hallowal
<<<<<
```

verarbeitet.

Kommandos in diesem Kapitel

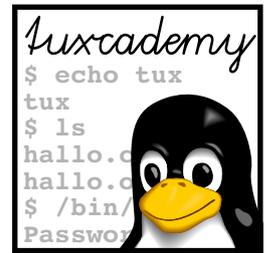
cmp	Vergleicht zwei Dateien byteweise	cmp(1)	101
mktemp	Erzeugt einen eindeutigen temporären Dateinamen (sicher)	mktemp(1)	100

Zusammenfassung

- sed ist ein »Stromeditor«, der seine Standardeingabe liest und modifiziert auf die Standardausgabe schreibt.
- sed erlaubt die flexible Adressierung von Eingabezeilen über ihre Position oder ihren Inhalt sowie die Beschreibung von Zeilenbereichen in der Eingabe.
- Diverse Kommandos zur Textmodifikation stehen zur Verfügung.

Literaturverzeichnis

- DR97** Dale Dougherty, Arnold Robbins. *sed & awk*. Sebastopol, CA: O'Reilly & Associates, 1997, 2. Auflage. ISBN 1-56592-225-5.
<http://www.oreilly.de/catalog/sed2/>
- Rob02** Arnold Robbins. *sed & awk – kurz & gut*. Sebastopol, CA: O'Reilly & Associates, 2002, 2. Auflage. ISBN 3-89721-246-3.
<http://www.oreilly.de/catalog/sedawkrepr2ger/>



7

Die awk-Programmiersprache

Inhalt

7.1	Was ist awk?	106
7.2	awk-Programme.	107
7.3	Ausdrücke und Variable	109
7.4	awk in der Praxis	113

Lernziele

- Die Programmiersprache awk kennenlernen
- Einfache awk-Programme formulieren können
- awk in Shellskripten einsetzen können

Vorkenntnisse

- Kenntnisse über Shellprogrammierung (aus den vorigen Kapiteln)
- Programmier-Erfahrung in anderen Sprachen ist hilfreich

7.1 Was ist awk?

awk ist eine Programmiersprache zur Verarbeitung von Textdateien. Der Name setzt sich aus den Anfangsbuchstaben der Nachnamen der Erfinder Alfred V. Aho, Peter J. Weinberger und Brian W. Kernighan zusammen und leitet sich nicht etwa vom englischen Wort »awkward« ab, was soviel wie schwierig, unbequem bedeutet.



Auf dem awk-Buch von Aho, Weinberger und Kernighan [AKW88] ist ein Alk (*Alca torda*) abgebildet. Dieser nordische Seevogel heißt auf Englisch *auk*, was genau wie *awk* ausgesprochen wird. (Alke haben übrigens nichts mit Pinguinen zu tun.)

awk in Linux Linux-Distributionen enthalten in der Regel nicht den ursprünglichen awk von AT&T, sondern kompatible und mehr oder weniger erweiterte Implementierungen wie mawk oder gawk (GNU awk). Für unsere Zwecke genügt es, über awk zu reden, da die Erweiterungen für uns nicht wirklich relevant sind (wir weisen gegebenenfalls darauf hin).



Welche Sorte awk Sie haben, kann bei der Suche nach der Dokumentation wichtig sein. Nicht immer wird dafür gesorgt, dass bei einem »man awk« tatsächlich die Handbuchseite erscheint (es hängt von Ihrer Distribution ab); notfalls müssen Sie »man gawk« oder »man mawk« versuchen. GNU awk hat außerdem eine Info-Dokumentation, die Sie (je nach Distribution) möglicherweise als gesondertes Paket installieren müssen.

awk als Programmiersprache awk als Programmiersprache zu bezeichnen wirkt auf viele Leute, die sich nicht wirklich als »Programmierer« sehen, eher abschreckend. Wenn Sie damit ein Problem haben, dann betrachten Sie awk als besonders leistungsfähiges Werkzeug zum Analysieren und Ändern von Dateien – eine Art Super-sed-cut-sort-paste-grep. Mit awk können Sie zum Beispiel formatierte Berichte aus Protokolldateien generieren oder diverse andere Operationen auf »strukturierten Daten« ausführen – etwa »Tabellen« mit Tabulatorzeichen zwischen den Spalten. awk kann – unter anderem –:

- Textdateien interpretieren, die aus »Datensätzen« bestehen, mit »Feldern« innerhalb der Datensätze;
- Daten in Variablen und Feldern speichern;
- arithmetische, logische und Zeichenketten-Operationen ausführen;
- Schleifen und bedingte Anweisungen bearbeiten;
- Funktionen definieren;
- die Ausgabe von Kommandos weiterbearbeiten.

Programmiermodell awk liest aus der Standardeingabe oder aus namentlich angegebenen Dateien Text (normalerweise) zeilenweise ein. Jede Zeile wird in Felder unterteilt und bearbeitet. Die Ergebnisse werden dann auf die Standardausgabe oder in eine benannte Datei geschrieben.

awk-»Programme« befinden sich in der Lücke zwischen Shellskripten und Programmen in Sprachen wie Perl oder Tcl/Tk. Der wesentliche Unterschied von awk zu anderen Programmiersprachen wie den Shells, C oder Tcl/Tk besteht in der »datenorientierten« Arbeitsweise, während die typischen Programmiersprachen »funktionsorientiert« wirken. Ein awk-Programm wirkt im Prinzip wie eine Schleife über die Eingabedatensätze (meist Zeilen), die durchlaufen wird, bis keine Daten mehr in der Eingabe stehen oder das Programm verlassen wird. Der Steuerfluss ist also maßgeblich durch die Daten gegeben. In den meisten anderen Programmiersprachen wird das Hauptprogramm dagegen einmalig initiiert und Funktionen (die gegebenenfalls Daten einlesen) beeinflussen den Fortschritt der Berechnung.

7.2 awk-Programme

Im einfachsten Fall funktioniert `awk` nicht unähnlich zu `sed`: Sie können Zeilen adressieren und auf die passenden Zeilen anschließend Kommandos anwenden. Ein `grep`-Äquivalent in `awk` würde zum Beispiel so aussehen:

```
awk '/a.*a.*a/ { print }'
```

gibt alle Zeilen seiner Eingabe aus, die mindestens drei »a« enthalten. In den geschweiften Klammern können ein oder mehrere Kommandos stehen, die auf die Zeilen angewendet werden, die auf den regulären Ausdruck zwischen den Schrägstrichen passen.

Kommandos

Es ist oft bequemer, `awk`-»Skripte« in ihre eigenen Dateien zu schreiben. Solche Dateien können Sie mit »`awk -f <Skriptdatei>`« ausführen. Zeilen mit »#« am Anfang gelten wie in der Shell als Kommentarzeilen.

awk-»Skripte«

Kommentarzeilen



Auch hier spricht natürlich nichts gegen direkt ausführbare `awk`-Skripte der Form

```
#!/usr/bin/awk -f
/a.*a.*a/ { print }
```

Hier ist ein `awk`-Skript, das für alle Zeilen, in denen eine Zahl steht, eine Nachricht ausgibt:

```
#!/usr/bin/awk -f
/[0-9]+/ { print "Diese Zeile enthält eine Zahl." }
```

Zeilen, in denen keine Zahl steht, werden ignoriert.

Für jede Eingabezeile wird geprüft, welche `awk`-Skriptzeilen auf sie passen, und alle `awk`-Kommandofolgen aus passenden Zeilen werden ausgeführt. Das folgende Skript `classify` versucht Zeilen gemäß ihres Inhalts zu klassifizieren:

awk und Eingabe

```
#!/usr/bin/awk -f
# classify -- klassifiziert Eingabezeilen
/[0-9]+/ { print "Diese Zeile enthält eine Zahl." }
/[A-Za-z]+/ { print "Diese Zeile enthält ein Wort." }
/^[^$]/ { print "Diese Zeile ist leer." }
```

Hier ist ein Beispiel für das `classify`-Skript:

```
$ awk -f classify
123
Diese Zeile enthält eine Zahl.
bla
Diese Zeile enthält ein Wort.
↵
Diese Zeile ist leer.
123 bla
Diese Zeile enthält eine Zahl.
Diese Zeile enthält ein Wort.
```

Sie sehen hier, dass die Zeile »123 bla« auf zwei der Regeln passte. Es ist möglich, Regeln so zu formulieren, dass immer nur eine von ihnen passt.

`awk` nimmt an, dass die Eingabe strukturiert ist und nicht nur ein Strom von Bytes. Im Standardfall wird jede Eingabezeile als »Datensatz« betrachtet und an Leerzeichen in »Felder« aufgeteilt.

Eingabe: strukturiert



Im Gegensatz zu Programmen wie `sort` und `cut` interpretiert `awk` Folgen von Leerzeichen als *einen* Feldtrenner, anstatt zwischen zwei aufeinanderfolgenden Leerzeichen ein leeres Feld zu vermuten.

Sie können sich mit den `awk`-Ausdrücken `$1`, `$2` usw. auf die einzelnen Felder eines Datensatzes beziehen:

```
$ ls -l *.sh | awk '{ print $9, $5 }'
dump_pwd.sh 113
dwwm.sh 1220
wmm-sed.sh 1513
wmm.sh 1132
zahlenraten.sh 323
```

Sie müssen natürlich aufpassen, dass die Shell die »\$« nicht zu expandieren versucht!



Der Ausdruck »\$0« liefert den kompletten Eingabedatensatz (alle Felder).

Bei dieser Gelegenheit lernen Sie auch gleich noch, dass eine Folge von `awk`-Kommandos keinen regulären Ausdruck vor sich *braucht*: Ein »{...}« ohne regulären Ausdruck wird auf *jeden* Eingabedatensatz angewendet.



Hier macht sich `awk` schon als verbessertes `cut` nützlich: Erinnern Sie sich, dass `cut` die Spalten in der Ausgabe nicht gegenüber ihrer Reihenfolge in der Eingabe umsortieren konnte.

Das Trennzeichen für die Eingabefelder können Sie mit der `awk`-Option `-F` ändern:

```
$ awk -F: '{ print $1, $5 }'
root root
daemon daemon
bin bin
<<<<<<
```

Felder in der Ausgabe werden durch Leerzeichen getrennt (wie Sie das ändern können, erfahren Sie später).

`BEGIN` und `END` `awk` erlaubt es außerdem, Kommandofolgen anzugeben, die am Anfang der Programmausführung – bevor Eingabe gelesen wurde – und am Ende der Eingabe – nachdem der letzte Datensatz gelesen wurde – ausgeführt werden. Hiermit können Sie Initialisierungen vornehmen oder Gesamtergebnisse ausgeben. Das Kommando

```
ls -l *.txt | awk '
BEGIN { sum = 0 }
      { sum = sum + $5 }
END   { print sum }'
```

zum Beispiel addiert die Längen aller Dateien mit der Endung `*.txt` im aktuellen Verzeichnis und gibt am Ende das Ergebnis aus. `sum` ist eine **Variable**, die die aktuelle Zwischensumme speichert; Variable in `awk` benehmen sich ähnlich wie Shellvariable, bis darauf, dass Sie auf ihren Wert zugreifen können, ohne ein »\$« vor den Variablennamen setzen zu müssen.



Für Kenner: `awk`-Variable können wahlweise Zeichenketten oder (Gleitkomma-)Zahlen enthalten. Diese Datentypen werden nach Bedarf ineinander umgewandelt.

7.3 Ausdrücke und Variable

Wir können das Dateigrößen-Summierprogramm aus dem vorigen Abschnitt noch weiter verfeinern. Zum Beispiel können wir die Dateien zählen und die Anzahl der betrachteten Dateien ausgeben:

```
#!/usr/bin/awk -f
# filesum -- Dateigrößen addieren
{ sum += $5
  count++
}
END { print sum " Bytes in " count " Dateien" }
```

Die »BEGIN«-Regel ist nicht wirklich nötig, da neue Variable beim ersten Benutzen den Standardwert 0 bekommen. »sum += \$5« ist dasselbe wie »sum = sum + \$5«, und »count++« wiederum dasselbe wie »count = count + 1«. (C-Programmierer sollten sich hier ganz zu Hause fühlen.) Das ganze funktioniert dann so:

```
$ ls -l *.sh | filesum
4301 Bytes in 5 Dateien
```

Dieses simple Programm ist nicht ohne seine Probleme. Wenn Sie sich nicht eine bestimmte Menge von Dateien aus einem Verzeichnis anzeigen lassen (wie eben alle Dateien, deren Namen auf ».sh« enden), sondern ein komplettes Verzeichnis – denken Sie an »ls -l .«, dann bekommen Sie am Anfang der Ausgabe eine Zeile mit der Gesamtzahl der von Dateien im Verzeichnis belegten »Datenblöcke« (bei Linux normalerweise Kibibytes) angezeigt:

```
total 1234
```

Diese Zeile hat kein fünftes Feld, verfälscht also die Gesamtsumme nicht, aber wird als Eingabezeile und damit als Datei gezählt. Ein anderes Problem betrifft Zeilen für Unterverzeichnisse à la

```
drwxr-xr-x  3 anselm  anselm  4096 May 28 12:59 subdir
```

Die »Dateigröße« in diesem Eintrag hat nichts zu bedeuten. Auch andere Dateitypen (etwa Gerätedateien) stiften nur Verwirrung.

Um diese Probleme zu umgehen, sind zwei Beobachtungen wichtig: Die Feldanzahl von »interessanten« Zeilen ist größer als 2, und wir wollen nur solche Zeilen betrachten, die mit »-« anfangen. Letzteres ist leicht umzusetzen:

```
/^-/ { ... }
```

Um ersteres zu beachten, müssen Sie wissen, dass `awk` die Anzahl der in einem Datensatz gefundenen Felder in der Variablen `NF` zugänglich macht. Wir müssen nur prüfen, ob diese Variable einen Wert größer 2 hat. Sind diese Bedingung *und* die »Beginnt mit -«-Bedingung erfüllt, betrachten wir die Zeile. Also:

```
#!/usr/bin/awk -f
# filesum2 -- Dateigrößen addieren
NF > 2 && /^-/ {
  sum += $5
  count++
}
END {
  print sum " Bytes in " count " Dateien"
}
```

Der Gleichheits-Operator ist in *awk* (wie in C) »=*«, um eine Verwechslung mit dem Zuweisungs-Operator »=*« zu erschweren. Ebenfalls wie in C ist »&&« der logische Und-Operator; genau wie in C wertet er seine rechte Seite nur aus, wenn die linke »wahr«, also einen von 0 verschiedenen Wert, ergibt.**



Dies ist genau andersherum als bei der Shell – die Kommandos »if«, »while« & Co. und der »&&«-Operator der Shell bewerten einen *Rückgabewert* von 0 als »Erfolg«.

awk-Ausdrücke *awk*-Ausdrücke können unter anderem die gängigen Operatoren für die Grundrechenarten (»+«, »-«, »*« und »/«) und logischen Vergleiche (»<«, »<=« (≤), »>«, »>=« (≥), »==« und »!=« (≠)) enthalten. Außerdem gibt es zum Beispiel noch die Testoperatoren für reguläre Ausdrücke »~« und »!~«, mit denen Sie prüfen können, ob eine Zeichenkette auf einen regulären Ausdruck passt (oder nicht passt):

```
$1 ~ /a.*a.*a/ { ... }
```

führt die Kommandos genau dann aus, wenn das erste Feld mindestens drei »a« enthält.

awk-eigene Operatoren Auch einige andere Operatoren von *awk* sind nicht der Programmiersprache C entliehen. Das »\$« könnten Sie zum Beispiel auch als »Feldzugriffoperator« interpretieren: »\$3« liefert Ihnen den Wert des 3. Feldes (falls vorhanden) im aktuellen Datensatz, aber mit »\$NF« bekommen Sie immer den Inhalt des *letzten* Feldes, egal wie viele Felder der aktuelle Datensatz hat. Zwei Zeichenketten (oder Variablenwerte) können Sie aneinanderhängen, indem Sie sie einfach nebeneinanderschreiben (durch ein Leerzeichen getrennt):

```
$ awk '{ print $1 "schwerenot" }'  
Schock  
Schockschwerenot
```



In *awk* fehlen gegenüber der Programmiersprache C die bitweisen logischen Operatoren »|«, »&« und »^« sowie die Schiebeoperatoren »<<<« und »>>>«. (Wenn Sie Bits schubsen wollen, müssen Sie wohl oder übel C benutzen.) (Oder Perl.) In *awk* gibt es einen »^«-Operator, aber der steht für Exponentiation.

(Eine komplette Zusammenstellung der *awk*-Operatoren finden Sie in der *awk*-Dokumentation.)

Variable Variable in *awk* sind, wie erwähnt, »typenlos«, können also Zeichenketten und Zahlen aufnehmen und werden jeweils passend interpretiert. Jedenfalls soweit das möglich ist:

```
$ awk 'BEGIN { a = "123abc"; print 2*a; exit }'  
246  
$ awk 'BEGIN { a = "abc"; print 2*a; exit }'  
0
```

Die Namen von Variablen beginnen immer mit einem Buchstaben und können ansonsten Buchstaben, Ziffern und die Unterstreichung (»_«) enthalten.

Systemvariable *awk* definiert außer NF noch einige andere »Systemvariable«: FS ist der Eingabe-Feldseparator, den Sie über die Option -F setzen können (eine Zuweisung an »FS« in einem BEGIN-Kommando tut es aber auch). RS ist der Eingabe-Datensatzseparator, also das Zeichen, das »Datensätze« trennt. Normalerweise ist das ein Zeilentrenner, aber es hindert Sie niemand daran, etwas Anderes einzustellen. Der besondere Wert »""« steht für eine Leerzeile. – Dies macht es einfach, Dateien zu bearbeiten, die keine Zeilenstruktur aufweisen, sondern eine »Blockstruktur« wie die folgende:

```
Hugo
Schulz
Lindenstraße 43c
12345
Königs-Musterhausen
(0123) 456789

Emil
Huber
Goldregenweg 33
65432
Beispielstadt
(06543) 3210
<<<<<
```

Sie müssen dazu nur FS und RS auf passende Werte setzen:

```
#!/usr/bin/awk -f
# Gibt eine Telefonliste aus
BEGIN { FS = "\n"; RS = "" }
      { print "$1 $2", $NF }
```

Außer »einfachen« Variablen unterstützt `awk` auch Felder, also indizierte Gruppen von Variablen unter einem Namen. Felder sind Ihnen schon bei der Bash begegnet – denken Sie an das `dwm`-Skript –; im Gegensatz zur Bash mit ihren numerischen Indizes erlaubt `awk` aber beliebige Zeichenketten als Feldindex. Man spricht hier auch von »assoziativen Feldern«. Dies ist ein extrem mächtiges Werkzeug, wie das folgende Skript zeigt:

```
#!/usr/bin/awk -f
# shellusers -- gibt aus, welche Shell von wem benutzt wird
BEGIN { FS = ":" }
      { use[$NF] = use[$NF] " " $1 }
END {
    for (i in use) {
        print i ":" use[i]
    }
}
```

Wenn Sie `shellusers` mit `/etc/passwd` als Parameter aufrufen, liefert es Ihnen eine Liste der verwendeten Login-Shells mit ihren jeweiligen Benutzern:

```
# shellusers /etc/passwd
/bin/sync: sync
/bin/bash: root anselm tux
/bin/sh: daemon bin sys games man lp mail news uucp proxy>
< postgres www-data backup operator list irc gnats nobody
/bin/false: hermes identd mysql partimag sshd postfix>
< netsaint telnetd ftp bind
```

Dabei dient die Kommandofolge ohne regulären Ausdruck zum Sammeln der Daten und das `END`-Kommando zur Ausgabe; das `for`-Kommando leitet eine Schleife ein, bei der die Variable `i` sukzessive auf jeden Index des Feldes `use` gesetzt wird (die Reihenfolge ist nicht vorhersagbar).

In `awk`-Ausdrücken können auch Funktionen angesprochen werden. `awk` definiert einige vor, zum Beispiel die arithmetischen Funktionen »int« (ganzzahligen Anteil einer Zahl bestimmen), »sqrt« (Quadratwurzel) oder »log« (Logarithmus).

Die Möglichkeiten von `awk` entsprechen hier etwa denen eines technisch-wissenschaftlichen Taschenrechners. Es gibt auch Funktionen, die sich mit Zeichenketten befassen: »length« bestimmt die Länge einer Zeichenkette, »substr« schneidet ein Stück aus einer Zeichenkette aus und »sub« entspricht dem »s«-Operator von `sed`. Sie können übrigens auch selber Funktionen definieren. Betrachten Sie dazu das folgende Beispiel:

Eigene Funktionen

```
#!/usr/bin/awk -f
# triple -- verdreifache Zahlen

function triple(n) {
    return 3*n
}

{ print $1, triple($1) }
```

Dieses Programm liest eine Datei mit Zahlen (eine pro Zeile) und gibt die ursprüngliche Zahl und das Dreifache dieser Zahl aus:

```
$ triple
3
3 9
11
11 33
```

Der »Körper« der Funktion kann aus einem oder mehreren `awk`-Kommandos bestehen; das `return`-Kommando dient dazu, einen Wert als Funktionsergebnis zurückzuliefern.

Lokale Variable

Die Variablen in der Parameterliste einer Funktion (hier `n`) werden an die Funktion übergeben und sind dort »lokal«, das heißt, sie können verändert werden, ohne dass das Auswirkungen außerhalb der Funktion hat. Alle anderen Variablen sind »global«, also überall im `awk`-Programm zu sehen. Insbesondere haben Sie in `awk` nicht die Möglichkeit, innerhalb einer Funktion zusätzliche lokale Variable zu definieren. Sie können sich aber aus der Affäre ziehen, indem Sie der Funktion ein paar zusätzliche »Parameter« übergeben, die Sie beim Funktionsaufruf nicht verwenden. Hier ist als Illustration eine Funktion, die die Elemente eines Feldes `F` sortiert, das numerische Indizes von 1 bis `N` verwendet:

```
function sort(F, N, i, j, temp) {
    # Sortieren durch Einfügen
    for (i = 2; i <= N; i++) {
        for (j = i; F[j-1] > F[j]; j--) {
            temp = F[j]; F[j] = F[j-1]; F[j-1] = temp
        }
    }
    return
}
```

Die `for`-Schleife führt hier ihr erstes Argument aus (`i = 2`). Dann wiederholt sie das Folgende: Sie wertet ihr zweites Argument (`i <= N`) aus. Ergibt das ein »wahreres« (von 0 verschiedenes) Ergebnis, wird der Schleifenkörper (hier eine zweite `for`-Schleife) ausgeführt und anschließend das dritte Argument (`i++`). Dies wird wiederholt, bis das zweite Argument 0 als Ergebnis liefert.



Dies ist sicherlich nicht der genialste mögliche Sortieralgorithmus. Kenner der Materie werden unmittelbar feststellen, dass hier zum Sortieren von n Werten eine Anzahl von Schritten gebraucht wird, die proportional zu n^2 ist, und das lässt sich mit geschickteren, aber aufwendigeren Verfahren problemlos unterbieten. Aber wir sind ja hier, um `awk` zu lernen und nicht »Sortieren und Suchen«.

Aufgerufen würde diese Funktion zum Beispiel wie

```
{
  a[1] = "Hund"; a[2] = "Katze"
  a[3] = "Maus"; a[4] = "Baum"
  sort(a, 4)
  for (i = 1; i <= 4; i++) {
    print i ": " a[i]
  }
}
```

Beachten Sie die Ausgabe der Feldelemente über eine »zählende« for-Schleife; eine »for (i in a)«-Schleife hätte die Elemente in einer nicht vorhersehbaren Reihenfolge geliefert (und damit das Sortieren sinnlos gemacht).

Übungen



7.1 [3] In einer früheren Version dieses Kapitels stand das Programm `filesun2` wie folgt:

```
#!/usr/bin/awk -f
# filesun2 -- Dateigrößen addieren
NF == 9 && /^-/ {
  sum += $5
  count++
}
END {
  print sum " Bytes in " count " Dateien"
}
```

Warum funktioniert das nicht immer? (*Tipp*: Testen Sie das Programm in verschiedenen Sprachumgebungen, etwa `LANG=C` und `LANG=de_DE@euro`.)

7.4 awk in der Praxis

Hier noch ein paar `awk`-Beispiele aus dem »wirklichen Leben«:

Spiel und Spaß mit der Shell-History (Aus der GNU-`awk`-Dokumentation.) Die Bash speichert Ihre Kommandos in der Datei `~/.bash_history` – wenn Sie dasselbe Kommando mehrmals hintereinander geben, auch mehrmals. Angenommen, Sie wollen diese Datei verkleinern und jedes Kommando nur einmal ablegen. Es gibt zwar das Kommando `uniq`, aber das entfernt nur unmittelbar aufeinanderfolgende Dubletten und arbeitet am besten mit vorsortierter Eingabe; in der Shell-History sollte die Reihenfolge der Kommandos aber grundsätzlich erhalten bleiben. Anders gesagt, bei einer Eingabe wie

```
abc
def
abc
abc
ghi
def
def
ghi
abc
```

ist nicht die `uniq`-Ausgabe

```
abc
def
abc
ghi
def
ghi
abc
```

gewünscht, sondern etwas wie

```
abc
def
ghi
```

Die assoziativen Felder von `awk` machen das einfach: Wir zählen das Vorkommen jeder Zeile in einem assoziativen Feld `data`, das durch den kompletten Text des Kommandos indiziert wird. Nur wenn eine Zeile noch nie gesehen wurde (ihr Eintrag in diesem Feld also Null ist), geben wir die Zeile tatsächlich aus (`»print«` ist dasselbe wie `»print $0«`).

```
#!/usr/bin/awk -f
# histsort -- Shell-History-Datei kompaktieren
{
    if (data[$0]++ == 0) {
        print
    }
}
```

Fallunterscheidungen Hier sehen Sie, dass `awk` auch `if`-Fallunterscheidungen kennt; die Syntax ist (wie üblich) an C angelehnt:

```
if (<Bedingung>) {
    <Kommandos>
} [else {
    <Kommandos>
}]
```

Der Ausdruck `»data[$0]++ == 0«` ist ein gängiges Idiom; er ist genau dann `»wahr«`, wenn der Wert von `»$0«` zum ersten Mal gesehen wurde.

Wissen Sie, welche Unix-Kommandos Sie am meisten verwenden? Auch diese Frage kann `awk` Ihnen anhand der Shell-History beantworten. Die nötigen Daten dafür sammelt das vorstehende Programm schon in `data`; wir müssen sie nur noch ausgeben. Der Einfachheit halber sparen wir uns die Ausgabe der tatsächlichen Kommandos beim Zählen:

```
#!/usr/bin/awk -f
# histcount -- Kommandos in der Shell-History-Datei zählen
{
    data[$0]++
}
END {
    for (c in data) {
        print data[$c], $c
    }
}
```

Wie bei `»in«` üblich werden die Kommandozeilen in einer nicht vorhersagbaren Reihenfolge ausgegeben. Das können Sie durch einen nachgeschalteten Aufruf

des `sort`-Kommandos korrigieren, da nicht jede `awk`-Version eine Sortieroutine anbietet.



Das Programm `histcount` betrachtet das komplette Kommando (mit Argumenten). Wenn Sie sich nur für die Kommandos im eigentlichen Sinne, ohne Argumente, interessieren, können Sie statt `$0` auch `$1` verwenden. Beachten Sie dazu allerdings Übung 7.3.

Doppelte Wörter Ein gängiger Fehler in Texten sind versehentliche Wortverdopplungen, etwas wie: »Das Ergebnis des des Programms ist ...«. Mitunter steht das erste der beiden Wörter am Ende einer Zeile und das zweite am Anfang der nächsten, was die Sache schwer zu finden macht. Hier ist ein `awk`-Skript, das dabei hilft – es betrachtet die Wörter jeder Zeile und speichert außerdem das letzte Wort jeder Zeile zum Vergleich mit dem ersten der nächsten Zeile.

Der Einfachheit halber gehen wir davon aus, dass die Eingabe komplett in Kleinbuchstaben ist und alle Nichtbuchstaben zu Leerzeichen gemacht wurden – keine große Einschränkung, denn das lässt sich durch etwas wie

```
tr '[:upper:]' '[:lower:]' | tr -cs '[:alpha:]' ' '
```

leicht erreichen¹ (es geht auch direkt in `awk`, jedenfalls GNU-`awk`, aber die Details ersparen wir Ihnen). Das `awk`-Skript selbst ist etwas wie

```
#!/usr/bin/awk -f
# dupwords -- doppelte Wörter entfernen

{
    if (NF == 0) {
        next
    }
    if ($1 == prev) {
        printf("%s:%d: %s doppelt\n", FILENAME, FNR, $1)
    }
    for (i = 2; i <= NF; i++) {
        if ($i == $(i-1)) {
            printf("%s:%d: %s doppelt\n", FILENAME, FNR, $i)
        }
    }
    prev = $NF
}
```

Hierbei entspricht `printf` dem gleichnamigen Kommando der Shell und der C-Bibliothek zur formatierten Ausgabe: Die »Formatschlüssel« `%s` und `%d` werden durch die entsprechenden späteren Argumente als Zeichenkette (für `%s`) und Zahl (`%d`) ersetzt, das erste `%s` also durch den Wert von `FILENAME` (den Namen der aktuellen Eingabedatei), das `%d` durch die Zeilennummer in der aktuellen Eingabedatei (`FNR`) und das zweite `%s` durch das fragliche Wort. Beachten Sie auch den Zugriff auf das »vorige« Wort über »`$(i-1)`«. Wenn man gerade von der Shell kommt, ist es sehr leicht, dem Irrtum anheimzufallen, `$` sei einfach nur ein Präfix für Variablennamen. In Wirklichkeit ist es in `awk` ein echter Operator zum Zugriff auf die Felder der Eingabe – was dahinter steht, wird ausgewertet, und daraus ergibt sich die zu holende Feldnummer.

König Fußball Hier ist ein Thema, das sicher vielen von Ihnen sympathisch ist: die Fußball-Bundesliga. Gegeben ist die Datei `b103.txt`, die die Ergebnisse der Bundesliga von 2003/04 enthält:

¹Traditionell wäre das »`tr A-Z a-z | tr -cs a-z ' '`«; die Fassung mit den POSIX-Zeichenklassen funktioniert aber auch für nichtenglische Texte.

```

1:Bayern München:Eintracht Frankfurt:3:1:6300
1:Schalke 04 Gelsenkirchen: Borussia Dortmund:2:2:61010
1:Hamburger SV:Hannover 96:0:3:53224
1:Bayer Leverkusen:SC Freiburg:4:1:22500
1:Hertha BSC Berlin:Werder Bremen:0:3:40000
1:1.FC Kaiserslautern:1860 München:0:1:35629
1:VfL Wolfsburg:VfL Bochum:3:2:20000
1:Hansa Rostock:VfB Stuttgart:0:2:23500
1: Borussia Mönchengladbach:1.FC Köln:1:0:34500
2:VfL Bochum:Hamburger SV:1:1:20400
2: Borussia Dortmund:VfL Wolfsburg:4:0:72500
2:1860 München:Schalke 04 Gelsenkirchen:1:1:33000
2:1.FC Köln:1.FC Kaiserslautern:1:2:33000
<<<<<<

```

Dabei ist das erste Feld der Spieltag, das zweite und dritte die konkurrierenden Mannschaften, das vierte und fünfte das Spiel-Ergebnis und das sechste die Anzahl der Zuschauer im Stadion.

Fangen wir mit etwas Einfachem an. Wie viele Zuschauer haben sich am 1. Spieltag auf den Weg ins Stadion gemacht? Dafür müssen Sie nur die letzten Spalten der Zeilen für den betreffenden Spieltag aufaddieren:

```

#!/usr/bin/awk -f
# zuschauer -- Addiert Zuschauer für den 1. Spieltag

$1 == 1 { z += $6 }
END { print z }

```

Also:

```

$ awk -F: -f zuschauer b103.txt
296663

```

awk-Variable auf der Kommandozeile

Wenn Sie die Zuschauerzahlen für irgendeinen beliebigen Spieltag interessieren, können Sie die Nummer des Spieltags als Parameter übergeben. Die Abfrage sieht dann etwas anders aus:

```

$1 == tag { z += $6 }

```

Und der Aufruf ist

```

$ awk -F: -f zuschauer tag=2 b103.txt
257200

```

Sie können nämlich zwischen die awk-Optionen und die Dateinamenargumente Zuweisungen an awk-Variable einbauen; Bedingung ist nur, dass awk jede Zuweisung als ein Argument zu sehen bekommt – rund um das »=« dürfen also keine Leerzeichen stehen.

Wie sieht die Tabelle der Bundesliga nach dem *n*-ten Spieltag aus? Hierfür ist etwas mehr Arbeit nötig: Wir müssen für jedes Spiel entscheiden, welche Mannschaft von beiden gewonnen hat, und die Punkte und die Tordifferenz berechnen. Das könnte ungefähr so aussehen:

```

BEGIN { FS = ":"; OFS = ":" }
$1 <= tag {
    if ($4 > $5) {
        punkte[$2] += 3
    } else if ($4 < $5) {

```

```

    punkte[$3] += 3
  } else {
    punkte[$2]++; punkte[$3]++
  }
  tore[$2] += $4 - $5; tore[$3] += $5 - $4
}

```

(Die Variable OFS ist der »Ausgabe-Feldtrenner«, also das, was von awk in Kommandos wie »print a, b« (mit Komma) zwischen die Ausdrücke geschrieben wird.) Wir addieren die Punkte im Feld punkte und die Tordifferenzen im Feld tore, beide indiziert durch die Mannschaftsnamen. Ausgeben können wir die Tabelle mit etwas wie

```

END {
  for (team in punkte) {
    print team, punkte[team], tore[team]
  }
}

```

Zum Beispiel:

```

$ awk -f bltab tag=1 bl03.txt
1.FC Kaiserslautern:0:-1
Borussia Mönchengladbach:3:1
Bayer Leverkusen:3:3
Bayern München:3:2
Hansa Rostock:0:-2
Borussia Dortmund:1:0
Hertha BSC Berlin:0:-3
Hamburger SV:0:-3
Schalke 04 Gelsenkirchen:1:0
VfL Wolfsburg:3:1

```

Sie müssen sich nicht wirklich gut mit Fußball auskennen, um zu sehen, dass da offenbar etwas faul ist. Zunächst ist die Tabelle offensichtlich nicht richtig sortiert (was nicht weiter überraschend ist, wegen »team in punkte« ist damit ja nicht zu rechnen) – aber sind nicht eigentlich mehr als 10 Mannschaften in der Bundesliga? Offensichtlich unterschlägt unser Programm einige Teams, und nach kurzer Überlegung sollte Ihnen auch klar werden, warum: Im punkte-Feld stehen nur die Mannschaften, die tatsächlich in der Liga gepunktet haben, und das sind nach dem ersten Spieltag halt in der Regel noch nicht alle (wenn wir gleich die Abschlußtable aufgestellt hätten, wäre uns dieser Fehler vielleicht gar nicht aufgefallen). Wir müssen also dafür sorgen, dass auch die Verlierermannschaften einen Eintrag in punkte bekommen, und das geht am einfachsten über eine »nutzlose« Addition weiter oben:

```

<<<<<<
  if ($4 > $5) {
    punkte[$2] += 3; punkte[$3] += 0;
  } else if ($4 < $5) {
    punkte[$2] += 0; punkte[$3] += 3;
  } else {
<<<<<<

```

Damit ergibt sich die zumindest numerisch korrekte Tabelle

```

1.FC Kaiserslautern:0:-1
VfB Stuttgart:3:2
1.FC Köln:0:-1

```

```

Borussia Mönchengladbach:3:1
Bayer Leverkusen:3:3
Eintracht Frankfurt:0:-2
Bayern München:3:2
Hansa Rostock:0:-2
1860 München:3:1
Werder Bremen:3:3
SC Freiburg:0:-3
Borussia Dortmund:1:0
VfL Bochum:0:-1
Hertha BSC Berlin:0:-3
Hamburger SV:0:-3
Schalke 04 Gelsenkirchen:1:0
VfL Wolfsburg:3:1
Hannover 96:3:3

```

die dann nur noch sortiert werden muss.

Plattenplatzverbrauch pro Linux-Gruppe Wollen Sie herausfinden, welche primäre Gruppe auf Ihrem System die platzgierigsten Benutzer hat? Mit etwas wie »`du -s /home/*`« erhalten Sie zwar den Platzverbrauch pro Benutzer, aber das sagt Ihnen noch nichts über die Gruppen. Hierzu müssen Sie die benutzerbezogene Liste mit der Kennwortdatei korrelieren. Angesichts des `du`-Ausgabeformats

```

1234  /home/anselm
56    /home/tux
567   /home/hugo
342   /home/emil

```

sieht ein entsprechendes `awk`-Skript ungefähr so aus:

```

BEGIN {
    readgroups()
}
{
    sub(/^\/.*\/, "", $2)
    used[usergroup[$2]] += $1
}
END {
    for (g in used) {
        print used[g] " " g
    }
}

```

Die »`readgroups()`«-Funktion (die wir gleich zeigen) konstruiert ein Feld namens `usergroup`, das für jeden Benutzernamen den Namen der primären Gruppe angibt. Dieses Feld benutzen wir, um im Feld `used` den Plattenplatz aufzuaddieren, den die Heimatverzeichnisse der Gruppenmitglieder belegen (alle Benutzer, für die in `usergroup` derselbe Wert steht, sind in derselben Gruppe). Den Benutzernamen erhalten wir aus der `du`-Ausgabe, indem wir mit `sub` alles vom ersten bis zum letzten Schrägstrich entfernen, was etwas unappetitlich aussieht. Am Ende der Eingabe werden die Gruppen und ihr Platzbedarf ausgegeben.

Nun noch die »`readgroups()`«-Funktion:

```

# Lies die Gruppennamen der Benutzer nach USERGROUP (indiziert
# durch Benutzernamen). GROUPNAME und OLDFS sind lokale Variable.

function readgroups(groupname, oldfs) {

```

```

oldfs = FS
FS = ":"
while (getline <"/etc/group") {
    groupname[$3] = $1
}
close ("/etc/group")

while (getline <"/etc/passwd") {
    usergroup[$1] = groupname[$4]
}
close ("/etc/passwd")
FS = oldfs
}

```

Hier lernen Sie die awk-Funktionen zum Dateizugriff kennen: `getline` versucht, sich eine Zeile aus der angegebenen Datei zu holen, und liefert 1, wenn es noch eine Zeile gab oder 0 am Dateiende – die Datei wird beim ersten `getline` geöffnet und dann nur noch weiter gelesen. Mit `close` können Sie eine Datei nach Gebrauch schließen; das ist selten wirklich nötig, aber eine gute Angewohnheit, denn viele awk-Implementierungen können nur sehr wenige Dateien gleichzeitig offen halten (10 oder so). Bemerken Sie auch, dass wir zum Lesen von `/etc/group` und `/etc/passwd` den Feldtrenner von awk undefinieren müssen; wir merken uns den ursprünglichen Wert in `oldfs` und setzen ihn am Ende der Funktion wieder in Kraft, um das aufrufende Programm nicht zu verwirren.

Übungen

-  7.2 [!1] Geben Sie eine Shell-Kommandozeile an, mit der Sie eine Liste der von Ihnen am meisten gebrauchten Shell-Kommandos in absteigender Reihenfolge der Häufigkeit ausgeben können.
-  7.3 [!3] Unter welchen Umständen liefert das Programm `histcount` falsche Resultate, wenn Sie mit `$1` nur das erste Wort der Kommandozeile betrachten?
-  7.4 [!2] Schreiben Sie ein awk-Programm, das die Wörter in einem Text zählt und mitsamt ihrer Häufigkeit ausgibt.
-  7.5 [!1] Wie können Sie die unsortierte Bundesliga-Tabelle, die das Programm `bltab` liefert, angemessen sortieren?
-  7.6 [!2] Schreiben Sie ein awk-Programm, das für jeden Bundesligaverein die Stadionzuschauer aufaddiert und ausgibt.
-  7.7 [3] Schreiben Sie ein Skript (am besten mit `awk` und `sort`), das eine »schöne« Bundesliga-tabelle der Form

PL	MANNSCHAFT	SP	G	U	V	PUNKTE	TORE
1	Werder Bremen	34	22	8	4	74	41
2	Bayern München	34	20	8	6	68	31
3	Bayer Leverkusen	34	19	8	7	65	34
4	VfB Stuttgart	34	18	10	6	64	28
5	VfL Bochum	34	15	11	8	56	18
6	Borussia Dortmund	34	16	7	11	55	11
7	Schalke 04 Gelsenkirchen	34	13	11	10	50	7
8	Hamburger SV	34	14	7	13	49	-13
9	Hansa Rostock	34	12	8	14	44	1
10	VfL Wolfsburg	34	13	3	18	42	-5
11	Borussia Mönchengladbach	34	10	9	15	39	-9

12 Hertha BSC Berlin	34	9	12	13	39	-17
13 1.FC Kaiserslautern	34	11	6	17	39	-23
14 SC Freiburg	34	10	8	16	38	-25
15 Hannover 96	34	9	10	15	37	-14
16 Eintracht Frankfurt	34	9	5	20	32	-17
17 1860 München	34	8	8	18	32	-23
18 1.FC Köln	34	6	5	23	23	-25

erzeugt.² (*Tipp*: Schauen Sie in der awk-Dokumentation nach printf).



7.8 [2] Wenn Sie die Ausgabe von Übung 7.7 mit der tatsächlichen Bundesliga-Tabelle der Saison 2003/2004 vergleichen, wird Ihnen eine Anomalie auffallen: Die »echten« Plätze 13 bis 15 lauten

13 SC Freiburg	34	10	8	16	38	-25
14 Hannover 96	34	9	10	15	37	-14
15 1.FC Kaiserslautern	34	11	6	17	37	-23

Den Recken vom Betzenberg wurden nämlich wegen Lizenzverstößen zwei Punkte aberkannt. Wie können Sie diesen Umstand in Ihrer Tabellensoftware berücksichtigen?



7.9 [2] Schreiben Sie ein awk-Programm, das die Ausgabe von »du -s /home/*« übernimmt und den Platzbedarf der Heimatverzeichnisse »grafisch« darstellt, etwa so:

```
hugo      *****
emil      *****
tux       ***
anselm    *****
```

Kommandos in diesem Kapitel

awk	Programmiersprache für Textverarbeitung und Systemverwaltung	
		awk(1) 106
uniq	Ersetzt Folgen von gleichen Zeilen in der Eingabe durch die erste solche	
		uniq(1) 113

²Anhänger von SC Freiburg und Hannover 96 beachten bitte Übung 7.8.

Zusammenfassung

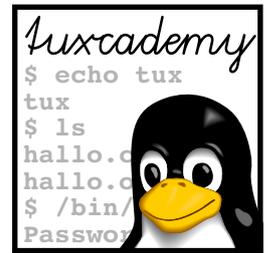
- `awk` ist eine einfache Programmiersprache für Textverarbeitungszwecke (im weitesten Sinne).
- Unter Linux sind verschiedene `awk`-Implementierungen gebräuchlich.
- `awk` liest seine Eingabe zeilenweise und kann für jede Eingabezeile eine Operation oder Operationen durchführen. Die Ergebnisse des Einlesevorgangs können weiter manipuliert und auf die Standardausgabe oder eine benannte Datei geschrieben werden.
- Sie können über reguläre Ausdrücke, Feldvergleiche usw. bestimmen, auf welche Zeilen eine Operation angewendet werden soll.
- Die besonderen »Auswahlkriterien« `BEGIN` und `END` erlauben die Ausführung von Kommandos vor bzw. nach dem Lesen der Eingabe.
- `awk` erlaubt die Verwendung von Variablen und unterstützt die meisten arithmetischen Ausdrücke der Programmiersprache C. Sie können auch Ihre eigenen Funktionen definieren.

Literaturverzeichnis

AKW88 Alfred V. Aho, Brian W. Kernighan, Peter J. Weinberger. *The AWK Programming Language*. Reading, MA: Addison-Wesley, 1988. ISBN 0-201-07981-X.
<http://cm.bell-labs.com/cm/cs/awkbook/>

DR97 Dale Dougherty, Arnold Robbins. *sed & awk*. Sebastopol, CA: O'Reilly & Associates, 1997, 2. Auflage. ISBN 1-56592-225-5.
<http://www.oreilly.de/catalog/sed2/>

Rob02 Arnold Robbins. *sed & awk – kurz & gut*. Sebastopol, CA: O'Reilly & Associates, 2002, 2. Auflage. ISBN 3-89721-246-3.
<http://www.oreilly.de/catalog/sedawkrepr2ger/>



8

SQL

Inhalt

8.1	Warum SQL?	124
8.1.1	Überblick.	124
8.1.2	SQL einsetzen	126
8.2	Tabellen definieren	127
8.3	Datenmanipulation und Abfragen.	129
8.4	Relationen	133
8.5	Praktische Beispiele	136

Lernziele

- Verstehen, wofür SQL benutzt wird
- Einfache Tabellen in SQL definieren können
- Einfache Datenmanipulation und Abfragen in SQL beherrschen

Vorkenntnisse

- Umgang mit Linux-Kommandos
- Umgang mit einem Texteditor

8.1 Warum SQL?

8.1.1 Überblick

Tupel
Tabellen

SQL (kurz für *Structured Query Language*) ist eine standardisierte Sprache zur Definition, Abfrage und Manipulation relationaler Datenbanken. Relationale Datenbanken verwalten Datensätze (im Jargon »Tupel« genannt) in »Tabellen«; Sie können sich das informell veranschaulichen, indem Sie sich ein großes Blatt Papier vorstellen, das in Zeilen und Spalten aufgeteilt ist. Die Zeilen stellen einzelne Datensätze und die Spalten einzelne Eigenschaften eines Datensatzes dar (Tabelle 8.1). Das Datenbanksystem macht es einfach, aus einer solchen Menge von Tupeln diejenigen herauszusuchen, die gewisse Anforderungen erfüllen (beispielsweise die Nachnamen aller Personen, die ein Raumschiff namens »USS Enterprise« kommandieren).

Normalisierung

Interessant wird das Ganze durch das Konzept der »Normalisierung«: Wenn Sie sich das Beispiel genau anschauen, wird Ihnen auffallen, dass die Namen von einigen Personen, Raumschiffen und Filmen mehrmals vorkommen. Das ist nicht wünschenswert, da man so Änderungen an verschiedenen Stellen in der Datenbank machen müsste. Es besteht die Gefahr, dass man eine Stelle vergisst und die Daten so inkonsistent werden. Statt dessen wird der Datenbestand »normalisiert«: Wir wissen, dass der »James T. Kirk« aus den ersten beiden Tupeln tatsächlich ein und dieselbe Person ist, die »USS Enterprise« aus den ersten drei Tupeln und die aus dem vierten jedoch zwei verschiedene Raumschiffe¹. Wir können unsere Tabelle also aufteilen in drei Tabellen, je eine für die Personen, Schiffe und Filme. Das ist in Tabelle 8.2 zu sehen. Beachten Sie die folgenden Punkte:

- Fremdschlüssel
- Die ursprüngliche Tabelle ist zur neuen Tabelle »Person« mutiert, aus der die Spalten mit dem Schiffsnamen und dem Filmnamen entfernt wurden. Statt dessen wird das Schiff nur noch über einen »Fremdschlüssel« angegeben, der auf ein Tupel in der Tabelle »Schiff« verweist. Dieser Fremdschlüssel erlaubt es uns, den Umstand auszudrücken, dass Willard Decker und James T. Kirk die »alte« USS Enterprise kommandiert haben, Jean-Luc Picard hingegen die »neue«. Man spricht hier auch von einer »1 : n-Beziehung«, weil dasselbe Schiff im Laufe seiner Existenz mehrere Kommandanten haben kann.
 - Da sowohl dieselbe Person in mehreren Filmen auftreten als auch derselbe Film mehr als eine Person aus der Liste enthalten kann, lässt die Beziehung zwischen Personen und Filmen sich nicht wie die zwischen Personen und Schiffen über einen einfachen Fremdschlüssel in der Tabelle »Person« abbilden. (So etwas nennt man auch eine »m : n-Beziehung«.) Statt dessen führen wir eine weitere Tabelle ein, deren Tupel jeweils eine Aussage der Form »Person x erscheint in Film y « darstellen.

¹Naja, Trekkies wie wir wissen das. Es *nicht* zu wissen ist aber vielleicht nicht wirklich ehrenrührig.

Vorname	Nachname	Schiff	Film
James T.	Kirk	USS Enterprise	Star Trek
James T.	Kirk	USS Enterprise	Star Trek: Generations
Willard	Decker	USS Enterprise	Star Trek
Jean-Luc	Picard	USS Enterprise	Star Trek: Generations
Han	Solo	Millennium Falcon	Star Wars 4
Han	Solo	Millennium Falcon	Star Wars 5
Wilhuff	Tarkin	Death Star	Star Wars 4
Malcolm	Reynolds	Serenity	Serenity

Bild 8.1: Eine Datenbanktabelle: Berühmte Raumschiffkommandanten im Kino

Person	Vorname	Nachname	Schiff	PersonFilm	Person	Film
1	James T.	Kirk	1	1	1	1
2	Willard	Decker	1	2	1	2
3	Jean-Luc	Picard	2	3	2	1
4	Han	Solo	3	4	3	2
5	Wilhuff	Tarkin	4	5	4	3
6	Malcolm	Reynolds	5	6	4	4
				7	5	3
				8	6	5

Schiff	Name	Film	Titel	Jahr	Budget (Mio.\$)
1	USS Enterprise	1	Star Trek	1979	46
2	USS Enterprise	2	Star Trek: Generations	1994	35
3	Millennium Falcon	3	Star Wars 4	1977	11
4	Death Star	4	Star Wars 5	1980	33
5	Serenity	5	Serenity	2005	39

Bild 8.2: Berühmte Raumschiffkommandanten im Kino (normalisiert)

- Wir haben ein paar Spalten zur Tabelle »Film« hinzugefügt, um die Daten interessanter zu machen.



Den Umstand, dass eine unserer Personen im Laufe ihrer Karriere mehrere verschiedene Raumschiffe kommandieren könnte, lassen wir hier im Interesse der Überschaubarkeit außer Acht.

Das »Datenmodell« ist so natürlich auf den ersten Blick etwas unübersichtlicher, aber dafür ist hier jede Information nur an einer einzigen Stelle abgelegt, was es im »wirklichen Leben« wesentlich leichter macht, den Überblick zu behalten.



Relationale Datenbanken wurden ursprünglich 1970 von Edgar F. Codd postuliert. Sie bilden heutzutage das Rückgrat der computergestützten Datenverarbeitung. Relationale Datenbanken sind zwar nur bedingt dazu geeignet, die objektorientierten Datenstrukturen moderner Software zu speichern, aber haben im Gegensatz zu den meisten anderen Datenbank-Ansätzen, etwa den »objektorientierten Datenbanken« den Vorteil, dass ihnen eine relativ gut verstandene mathematische Theorie (die »relationale Algebra«) zugrunde liegt und sie sich halbwegs effizient implementieren lassen.



Die erste Version von SQL (damals noch unter dem Namen SEQUEL) wurde in den frühen 1970er Jahren von Donald D. Chamberlin und Raymond F. Boyce entwickelt und bildete die Grundlage von IBMs erster relationaler Datenbank, System R. SQL wurde 1986 zum ersten Mal standardisiert und seitdem weiterentwickelt. Die aktuelle Version des Sprachstandards ist ISO 9075:2008 (vulgo ISO SQL:2008) und wurde, wer hätte es gedacht, im Juli 2008 amtlich.



Die offizielle Aussprache von SQL ist »Ess-Kuh-Ell«. Hin und wieder sprechen Leute den Namen auch aus wie das englische Wort *sequel*.

Übungen



8.1 [!2] Wie würden Sie im Datenmodell außer den Kommandanten der Raumschiffe auch noch andere Besatzungsmitglieder repräsentieren?



8.2 [2] Wie würden Sie im Datenmodell den Regisseur eines Films unterbringen?

8.1.2 SQL einsetzen

Aus der kommerziellen Datenverarbeitung ist SQL, wie erwähnt, nicht mehr wegzudenken – eine komplette Industrie lebt relativ gut von Implementierung und Support von Produkten, die relationale Datenbanksysteme implementieren. Hier sind einige unter Linux verfügbare relationale Datenbankprodukte auf der Basis von SQL:

MySQL und PostgreSQL Diese beiden Pakete sind vielleicht die ersten, an die man denkt, wenn man »Linux« und »SQL« assoziiert. Beide sind frei verfügbar und weit verbreitet und bieten eine solide Grundlage für gängige, nicht zu komplexe Anwendungen etwa im World-Wide-Web-Bereich.



Wie üblich in der Open-Source-Szene toben rege Glaubenskriege zwischen den Verfechtern der beiden Pakete. PostgreSQL-Jünger kritisieren, dass MySQL nicht den kompletten SQL-Standard umsetze, während MySQL-Anhänger argumentieren, dass die Teile, die MySQL implementiert, völlig ausreichen und die Geschwindigkeit von MySQL den Verzicht auf den Rest versüße. Wir werden hier keine Empfehlung aussprechen; beide Produkte sind frei verfügbar und können nach Bedarf getestet werden.

Oracle, Sybase, DB2 & Co. Für »unternehmenskritische« Anwendungen steht ein ganzer Zoo von kommerziell entwickelten und unterstützten Datenbankprodukten bereit, die (auch) auf Linux laufen. Diese Implementierungen stehen dem, was andere Plattformen, etwa Windows oder traditionelles Unix, zu bieten haben, in nichts nach und können ohne weiteres als Basis für alle Arten von Datenbankanwendungen genutzt werden.



Während Sie MySQL und PostgreSQL auf jedem beliebigen Linux-Rechner einsetzen können, unterstützen die Datenbankanbieter offiziell in der Regel nur bestimmte Plattformen, typischerweise die »Enterprise«-Versionen von Firmen wie Red Hat und Novell/SUSE, auf denen sie ihre Produkte »zertifizieren« – soll heißen, der Anbieter probiert das Produkt auf der betreffenden Plattform gründlich aus, verkündet, dass es funktioniert und ist anschließend bereit, zahlenden Kunden, die genau diese Plattform auch laufen haben, bei Problemen aus der Patsche zu helfen. Sie können natürlich ohne weiteres versuchen, Oracle & Co. auch auf anderen Linux-Distributionen als den offiziell zertifizierten zum Laufen zu bringen, und haben auch gute Chancen, dass das klappt (so unterschiedlich sind Linuxe auch nicht mehr). Allerdings sind Sie dann bei Problemen auf sich gestellt, was den Nutzen einer teuren kommerziellen Datenbank in Zweifel zieht – denn dann können Sie für die allermeisten Anwendungen auch auf MySQL oder PostgreSQL zurückgreifen.



Übrigens ist es kein größeres Problem, auch für PostgreSQL und MySQL »kommerziellen« Support zu bekommen (zu kommerziellen Preisen).

SQLite Während die anderen Pakete typischerweise einen »Datenbankserver« als eigenständigen Prozess laufen lassen, mit dem Anwendungsprogramme sich dann über das Netz verbinden können, wird SQLite als Bibliothek direkt ins Anwendungsprogramm eingebunden, ist ohne Konfiguration einsetzbar und liest und schreibt lokale Dateien. SQLite unterstützt den größten Teil des SQL92-Standards, inklusive Sachen wie Transaktionen und Trigger, mit denen MySQL sich unter Umständen noch schwer tut. SQLite eignet sich für den Einsatz in Geräten mit wenig Speicher wie MP3-Playern oder als Datenformat für Anwendungsprogramme.

Im Rest dieses Kapitel verwenden wir SQLite zur Illustration von SQL.



Bei Debian GNU/Linux und Ubuntu installieren Sie SQLite bequem über eines der Kommandos



```
# aptitude install sqlite3 als root
$ sudo aptitude install sqlite3 als normaler Benutzer
$ sudo apt-get install sqlite3 bei Ubuntu
```

Achten Sie darauf, dass Sie sich `sqlite3` holen – es gibt auch `sqlite`, aber das ist eine ältere Version, die nur noch aus Kompatibilitätsgründen angeboten wird.



Bei der SUSE ist SQLite (3) seit einiger Zeit Bestandteil der Standardinstallation, genau wie bei den Red-Hat-Distributionen. Sie müssen also nichts Besonderes machen, um die Beispiele in diesem Kapitel ausprobieren zu können – alles, was Sie dafür brauchen, ist schon installiert.

Übungen



8.3 [2] Unter welchen Bedingungen würden Sie ein frei verfügbares SQL-Datenbankprodukt wie MySQL oder PostgreSQL für eine Web-Präsenz einsetzen? Ändert sich Ihre Einschätzung in Abhängigkeit davon, ob die Web-Präsenz ein reines Hobbyprojekt ist oder unternehmenskritische Aufgaben erfüllt?



8.4 [2] Die Autoren von SQLite empfehlen, SQLite zur Speicherung von Anwendungsdaten zu benutzen (denken Sie zum Beispiel an Tabellen einer Tabellenkalkulation oder Konfigurationsdaten eines Web-Browsers). Welche Vor- und Nachteile dieses Ansatzes sehen Sie?

8.2 Tabellen definieren

Bevor Sie eine SQL-Datenbank mit Daten füllen können, müssen Sie definieren, wie die einzelnen Tabellen heißen sollen, welche Namen die Spalten haben sollen und welche Werte in einer Spalte abgelegt werden dürfen. SQL unterstützt eine reichhaltige Auswahl von Datentypen wie »Zeichenkette« oder »ganze Zahl«, auf die Sie bei der Definition der Spalten einer Tabelle zurückgreifen können. Die Gesamtheit der Tabellendefinitionen einer SQL-Datenbank nennen wir übrigens auch »Datenbankschema«.

Datentypen

Datenbankschema



Wir können hier aus Platz- und Zeitgründen nicht den kompletten Sprachumfang von SQL diskutieren. Gerade bei der Tabellendefinition gibt es auch große Unterschiede zwischen den einzelnen SQL-Datenbanken. Wir beschränken uns auf das absolute Minimum, auch weil das *Anlegen* von Tabellen nicht für die Prüfung LPI-102 relevant ist.

In SQL-Syntax könnte eine Definition der Tabelle »Person« aus unserem Beispiel ungefähr so aussehen:

```
CREATE TABLE person (
  id      INTEGER PRIMARY KEY,
  vorname VARCHAR(20),
  nachname VARCHAR(20),
  schiff_id INTEGER
);
```

Dabei stehen `INTEGER` und `VARCHAR(20)` für die SQL-Datentypen »ganze Zahl« und »Zeichenkette von bis zu 20 Zeichen Länge«. Mit »PRIMARY KEY« wird die Spalte `id`,

```

CREATE TABLE person (
  id      INTEGER PRIMARY KEY,
  vorname VARCHAR(20),
  nachname VARCHAR(20),
  schiff_id INTEGER
);

CREATE TABLE film (
  id      INTEGER PRIMARY KEY,
  titel   VARCHAR(40),
  jahr    INTEGER,
  budget  INTEGER
);

CREATE TABLE schiff (
  id      INTEGER PRIMARY KEY,
  name    VARCHAR(20)
);

CREATE TABLE personfilm (
  id      INTEGER PRIMARY KEY,
  person_id INTEGER,
  film_id  INTEGER
);

```

Bild 8.3: Das komplette Schema für die Datenbank

Primärschlüssel die der »laufenden Nummer« der Person in der Beispieldatenbank in Tabelle 8.2 entspricht, zum »Primärschlüssel« erklärt. Das heißt, die Datenbank kümmert sich darum, dass jeder Wert in dieser Spalte nur einmal vorkommt, und die Werte hier kommen als »Fremdschlüssel« in Tupeln aus anderen Tabellen in Frage (im Falle von person etwa in der Tabelle, die Personen und Filme zusammenbringt).

 Es ist eine gängige Konvention, Fremdschlüsseln den Namen der Tabelle, auf die sie »zeigen«, mit dem Suffix `_id` zu geben. `schiff_id` ist also ein Fremdschlüssel in die Tabelle `schiff`.

 Wenn Sie sich etwas mit SQL auskennen, werden Sie sich vermutlich daran stören, dass wir den Fremdschlüssel `schiff_id` einfach nur als `INTEGER` definiert haben. Gestatten Sie uns für heute diese simple Sicht der Dinge.

 Groß- und Kleinschreibung ist in SQL übrigens egal. Wir verfolgen die gängige Konvention, die Namen von Tabellen und Spalten klein und alles, was zu SQL selbst gehört, groß zu schreiben, aber das können Sie im Grunde machen, wie Sie mögen. Allerdings tun Sie sich und uns einen Gefallen, wenn Sie mit sich selber konsistent bleiben.

In Bild 8.3 sehen Sie das komplette SQL-Schema für unsere Beispieldatenbank. Mit dem Schema in der Datei `commanders-schema.sql` können Sie in SQLite die tatsächliche Datenbank zum Beispiel wie folgt initialisieren:

```
$ sqlite3 comm.db <commanders-schema.sql
```

Hierbei wird die Datenbank in der Datei `comm.db` untergebracht. SQLite übernimmt immer den Namen der Datenbankdatei als Parameter; wenn die Datei schon existiert, wird sie geöffnet, ansonsten wird sie neu angelegt.

Wenn Sie `sqlite3` aufrufen, ohne die Standardeingabe umzulenken, bekommen Sie eine interaktive Sitzung:

```

$ sqlite3 comm.db
SQLite version 3.5.9
Enter ".help" for instructions
sqlite> .tables
film      person   personfilm  schiff
sqlite> .schema schiff
CREATE TABLE schiff (
  id      INTEGER PRIMARY KEY,
  name    VARCHAR(20),
);
sqlite> _

```



SQLite bietet diverse »Verwaltungskommandos« an, deren Namen alle mit einem Punkt anfangen – im Beispiel zeigen wir `.tables`, das die Tabellen in der Datenbank auflistet, und `.schema`, mit dem Sie sich das Datenbankschema anschauen können. Es gibt aber noch einige mehr; `.help` zeigt die komplette Liste.

Übungen



8.5 [!2] Erstellen Sie eine SQLite-Datenbank auf der Basis des in diesem Abschnitt angegebenen Datenbankschemas für die Raumschiff-Kommandanten.



8.6 [3] Realisieren Sie die Erweiterungen aus Übung 8.1 und Übung 8.2 als SQL-Schema.

8.3 Datenmanipulation und Abfragen

SQL erlaubt nicht nur die Definition von Datenbankschemata, sondern auch das Einbringen, Ändern, Abfragen und Löschen von Daten (Tupeln). Maßgeblich dafür ist natürlich das jeweils aktuelle Datenbankschema. Sie könnten zum Beispiel ein paar Raumschiffe und Filme in die Datenbank aufnehmen:

```
sqlite> INSERT INTO schiff VALUES (1, 'USS Enterprise');
sqlite> INSERT INTO schiff VALUES (2, 'USS Enterprise');
sqlite> INSERT INTO film VALUES (1, 'Star Trek', 1979, 46);
sqlite> INSERT INTO film VALUES (2, 'Star Trek: Generations',
...> 1994, 35);
```



Beachten Sie, dass wir hier die Werte für die Primärschlüssel explizit vergeben. Das ist in einer größeren Datenbank eher lästig, da Sie für jedes neue Tupel herausfinden müßten, was der korrekte Wert ist – es ist bequemer, die Datenbank selbst den richtigen Wert eintragen zu lassen, und die meisten SQL-Datenbanken können das auch. Bei SQLite muss der Primärschlüssel als »INTEGER PRIMARY KEY« definiert sein, und Sie müssen statt eines expliziten Werts dafür den »magischen« Wert NULL angeben:

```
sqlite> INSERT INTO film VALUES (NULL, 'Star Wars 4', 1977, 11);
```

Auch Personen und Tupel in der Person-Film-Beziehung können Sie entsprechend eingeben:

```
sqlite> INSERT INTO person VALUES (1, 'James T.', 'Kirk', 1);
sqlite> INSERT INTO person VALUES (2, 'Willard', 'Decker', 1);
sqlite> INSERT INTO personfilm VALUES (NULL, 1, 1);
sqlite> INSERT INTO personfilm VALUES (NULL, 1, 2);
sqlite> INSERT INTO personfilm VALUES (NULL, 2, 1);
```

Beachten Sie, wie wir für die Fremdschlüssel `schiff_id` in der Tabelle `person` sowie `person_id` und `film_id` in `personfilm` die Primärschlüssel der jeweiligen Tupel aus den anderen Tabellen angeben.



Wenn Sie sich ein bisschen mit Programmieren auskennen, werden Sie dabei vielleicht ein mulmiges Gefühl kriegen. Immerhin garantiert Ihnen niemand, dass es in der »anderen Tabelle« überhaupt ein Tupel mit dem betreffenden Primärschlüssel *gibt*². Das ist kein prinzipielles Problem von SQL, sondern eher eins unserer einfachen Beispiele hier – »gute« SQL-Datenbanken (nicht SQLite und MySQL auch nicht immer) unterstützen

referentielle Integrität ein Konzept namens »referentielle Integrität«, das genau dieses Problem lösen hilft. Dabei ist es möglich, im Schema zu definieren, dass `schiff_id` ein Fremdschlüssel auf die Tabelle `schiff` ist, und die Datenbank stellt dann sicher, dass die Werte für `schiff_id` vernünftig sind. Referentielle Integrität beinhaltet auch andere nette Eigenschaften; in unserem einfachen Beispiel müssten Sie, wenn Sie zum Beispiel das Tupel für James T. Kirk aus der `person`-Tabelle löschen, selber dafür sorgen, auch die Tupel in `personfilm` zu entfernen, die James T. Kirk mit Filmen verbinden. Bei einer Datenbank, die referentielle Integrität betrachtet, könnte das automatisch passieren.

Abfragen Mit dem Datenbestand aus Tabelle 8.2 in unserer Datenbank können wir uns jetzt einige Abfragen anschauen. Alle Tupel einer Tabelle erhalten wir wie folgt:

Alle Tupel

```
sqlite> SELECT * FROM schiff;
1|USS Enterprise
2|USS Enterprise
3|Millennium Falcon
4|Death Star
5|Serenity
```

bestimmte Spalten Der Stern (»*«) steht dabei für »alle Spalten«. Wenn Sie nur bestimmte Spalten haben möchten, müssen Sie sie aufzählen:

```
sqlite> SELECT vorname, nachname FROM person;
James T.|Kirk
Willard|Decker
Jean-Luc|Picard
Han|Solo
Wilhuff|Tarkin
Malcolm|Reynolds
```

Ausdrücke Sie sind nicht darauf angewiesen, die Werte von Spalten so auszugeben, wie sie in der Datenbank stehen, sondern können mit ihnen »Ausdrücke« bilden. Im folgenden Beispiel geben wir die Vor- und Nachnamen der Kommandanten ohne den häßlichen Strich aus. Der »||«-Operator hängt zwei Zeichenketten aneinander:

```
sqlite> SELECT nachname || ', ' || vorname FROM person;
Kirk, James T.
Decker, Willard
Picard, Jean-Luc
Solo, Han
Tarkin, Wilhuff
Reynolds, Malcolm
```

Rechnen können Sie natürlich auch:

```
sqlite> SELECT titel, budget * 0.755 FROM film;
Star Trek|34.73
Star Trek: Generations|26.425
Star Wars 4|8.305
Star Wars 5|24.915
Serenity|29.445
```

(Ob es einen großen Sinn ergibt, Dollars von 1977 nach dem Kurs von Januar 2009 in Euro umzurechnen, steht natürlich auf einem anderen Blatt.)

Aggregatfunktionen »Aggregatfunktionen« machen es möglich, Operationen wie Summen- oder Mittelwertbildung auf Spalten aller Tupel anzuwenden. Zum Beispiel können Sie die Anzahl und das durchschnittliche Budget aller Filme in der Datenbank (Inflation wird ignoriert) wie folgt bestimmen:

²Es sei denn, Sie programmieren in C; in diesem Fall ist da natürlich nichts Schlimmes dabei.

```
sqlite> SELECT COUNT(budget), AVG(budget) FROM film;
5|32.8
```

Hier bekommen Sie natürlich nur noch ein Tupel als Resultat.



Das »COUNT(budget)« wird Sie vielleicht etwas überraschen, aber es steht für »die Anzahl aller Tupel in der Tabelle, deren Spalte budget einen Wert«. Es wäre ja möglich, dass das Budget für einen Film nicht bekannt ist – »Star Trek« ist da übrigens hart an der Grenze –, und in diesem Fall könnten Sie den Wert NULL dort eintragen (nicht zu verwechseln mit »0« für einen Film, dessen Produktion nichts gekostet hat). Solche Filme werden bei der Aggregatbildung dann einfach übergangen. Wenn Sie die Anzahl aller Filme wissen wollen, egal ob deren Budget bekannt ist oder nicht, können Sie »COUNT(*)« sagen.

Interessant gerade im Zusammenhang mit Aggregatfunktionen ist auch die Möglichkeit, Tupel »gruppenweise« zu betrachten. Angenommen, uns interessiert das durchschnittliche Budget der Filme, die in einem Jahrzehnt produziert wurden, für alle Jahrzehnte in der Datenbank. Dann können wir etwas benutzen wie

Gruppierung

```
sqlite> SELECT jahr/10, AVG(budget) FROM film GROUP BY jahr/10;
197|28.5
198|33.0
199|35.0
200|39.0
```

(Vielleicht ist das nicht das genialste mögliche Ausgabeformat, aber es tut, was es soll.) Die »GROUP BY«-Klausel definiert dabei, dass alle Tupel, bei denen jahr/10 denselben Wert ergibt, gemeinsam betrachtet werden sollen, und die Spaltenangaben beziehen sich dann jeweils auf alle Tupel in einer solchen Gruppe. Das heißt, das AVG(budget) berechnet nicht mehr den Durchschnitt über *alle* Tupel, sondern nur noch über diejenigen Tupel, die sich eine Gruppe teilen.



Die Namen der Spalten in der Ausgabe haben sich bisher aus den Namen der Spalten der Eingabe-Tupel bestimmt. SQL gibt Ihnen aber die Möglichkeit, sich für die Spalten in der Ausgabe eigene Namen zu wünschen. Gerade bei etwas komplexeren Abfragen kann das die Übersichtlichkeit erhöhen:

Spaltennamen

```
sqlite> SELECT jahr/10 AS jahrzehnt, AVG(budget)
...> FROM film GROUP BY jahrzehnt;
```

ist etwas weniger umständlich zu lesen als das Original.



Wie Ihre Ausgabe tatsächlich aussieht, hängt vor allem von Ihrem Datenbanksystem ab. SQLite ist da standardmäßig ziemlich primitiv, was wohl am ehesten darauf zurückzuführen ist, dass es im Geiste des »Unix-Werkzeugkastens« versucht, Ausgabe zu produzieren, die leicht von anderen Programmen weiterzuverarbeiten und frei von überflüssigem Gelaber ist. Für den interaktiven Gebrauch können Sie jedoch bequemere Ausgabeformate einstellen:

```
sqlite> .headers on
sqlite> .mode tabs
sqlite> SELECT jahr/10 AS jahrz, AVG(budget)
...> FROM film GROUP BY jahrz
jahrz  avg(budget)
197    28.5
198    33.0
199    35.0
200    39.0
```

Hier werden die einzelnen Spalten durch Tabs getrennt. Mit »`.mode column`« erhalten Sie ein Format, bei dem Sie über das Kommando `.width` explizite Spaltenbreiten vorgeben können:

```
sqlite> .mode column
sqlite> .width 10 12
sqlite> SELECT jahr/10 AS jahrzehnt, AVG(budget)
...> FROM film GROUP BY jahrzehnt
jahrzehnt  avg(budget)
-----
197        28.5
198        33.0
199        35.0
200        39.0
```

Tupel auswählen Oft wollen Sie nicht mit allen Tupeln einer Tabelle etwas anfangen, sondern nur mit einem Teil davon, den Sie gemäß gewisser Kriterien auswählen wollen. Auch das geht mit `SELECT`. Hier zum Beispiel die Liste aller Filme in unserer Datenbank, die seit 1980 gedreht wurden:

```
sqlite> SELECT titel, jahr FROM film WHERE jahr >= 1980;
Star Trek: Generations|1994
Star Wars 5|1980
Serenity|2005
```

Sie können die Ausgabe auch sortieren:

```
sqlite> SELECT titel, jahr FROM film WHERE jahr >= 1980
...> ORDER BY jahr ASC;
Star Wars 5|1980
Star Trek: Generations|1994
Serenity|2005
```

»ASC« steht hier für »aufsteigend« (engl. *ascending*), das Gegenteil wäre »DESC« (engl. *descending*):

```
sqlite> SELECT titel FROM film ORDER BY budget DESC;
Star Trek
Serenity
Star Trek: Generations
Star Wars 5
Star Wars 4
```

Sub-SELECTs In der `WHERE`-Klausel dürfen Sie andere `SELECT`-Kommandos verwenden, vorausgesetzt diese liefern etwas, was zu dem betreffenden Ausdruck passt. Hier ist die Liste der Filme mit überdurchschnittlichem Budget:

```
sqlite> SELECT titel, jahr FROM film
...> WHERE budget > (SELECT AVG(budget) FROM FILM);
Star Trek|1979
Star Trek: Generations|1994
Star Wars 5|1980
Serenity|2005
```

Tupel ändern Ändern können Sie Tupel mit dem Kommando `UPDATE`:

```
sqlite> UPDATE person SET vorname='James Tiberius' WHERE id=1;
sqlite> SELECT vorname, nachname FROM person WHERE id=1;
James Tiberius|Kirk
```

Die WHERE-Klausel ist hier extrem wichtig, damit Änderungen gezielt stattfinden. Ein Ausrutscher und die Katastrophe ist perfekt:

```
sqlite> UPDATE person SET vorname='James Tiberius';
sqlite> SELECT vorname || ' ' || nachname FROM person;
James Tiberius Kirk
James Tiberius Decker
James Tiberius Picard
James Tiberius Solo
James Tiberius Tarkin
James Tiberius Reynolds
```

Wobei Sie sich das natürlich auch sinnvoll zu Nutze machen können:

```
sqlite> UPDATE film SET budget=budget * 0.755; Alles in Euro
```

Zu guter Letzt können Sie mit DELETE FROM Tupel löschen. Die Warnung über Tupel löschen WHERE gilt auch hier:

```
sqlite> DELETE FROM person WHERE nachname='Tarkin';
```

Alle Tupel in einer Tabelle löschen Sie mit

```
sqlite> DELETE FROM person; Kids, don't try this at home
```



Denken Sie an unsere Anmerkung oben über »referentielle Integrität«: Je nachdem, wie potent Ihr Datenbanksystem ist, müssen Sie gegebenenfalls selber darauf achten, dass zusammen mit einem Tupel auch solche Tupel verschwinden, die Fremdschlüssel auf dieses Tupel enthalten.

Übungen



8.7 [1] Fügen Sie die Tupel aus Tabelle 8.2 in die SQLite-Datenbank aus Übung 8.5 ein. Wenn Sie Science-Fiction-Filme mögen, dann erweitern Sie den Datenbestand noch ein bisschen. (Wir sind zum Beispiel große Fans von »Galaxy Quest«.)



8.8 [2] Geben Sie ein SQL-Kommando an, das alle vor 1985 produzierten Filme in der Datenbank auflistet, deren Budget unter 40 Millionen Dollar betrug.

8.4 Relationen

Wirklich interessant werden SQL-Abfragen da, wo Sie mehrere Tabellen zusammenbringen. Es könnte zum Beispiel sein, dass Sie eine Liste aller Kommandanten zusammen mit den Namen ihrer Schiffe haben möchten (in den Tupeln in person stehen ja nur die Primärschlüssel der Schiffe):

```
sqlite> SELECT * FROM person, schiff
...> WHERE person.schiff_id=schiff.id;
1|James T.|Kirk|1|1|USS Enterprise
2|Willard|Decker|1|1|USS Enterprise
3|Jean-Luc|Picard|2|2|USS Enterprise
4|Han|Solo|3|3|Millennium Falcon
5|Wilhuff|Tarkin|4|4|Death Star
6|Malcolm|Reynolds|5|5|Serenity
```


In unserem Spielzeugbeispiel ist das noch nicht wirklich ein Problem, aber wenn Sie sich vorstellen, dass das Finanzamt die Arbeitgeber aller Steuerzahler bestimmen möchte, dann sind die Dimensionen doch etwas anders.

Aus diesem Grund gibt es in SQL die Möglichkeit, direkt zu sagen, welche Kombinationen von Tupeln Sie interessant finden, anstatt *alle* möglichen Kombinationen zu erzeugen und dann die uninteressanten zu verwerfen. Und das sieht so aus:

```
sqlite> SELECT *
...> FROM person JOIN schiff ON person.schiff_id=schiff.id;
                                     Ergebnis: siehe oben
```



Ob das in der Praxis wirklich ein Problem darstellt, hängt auch von Ihrem Datenbanksystem ab. Ein großer Anteil des Entwicklungsaufwands für ein Datenbanksystem geht in die »Abfrageoptimierung«, also den Teil, der entscheidet, wie genau SELECT-Kommandos abgearbeitet werden. Clevere Datenbanksysteme können erkennen, dass das erste Beispiel und das Beispiel mit JOIN im Prinzip dasselbe tun und beide auf dieselbe (effiziente) Weise abhandeln. SQLite zum Beispiel erzeugt für beide Anfragen denselben Bytecode. Auf der anderen Seite sind diese Anfragen noch sehr simpel, und im wirklichen Leben werden Sie es mit komplizierteren Anfragen zu tun bekommen, die einem Abfrageoptimierer möglicherweise so starkes Kopferbrechen machen, dass er offensichtliche Vereinfachungen nicht mehr erkennt. Wir empfehlen Ihnen zur Sicherheit die Verwendung der JOIN-Form.

Abfrageoptimierung

Wenn Sie wissen wollen, welcher Kommandant in welchem Film aufgetreten ist, müssen Sie die personfilm-Tabelle heranziehen:

```
sqlite> SELECT vorname, nachname, titel
...> FROM person p JOIN personfilm pf ON p.id=pf.person_id
...> JOIN film f ON pf.film_id=f.id;
James T.|Kirk|Star Trek
James T.|Kirk|Star Trek: Generations
Willard|Decker|Star Trek
Jean-Luc|Picard|Star Trek: Generations
Han|Solo|Star Wars 4
Han|Solo|Star Wars 5
Wilhuff|Tarkin|Star Wars 4
Malcolm|Reynolds|Serenity
```

Auch Relationen zwischen drei (und mehr) Tabellen sind demnach kein Problem – Sie müssen nur den Überblick behalten!

Hier sind noch ein paar Beispiele für relationale Abfragen. Zuerst die Liste der Kommandanten, die in Filmen seit 1980 aufgetreten sind:

```
sqlite> SELECT jahr, titel, vorname || ' ' || nachname
...> FROM person p JOIN personfilm pf ON p.id=pf.person_id
...> JOIN film f ON pf.film_id=f.id
...> WHERE jahr >= 1980 ORDER BY jahr ASC;
1980|Star Wars 5|Han Solo
1994|Star Trek: Generations|James T. Kirk
1994|Star Trek: Generations|Jean-Luc Picard
2005|Serenity|Malcolm Reynolds
```

Hier eine Liste der Filme, in denen zwei oder mehr Kommandanten auftreten:

```
sqlite> SELECT titel
...> FROM film f JOIN personfilm pf ON f.id=pf.film_id
...> GROUP BY film_id HAVING COUNT(*) > 1;
```

```
Star Trek
Star Trek: Generations
Star Wars 4
```

Die »GROUP BY«-Klausel sorgt hier dafür, dass Tupel in personfilm, die sich auf denselben Film beziehen, gemeinsam betrachtet werden. HAVING (das wir noch nicht besprochen haben) ist ähnlich zu WHERE, aber greift erst nach der Gruppierung und erlaubt die Verwendung von Aggregatfunktionen (was WHERE nicht zulässt); das COUNT(*) im HAVING zählt also die Tupel in der jeweiligen Gruppe.

Übungen



8.9 [!1] Welche Ausgabe liefert das SQL-Kommando

```
SELECT * FROM person, schiff
```

bei unserer Beispiel-Datenbank?

8.5 Praktische Beispiele

Nachdem Sie jetzt die Anfangsgründe von SQL kennengelernt haben, werden Sie sich sicherlich fragen, was Ihnen das für die Praxis bringt (wenn Sie nicht sowieso schon mit Datenbanken zu tun haben, aber dann ist dieses ganze Kapitel wahrscheinlich für Sie ein alter Hut). In diesem Abschnitt zeigen wir Ihnen noch ein paar Ideen, was Sie mit einer SQL-Datenbank wie SQLite im »täglichen Leben« anfangen können.

Firefox (oder, für Debian-GNU/Linux-Anwender, Iceweasel) verwendet seit der Version 3 SQLite für eine steigende Anzahl seiner internen Verwaltungsdateien. In einigen davon können Sie herumschnüffeln und interessante Dinge lernen; alles, was im Verzeichnis ~/.mozilla/firefox/*.Standard-Benutzer zu finden ist (der Stern ist in Wirklichkeit ein Code, um den Namen eindeutig zu machen) und auf .sqlite endet, ist potentiell Freiwild.

Die Datei formhistory.sqlite zum Beispiel enthält die Vorschlagswerte, die Firefox in Web-Formulare einträgt; das Schema ist relativ überschaubar:

```
CREATE TABLE moz_formhistory (
  id INTEGER PRIMARY KEY,
  fieldname LONGVARCHAR,
  value LONGVARCHAR
);
```

Sie können sich also zum Beispiel mit

```
sqlite> SELECT value FROM moz_formhistory WHERE fieldname='address';
```

anschauen, was Firefox Ihnen vorschlägt, wenn ein Textfeld in einem Web-Formular den (internen HTML-)Namen address hat. Ebenso haben Sie so die Möglichkeit, Vorschlagswerte, die Sie nerven, gezielt loszuwerden (was Firefox Ihnen selber nicht erlaubt) – ein passendes DELETE FROM schafft vollendete Tatsachen.



Wenn Sie sicher sein wollen, machen Sie das, wenn Firefox gerade nicht läuft – aber es sollte auch so klappen.

Ebenfalls seit Firefox 3 enthält die Datei places.sqlite Interessantes wie Ihre Lesezeichen (in moz_bookmarks), die angesurften Seiten (in moz_historyvisits und moz_places) und so manches andere mehr.

Amarok Benutzen Sie das KDE-Musikverwaltungsprogramm Amarok? Dann können Sie zum Beispiel bequem Ihre persönlichen »Top Ten« herausfinden:

```
$ sqlite3 ~/.kde/share/apps/amarok/collection.db \  
> 'SELECT url, playcounter FROM statistics ORDER BY  
> playcounter DESC LIMIT 10;'
```

(Das »LIMIT 10« am Schluss begrenzt das Ergebnis auf maximal 10 Tupel.) Wenn Sie statt dem URL der Datei für die Stücke lieber Künstler und Titel sehen wollen, ist die Abfrage nur etwas komplizierter:

```
$ sqlite3 ~/.kde/share/apps/amarok/collection.db \  
> 'SELECT title, playcounter FROM statistics s  
> JOIN tags t ON s.url=t.url  
> ORDER BY playcounter DESC LIMIT 10;'
```



Eine etwas bequemere Schreibweise für den JOIN-Ausdruck ist übrigens JOIN tags USING(url), da in beiden Tabellen die Spalte url verwendet wird, um die Verbindung herzustellen.

Schauen Sie sich das Amarok-Schema mal mit .schema an. Ihnen fallen bestimmt andere interessante Anwendungen ein.

Kalender (für Arme) Wenn Sie zu den Leuten gehören, die schon mal versehentlich Tante Friedas Geburtstag vergessen, dann sollten Sie um des lieben Familienfriedens willen versuchen, dass das nicht zu häufig vorkommt. Es wäre nett, bei Bedarf Aufklärung über die Familienfeste und Gedenktage der näheren Zukunft bekommen zu können – idealerweise mit etwas Vorwarnung, um sich noch um Geschenke, Glückwunschkarten und ähnliches kümmern zu können.

Da Sie sich ja gut mit der Shell auskennen, sollte es kein Problem sein, einen kleinen »Kalender« zusammenzubasteln, der Ihnen dabei hilft, den Überblick zu behalten. Natürlich wollen wir hier weder Evolution noch KOrganizer oder Google Calendar Konkurrenz machen, sondern backen lieber kleine (wenn auch hoffentlich leckere) Brötchen.

Am Anfang unserer Überlegungen steht das Datenbankschema. Wir möchten gerne verschiedene Sorten von Terminen speichern (Geburtstage, Hochzeitstage, Jubiläen) sowie für jeden Termin nicht nur die Kategorie, sondern auch das Datum und eine Erklärung (damit wir wissen, worum es überhaupt geht). Hier ist ein Vorschlag für die Kategorie:

```
CREATE TABLE type (  
  id INTEGER PRIMARY KEY,  
  abk VARCHAR(1),  
  name VARCHAR(100)  
);
```

*Abkürzung
Langer Name*

Und hier die eigentlichen Ereignisse:

```
CREATE TABLE event (  
  id INTEGER PRIMARY KEY,  
  type_id INTEGER,  
  jahr INTEGER,  
  datum VARCHAR(5),  
  beschreibung VARCHAR(100)  
);
```

Fremdschlüssel

Wir speichern das Datum des Ereignisses getrennt nach Jahr (Spalte jahr) und Monat bzw. Tag in der Form MM-TT (Spalte datum), aus Gründen, die hoffentlich gleich noch klar werden. Einträge können wir wie folgt machen:

```

INSERT INTO type VALUES (1, 'G', 'Geburtstag');
INSERT INTO type VALUES (2, 'H', 'Hochzeitstag');
INSERT INTO type VALUES (3, 'J', 'Jubiläum');

INSERT INTO event VALUES (1, 1, 1926, '01-17', 'Tante Frieda');
INSERT INTO event VALUES (2, 1, 1934, '01-21', 'Onkel Heinz');
INSERT INTO event VALUES (3, 2, 2002, '02-05', 'Susi und Martin');

```

Jetzt können wir daran gehen, die interessante Frage zu beantworten: Was tut sich in der nächsten Woche? Mit anderen Worten: Wenn wir die Termine in events ins aktuelle Jahr »teleportieren«, welche davon fallen dann in den Zeitraum von »heute« bis »heute plus sieben Tage«? (Wir gehen mal davon aus, dass eine Woche reicht, um eine Glückwunschkarte oder ein Geschenk zu besorgen. Der Schreibwarenladen um die Ecke wird schon was haben, und Amazon & Co. liefern notfalls ja auch recht schnell.)

Vereinfacht wird das durch die Datumsfunktionen von SQLite, die wir bisher noch nicht beleuchtet haben. Die Funktion DATE() akzeptiert als erstes Argument zum Beispiel ein Datum in der üblichen »internationalen« Schreibweise (also erst das Jahr, dann den Monat und dann den Tag, durch Bindestriche getrennt – etwa 2009-01-14) oder die Zeichenkette now (mit der naheliegenden Bedeutung). Als zweites und folgende Argumente können Sie dann »Modifikatoren« angeben, die das Datum aus dem ersten Argument ändern. Den Zeitpunkt »heute plus sieben Tage« bekommen Sie also einfach mit

```

sqlite> SELECT DATE('now', '+7 days');
2009-01-20

```

Heute ist übrigens der 13. Januar 2009

Damit ist die Lage klar: Alle »aktuellen« Termine finden wir mit einer Abfrage wie

```

sqlite> SELECT DATE('2009-' || datum) AS d, beschreibung, jahr
...> FROM event
...> WHERE d >= DATE('now') AND d <= DATE('now', '+7 days')
2009-01-17|Tante Frieda|1926

```



Statt

```
d >= DATE('now') AND d <= DATE('now', '+7 days')
```

können Sie auch

```
d BETWEEN DATE('now') AND DATE('now', '+7 days')
```

schreiben.

Jetzt können wir das Ganze nett in einem Shellskript verpacken, das für die Bestimmung des aktuellen Jahrs sorgt (in SQL etwas schwerfällig) und die Ausgabe nett formatiert (in SQL *verflix*t schwerfällig). Das Resultat könnte zum Beispiel so aussehen wie in Bild 8.4 und mit einer Datenbankdatei in `~/cal.db` eine Ausgabe produzieren wie

```

$ calendar-upcoming 60
2009-01-17: Tante Frieda (83. Geburtstag)
2009-01-21: Onkel Heinz (75. Geburtstag)
2009-02-05: Susi und Martin (7. Hochzeitstag)
2009-02-06: ROM-TOS-Jubiläum (23. Jubiläum)
2009-02-17: Kaninchenzuchtverein Langohr (124. Jubiläum)

```

Nächste zwei Monate

Es sind viele Erweiterungen denkbar. Schauen Sie dazu in die Übungen.

```
#!/bin/bash
# calendar-upcoming [Grenze]

caldb=$HOME/.cal.db
year=$(date +%Y)
limit=${1:-14}

sqlite3 $caldb \
  "SELECT DATE('$year-' || datum) AS d, name, beschreibung, jahr
  FROM event JOIN type ON event.type_id=type.id
  WHERE d >= DATE('now') AND d <= DATE('now', '+$limit days')
  ORDER BY d ASC;" \
| awk -F'|' '{ print $1 ": " $3 " (" year-$4 ". " $2 ")" }' year=$year
```

Bild 8.4: Das Skript calendar-upcoming

Übungen



8.10 [3] Erweitern Sie den »Kalender für Arme« um eine Kategorie »Feiertag«. Bei Feiertagen soll kein »Alter« in der Ausgabe erscheinen, also etwas wie

```
2009-02-17: Kaninchenzuchtverein Langohr (124. Jubiläum)
2009-02-23: Rosenmontag (Feiertag)
2009-03-01: Cousin Fritz (29. Geburtstag)
```



8.11 [3] Schreiben Sie ein Shellskript calendar-holidays, das das Datum des Ostersonntags in einem Jahr in der Form 2009-04-12 übergeben bekommt und die Feiertage für das betreffende Jahr in die Kalender-Datenbank einträgt.



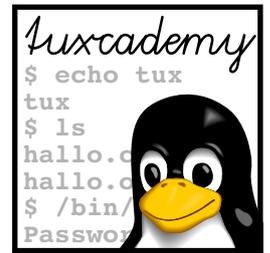
8.12 [4] (Fortsetzung der vorigen Aufgabe, größeres Projekt) Schreiben Sie ein Programm, das für eine gegebene Jahreszahl das Osterdatum berechnet. Informationen darüber finden Sie zum Beispiel unter http://de.wikipedia.org/wiki/Gaußsche_Osterformel. (*Tipp:* Benutzen Sie awk.) Schreiben Sie calendar-holidays so um, dass es dieses Programm benutzt – es braucht dann natürlich nur noch die Jahreszahl und nicht mehr das Osterdatum.



8.13 [2] Sorgen Sie dafür, dass Sie jeden Morgen die aktuellen Termine in Ihrem E-Mail-Postfach vorfinden. (Dazu brauchen Sie wahrscheinlich den Inhalt von Kapitel 9, also kommen Sie ggf. später auf diese Aufgabe zurück.)

Zusammenfassung

- SQL (kurz für *Structured Query Language*) ist eine standardisierte Sprache zur Definition, Abfrage und Manipulation relationaler Datenbanken.
- Normalisierung ist wichtig für die Konsistenz von relationalen Datenbanken.
- Für Linux stehen diverse SQL-Datenbankprodukte zur Verfügung.
- Die Gesamtheit der Tabellendefinitionen einer SQL-Datenbank heißt »Datenbankschema«.
- SQL erlaubt nicht nur die Definition von Datenbankschemata, sondern auch das Einbringen, Ändern, Abfragen und Löschen von Daten (Tupeln).
- Aggregatfunktionen machen es möglich, Operationen wie Summen- oder Mittelwertbildung auf Spalten aller Tupel anzuwenden.
- Über Relationen können Sie Informationen aus mehreren Tabellen gemeinsam verarbeiten.
- Viele Anwendungsprogramme unter Linux setzen SQLite zur Datenhaltung ein.



9

Zeitgesteuerte Vorgänge – at und cron

Inhalt

9.1	Allgemeines.	142
9.2	Einmalige Ausführung von Kommandos	142
9.2.1	at und batch	142
9.2.2	at-Hilfsprogramme	144
9.2.3	Zugangskontrolle.	145
9.3	Wiederholte Ausführung von Kommandos	145
9.3.1	Aufgabenlisten für Benutzer	145
9.3.2	Systemweite Aufgabenlisten	147
9.3.3	Zugangskontrolle.	148
9.3.4	Das Kommando crontab	148
9.3.5	Anacron	148

Lernziele

- Kommandos mit at zu einem zukünftigen Zeitpunkt ausführen können
- Kommandos periodisch mit cron ausführen können
- anacron kennen und einsetzen können

Vorkenntnisse

- Umgang mit Linux-Kommandos
- Umgang mit einem Texteditor

9.1 Allgemeines

Ein wichtiger Bestandteil der Systemverwaltung besteht darin, wiederholt ablaufende Vorgänge zu automatisieren. Eine denkbare Aufgabe wäre, dass sich der Mailserver eines Firmennetzwerkes in regelmäßigen Abständen beim Internet-Provider einwählt und die eingegangenen Nachrichten abholt. Ferner könnten alle Mitglieder einer Projektgruppe eine halbe Stunde vor der wöchentlichen Besprechung eine schriftliche Erinnerung erhalten. Auch Administrationsaufgaben wie Dateisystemtests oder Datenarchivierung lassen sich automatisch in der Nacht ausführen, da zu dieser Zeit die Systembelastung naturgemäß deutlich geringer ist.

9.2 Einmalige Ausführung von Kommandos

9.2.1 at und batch

Mit Hilfe des Kommandos `at` lassen sich beliebige Shell-Befehle zu einem späteren Zeitpunkt, also zeitversetzt, einmalig ausführen. Wenn Kommandos hingegen in regelmäßigen Intervallen wiederholt ausgeführt werden sollen, ist die Verwendung von `cron` (Abschnitt 9.3) vorzuziehen.

Die Idee hinter `at` ist, einen Zeitpunkt vorzugeben, zu dem dann ein Kommando oder eine Folge von Kommandos ausgeführt wird. Etwa so:

```
$ at 01:00
warning: commands will be executed using /bin/sh
at> tar cvzf /dev/st0 $HOME
at> echo " Backup fertig" | mail -s Backup $USER
at> 
Job 123 at 2006-11-08 01:00
```

Hiermit würden Sie um 1 Uhr nachts eine Sicherheitskopie Ihres Heimatverzeichnisses auf Band schreiben (Band einlegen nicht vergessen!) und sich anschließend per Mail eine Vollzugsmeldung schicken lassen.

Zeitangabe

Das Argument von `at` ist eine Zeitangabe für die Ausführung der Kommandos. Zeiten im Format » $\langle HH \rangle : \langle MM \rangle$ « bezeichnen den nächstmöglichen solchen Zeitpunkt: Wird das Kommando »`at 14:00`« um 8 Uhr früh angegeben, bezieht es sich auf denselben Tag; wird es um 16 Uhr angegeben, auf den Folgetag.



Sie können die Zeitpunkte durch Anhängen von `today` und `tomorrow` eindeutig machen: »`at 14:00 today`«, vor 14 Uhr gegeben, bezieht sich auf heute, »`at 14:00 tomorrow`« auf morgen.

Möglich sind auch angelsächsische Zeitangaben wie `01:00am` oder `02:20pm` sowie die symbolischen Namen `midnight` (0 bzw. 24 Uhr), `noon` (12 Uhr) und `teatime` (16 Uhr) (!); der ebenfalls erlaubte Zeitpunkt `now` ist vor allem in Verbindung mit relativen Zeitangaben (siehe unten) sinnvoll.

Datumsangabe

Neben Uhrzeiten versteht `at` auch Datumsangaben in der Form » $\langle MM \rangle \langle TT \rangle \langle JJ \rangle$ « und » $\langle MM \rangle / \langle TT \rangle / \langle JJ \rangle$ « (also nach amerikanischem Brauch, mit dem Monat vor dem Tag) sowie » $\langle TT \rangle . \langle MM \rangle . \langle JJ \rangle$ « (für uns Europäer). Daneben sind ausgeschriebene amerikanische Datumsangaben der Form » $\langle \text{Monatsname} \rangle \langle \text{Tag} \rangle$ « und » $\langle \text{Monatsname} \rangle \langle \text{Tag} \rangle \langle \text{Jahr} \rangle$ « erlaubt. Wenn Sie nur ein Datum angeben, werden die Kommandos zur aktuellen Zeit am betreffenden Tag ausgeführt; Sie können auch Zeit- und Datumsangabe kombinieren, müssen dann aber das Datum nach der Zeit anführen:

```
$ at 11:11 November 11
warning: commands will be executed using /bin/sh
at> echo 'Helau!'
```

```
at>  +   
Job 124 at 2003-11-11 11:11
```

Außer »expliziten« Zeit- und Datumsangaben können Sie auch »relative« Angaben machen, indem Sie eine Differenz zu einem gegebenen Zeitpunkt bestimmen: »relative« Angaben

```
$ at now + 5 minutes
```

führt die Kommandos in fünf Minuten aus, während

```
$ at noon + 2 days
```

sich auf 12 Uhr am übermorgigen Tag bezieht (jedenfalls solange das at-Kommando vor 12 Uhr angegeben wurde). at unterstützt die Zeiteinheiten `minutes`, `hours`, `days` und `weeks`.



Eine einzige Verschiebung um eine einzige Zeiteinheit muss reichen: Kombinationen wie

```
$ at noon + 2 hours 30 minutes
```

oder

```
$ at noon + 2 hours + 30 minutes
```

sind leider nicht erlaubt. Natürlich können Sie jede beliebige Verschiebung in Minuten ausdrücken ...

at liest die Kommandos von der Standardeingabe, also normalerweise der Tastatur; mit der Option »-f *(Datei)*« können Sie stattdessen eine Datei angeben. Kommandos



at versucht, die Kommandos in einer Umgebung auszuführen, die der zum Zeitpunkt des at-Aufrufs möglichst ähnelt. Das aktuelle Arbeitsverzeichnis, die `umask` und die aktuellen Umgebungsvariablen (außer `TERM`, `DISPLAY` und `_`) werden gesichert und vor der Kommandoausführung wieder aktiviert.

Eine etwaige Ausgabe der per at gestarteten Kommandos – Standardausgabe- und Standardfehlerausgabekanal – bekommen Sie per Mail geschickt. Ausgabe



Wenn Sie vor dem Aufruf von at mit `su` eine andere Identität angenommen haben, werden die Kommandos mit dieser Identität ausgeführt. Die Ausgabemails gehen aber trotzdem an Sie.

Während Sie mit at Kommandos zu einem bestimmten Zeitpunkt ausführen können, macht der (ansonsten analog funktionierende) Befehl `batch` es möglich, eine Folge von Kommandos zum »nächstmöglichen Zeitpunkt« auszuführen. Wann das passiert, hängt von der aktuellen Systemlast ab; hat das System gerade viel zu tun, müssen batch-Jobs warten. Ausführung zum »nächstmöglichen Zeitpunkt«



Eine at-artige Zeitangabe ist bei `batch` erlaubt, aber nicht vorgeschrieben. Wird sie angegeben, so werden die Kommandos »irgendwann nach« dem genannten Zeitpunkt ausgeführt, so als wären sie gerade dann per `batch` eingereicht worden.



`batch` ist nicht geeignet für Umgebungen, wo die Benutzer um Ressourcen wie Rechenzeit konkurrieren. Hierfür müssen andere Systeme eingesetzt werden.

Übungen



9.1 [!1] Nehmen wir an, jetzt ist es der 1. März, 15 Uhr. Wann werden die mit den folgenden Kommandos eingereichten Aufträge ausgeführt?

1. at 17:00
2. at 02:00pm
3. at teatime tomorrow
4. at now + 10 hours



9.2 [1] Verwenden Sie das Programm `logger`, um in 3 Minuten eine Nachricht ins Systemprotokoll zu schreiben.

9.2.2 at-Hilfsprogramme

Das System reiht die mit `at` registrierten Aufträge in eine Warteschlange ein. Diese Warteschlange anschauen können Sie mit `atq` inspizieren (Sie sehen aber nur Ihre eigenen Aufträge, es sei denn, Sie sind `root`):

```
$ atq
123    2003-11-08 01:00 a hugo
124    2003-11-11 11:11 a hugo
125    2003-11-08 21:05 a hugo
```



Das »a« in der Liste bezeichnet die »Auftragsklasse«, einen Buchstaben zwischen »a« und »z«. Sie können mit der Option `-q` für `at` eine Auftragsklasse bestimmen; Aufträge in Klassen mit »größeren« Buchstaben werden mit höheren nice-Werten ausgeführt. Standard ist »a« für `at`- und »b« für `batch`-Aufträge.



Ein gerade laufender Auftrag ist in der besonderen Auftragsklasse »=«.

Auftrag stornieren

Mit `atrm` können Sie einen Auftrag stornieren. Hierzu müssen Sie dessen Auftragsnummer angeben, die Sie bei der Einreichung genannt oder mit `atq` gezeigt bekommen haben. Wenn Sie nachschauen wollen, aus welchen Kommandos der Auftrag besteht, können Sie das mit »`at -c <Auftragsnummer>`«.

Daemon

Zuständig für die tatsächliche Ausführung der `at`-Aufträge ist ein Daemon namens `atd`. Dieser wird üblicherweise beim Systemstart geladen und wartet im Hintergrund auf Arbeit. Beim Start von `atd` sind einige Optionen möglich:

- b** (engl. *batch*, Stapel) Legt das minimale Intervall für den Start zweier `batch`-Aufträge fest. Voreinstellung ist hier 60 Sekunden.
- l** (engl. *load*, Last) Legt einen Grenzwert für die Systembelastung fest, oberhalb der `batch`-Aufträge nicht ausgeführt werden. Voreinstellung ist hier ein Wert von 0,8.
- d** (engl. *debug*, »Entwanzen«) aktiviert den Debug-Modus, d. h., alle Fehlermeldungen werden nicht per `syslogd` protokolliert, sondern auf die Standardfehlerausgabe geschrieben.

Der Daemon `atd` benötigt zu seiner Arbeit die folgenden Verzeichnisse:

- Im Verzeichnis `/var/spool/atjobs` werden die `at`-Aufträge abgelegt. Der Zugriffsmodus sollte 700 sein, der Eigentümer ist `at`.
- Das Verzeichnis `/var/spool/atpool` dient zur Zwischenspeicherung von Ausgaben. Auch hier sollten der Eigentümer `at` und der Zugriffsmodus 700 sein.

Übungen



9.3 [1] Registrieren Sie mit `at` einige Aufträge und lassen Sie sich eine Auftragsliste anzeigen. Entfernen Sie die Aufträge wieder.



9.4 [2] Wie würden Sie eine Liste von `at`-Aufträgen erzeugen, die nicht nach der Auftragsnummer, sondern nach dem vorgesehenen Ausführungszeitpunkt sortiert ist?

9.2.3 Zugangskontrolle

Die Dateien `/etc/at.allow` und `/etc/at.deny` bestimmen, wer mit `at` und `batch` Aufträge einreichen darf. Wenn die Datei `/etc/at.allow` existiert, sind nur die darin eingetragenen Benutzer berechtigt, Aufträge einzureichen. Gibt es die Datei `/etc/at.allow` nicht, aber die Datei `/etc/at.deny`, dann dürfen diejenigen Benutzer Aufträge einreichen, die *nicht* in der Datei stehen. Existiert weder die eine noch die andere, dann stehen `at` und `batch` nur `root` zur Verfügung.



Debian GNU/Linux liefert eine `/etc/at.deny`-Datei aus, in der die Namen diverser Systembenutzer stehen (etwa `alias`, `backup`, `guest` und `www-data`). Damit werden diese Benutzeridentitäten von der Verwendung von `at` ausgeschlossen.



Die Voreinstellung von Ubuntu entspricht auch hier der von Debian GNU/Linux.



Red Hat liefert eine leere `/etc/at.deny`-Datei aus; damit darf jeder Benutzer Aufträge einreichen.



Die OpenSUSE-Voreinstellung gleicht (interessanterweise) der von Debian GNU/Linux und Ubuntu – diverse Systembenutzer dürfen `at` nicht verwenden. (Den explizit ausgeschlossenen Benutzer `www-data` zum Beispiel gibt es bei OpenSUSE überhaupt nicht; Apache läuft mit der Identität von `wwwrun`.)

Übungen



9.5 [1] Wer darf auf Ihrem System `at` und `batch` benutzen?

9.3 Wiederholte Ausführung von Kommandos

9.3.1 Aufgabenlisten für Benutzer

Im Unterschied zu den `at`-Kommandos dient der Daemon `cron` zur automatischen Ausführung von sich regelmäßig wiederholenden Aufgaben. Gestartet werden sollte `cron` – wie auch der `atd` – beim Systemstart mit einem Init-Skript; Sie müssen sich darum normalerweise nicht kümmern, da `cron` und `atd` so wichtige Bestandteile eines Linux-Systems sind, dass keine der wichtigen Distributionen auf sie verzichtet.

Jeder Anwender hat seine eigene Aufgabenliste (vulgo `crontab`), die im Verzeichnis `/var/spool/cron/crontabs` (bei Debian GNU/Linux und Ubuntu; bei SUSE: `/var/spool/cron/tabs`, bei Red Hat: `/var/spool/cron`) unter dem jeweiligen Benutzernamen abgelegt ist. Die dort beschriebenen Kommandos werden mit den Rechten des betreffenden Anwenders ausgeführt.



Auf Ihre Aufgabenliste im `cron`-Verzeichnis haben Sie keinen direkten Zugriff, sondern müssen das Programm `crontab` bemühen (siehe unten). Beachten Sie hierzu auch Übung 9.6.

Syntax crontab-Dateien sind zeilenweise aufgebaut; jede Zeile beschreibt einen (wiederkehrenden) Zeitpunkt und ein Kommando, das zu dem betreffenden Termin ausgeführt werden soll. Leerzeilen und Kommentarzeilen (mit einem »#« am Anfang) werden ignoriert. Die übrigen Zeilen bestehen aus fünf Zeitfeldern und dem auszuführenden Kommando; die Zeitfelder beschreiben respektive die Minute (0–59), die Stunde (0–23), der Tag im Monat (1–31), der Monat (1–12 oder der englische Name) und den Wochentag (0–7, 0 und 7 stehen für Sonntag, oder der englische Name), zu denen das Kommando ausgeführt werden soll. Alternativ ist jeweils ein Sternchen (»*«) erlaubt, das für »egal« steht. Zum Beispiel bedeutet

```
58 19 * * * echo "Gleich kommt die Tagesschau"
```

dass das Kommando täglich um 19.58 Uhr ausgeführt wird (Tag, Monat und Wochentag sind egal).



Das Kommando wird ausgeführt, wenn Stunde, Minute und Monat genau stimmen und *mindestens eine* der beiden Tagesangaben – Tag im Monat oder Wochentag – zutreffen. Die Spezifikation

```
1 0 13 * 5 echo "Kurz nach Mitternacht"
```

gibt also an, dass die Meldung an jedem Monats-Dreizehnten *und* an jedem Freitag ausgegeben wird, nicht nur am Freitag, dem 13.



Die letzte Zeile in einer crontab-Datei *muss* mit einem Zeilentrenner aufhören, sonst wird sie ignoriert.

cron akzeptiert in den Zeitfeldern nicht nur einzelne Zahlen, sondern erlaubt auch kommaseparierte Listen. Die Angabe »0,30« im Minutenfeld würde also dazu führen, dass das Kommando zu jeder »vollen halben« Stunde ausgeführt wird. Außerdem sind Bereiche erlaubt: »8-11« ist äquivalent zu »8,9,10,11«, »8-10,14-16« entspricht »8,9,10,14,15,16«. Ebenfalls gestattet ist die Angabe einer »Schrittweite« in Bereichen. Die Spezifikation »0-59/10« im Minutenfeld ist gleichbedeutend mit »0,10,20,30,40,50«. Wird – wie hier – der komplette Wertebereich abgedeckt, so könnten Sie auch »*/10« schreiben.

Die bei Monats- und Wochentagsangaben erlaubten Namen bestehen jeweils aus den ersten drei Buchstaben des englischen Monats- oder Wochentagsnamen (also zum Beispiel may, oct, sun oder wed). Bereiche und Listen von Namen sind nicht erlaubt.

Der Rest der Zeile bestimmt das auszuführende Kommando, das von cron an /bin/sh (bzw. die in der Variablen SHELL benannte Shell, siehe unten) übergeben wird.



Prozentzeichen (%) im Kommando müssen mit einem Rückstrich versteckt werden (also »\%«), sonst werden sie in Zeilentrenner umgewandelt. Dann gilt das Kommando als am ersten Prozentzeichen beendet; die folgenden »Zeilen« werden dem Kommando auf der Standardeingabe verfüttert.



Übrigens: Wenn Sie als Systemadministrator möchten, dass bestimmte Kommandos nicht, wie sonst bei cron üblich, bei der Ausführung per syslogd protokolliert werden, können Sie dies durch die Angabe eines »-« als erstes Zeichen der Zeile unterdrücken.

Außer Kommandos mit Wiederholungsangaben können die crontab-Zeilen auch Zuweisungen an Umgebungsvariablen enthalten. Diese haben die Form »<Variablenname>=<Wert>« (wobei im Gegensatz zur Shell vor und nach dem »=« auch Leerplatz stehen darf). Enthält der <Wert> Leerzeichen, sollte er mit Anführungszeichen eingefasst werden. Die folgenden Variablen werden automatisch voreingestellt:

SHELL Mit dieser Shell werden die eingegebenen Befehle ausgeführt. Voreingestellt ist dabei `/bin/sh`, aber auch die Angabe anderer Shells ist erlaubt.

LOGNAME Der Benutzername wird aus `/etc/passwd` übernommen und kann nicht geändert werden.

HOME Das Heimatverzeichnis wird ebenfalls aus `/etc/passwd` übernommen. Hier ist jedoch eine Änderung des Variablenwerts zulässig.

MAILTO An diese Adresse schickt `cron` Nachrichten mit der Ausgabe des aufgerufenen Kommandos (sonst gehen sie an den Eigentümer der `crontab`-Datei). Soll `cron` überhaupt keine Nachrichten verschicken, muss die Variable leer gesetzt sein, d. h. `MAILTO=""`.

9.3.2 Systemweite Aufgabenlisten

Neben den benutzerbezogenen Dateien existiert noch eine systemweite Aufgabenliste. Diese findet sich in `/etc/crontab` und gehört dem Systemadministrator `root`, `/etc/crontab` der sie als einziger ändern darf. Die Syntax von `/etc/crontab` ist geringfügig anders als die der benutzereigenen `crontab`-Dateien; zwischen den Zeitangaben und dem auszuführenden Kommando steht hier noch der Name des Benutzers, mit dessen Rechten das Kommando ausgeführt werden soll.

 Diverse Linux-Distributionen haben ein `/etc/cron.d`-Verzeichnis; in diesem Verzeichnis können Dateien stehen, die als »Erweiterungen« von `/etc/crontab` interpretiert werden. Per Paketmanagement installierte Softwarepakete können so leichter den `cron`-Dienst benutzen, als wenn sie Zeilen zur `/etc/crontab` hinzufügen müssten.

 Populär sind auch Verzeichnisse `/etc/cron.hourly`, `/etc/cron.daily` und so weiter. In diesen Verzeichnissen können Softwarepakete (oder der Systemadministrator) Dateien ablegen, deren Inhalt dann stündlich, täglich, ... ausgeführt wird. Diese Dateien sind keine `crontab`-Dateien, sondern »normale« Shellskripte.

`cron` liest die Aufgabenlisten – aus benutzereigenen `crontab`-Dateien, der systemweiten `/etc/crontab` und den Dateien in `/etc/cron.d`, sofern vorhanden – nur einmal beim Start ein und behält sie danach im Speicher. Das Programm schaut allerdings jede Minute nach, ob die `crontab`-Dateien geändert wurden. Zu diesem Zweck wird einfach die `mtime`, der Zeitpunkt der letzten Änderung, herangezogen. Wenn `cron` hier eine Veränderung bemerkt, wird die Aufgabenliste automatisch neu aufgebaut. In diesem Fall ist also kein expliziter Neustart des Daemons erforderlich. `crontab`-Änderungen und `cron`

Übungen

 **9.6** [2] Wieso können Sie als Benutzer nicht auf Ihre Aufgabenliste in `/var/spool/cron/crontabs` (oder dem Äquivalent für Ihre Distribution) zugreifen? Wie wird arrangiert, dass `crontab` das kann?

 **9.7** [2] Wie können Sie dafür sorgen, dass eine Aufgabe nur am Freitag, dem 13., ausgeführt wird?

 **9.8** [3] Wie stellt das System sicher, dass die Aufgaben in `/etc/cron.hourly`, `/etc/cron.daily`, ... tatsächlich einmal in der Stunde, einmal am Tag usw. ausgeführt werden?

9.3.3 Zugangskontrolle

Welche Anwender überhaupt mit cron arbeiten dürfen, ist ähnlich wie bei at in zwei Dateien festgelegt. In `/etc/cron.allow` (gelegentlich `/var/spool/cron/allow`) sind die Benutzer aufgeführt, die cron verwenden dürfen. Wenn die Datei nicht existiert, aber es dafür die Datei `/etc/cron.deny` (manchmal `/var/spool/cron/deny`) gibt, sind in letzterer diejenigen Benutzer aufgelistet, die *nicht* in den Genuss der automatisierten Befehlsausführung kommen dürfen. Sind beide Dateien nicht vorhanden, hängt es von der jeweiligen Konfiguration ab, ob nur root die Dienste von cron beanspruchen darf oder sozusagen »gleiches Recht für alle« herrscht und cron jedem Benutzer zur Verfügung steht.

9.3.4 Das Kommando crontab

Die einzelnen Benutzer können ihre crontab-Datei nicht von Hand ändern, da das System sie vor ihnen versteckt. Lediglich die systemweite Aufgabenliste `/etc/crontab` ist ein Fall für den Lieblingsseditor von root.

Aufgabenlisten verwalten

Statt einen Editor direkt aufzurufen, sollten alle Benutzer das Kommando `crontab` verwenden. Hiermit können Aufgabenlisten erstellt, eingesehen, verändert und auch wieder entfernt werden. Mit

```
$ crontab -e
```

können Sie Ihre crontab-Datei mit dem Editor bearbeiten, dessen Name in den Umgebungsvariablen `VISUAL` bzw. `EDITOR` festgelegt ist – ersatzweise dem Editor `vi`. Nach dem Verlassen des Editors wird die modifizierte crontab-Datei automatisch installiert. Statt der Option `-e` können Sie auch den Namen einer Datei angeben, deren Inhalt dann als Aufgabenliste installiert wird. Der Dateiname »-« steht hierbei stellvertretend für die Standardeingabe.

Mit der Option `-l` gibt `crontab` Ihre crontab-Datei auf der Standardausgabe aus; mit der Option `-r` wird eine installierte Aufgabenliste ersatzlos gelöscht.



Mit der Option »-u *(Benutzername)*« können Sie sich auf einen anderen Benutzer beziehen (wofür Sie in der Regel root sein müssen). Dies ist vor allem wichtig, wenn Sie `su` benutzen; in diesem Fall sollten Sie immer mit der `-u` Option operieren, um sicherzustellen, dass Sie die richtige crontab-Datei erwischen.

Übungen



9.9 [1] Registrieren Sie mit dem Programm `crontab` eine Aufgabe, die jede Minute das aktuelle Datum an die Datei `/tmp/date.log` anhängt. Wie können Sie bequem erreichen, dass dasselbe alle zwei Minuten passiert?



9.10 [1] Verwenden Sie `crontab`, um den Inhalt Ihrer Aufgabenliste auf der Standardausgabe auszugeben und anschließend wieder zu löschen.



9.11 [2] (Für Administratoren.) Sorgen Sie dafür, dass der Benutzer `hugo` den cron-Dienst *nicht* verwenden darf. Vergewissern Sie sich, dass Ihre Maßnahme funktioniert.

9.3.5 Anacron

Mit `cron` können Sie Kommandos wiederholt zu vorgegebenen Zeitpunkten ausführen. Offensichtlich funktioniert das aber nur, wenn der Computer zum betreffenden Zeitpunkt eingeschaltet ist – es hat keinen großen Zweck, auf einem Arbeitsplatzrechner einen cron-Job für 2 Uhr morgens zu konfigurieren, wenn der Rechner außerhalb der üblichen Bürozeiten aus Stromspargründen ausgeschaltet ist. Auch mobile Rechner sind oft zu ungewöhnlichen Zeiten ein- oder ausgeschaltet, so dass es schwierig ist, die periodischen automatischen Aufräumungsarbeiten einzuplanen, die für ein Linux-System nötig sind.

Das Programm `anacron` (von Itai Tzur, aktuell gewartet von Pascal Hakim) kann ähnlich wie `cron` Jobs ausführen, die im täglichen, wöchentlichen oder monatlichen Rhythmus laufen sollen. (Tatsächlich gehen beliebige Abstände von n Tagen.) Dabei ist es nur wichtig, dass der Rechner am betreffenden Tag lange genug eingeschaltet ist, dass die Jobs ausgeführt werden können – es ist egal, wann am Tag. Allerdings wird `anacron` maximal einmal am Tag aktiv; wenn Sie eine höhere Frequenz brauchen (Stunden oder Minuten), führt `anacron` kein Weg vorbei.



Im Gegensatz zu `cron` ist `anacron` relativ primitiv, was die Verwaltung von Jobs angeht. Mit `cron` kann potentiell jeder Benutzer Jobs anlegen; bei `anacron` ist das das Privileg des Systemverwalters `root`.

Die Jobs für `anacron` werden in der Datei `/etc/anacrontab` festgelegt. Neben den üblichen Kommentar- und Leerzeilen (die ignoriert werden) enthält sie Zuweisungen an Umgebungsvariablen der Form

```
SHELL=/bin/sh
```

und Job-Beschreibungen der Form

```
7 10 weekly run-parts /etc/cron.weekly
```

Dabei steht die erste Zahl (hier 7) für den Zeitabstand (in Tagen), in dem einzelne Aufrufe des Jobs stattfinden sollen. Die zweite Zahl (10) gibt an, wieviele Minuten nach dem Start von `anacron` der Job gestartet werden soll. Als Nächstes kommen ein Name für den Job (hier `weekly`) und schließlich das auszuführende Kommando. Überlange Zeilen können mit einem `»\«` am Zeilenende umbrochen werden.



Der Jobname darf alle Zeichen enthalten außer Freiplatz und dem Schrägstrich. Er wird verwendet, um den Job in Protokollnachrichten zu identifizieren, und `anacron` benutzt ihn auch als Namen für die Datei, mit der er den Zeitstempel der letzten Ausführung protokolliert. (Diese Dateien stehen normalerweise in `/var/spool/anacron`.)

Wenn `anacron` gestartet wird, liest es `/etc/anacrontab` und prüft für jeden Job, ob er während der letzten t Tage ausgeführt wurde, wobei t der Zeitabstand aus der Jobdefinition ist. Wenn nein, dann wartet `anacron` die in der Jobdefinition angegebene Anzahl von Minuten ab und startet das Shell-Kommando.



Sie können auf der Kommandozeile von `anacron` einen Jobnamen angeben, um (gegebenenfalls) nur diesen Job auszuführen. Alternativ sind auf der Kommandozeile auch Shell-Suchmuster erlaubt, damit Sie Gruppen von (geschickt vergebenen) Jobnamen mit einem `anacron`-Aufruf ausführen können. Beim Aufruf von `anacron` gar keinen Jobnamen anzugeben ist dasselbe wie der Jobname `»*«`.



Den Zeitabstand zwischen Ausführungen eines Jobs können Sie auch symbolisch angeben; gültige Werte sind `@daily`, `@weekly`, `@monthly`, `@yearly` und `@annually` (die beiden letzten sind äquivalent).



In der Definition einer Umgebungsvariablen werden Leerzeichen links vom `»=«` ignoriert. Rechts vom `»=«` sind sie Teil des Wertes der Variablen. Definitionen gelten bis zum Dateiende oder bis zu einer neuen Definition derselben Variablen.



Einige »Umgebungsvariable« haben eine Sonderbedeutung für `anacron`. Mit `RANDOM_DELAY` können Sie eine zusätzliche zufällige Verzögerung für die Jobstarts festlegen: Wenn Sie die Variable auf eine Zahl t setzen, dann wird eine zufällige Anzahl von Minuten zwischen 0 und t zur in der Job-Beschreibung angegebenen Verzögerung addiert. Mit `START_HOURS_RANGE` können Sie einen Bereich von Stunden (auf der Uhr) angeben, während dem die Jobs gestartet werden sollen. Etwas wie

```
START_HOURS_RANGE=10-12
```

erlaubt Job-Starts nur zwischen 10 und 12 Uhr. Wie cron schickt anacron die Ausgabe von Jobs an die Adresse, die mit der Variablen MAILTO angegeben wurde, ersatzweise den Benutzer, der anacron ausführt (in der Regel root).

Normalerweise führt anacron die Jobs unabhängig voneinander und ohne Rücksicht auf Überlappungen aus. Mit der Option `-s` werden die Jobs »seriell« ausgeführt, das heißt, anacron startet einen neuen Job erst, wenn der vorige fertig ist.

Im Gegensatz zu cron ist anacron kein Hintergrunddienst, sondern wird beim Systemstart angestoßen, um allfällige liegengebliebene Jobs auszuführen (die Verzögerung in Minuten dient dazu, die Jobs zu verschieben, bis das System vernünftig läuft, um den Systemstart nicht noch weiter zu verlangsamen). Später können Sie anacron einmal am Tag mit cron ausführen, um dafür zu sorgen, das es auch funktioniert, wenn das System einmal unvorhergesehen lange läuft.



Sie können durchaus cron und anacron auf demselben System installiert haben. Während anacron normalerweise die eigentlich für cron gedachten Jobs in `/etc/cron.daily`, `/etc/cron.weekly` und `/etc/cron.monthly` ausführt, wird dafür gesorgt, dass anacron nichts macht, wenn cron aktiv ist. (Siehe hierzu Übung 9.13.)

Übungen



9.12 [!2] Überzeugen Sie sich, dass anacron so funktioniert wie behauptet. (*Tipp*: Wenn Sie nicht tagelang warten wollen, dann manipulieren Sie dazu kreativ die Zeitstempel-Dateien in `/var/spool/anacron`.)



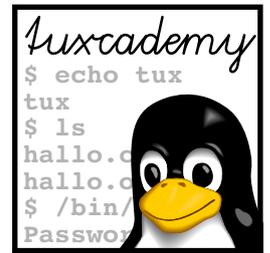
9.13 [2] Wie vermeidet man, dass auf einem lang laufenden System, das sowohl cron und anacron installiert hat, anacron dem regulären cron in die Quere kommt? (*Tipp*: Untersuchen Sie den Inhalt von `/etc/cron.daily` & Co.)

Kommandos in diesem Kapitel

anacron	Führt periodische Jobs aus, wenn der Computer nicht immer läuft	anacron(8)	148
at	Registriert Kommandos zur zeitversetzten Ausführung	at(1)	142
atd	Daemon für die zeitversetzte Ausführung von Kommandos über at	atd(8)	144
atq	Programm zur Abfrage der Warteschlangen für zeitversetzte Kommandoausführung	atq(1)	144
atrm	Storniert zeitversetzt auszuführende Kommandos	atrm(1)	144
crontab	Programm zur Verwaltung von regelmäßig auszuführenden Kommandos	crontab(1)	148

Zusammenfassung

- Mit `at` kann man Kommandos registrieren, die zu einem festen späteren Zeitpunkt ausgeführt werden.
- Das Kommando `batch` erlaubt die Ausführung von Kommandos, wenn das System dafür Kapazität frei hat.
- `atq` und `atrm` dienen zur Verwaltung der Auftragswarteschlangen. Der Daemon `atd` kümmert sich um die tatsächliche Ausführung der Aufträge.
- Der Zugang zu `at` und `batch` wird über die Dateien `/etc/at.allow` und `/etc/at.deny` gesteuert.
- Der `cron`-Daemon dient zur periodischen Wiederholung von Kommandos.
- Benutzer können eigene Aufgabenlisten (`crontabs`) haben.
- Eine systemweite Aufgabenliste existiert in `/etc/crontab` und – bei vielen Distributionen – im Verzeichnis `/etc/cron.d`.
- Der Zugriff auf `cron` wird über die Dateien `/etc/cron.allow` und `/etc/cron.deny` ähnlich geregelt wie der Zugriff auf `at`.
- Das Kommando `crontab` dient zur Verwaltung von `crontab`-Dateien.



10

Lokalisierung und Internationalisierung

Inhalt

10.1 Überblick.	154
10.2 Zeichencodierungen.	154
10.3 Spracheneinstellung unter Linux	158
10.4 Lokalisierungs-Einstellungen	160
10.5 Zeitzone	163

Lernziele

- Die gängigen Zeichencodierungen kennen
- Textdateien von einer Zeichencodierung in eine andere konvertieren können
- Die Umgebungsvariablen zur Spracheneinstellung kennen
- Die Linux-Infrastruktur zur Lokalisierung kennen
- Verstehen, wie Linux mit Zeitzone umgeht

Vorkenntnisse

- Umgang mit Linux-Kommandos
- Umgang mit einem Texteditor

10.1 Überblick

»Internationalisierung« (engl. *internationalisation* oder kurz *i18n*, weil zwischen dem ersten und dem letzten Buchstaben des Worts 18 andere Buchstaben stehen) ist die Ausgestaltung eines Softwaresystems, so dass eine spätere Lokalisierung möglich ist. »Lokalisierung« (engl. *localisation* oder *l10n*) ist die Anpassung eines internationalisierten Systems an die Gepflogenheiten eines bestimmten Kulturkreises. Dazu gehört in erster Linie die Sprache für die Benutzungsoberfläche und allfällige Meldungen des Systems sowie die zu verarbeitenden Daten (die gegebenenfalls besondere Schriften und Eingabemethoden benötigt), aber auch Aspekte wie standardisierte Schreibweisen für Datums- und Zeitangaben, Geldbeträge, die Reihenfolge der Zeichen im Alphabet und einiges mehr. Bei grafischen Programmen können sogar bestimmte Farben der Lokalisierung unterliegen; in der westlichen Welt suggeriert zum Beispiel die Farbe Rot Gefahr, aber das ist nicht überall sonst so.

Für Linux als Betriebssystem-Kernel ist Internationalisierung nicht wirklich ein bedeutsames Thema, da der Kernel sich mit den meisten Problemfeldern, die einer Internationalisierung bedürfen, überhaupt nicht beschäftigt. (Man ist sich relativ einig, dass der Kernel zum Beispiel nicht mit Systemmeldungen in allen möglichen Sprachen überfrachtet werden sollte; von jemandem, der diese Meldungen überhaupt zu sehen bekommt, wird erwartet, dass er genug Englisch kann, um sie zu verstehen.) Linux-Distributionen dagegen enthalten riesige Mengen von Anwendersoftware, die von einer Lokalisierung profitieren dürfte. Entsprechend stehen die gängigen großen Distributionen auch in einer Vielzahl von lokalisierten Versionen zur Verfügung. Daneben gibt es diverse spezielle Linux-Distributionen, die sich auf einzelne Kulturkreise beschränken und diese besonders gut zu unterstützen versuchen.



Während große kommerzielle Softwarehersteller die Lokalisierung ihrer Software entweder von örtlichen Niederlassungen oder bezahlten Partnerfirmen vornehmen lassen, beruht die Lokalisierung von Open-Source-Software wie den gängigen Linux-Anwendungen in weiten Teilen auf der Mithilfe von Freiwilligen. Der Vorteil davon ist, dass sich in der Regel auch für sehr ausgefallene Sprachen Freiwillige finden lassen, die eine Software übersetzen und anpassen können – Sprachen, die zu unterstützen sich für einen kommerziellen Hersteller nie rechnen würde. Auf der anderen Seite stellt der Einsatz von Freiwilligen besondere Herausforderungen an die Qualitätskontrolle. Legendär ist die Anekdote, dass ein KDE-Entwickler aus der arabischen Welt bei der Lokalisierung alle Verweise auf den Staat Israel durch »Besetztes Palästina« ersetzt hat; ein Schritt, der bei der KDE-Gemeinde insgesamt nicht wirklich gut ankam ...

10.2 Zeichencodierungen

Vorbedingung für die Internationalisierung und Lokalisierung von Programmen in fremden Sprachen (unter dem Strich »alles außer Englisch«) ist, dass das System den Zeichenvorrat der betreffenden Sprache anzeigen können muss. Der traditionelle Zeichencode für Computer ist der ASCII oder *American Standard Code for Information Interchange*, der, wie der Name sagt, für amerikanisches Englisch gedacht ist – in der Frühzeit elektronischer Rechenanlagen reichte das aus, aber alsbald wurde es notwendig, auch auf die Belange von »Fremdsprachen« Rücksicht zu nehmen.

Zeichenvorrat Steuerzeichen **ASCII** Der ASCII kann 128 verschiedene Zeichen darstellen, von denen 33 (die Positionen 0–31 und die Position 127) für Steuerzeichen wie »Zeilenvorschub«, »Tabulator« oder »Glocke« reserviert sind. Die anderen 95 Zeichen umfassen

Tabelle 10.1: Die wichtigsten Teile von ISO/IEC 8859

Teil	Trivialname	Umfang
ISO 8859-1	Latin-1 (Westeuropa)	Die meisten westeuropäischen Sprachen: Dänisch, Deutsch, Englisch, Färöisch, Finnisch*, Französisch*, Isländisch, Irisch, Italienisch, Niederländisch*, Norwegisch, Portugiesisch, Rätoromanisch, schottisches Gälisch, Spanisch und Schwedisch. Daneben einige andere Sprachen, unter anderem Afrikaans, Albanisch, Indonesisch und Swahili. (* = teilweise)
ISO 8859-2	Latin-2 (Mitteleuropa)	Mittel- und osteuropäische Sprachen, die das lateinische Alphabet verwenden, etwa Bosnisch, Kroatisch, Polnisch, Serbisch, Slowakisch, Slowenisch, Tschechisch und Ungarisch.
ISO 8859-3	Latin-3 (Südeuropa)	Maltesisch, Türkisch und Esperanto.
ISO 8859-4	Latin-4 (Nordeuropa)	Estnisch, Lettisch, Litauisch, Grönländisch und Sami.
ISO 8859-5	Latin/Cyrillic	Die meisten slawischen Sprachen mit kyrillischem Alphabet, etwa Bulgarisch, Mazedonisch, Russisch, Serbisch, Ukrainisch (teilweise) und Weißrussisch.
ISO 8859-6	Latin/Arabic	Enthält die gängigsten arabischen Zeichen, ist aber nicht für Sprachen mit arabischer Schrift außer Arabisch geeignet. Arabisch wird von rechts nach links geschrieben!
ISO 8859-7	Latin/Greek	Modernes Griechisch und Altgriechisch (ohne Akzente)
ISO 8859-8	Latin/Hebrew	Modernes Hebräisch
ISO 8859-9	Latin-5 (Türkisch)	Entspricht im wesentlichen ISO 8859-1, mit türkischen Zeichen statt den isländischen. Auch für Kurdisch.
ISO 8859-10	Latin-6 (Nordisch)	Eine andere Anordnung von Latin-4.
ISO 8859-11	Latin/Thai	Für Thai.
ISO 8859-12	Latin/Devanagari	Wurde nie fertiggestellt.
ISO 8859-13	Latin-7 (Baltikum)	Enthält noch mehr Zeichen für baltische Sprachen, die in Latin-4 und Latin-6 nicht vorkommen.
ISO 8859-14	Latin-8 (Keltisch)	Keltische Sprachen wie Gälisch und Bretonisch.
ISO 8859-15	Latin-9	Im wesentlichen Latin-1, aber mit dem Euro-Zeichen und den für Estnisch, Finnisch und Französisch fehlenden Zeichen anstelle einiger fast nie gebrauchter anderer.
ISO 8859-16	Latin-10	Für Albanisch, Italienisch, Kroatisch, Polnisch, Rumänisch, Slowenisch und Ungarisch, aber auch Deutsch, Finnisch, Französisch und irisches Gälisch. Mit Euro-Zeichen. Ohne praktische Bedeutung.

Groß- und Kleinbuchstaben, Ziffern und eine Auswahl von gängigen Sonderzeichen, vor allem Interpunktion.



In Deutschland war die Zeichencodierung DIN 66003 gebräuchlich, die im wesentlichen dem ASCII entsprach, bis darauf, dass die eckigen und geschweiften Klammern, der vertikale Balken und der Rückstrich durch Umlaute, die Tilde durch das scharfe S und der Klammeraffe (»@«) durch das Paragraphenzeichen ersetzt wurde. Das war für Texte annehmbar, aber nicht notwendigerweise für C-Programme. (Der Autor dieser Zeilen erinnert sich noch an den ersten Drucker, mit dem er zu tun hatte, einen Centronics 737-2, der mit einem »Mäuseklavier«, also einer Bank von Mikroschaltern, von ASCII auf DIN 66003 und zurück umgeschaltet werden musste – ein mühseliges Unterfangen.)

DIN 66003

ISO/IEC 8859 Später stieg man unter dem zunehmenden Druck internationaler Computeranwender vom ASCII mit seinen 128 Zeichen auf erweiterte Zeichensätze um, die alle 256 möglichen Werte eines Bytes zur Zeichencodierung nutzten. Hervorzuheben ist hier der Standard ISO/IEC 8859, der Zeichencodierungen für viele verschiedene Sprachen vorsieht. Tatsächlich besteht ISO/IEC 8859 aus einer

256 Zeichen

Informationsaustausch	<p>Anzahl von nummerierten, separat veröffentlichten Teilen, die auch als Standards angesehen werden können. Tabelle 10.1 gibt einen Überblick.</p> <p>Der Schwerpunkt der ISO/IEC-8859-Standards liegt auf Informationsaustausch, nicht eleganter Typografie, so dass diverse Zeichen, die für schöne Drucksachen erforderlich sind – etwa Ligaturen oder »typografische« Anführungszeichen – in den Zeichencodierungen fehlen. Bei der Zusammenstellung der Codetabellen konzentrierte man sich auf Zeichen, die bereits auf Computertastaturen vorhanden waren, und ging auch gewisse Kompromisse ein. Beispielsweise wurden die französischen »Œ«- und »œ«-Ligaturen nicht in Latin-1 aufgenommen, da man sie auch als »OE« und »oe« schreiben kann und der Platz in der Tabelle anderweitig benötigt wurde.</p>
Grenzen	<p>Fernöstliche Sprachen wie Chinesisch oder Japanisch werden von ISO/IEC 8859 nicht adressiert, da ihr Zeichenvorrat viel größer ist als die 256 Zeichen, die in eine einzelne ISO/IEC-8859-Codetabelle passen. Auch einige andere Alphabete der Welt sind nicht Gegenstand eines ISO/IEC-8859-Standards.</p>
Zwei für alle	<p>Unicode und ISO 10646 Unicode und ISO 10646 (das »Universal Character Set«) sind parallele Bemühungen, <i>alle</i> Alphabete der Welt in einer gemeinsamen Zeichencodierung zusammenzufassen. Die beiden Standards wurden zuerst getrennt entwickelt, aber dann – nachdem die Softwarefirmen der Welt heftigst gegen die Komplexität von ISO 10646 gemeutert hatten – zusammengelegt. Heutzutage standardisieren Unicode und ISO 10646 dieselben Zeichen mit denselben Zeichencodes; der Unterschied zwischen den beiden ist, dass ISO 10646 eine reine Zeichentabelle ist (im Prinzip ein erweitertes ISO 8859), während Unicode zusätzliche Regeln enthält, etwa für Sortierung, Normalisierung und bidirektionales Schreiben (für Sprachen wie Arabisch und Hebräisch). In diesem Sinne ist ISO 10646 eine »Untermenge« von Unicode; bei Unicode haben Zeichen noch diverse Zusatzeigenschaften, die zum Beispiel beschreiben, wie ein Zeichen sich mit anderen kombinieren läßt (was zum Beispiel für Arabisch wichtig ist, wo das Aussehen eines Zeichens davon abhängt, ob es am Anfang, in der Mitte oder am Ende eines Worts steht).</p>
ISO 10646 ohne Unicode	<p> Das Unix/Linux-Programm xterm ist ein Beispiel für ein Programm, das ISO 10646 unterstützt, aber nicht Unicode. Es kann alle ISO-10646-Zeichen darstellen, die sich direkt aus der Zeichencode-Tabelle ergeben und nur in eine Richtung geschrieben werden, und kann einige aus mehreren Zeichen (etwa einem »Basiszeichen« und Akzenten) zusammengesetzte Zeichen anzeigen, aber kein Hebräisch, Arabisch oder Devanagari. Unicode korrekt zu implementieren ist absolut kein Kinderspiel.</p>
Zeichenvorrat	<p>ISO 10646 enthält nicht nur Buchstaben und Ziffern, sondern auch Ideogramme (etwa chinesische und japanische Zeichen), mathematische Zeichen und vieles andere mehr. Jedes solche Zeichen wird durch einen eindeutigen Namen und eine Ganzzahl, seinen Codepunkt, identifiziert. Insgesamt stehen über 1,1 Millionen Codepunkte zur Verfügung, von denen allerdings nur die ersten 65.536 weithin verwendet werden. Diese nennt man auch die <i>basic multilingual plane</i> oder BMP.</p>
Codepunkte	<p>Unicode- bzw. ISO-10646-Codepunkte notiert man in der Form U+0040, wobei die vier Ziffern eine hexadezimale Zahl darstellen.</p>
Codierungsformen	<p>UCS-2 und UTF-8 Unicode und ISO 10646 spezifizieren Codepunkte, also Nummern für die Zeichen im Code, aber geben zunächst nicht an, wie man mit diesen Codepunkten tatsächlich umgehen kann. Dafür sind sogenannte Codierungsformen definiert, die erklären, wie man die Codepunkte in einem Computer darstellt.</p> <p>UCS-2 Die einfachste Codierungsform ist UCS-2, bei der für jedes Zeichen ein einzelner »Codewert« zwischen 0 und 65.535 verwendet wird, der in zwei Bytes dargestellt wird. Das heißt, UCS-2 ist beschränkt auf Zeichen in der BMP. UCS-2 hat auch das Problem, dass es für Daten aus der westlichen Welt, die in der Regel mit einer höchstens 8 Bit breiten Codierung wie ASCII oder ISO Latin-1 dargestellt</p>

werden können, den doppelten Speicherplatz impliziert, da statt einem plötzlich zwei Byte pro Zeichen verbraucht werden.



Statt UCS-2 verwenden Systeme wie Windows inzwischen eine Codierungsform namens UTF-16, die es gestattet, auch Codepunkte außerhalb der BMP darzustellen. Das Speicherplatzproblem existiert trotzdem. UTF-16

UTF-8 ist in der Lage, jedes Zeichen in ISO 10646 darzustellen, aber ist dennoch rückwärtskompatibel mit ASCII und ISO-8859-1. Es codiert die Codepunkte U+0000 bis U+10FFFF (also das 32fache von UCS-2) in ein bis vier Bytes, wobei die ASCII-Zeichen mit einem Byte auskommen. UTF-8 erfüllt die folgenden Entwurfsanforderungen: UTF-8

ASCII-Zeichen stehen für sich selbst Dadurch wird UTF-8 kompatibel zu allen Programmen, die mit Bytestrings umgehen können, also beliebigen Folgen von 8-Bit-Bytes, aber einigen ASCII-Zeichen eine Sonderbedeutung geben. So ist es leicht, existierende Systeme auf UTF-8 umzustellen.

Kein erstes Byte taucht in der Mitte eines Zeichens auf Wenn ein oder mehrere vollständige Bytes verlorengehen oder entstellt werden, dann ist es trotzdem möglich, den Anfang des nächsten Zeichens zu finden.

Das erste Byte eines Zeichens bestimmt die Byteanzahl Auf diese Weise wird sichergestellt, dass eine Bytefolge, die ein bestimmtes Zeichen darstellt, nicht Teil einer längeren Bytefolge sein kann, die für ein anderes Zeichen steht. Damit ist es effizient möglich, in Zeichenketten auf Byteebene nach Teilzeichenketten zu suchen.

Die Bytes FE und FF kommen nicht vor Diese Bytes stehen am Anfang eines UCS-2-Textes und dienen dazu, herauszufinden, in welcher Reihenfolge ein Rechner die beiden Bytes eines Worts darstellt. Da beide keine gültigen Bytes in einem UTF-8-Datenstrom sind, besteht so keine Verwechslungsgefahr zwischen UCS-2- und UTF-8-Daten.

UTF-8 ist inzwischen die Methode der Wahl für die Darstellung von Unicode-Daten auf einem Linux-System.



Wenn Sie genau wissen wollen, wie UTF-8 funktioniert, sollten Sie die Handbuchseite `utf-8(7)` konsultieren, die den Codierungsvorgang erklärt und mit Beispielen illustriert.

Das Kommando `iconv` erlaubt die Umwandlung zwischen verschiedenen Zeichencodierungen. Im einfachsten Fall konvertiert es den Inhalt einer auf der Kommandozeile benannten Datei (oder mehrerer Dateien) von einer angegebenen Zeichencodierung in die aktuell gültige. Das Resultat landet auf der Standardausgabe: `iconv`

```
$ iconv -f LATIN9 test.txt >test-utf8.txt
```

Sie können auch die gewünschte Zielcodierung angeben:

```
$ iconv -f UTF-8 -t LATIN9 test-utf8.txt >test-l9.txt
```

Mit `--output` (oder `-o`) können Sie eine Datei für die Ausgabe auswählen:

```
$ iconv -f LATIN9 -o test-utf8.txt test.txt
```

Ohne Eingabedateien liest `iconv` die Standardeingabe:

```
$ grep bla test.txt | iconv -f LATIN9 -o grep.out
```



Mit der Option `-l` können Sie sich ein Bild davon machen, welche Zeichencodierungen `iconv` kennt (was nicht notwendigerweise bedeutet, dass es erfolgreich zwischen beliebigen Paaren davon konvertieren kann).

Ungültige Zeichen



Wenn `iconv` in seiner Eingabe auf ein Zeichen stößt, das es nicht in die gewünschte Ausgabecodierung übertragen kann, dann meldet es einen Fehler und streicht die Segel. Als Abhilfe können Sie an die Zielcodierung eines der Suffixe `//TRANSLIT` oder `//IGNORE` anhängen. Mit `//IGNORE` werden alle in der Zielcodierung nicht vorhandenen Zeichen unter den Teppich gekehrt, während sie mit `//TRANSLIT`, wo möglich, durch ein oder mehrere ähnliche Zeichen approximiert werden:

```
$ echo xääüy | iconv -f UTF-8 -t ASCII//IGNORE
xy
iconv: ungültige Eingabe-Sequenz an der Stelle 9
$ echo xääüy | iconv -f UTF-8 -t ASCII//TRANSLIT
xaeoeuey
```

Die Option `-c` unterdrückt ungültige Zeichen kommentarlos:

```
$ echo xääüy | iconv -c -f UTF-8 -t ASCII
xy
```

10.3 Spracheneinstellung unter Linux

Die Sprache, die ein Linux-System »spricht«, wird in der Regel bei der Installation aus einem komfortablen Menü ausgewählt und später nicht mehr geändert. Notfalls bieten die Arbeitsumgebungen KDE und GNOME ebenfalls eine komfortable Sprachauswahl an. Als Linux-Anwender kommen Sie also selten in die Verlegenheit, die Sprache explizit und über die Kommandozeile ändern zu müssen, aber auch das ist möglich.

Sprache: pro Sitzung Als erstes müssen wir festhalten, dass die »Systemsprache« keineswegs eine Eigenschaft des kompletten Systems ist, sondern sich jeweils nur auf eine »Sitzung« bezieht. Normalerweise wird Ihre Login-Shell oder Ihre Arbeitsumgebung mit einer bestimmten Spracheneinstellung initialisiert, und alle Prozesse, die von dieser abstammen, »erben« diese Einstellung so, wie zum Beispiel auch das aktuelle Verzeichnis oder die Ressourcenlimits von einem Prozess an seine Kindprozesse vererbt werden. Es spricht also nichts dagegen, dass Sie das System mit einer deutschen Spracheneinstellung verwenden, während gleichzeitig jemand über das Netz oder an einem anderen Bildschirm angemeldet ist, der eine englische oder französische Spracheneinstellung benutzt.



Genau genommen hält Sie auch niemand davon ab, dass Sie selbst in einem oder mehreren Fenstern eine andere Sprache einstellen als die Standardsprache Ihrer Sitzung.

Maßgeblich für die Sprache einer Sitzung ist der Wert der Umgebungsvariablen `LANG`. Im einfachsten Fall besteht er aus einem Sprachencode gemäß ISO 639, gefolgt von einem Unterstrich, gefolgt von einem Ländercode gemäß ISO 3166, also zum Beispiel etwas wie

de_DE	<i>Deutsch in Deutschland</i>
de_AT	<i>Deutsch in Österreich</i>

Die Länderunterscheidung ist wichtig, da die Sprachen zweier Länder sich durchaus unterscheiden können, selbst wenn sie eigentlich dieselbe Sprache verwenden. Unsere österreichischen Freunde sagen bekanntlich »Jänner« statt »Januar«

und geben ihren Gemüsen merkwürdige Namen wie »Paradeiser« und »Karfiol« – ersteres ist wichtig, wenn Ihr Linux-System das Datum ausgeben soll, und bei letzterem ist es zumindest denkbar, dass ein Textverarbeitungsprogramm das Wort »Karfiol« in einem de_DE-Text genauso als merkwürdig anneckert wie das Wort »Blumenkohl« unter de_AT.



Wenn der Unterschied zwischen de_DE und de_AT Ihnen nicht krass genug ist, dann denken Sie mal über en_GB und en_US nach, oder über pt_PT und pt_BR.

Hinter dieser einfachen Spezifikation können noch Erweiterungen folgen, etwa eine Zeichencodierung (hinter einem Punkt) oder eine »Variante« (hinter einem @). Sie können also Werte verwenden wie

de_DE.ISO-8859-15	<i>Deutsch in Deutschland, gemäß ISO Latin-9</i>
de_AT.UTF-8	<i>Deutsch in Österreich, basierend auf Unicode</i>
de_DE@euro	<i>Deutsch in Deutschland, mit Euro (ISO Latin-9)</i>

Hier sehen Sie die Auswirkungen einer LANG-Änderung:

```
$ for i in de_DE de_AT en_US fi_FI fr_FR; do
> LANG=$i.UTF-8 date +"%B %Y"
> done
Januar 2009
Jänner 2009
January 2009
tammikuu 2009
janvier 2009
```

(Bei date steht der Formatschlüssel %B für »Monatsname gemäß der aktuellen Spracheinstellung«.)



Das Ganze setzt natürlich voraus, dass es auf dem betreffenden System überhaupt Unterstützung für die jeweilige Sprache gibt. Debian GNU/Linux zum Beispiel lässt Ihnen die Wahl, welche Einstellungen gültig sein sollen oder nicht. Wenn Sie sich eine Einstellung wünschen, die Ihr System nicht unterstützt, dann fällt das System auf einen eingebauten Standardwert zurück, und das ist in der Regel Englisch.

Die Umgebungsvariable LANGUAGE (nicht zu verwechseln mit LANG) wird nur von Programmen beachtet, die die GNU-gettext-Infrastruktur verwenden, um ihre Meldungen in verschiedene Sprachen zu übersetzen (und das sind unter Linux die meisten). Der offensichtlichste Unterschied zwischen LANGUAGE und LANG ist, dass LANGUAGE es erlaubt, mehrere Sprachen aufzuzählen (durch Doppelpunkte getrennt). Damit können Sie eine Präferenzliste angeben:

```
LANGUAGE=de_DE.UTF-8:en_US.UTF-8:fr_FR.UTF-8
```

bedeutet »Deutsch, oder sonst Englisch, oder sonst Französisch«. Die erste Sprache, für die ein Programm Systemmeldungen mitbringt, gewinnt. LANGUAGE hat (für Programme, die GNU gettext benutzen) Vorrang gegenüber LANG.

Übungen



10.1 [1] Was erscheint als Ausgabe des Kommandos

```
$ LANG=ja_JP.UTF-8 date +"%B %Y"
```

(installierte Unterstützung für diese Sprache vorausgesetzt)?

10.4 Lokalisierungs-Einstellungen

Tatsächlich beeinflusst der Wert von LANG nicht nur die Sprache, sondern die komplette »kulturelle Einstellung« eines Linux-Systems. Dazu gehören außerdem Sachen wie

Formatierung von Datums- und Zeitangaben In den USA ist es zum Beispiel üblich, ein Datum in der Form »Monat/Tag/Jahr« anzugeben:

<pre>\$ date +%x" 14.01.2009 \$ LANG=en_US.UTF-8 date +%x" 01/14/2009</pre>	<i>Landesabhängige Zeitangabe</i>
---	-----------------------------------

Ferner benutzen manche Länder (wieder die USA oder auch Großbritannien) eine 12-Stunden-Zeit, während zum Beispiel in Deutschland eine 24-Stunden-Zeit üblich ist: Was wir »15 Uhr« nennen würden, ist dort »3 p. m.«.

Formatierung von Zahlen und Geldbeträgen In Kontinentaleuropa verwenden wir ein Komma als Dezimaltrenner und Punkte, um die Lesbarkeit einer großen Zahl zu erhöhen:

299.792.458,0

In den USA dagegen ist es gerade umgekehrt:

299,792,458.0

 Diese Einstellung wirkt sich zunächst auf Programme aus, die die C-Funktionen printf() und scanf() verwenden. Andere Programme müssen die Einstellung explizit abfragen und in ihrer Ausgabe berücksichtigen.

Bei Geldbeträgen kommt dazu, dass zum Beispiel negative Salden manchmal durch ein vorgesetztes Minus und manchmal durch Klammern kenntlich gemacht werden (unter anderem).

Klassifizierung von Zeichen Es hängt von der verwendeten Sprache ab, welche Zeichen in der betreffenden Codetabelle als Buchstaben gelten, welche als Sonderzeichen und so weiter. Im Zeitalter von Unicode ist das nicht mehr so ein Problem, da die Codepunkte insgesamt standardisiert sind; komplizierter ist es, wenn zum Beispiel ISO-8859- oder gar ASCII-basierte Codierungen verwendet werden. In ASCII ist »[« zum Beispiel ziemlich eindeutig ein Sonderzeichen, das »Ä«, das in DIN 66003 denselben Platz in der Tabelle belegt, aber ebenso eindeutig ein Buchstabe. Dies hat auch Auswirkungen auf die Konvertierung zwischen Groß- und Kleinbuchstaben und ähnliches.

 Das deutsche scharfe S (»ß«) hat keine grafische Entsprechung als Großbuchstabe – ein Wort wie »Fuß« wird in Versalien »FUSS« geschrieben. (Bei Verwechslungsgefahr wurde sogar »SZ« empfohlen, etwa bei »MASSE« vs. »MASZE«, aber seit der neuen Rechtschreibung ist das abgeschafft.) Im Zusatz 4:2008 zu ISO 10646, der am 23. Juni 2008 veröffentlicht wurde, ist ein Codepunkt für ein »großes ß« (U+1E9E) enthalten, so dass grundsätzlich einer Behebung dieses Problems an der Wurzel nichts mehr im Weg steht. Wir müssen uns nur noch darüber einigen, wie dieses Zeichen tatsächlich aussehen soll. (»SS« wäre eine naheliegende Wahl.)

Tabelle 10.2: LC_*-Umgebungsvariable

Variable	Bedeutung
LC_ADDRESS	Formatierung von Adressen und Ortsangaben
LC_COLLATE	Zeichensortierung
LC_CTYPE	Zeichenklassifizierung und Groß- und Kleinschreibung
LC_MONETARY	Formatierung von Geldbeträgen
LC_MEASUREMENT	Maßeinheiten (metrisch oder anders)
LC_MESSAGES	Sprache für Meldungen und Gestalt von positiven bzw. negativen Antworten
LC_NAME	Formatierung von Eigennamen
LC_NUMERIC	Formatierung von Zahlen
LC_PAPER	Papierformat (umstritten)
LC_TELEPHONE	Formatierung von Telefonnummern
LC_TIME	Formatierung von Datums- und Zeitangaben
LC_ALL	Alle Einstellungen

Sortierreihenfolge von Zeichen Auch das ist keineswegs so eindeutig, wie man glauben mag. Schon in Deutschland gibt es zwei konkurrierende Methoden gemäß DIN 5007: In Lexika werden Umlaute als äquivalent zu ihren »Basiszeichen« angesehen (ein »ä« wird also für Zwecke der Sortierung als »a« interpretiert), während in Namenslisten, zum Beispiel Telefonbüchern, Umlaute gemäß ihrer »Ersatzschreibweise« sortiert werden (»ä« also wie »ae«). In beiden Fällen ist »ß« äquivalent zu »ss«.



Bei Namenslisten wünscht man sich offenbar, dass der Unterschied zwischen den Homophonen »Müller« und »Mueller« die Suche nicht zu sehr erschwert. Herrn Müller müßten Sie sonst zwischen Frau Muktadir und Herrn Muminovic suchen, während Frau Mueller zwischen Herrn Muders und Frau Muffert zu finden wäre – eine Unbequemlichkeit erster Güte. Im Lexikon dagegen sollte die Rechtschreibung des Suchbegriffs und damit dessen Einsortierung klar sein.

In Schweden dagegen stehen die Buchstaben »å«, »ä« und »ö« in dieser Reihenfolge am Ende des Alphabets (hinter »z«). In Großbritannien kommt »ä« nach »a« (also zwischen »az« und »b«), und gemeinerweise ist der Namensbestandteil »Mc« äquivalent zu »Mac« (die Sortierung ist also »Macbeth, McDonald, MacKenzie«). Sprachen auf der Basis von Ideogrammen wie Japanisch und Chinesisch sind noch unbequemer zu sortieren; in Wörterbüchern orientiert man sich an der Struktur der Ideogramme und ihrer Strichanzahl, während auf Computern bequemerweise nach der lateinischen Umschrift sortiert wird. (Wir hören an dieser Stelle auf, bevor Ihnen der Kopf platzt.)

Die LANG-Variable setzt neben der Sprache auch das alles mit einem Schlag auf die in einem bestimmten Kulturkreis (engl. *locale*) gültigen Werte. Daneben existiert aber auch die Möglichkeit, einzelne Aspekte der Lokalisierung separat einzustellen. Dafür gibt es eine Reihe von Umgebungsvariablen der Form LC_* (siehe Tabelle 10.2).



Wenn Sie wissen wollen, was diese Einstellungen tatsächlich beinhalten, können Sie das Kommando `locale` heranziehen, etwa so:

```
$ locale -k LC_PAPER
height=297
width=210
paper-codeset="UTF-8"
```

Auf diese Weise können Sie also herausfinden, dass Papierblätter in Deutschland in der Regel 297 mm hoch und 210 mm breit sind. Wir kennen das als »DIN A4«.



Die tatsächlichen *Definitionen*, die den Einstellungen zugrunde liegen, finden Sie im Verzeichnis `/usr/share/i18n/locales`. Grundsätzlich steht Ihnen nichts im Weg, wenn Sie eine eigene Einstellungsdatei entwerfen wollen (außer vielleicht der spärlichen Dokumentation). Das Programm `localedef` macht die eigentliche Arbeit.

`LC_ALL` Mit `LC_ALL` können Sie ähnlich wie mit `LANG` auf einen Schlag alle Kulturkreis-Einstellungen setzen. Bei der Bestimmung, welche Einstellung nun letzten Endes maßgeblich ist, geht das System wie folgt vor:

1. Wenn `LANG` gesetzt ist, dann gilt dessen Wert.
2. Wenn `LANG` nicht gesetzt ist, aber die `LC_*`-Variable für den betreffenden Bereich (etwa `LC_COLLATE`) gesetzt ist, dann gilt deren Wert.
3. Wenn weder `LANG` noch die passende `LC_*`-Variable gesetzt sind, aber die Variable `LC_ALL` gesetzt ist, dann gilt deren Wert.
4. Wenn überhaupt nichts dergleichen gesetzt ist, dann gilt ein fest ins Programm eingebauter Standardwert.

Beachten Sie also, dass Sie, wenn (wie üblich) die `LANG`-Variable gesetzt ist, mit den `LC_*`-Variablen machen können, was Sie wollen – ohne dass es irgendwelche Auswirkungen zeitigt.



Wenn Sie sich jetzt am Kopf kratzen und sich wundern, dann haben Sie völlig recht – warum sollte man überhaupt `LC_*`-Variable setzen, wenn `LANG`, die Umgebungsvariable, die das System beim Anmelden für Sie setzt, eh alles andere platt macht? Für uns ist das genauso ein Rätsel wie für Sie, aber es gibt ja notfalls `.bash_profile` und »`unset LANG`«.

Das Kommando »`locale -a`« liefert eine Liste von Werten, die das aktuelle System für `LANG` und die `LC_*`-Variablen unterstützt:

```
$ locale -a
C
de_AT.utf8
de_DE
de_DE@euro
de_DE.utf8
deutsch
<<<<<<
POSIX
```



Sachen wie `LANG=deutsch` sehen auf den ersten Blick verlockend aus, sind aber zu unkonkret, um nützlich zu sein. Offiziell verpönt sind sie außerdem, werden aber aus Kompatibilitätsgründen (noch) beibehalten. Machen Sie einen Bogen darum.

`C` Die magischen Werte `C` und `POSIX` (äquivalent) in der Liste beschreiben den fest
`POSIX` eingebauten Standard, den Programme verwenden, wenn sie keine gültige andere Einstellung finden. Dies ist nützlich, wenn Sie möchten, dass Programme wie `ls` ein definiertes Ausgabeformat liefern. Vergleichen Sie zum Beispiel

```
$ LANG=de_DE.UTF-8 ls -l /bin/ls
-rwxr-xr-x 1 root root 92312 4. Apr 2008 /bin/ls
$ LANG=ja_JP.UTF-8 ls -l /bin/ls
```

```
-rwxr-xr-x 1 root root 92312 2008-04-04 16:22 /bin/ls
$ LANG=fi_FI.UTF-8 ls -l /bin/ls
-rwxr-xr-x 1 root root 92312  4.4.2008 /bin/ls
$ LANG=C ls -l /bin/ls
-rwxr-xr-x 1 root root 92312 Apr  4 2008 /bin/ls
```

Wir führen dasselbe Kommando mit vier verschiedenen LANG-Einstellungen aus und erhalten vier verschiedene Resultate, die sich alle in der Datumsangabe unterscheiden. Gemeinerweise kann diese Datumsangabe, je nach Spracheinstellung, aus der Sicht von Programmen wie `awk` oder `»cut -d' '«` aus einem, zwei oder drei Feldern bestehen – fatal, wenn ein Skript diese Ausgabe analysieren soll! Sie tun also gut daran, in solchen Zweifelsfällen Programme wie `ls`, deren Ausgabeformat von der Sprache abhängt, per explizitem `LANG=C` auf einen Standard festzunageln, der auf jeden Fall existiert (bei allen anderen Einstellungen können Sie sich da nicht sicher sein).

Übungen



10.2 [2] Das Programm `printf(1)` (nicht zu verwechseln mit dem eingebauten Shellkommando `printf`) richtet sich beim Formatieren von Zahlen nach der `LC_NUMERIC`-Einstellung. Insbesondere formatiert ein Kommando wie

```
$ /usr/bin/printf "%g\n" 31415.92
```

eine Dezimalzahl mit dem für die aktuelle Spracheinstellung korrekten Dezimaltrenner und »Übersichtszeichen«. Experimentieren Sie mit dem Programm und verschiedenen Einstellungen für `LANG`.



10.3 [2] Finden Sie Programme außer `date`, `ls` und `printf`, die sich nach Sprach- bzw. Kulturkreiseinstellungen richten. (Übersetzte Fehlermeldungen sind zu einfach und zählen nicht, es muss schon etwas Interessantes sein.)

10.5 Zeitzonen

Zu guter Letzt müssen wir noch ein paar Dinge über Zeitzonen sagen und beschreiben, wie Linux mit ihnen umgeht. Spätestens wenn Sie mal eine weite Flugreise nach Westen oder Osten unternommen haben, dürfte Ihnen klar sein, dass es nicht überall auf der Erde immer gleich spät ist – steht bei uns in Europa die Sonne hoch am Himmel, ist es in Amerika möglicherweise noch dunkel und in Ostasien neigt der Tag sich schon wieder dem Ende zu. Das wäre alles kein Problem (Ihren Wecker stellen Sie ja auch nach dem Zeitsignal im Radio), wenn da nicht das Internet wäre, das es ermöglicht, Daten mit hoher Geschwindigkeit zwischen irgendwelchen Computern irgendwo auf der Welt auszutauschen. Und ob eine E-Mail nun um 12 Uhr europäischer, amerikanischer oder australischer Zeit geschrieben wurde, ist schon ein Unterschied von etlichen Stunden, auf den man Rücksicht nehmen möchte.

Heutige Computersysteme gestatten es also, festzulegen, in welcher Zeitzone der Rechner aufgestellt ist – typischerweise werden Sie bei der Systeminstallation danach gefragt, und wenn Sie nicht gerade mit dem Rechner im Gepäck auswandern, müssen Sie den einmal gesetzten Wert auch nie wieder ändern.



Die Zeitzonen der Erde richten sich im wesentlichen nach dem Umstand, dass ein Unterschied von 15 Grad in geografischer Länge einem Unterschied von einer Stunde auf der Uhr entspricht (was logisch ist, denn der komplette Erdumfang im Werte von 24 Stunden entspricht ja 360 Grad, und $360/24$

Tabelle 10.3: Die Sommerzeit in Deutschland

Zeitraum	Situation
vor 1916	Keine Zeitumstellung
1916	Sommerzeit vom 1. Mai bis zum 1. Oktober
1917–1918	Sommerzeit von Mitte April bis Mitte September
1919–1939	Keine Zeitumstellung
1940–1942	Die Uhr wurde am 1. April 1940 um eine Stunde vorgestellt und blieb so bis zum 2. November 1942 (!)
1943–1944	Sommerzeit von Ende März/Anfang April bis Anfang Oktober
1945	Sommerzeit vom 2. April bis 18. November, außerdem zwischen dem 24. Mai und dem 24. September »doppelte« Sommerzeit (die Uhr wurde nochmals eine Stunde vorgestellt)
1946–1949	Sommerzeit von Mitte April bis Anfang Oktober
1947	»Doppelte« Sommerzeit zwischen dem 11. Mai und dem 29. Juni
1950–1979	Keine Zeitumstellung
1980–1995	Sommerzeit vom letzten März- bis zum letzten Septembersonntag (war in der Bundesrepublik schon 1978 beschlossen worden, aber man musste sich mit der DDR einigen)
seit 1996	Sommerzeit vom letzten März- bis zum letzten Oktobersonntag (EU-Vereinheitlichung)

ist nun mal 15). Früher gab es keine Zeitzonen, sondern jeder Ort hatte seine eigene Zeit, wobei man Wert darauf legte, dass die Sonne um 12 Uhr mittags möglichst genau im Süden stehen sollte (vermutlich damit die Sonnenuhren stimmen). Die Einführung des Eisenbahnverkehrs und die mit der »Ortszeit« verbundenen Schwierigkeiten bei der Fahrplan-Erstellung machten das jedoch zusehends unpraktisch, so dass man sich zur Einführung der Zeitzonen entschloss, um den Preis, dass die Zeit auf der Uhr lokal nicht mehr mit der »astronomischen« Zeit übereinstimmt. Die Grenzen der Zeitzonen orientieren sich ohnehin nicht (ausschließlich) an Längengraden, sondern eher an politischen Grenzen.



Dass das in der Praxis durchaus merkbar sein kann, sieht man in Europa. Spanien zum Beispiel folgt der »mitteleuropäischen Zeit« (MEZ), die auf 15 Grad östlicher Länge bezogen ist. Das entspricht zum Beispiel Görlitz an der deutsch-polnischen Grenze. Wenn es in A Coruña an der spanischen Atlantikküste (fast 8,5 Grad *westlicher* Länge) auf der Uhr 12 Uhr mittags ist, dann ist die Sonne erst auf ungefähr halb elf. (Portugal verwendet übrigens dieselbe Zeit wie Großbritannien; dort ist es dann immerhin schon fast halb zwölf.)



Weiter verkompliziert wird das Ganze durch die »Sommerzeit« (engl. *daylight savings time*), bei der die Uhren in einer Zeitzone im Frühjahr künstlich eine Stunde vorgestellt werden und im Herbst wieder zurück. Die Sommerzeit ist ein rein politisches Phänomen, das aber trotzdem beachtet werden muss – in Deutschland hat sie eine durchaus bewegte Geschichte (Tabelle 10.3), die illustriert, warum Sie bei Linux nicht einfach »MEZ« als Zeitzone einstellen können, sondern sich »Europe/Berlin« wünschen müssen.

Wie bei den Sprach- und Kultureinstellungen gilt, dass die Zeitzone auf einem Linux-System keine eindeutige systemweite Einstellung ist, sondern zu den vererbaren Eigenschaften eines Prozesses gehört. Der Linux-Kernel misst die Zeit in Sekunden seit dem 1.1.1970, 0 Uhr UTC, so dass der Komplex »Zeitzone« lediglich eine Frage der Formatierung dieser (inzwischen recht großen) Sekunden-

zahl¹. Damit werden elegant alle Schwierigkeiten umgangen, die andere Betriebssysteme mit der Sommerzeit-Umstellung hatten und haben, und es besteht auch gar kein Problem darin, dass Ihr Kumpel aus Sydney sich über das Internet auf Ihrem Rechner anmeldet und die australische Zeit sieht, während Sie selbst natürlich MEZ verwenden. So gehört sich das.

Die Voreinstellung für die Zeitzone, die Sie bei der Installation des Systems machen, hinterlegt Linux in der Datei `/etc/timezone`:

```
$ cat /etc/timezone
Europe/Berlin
```

Die gültigen Zeitzonen können Sie den Namen der Dateien unter `/usr/share/zoneinfo` entnehmen:

```
$ ls /usr/share/zoneinfo
Africa/      Chile/      Factory    Iceland    MET        Portugal   Turkey
America/    CST6CDT    GB         Indian/    Mexico/    posix/     UCT
Antarctica/ Cuba       GB-Eire    Iran       Mideast/   posixrules Universal
Arctic/     EET        GMT        iso3166.tab MST        PRC        US/
Asia/       Egypt     GMT0       Israel     MST7MDT    PST8PDT    UTC
Atlantic/   Eire      GMT-0      Jamaica   Navajo     right/     WET
Australia/  EST       GMT+0      Japan     NZ         ROC        W-SU
Brazil/     EST5EDT   Greenwich  Kwajalein NZ-CHAT    ROK        zone.tab
Canada/     Etc/      Hongkong   Libya     Pacific/   Singapore  Zulu
CET         Europe/   HST        localtime@ Poland     SystemV/
```

Hierbei handelt es sich größtenteils um Verzeichnisse:

```
$ ls /usr/share/zoneinfo/Europe
Amsterdam  Chisinau   Kiev       Moscow     Sarajevo   Vatican
Andorra    Copenhagen Lisbon     Nicosia    Simferopol Vienna
Athens     Dublin     Ljubljana Oslo       Skopje     Vilnius
Belfast    Gibraltar London     Paris      Sofia      Volgograd
Belgrade   Guernsey  Luxembourg Podgorica  Stockholm  Warsaw
Berlin     Helsinki  Madrid     Prague     Tallinn    Zagreb
Bratislava Isle_of_Man Malta      Riga       Tirane     Zaporozhye
Brussels   Istanbul  Mariehamn Rome       Tiraspol   Zurich
Bucharest  Jersey    Minsk      Samara     Uzhgorod
Budapest   Kaliningrad Monaco     San_Marino Vaduz
```



Die Regel ist, dass die Zeitzone nicht etwa heißt wie die Hauptstadt des betreffenden Landes (das wäre zu einfach), sondern so wie die bevölkerungsreichste Stadt in dem Teil des betreffenden Landes, für den die Zeitzone gilt. Die Schweiz wird also durch `Europe/Zurich` abgedeckt (und nicht `Europe/Berne`), und Russland hat insgesamt 11 Zeitzonen, die allerdings nicht alle unter `Europe` gezählt werden. `Europe/Kaliningrad` ist zum Beispiel eine Stunde vor `Europe/Moscow`, und das wiederum ist wiederum zwei Stunden vor `Asia/Yekaterinburg`.



`/usr/share/zoneinfo` enthält auch einige »Bequemlichkeits-Zeitzone« wie `Poland` oder `Hongkong`. `Zulu` hat nichts mit Südafrika zu tun, sondern bezieht sich, wie Leser von Tom Clancy wissen, auf die Weltzeit (UTC), die gerne als `12:00Z` angegeben wird, wobei die NATO »Z« als »Zulu« buchstabiert.

`/etc/localtime` ist eine Kopie der Datei unter `/usr/share/zoneinfo`, die die Informationen für die Zeitzone enthält, die durch den Inhalt von `/etc/timezone` gegeben ist – zum Beispiel `/usr/share/zoneinfo/Europe/Berlin`.

¹Am 14. Februar 2009 um 0:31:30 Uhr MEZ sind seit dem Anbeginn der Linux-Zeitrechnung genau 1234567890 Sekunden vergangen. Fröhlichen Valentinstag!



Eigentlich sollte `/usr/share/zoneinfo` Zeitzonen für jeden Geschmack zur Verfügung stellen. Sollten Sie jemals in die Verlegenheit kommen, selber eine neue Zeitzone definieren zu müssen, dann können Sie das über den »Zeitzone-Compiler« `zic` tun. Mit `zdump` können Sie die Zeit in jeder beliebigen Zeitzone oder auch die »Vorgeschichte« der Sommerzeit in jeder Zeitzone abrufen. (Raten Sie mal, wo wir die Informationen für Tabelle 10.3 her haben.)

Sie können die systemweite Zeitzone »manuell« setzen, indem Sie die Dateien `/etc/timezone` und `/etc/localtime` anpassen:

```
# echo Asia/Tokyo >/etc/timezone
# cp /usr/share/zoneinfo/$(cat /etc/timezone) /etc/localtime
```

Darüber hinaus bieten die Distributionen verschiedene Hilfsmittel zur komfortableren Konfiguration der Zeitzone an.



Mit `tzselect` können Sie interaktiv eine Zeitzone auswählen. Das Programm fragt Sie nach Erdteil und Land und bietet Ihnen dann eine Auswahl der gültigen Zeitzonen an. Es liefert den Namen der Zeitzone auf der Standardausgabe, während die komplette Interaktion sich auf der Standardfehlerausgabe abspielt; Sie können also mit etwas wie

```
$ TZ=$(tzselect)
```

das Ergebnis in eine Umgebungsvariable holen. – Zur Änderung der systemweiten Zeitzone wird nicht `tzselect` verwendet, sondern der `debconf`-Mechanismus; rufen Sie

```
# dpkg-reconfigure tzdata
```

auf. Das hin und wieder noch erwähnte Programm `tzconfig` ist verpönt.



SUSE-Anwender können die systemweite Zeitzone über den YaST setzen. Für die »persönliche« Zeitzone gibt es nichts offensichtliches Bequemes.



Die systemweite Voreinstellung für die Zeitzone steht in der Datei `/etc/sysconfig/clock`. Außerdem gibt es ein Programm namens `timeconfig`, und das bei Debian GNU/Linux diskutierte `tzselect` steht ebenfalls zur Verfügung.

Unabhängig von der systemweiten Voreinstellung für die Zeitzone können Sie in der Umgebungsvariablen `TZ` den Namen einer Zeitzone hinterlegen, die dann für den betreffenden Prozess verwendet wird (und sich wie andere Umgebungsvariablen an Kindprozesse vererbt). Mit etwas wie

```
$ export TZ=America/New_York
```

können Sie zum Beispiel für eine Shell und alle Programme, die Sie aus ihr heraus starten, die Zeitzone `America/New_York` setzen.

Sie können die Zeitzone auch nur für ein einziges Kommando ändern:

```
$ TZ=America/New_York date
```

zeigt Ihnen die aktuelle Zeit in New York.



Mit der Variablen `TZ` können Sie eine Zeitzone auch beschreiben, ohne auf die Zeitzone-Daten in `/usr/share/zoneinfo` zurückgreifen zu müssen. Im einfachsten Fall geben Sie dazu einfach den (abgekürzten) Namen der Zeitzone und die Differenz zur UTC an. Letztere muss ein Vorzeichen haben (»+« für Zeitzonen westlich vom Nullmeridian, »-« für östlich), gefolgt von einer Zeitspanne der Form `HH:MM:SS` (der Minuten- und Sekundenanteil ist optional, und aktuell gibt es auch keine Zeitzonen mit einer Sekundenverschiebung). Etwas wie

```
$ export TZ=MEZ-1
```

würde also »mitteleuropäische Zeit« setzen, allerdings ohne Sommerzeit oder gar die deutsche Sommerzeit-Vorgeschichte. Um die Sommerzeit mit anzugeben, müssen Sie deren Namen, eine eventuelle Differenz zur Normalzeit (wenn sie nicht auf »eine Stunde vorgestellt« hinausläuft) und eine Regel für die Umstellung angeben. Die Umstellungsregel besteht aus einer Tagesangabe gefolgt von einer optionalen Zeitangabe (abgetrennt durch einen Schrägstrich), wobei die Tagesangabe eine von drei Formen haben kann:

Jn Die Nummer des Tags im Jahr, gezählt von 1 bis 365. Der 29. Februar, wenn es ihn gibt, zählt nicht mit.

n Die Nummer des Tags im Jahr, gezählt von 0 bis 365. In Schaltjahren zählt der 29. Februar mit.

Mm.w.d Tag *d* der Woche *w* in Monat *m*. *d* ist zwischen 0 (Sonntag) und 6 (Samstag), die Woche ist zwischen 1 und 5, wobei 1 für die erste und 5 für die letzte Woche mit einem Tag *d* steht (letzteres unabhängig davon, wie viele Tage *d* es im Monat tatsächlich gibt). *m* ist zwischen 1 und 12.

Die aktuell in Deutschland übliche Regel könnten Sie also mit

```
$ export TZ=MEZ-1MESZ,M3.5.0/2,M10.5.0/2
```

einstellen, wobei Ihnen aber auch hier die »Vorgeschichte« aus Tabelle 10.3 entgeht.

Übungen



10.4 [1] Warum ist `/etc/localtime` eine Kopie der relevanten Datei aus `/usr/share/zoneinfo`? Ein symbolisches Link würde doch auch reichen, oder man konsultiert direkt die richtige Datei in `/usr/share/zoneinfo`. Was denken Sie?



10.5 [!2] Stellen Sie sich vor, Sie sind Börsenmakler und brauchen einen schnellen Überblick über die Zeit in Tokyo, Frankfurt und New York. Wie können Sie das mit Linux verwirklichen?



10.6 [2] Geben Sie eine TZ-Umstellungsregel für das hypothetische Land Nirgendwonien an. Es gelten die folgenden Grundregeln:

- In Nirgendwonien gilt die Nirgendwonische Normalzeit (NNZ). 12 Uhr UTC entspricht 13:15 Uhr NNZ.
- Vom zweiten Mittwoch im April um 3 Uhr morgens bis zum 10. Oktober (in Schaltjahren dem 11. Oktober) um 21 Uhr gilt die Nirgendwonische Sommerzeit (NSZ), während der alle Uhren in Nirgendwonien um 25 Minuten vor gestellt werden.

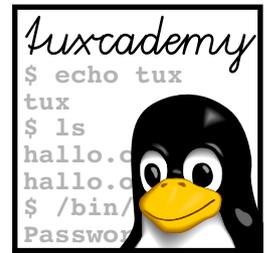
Wie können Sie Ihre Regel testen?

Kommandos in diesem Kapitel

iconv	Konvertiert zwischen Zeichencodierungen	iconv(1)	157
locale	Zeigt Informationen über die Lokalisierung an	locale(1)	161, 162
localedef	Übersetzt Locale-Definitionsdateien	localedef(1)	162
timeconfig	[Red Hat] Erlaubt die bequeme Festlegung der systemweiten Zeitzone	timeconfig(8)	166
tzselect	Erlaubt eine bequeme interaktive Wahl der Zeitzone	tzselect(1)	166
zdump	Gibt die aktuelle Zeit oder die Zeitzonendefinitionen für verschiedene Zeitzonen aus	zdump(1)	165
zic	Compiler für Zeitzonen-Dateien	zic(8)	165

Zusammenfassung

- Internationalisierung ist die Ausgestaltung eines Softwaresystems, so dass eine spätere Lokalisierung möglich ist. Lokalisierung ist die Anpassung eines internationalisierten Systems an die Gepflogenheiten eines bestimmten Kulturkreises.
- Gängige Zeichencodierungen unter Linux sind ASCII, ISO 8859 und ISO 10646 bzw. Unicode.
- UCS-2, UTF-8 und UTF-16 sind Codierungsformen von ISO 10646/Unicode
- Das Kommando `iconv` erlaubt die Umwandlung zwischen verschiedenen Zeichencodierungen.
- Die Sprache für einen Linux-Prozess können Sie über die Umgebungsvariable `LANG` einstellen.
- Die Umgebungsvariablen `LC_*` steuern zusammen mit `LANG` die Lokalisierung eines Linux-Prozesses.
- Das Kommando `locale` gibt Ihnen Zugriff auf die Lokalisierungsinformationen.
- In Skripten sollten Sie `LANG=C` verwenden, um lokalisierungsspezifische Einflüsse auf das Verhalten von Standardkommandos auszuschließen.
- Die systemweite Zeitzone entnimmt Linux der Datei `/etc/timezone`.
- Pro Prozess kann über die Umgebungsvariable `TZ` eine Zeitzone gesetzt werden.



11

Die Grafikoberfläche X11

Inhalt

11.1 Grundlagen	170
11.2 X11 konfigurieren.	175
11.3 Displaymanager	182
11.3.1 Grundlegendes zum Starten von X	182
11.3.2 Der Displaymanager LightDM	183
11.3.3 Andere Displaymanager	185
11.4 Informationen anzeigen	186
11.5 Der Schriftenserver	188
11.6 Fernzugriff und Zugriffskontrolle	191

Lernziele

- Struktur und Funktionsweise von X11 kennen
- Mit Displaynamen und der DISPLAY-Variablen umgehen können
- Methoden zum Start von X11 kennen
- Mit einem Displaymanager umgehen können
- Schriftenverwaltung und den Schriftenserver kennen
- Fernzugriff auf einen X-Server konfigurieren können

Vorkenntnisse

- Kenntnisse der grafischen Oberfläche eines anderen Betriebssystems (nützlich)

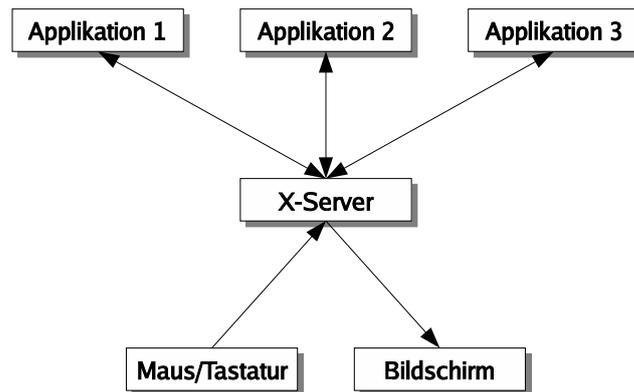


Bild 11.1: X Window System als Client-Server-System

11.1 Grundlagen

Das *X Window System*, kurz »X11« oder »X« (niemals »X Windows«), ist eine grafische Betriebsumgebung, die 1985–1987 am MIT (*Massachusetts Institute of Technology*) entwickelt wurde.

💡 X11 stammt ursprünglich aus dem MIT-Projekt »Athena« und wurde danach unter den Auspizien des »X-Konsortiums« weitergeführt, das später in der *Open Group* aufging. Die Software war von Anfang an frei verfügbar und enthielt neben diversen (portablen) Benutzerprogrammen auch einen Server mit Anpassungen an die meisten grafischen Arbeitsplatzrechner der damaligen Zeit.

X.org 💡 Die kanonische Implementierung von X11 für Linux heißt **X.org** (es gibt andere, aber die sind nicht von praktischer Bedeutung).

X-Protokoll Grundlage von X11 ist das **X-Protokoll**, eine Methode zur Übertragung von grafischen Grundoperationen über eine Netzverbindung. X11 ist ein Client-Server-System, wenn auch etwas anders als man es sonst kennt: Der **X-Server** läuft auf einem Arbeitsplatzrechner mit Grafikkarte, Maus, Grafiktablett oder ähnlichen Peripheriegeräten, und **X-Clients** – Anwendungsprogramme – schicken ihm Grafikbefehle über das X-Protokoll. Umgekehrt schickt der X-Server Ereignisse, etwa Tastendrucke und Mausbewegungen, an die X-Clients zurück.

💡 X-Clients und -Server tauschen X-Protokollnachrichten unter Linux normalerweise über Unix-Domain-Sockets aus, also schnelle Verbindungen für Programme auf demselben Rechner. Es spricht aber nichts dagegen, X-Protokollnachrichten über TCP/IP zu transportieren. X-Clients können also ohne Weiteres auf Rechnern laufen, die selber im Extremfall gar keine Grafikhardware haben, solange sie über das Netz mit einem X-Server auf einem anderen Rechner kommunizieren können. Hierzu gleich mehr.

💡 Die Operationen des X-Protokolls sind ziemlich primitiv – es unterstützt vor allem simple Grafikoperationen wie das Zeichnen von Punkten, Linien, Kreisen, Rechtecken und die Anzeige von Zeichenketten. Außerdem gibt es Funktionen zur Verwaltung von Fenstern (rechteckigen Bereichen auf dem Bildschirm, die Ziel von Ereignissen wie Mausklicks oder Tastendrucke sein können) und für die interne Organisation. Das heisst, wenn eine Anwendung zum Beispiel ein Aufklappenü anbietet, muss der X-Server der

Anwendung einen Mausklick auf den »Knopf« des Menüs melden; die Anwendung schickt dann die notwendigen Befehle zur Darstellung des Menüs und übernimmt – vom X-Server mit einem stetigen Strom von Mausbewegungs-Nachrichten versorgt – die Interaktion mit dem Benutzer, bis dieser sich schließlich für einen Menüeintrag entscheidet. Dabei entsteht natürlich einige Netzlast. Diese Interaktion zwischen Server und Anwendung auf sehr niedriger Ebene wird oft an X11 kritisiert. Es wäre durchaus möglich, mehr von dieser Interaktion in den Server zu verlagern – beispielsweise könnte man eine Methode vorsehen, mit der die Anwendung dem Server die Menüeinträge mitteilt, der Server bei Bedarf die komplette Interaktion mit dem Benutzer übernimmt, und die Anwendung am Ende nur noch den Index des gewählten Menüeintrags geschickt bekommt –; allerdings würde das bedeuten, dass der Server wissen muss, wie das Menü tatsächlich grafisch auf dem Bildschirm aussehen soll und wie die Interaktion im Detail funktioniert. Die bisherige Philosophie von X ist dagegen, dass der Server zwar Grafikoperationen zur Verfügung stellt, sich aber nicht darum kümmert, wie diese tatsächlich eingesetzt werden (*mechanism, not policy*) – und der Leidensdruck reicht vor allem angesichts der heute sehr schnellen Netzverbindungen nicht für einen solchen »Paradigmenwechsel« aus. Verschiedene andere Grafiksyste­me – Sun Microsystems' »NeWS« in den 1980er Jahren oder, aktueller, das »Berlin«-Projekt – haben versucht, diese Idee konsequenter umzusetzen, waren aber aus verschiedenen Gründen nicht erfolgreich; NeWS scheiterte an der damals noch nicht ausreichenden Rechenleistung von Workstations und Berlin daran, dass X11 einfach »gut genug« ist.

Es gibt minimale Systemanforderungen für Rechner, auf denen ein X11-Server laufen soll. Wir erwähnen diese hier nicht, um uns nicht lächerlich zu machen, da sie im 21. Jahrhundert von Mobiltelefonen und kleinen Netbooks fast um Größenordnungen übertroffen werden (nicht dass viele Mobiltelefone tatsächlich X11 benutzen, aber theoretisch wäre es kein Problem). Die einzige interessante Frage dieser Tage ist, ob es für den Grafikchip im Rechner (auf demselben Chip wie die CPU oder auf einer zusätzlichen Steckkarte) passende Treiber gibt. In der Regel ist das kein Problem, nur bei exotischer – also nicht von einem der drei »Platzhirsche« Intel, AMD und nVidia hergestellter – oder extrem neuer Hardware kann es (temporäre) Schwierigkeiten geben.



Ernstgemeinte Computergrafik ist heute 3D-Grafik, auch wenn das Dargestellte überhaupt nicht dreidimensional aussieht. Im Idealfall können Anwendungsprogramme mit etwas Schützenhilfe von X.org direkt die spezielle 3D-Hardware des Grafikchips ansprechen und in rasender Geschwindigkeit erstaunliche und wundersame Dinge auf den Bildschirm bringen. Im Nicht-Idealfall werden Teile der 3D-Grafik von der CPU berechnet, und das ist für »Büroanwendungen« immer noch schnell genug. Früher hatten Grafikchips Hardware-Beschleunigung für 2D-Grafik, aber da kein Betriebssystem heute noch 2D-Grafik benutzt, machen die Hersteller sich heutzutage nicht mehr die Mühe.



Für den Idealfall brauchen Sie einen Kernel-Treiber für Ihren Grafikchip und einen weiteren Treiber für X.org. Der erstere kümmert sich um die Grundkonfiguration der Grafik und um Basisoperationen, der letztere um die Operationen des X-Protokolls.



Grob gesagt wird Intel-Grafikhardware von Linux und X.org am besten unterstützt (was damit zu tun haben kann, dass einige wesentliche X-Entwickler für Intel arbeiten). Allerdings spielt sie, was die 3D-Leistung angeht, nicht in derselben Liga wie die besten Grafikchips der großen Hersteller AMD und nVidia. Für letztere gibt es sowohl frei verfügbare Treiber als auch von den Herstellern selbst vertriebene »proprietäre« Treiber, die

die Hardware besser ausnutzen, aber nicht im Quellcode zur Verfügung sehen.



X11 hat keine 3D-Grafikprimitive, und so beschränkt sich die »Schützenhilfe«, die X.org liefern kann, bei 3D-Clients in der Regel darauf, dem Client ein Stück Speicher auf der Grafikkarte zur Verfügung zu stellen, in das er mit direkten 3D-Operationen – typischerweise mit einer Grafiksprache wie OpenGL – seine Grafikausgabe zeichnet. Ein spezieller X11-Client, der »Compositor«, kümmert sich dann darum, die Grafikausgabe verschiedener Clients in der richtigen Reihenfolge »aufzustapeln« und mit Effekten wie Durchsichtigkeit, Schattenwurf usw. zu verbrämen.



Wayland

Der neueste Trend bei Linux-Grafik beruht auf der Beobachtung, dass der X-Server heute außer einem bisschen Schützenhilfe nichts Nützliches mehr tut: Der Compositor und die Clients machen die ganze eigentliche Arbeit. Was liegt also näher, als den X-Server komplett abzuschaffen? Die künftige Infrastruktur – Wayland – tut im Grunde genau das. Wayland implementiert einen Compositor, der direkt mit dem Grafikchip reden kann. Dadurch ist der X-Server im Wesentlichen überflüssig. Voraussetzung ist, dass die Clients für ihre Grafikausgabe statt X11-Operationen 3D-Operationen (in OpenGL) generieren, aber das lässt sich einigermaßen bequem erreichen, indem man die »Toolkits«, also die Programmierumgebungen für Clients, entsprechend anpasst. Die gängigen Toolkits enthalten heute bereits mehr oder weniger experimentelle Wayland-Unterstützung.

Die logische Trennung von X-Server und -Clients durch das X-Protokoll erlaubt es, dass Server und Clients auf verschiedenen Rechnern laufen. Grundsätzlich können auf einem einzigen Rechner diverse Clients gestartet werden, die jeweils mit anderen X-Servern kommunizieren¹. Die interessante Frage, die sich hier stellt, ist, wie ein Client herausfindet, mit welchem Server er reden soll, und die Antwort lautet »Über die Umgebungsvariable DISPLAY«. Um den X-Server auf dem Rechner `red.example.com` anzusprechen, müssen Sie

```
DISPLAY=red.example.com:0
```

Displaynamen setzen. Das »:0« am Ende bezieht sich dabei auf den ersten X-Server auf diesem Rechner. Etwas wie `red.example.com:0` nennt man auch einen **Displaynamen**.



Im Prinzip hält Sie niemand davon ab, auf demselben Rechner mehr als einen X-Server laufen zu lassen. Es gibt heute relativ billige »Port-Extender«, die Anschlüsse für eine Tastatur, eine Maus und einen Monitor haben und über USB mit einem PC verbunden werden. Sie können so auf einem normalen PC mehr als einen Arbeitsplatz haben, was zum Beispiel für die gängigen Büroanwendungen überhaupt kein Problem darstellt. In so einem Fall wird pro Arbeitsplatz ein X-Server konfiguriert.



Die weiteren X-Server auf dem Rechner `red.example.com` heißen dann entsprechend `red.example.com:1`, `red.example.com:2`, ...



Wenn Sie einen X-Server in dieser Form ansprechen, heißt das, dass Sie über TCP/IP mit ihm reden wollen. (Wenn die Zahl hinter dem Doppelpunkt n ist, dann versucht der Client, mit dem TCP-Port $6000 + n$ auf dem betreffenden Rechner Kontakt aufzunehmen. Dort lauscht dann der entsprechende

¹In den 1990er Jahren gab es das Konzept sogenannter »X-Terminals« – also spezialisierten Rechnern, auf denen außer einem X-Server nicht allzuviel lief und deren einziger Lebenszweck darin bestand, die Eingabe und Ausgabe von Clients auf einem (oder mehreren) Zentralrechner(n) zu verwalten, so wie man zuvor Textterminals hatte, die die Eingabe und Ausgabe von Programmen auf einem zentralen Rechner verwalteten. Irgendwann zeigte sich aber, dass PCs mit Linux und X11 wesentlich leistungsfähiger und flexibler waren (von »billiger« ganz zu schweigen), und X-Terminals (typischerweise in Gestalt von PCs mit Linux) sind heute auf Nischenanwendungen beschränkt.

X-Server auf Verbindungen.) Möchten Sie die Verbindung lieber über ein Unix-Domain-Socket herstellen – was unbedingt zu empfehlen ist, wenn Server und Client auf demselben Rechner laufen –, dann verwenden Sie die Namen `unix:0`, `unix:1` usw.



Sie können auch einfach nur `:0`, `:1`, ... benutzen. Das ist gleichbedeutend mit »Wähle die schnellstmögliche lokale Verbindung«. Bei Linux läuft das in der Regel auf das Unix-Domain-Socket heraus, aber andere Unix-Betriebssysteme unterstützen weitere Transportmechanismen, etwa Shared Memory.



Prinzipiell können Sie außer einem X-Server auf einem Rechner auch noch gezielt einen Bildschirm ansprechen, der von diesem X-Server kontrolliert wird, indem Sie einen Punkt und die Bildschirmnummer anhängen, also zum Beispiel `red.example.com:0.0` für den ersten Bildschirm des ersten X-Servers auf `red`. Wir erwähnen das hier nur der Vollständigkeit halber, denn selbst wenn Sie mehrere Monitore an einen Linux-Rechner anschließen, dann funktionieren diese normalerweise als ein großer »logischer« Bildschirm, was viel bequemer ist, wenn Sie tatsächlich davorsitzen – aber keine individuelle Adressierung der einzelnen Monitore zulässt.



Der Vorteil des großen logischen Bildschirms ist, dass Sie Fenster von einem Monitor auf den anderen verschieben können. Fenster können auch zur Hälfte auf einem Monitor liegen und zur Hälfte auf einem anderen. Normalerweise ist das heutzutage das, was man will. Bei separaten Bildschirmen ist es möglich, die Bildschirme verschieden anzusteuern (etwa einen als Farb- und einen als Schwarzweißmonitor, früher, als es noch Schwarzweißmonitore gab), aber dann müssen Sie sich beim Programmstart entscheiden, auf welchem der »Screens« Ihr Programm erscheinen soll, denn nachträgliches Verschieben ist nicht erlaubt.

Dass Sie über einen geeigneten Displaynamen die X-Server auf beliebigen entfernten Rechnern adressieren können, heißt noch lange nicht, dass diese X-Server tatsächlich mit Ihren Clients reden wollen – in der Tat wäre es eine gravierende Sicherheitslücke, wenn Sie beliebigen Clients (nicht bloß Ihren eigenen) Zugriff auf Ihren X-Server geben würden². Das bedeutet zum einen, dass es eine rudimentäre Zugangskontrolle gibt (siehe Abschnitt 11.6) und zum anderen, dass viele X-Server überhaupt nicht mehr auf direkte TCP-Verbindungen lauschen. Denn da ansonsten jeder, der den Datenverkehr zwischen Client und Server verfolgen kann, sich anschauen kann, was der Client auf den Bildschirm zeichnet (er muss ja nur die X11-Protokollnachrichten interpretieren), zieht man es heute meist vor, nur lokale Verbindungen zuzulassen und den Fernzugriff durch »X11 Forwarding« über die Secure Shell zu realisieren. Dabei wird das X-Protokoll von der Secure Shell verschlüsselt, so dass Mithörer nicht auf die Grafikausgabe (und die Maus- und Tastatureingaben des Benutzers) rückschließen können.

Sicherheitslücke

X11 Forwarding



Mehr über die Secure Shell und X11 Forwarding erfahren Sie in der Linup-Front-Schulungsunterlage *Linux-Administration II*.

Außer über die Umgebungsvariable `DISPLAY` können Sie den Server bei den meisten Clients auch auf der Kommandozeile mit einer Option wie »-display `red.example.com:0`« auswählen. Mit der Umgebungsvariablen ist es aber bequemer:

```
$ xclock -display red.example.com:0           Anzeige auf red
$ DISPLAY=red.example.com:0 xclock           ... dasselbe
$ export DISPLAY=red.example.com:0

Alle ab jetzt gestarteten X-Clients zeigen auf red an
```

²Ein feindlicher X-Client könnte zum Beispiel ein durchsichtiges Fenster über den kompletten Bildschirm legen, alle Ihre Tastendrucke abfangen und nach Kennwörtern, Kreditkartennummern und ähnlichem suchen. Oder er könnte Tausende von Fenstern öffnen und nervtötend piepen, ohne aufzuhören.

Wenn Sie sich in einer grafischen Umgebung anmelden, dann sollte diese Variable ohne Ihr Zutun korrekt gesetzt werden. Sie müssen sich also außer in Ausnahmefällen nicht selber darum kümmern.

Hier noch schnell ein paar Begriffsdefinitionen aus dem X11-Umfeld:

Fenstermanager (*window manager*) Ein spezieller X11-Client, dessen Aufgabe es ist, über die Platzierung (Position und Vordergrund/Hintergrund) der Fenster auf dem Bildschirm zu entscheiden. Clients dürfen zwar Vorschläge machen, aber der Fenstermanager (und damit der Benutzer) hat das letzte Wort. Es gibt diverse Fenstermanager mit unterschiedlichen Philosophien, etwa was überlappende Fenster angeht – manche Benutzer bevorzugen Fenstermanager, die den Bildschirm mit Fenstern »kacheln« und Überlappungen entweder vermeiden oder gar verbieten. Viele Fenstermanager fungieren heute auch als Compositor für 3D-Grafik.

Sitzungsverwaltung

Displaymanager Ein Programm, das sich um die Verwaltung des X-Servers (oder der X-Server) auf einem Rechner kümmert. Der Displaymanager erlaubt, dass ein Benutzer sich in einer grafischen Umgebung anmeldet, und baut anschließend eine grafische Sitzung für ihn auf. Viele Displaymanager unterstützen auch Sitzungsverwaltung (engl. *session management*), das heißt, sie versuchen beim Abmelden den aktuellen Zustand der Sitzung zu speichern und beim Wiederanmelden dann wieder herzustellen.



Wie gut Sitzungsverwaltung in der Praxis funktioniert, hängt davon ab, wie konsequent die X11-Clients dabei mitspielen. Nicht alle Clients können das wirklich gut, und fairerweise müssen wir anmerken, dass manche Clients es wirklich mit einem komplexen inneren Zustand zu tun haben, der nicht leicht zu speichern und wieder in Kraft zu setzen ist.

Toolkit Eine Programmierumgebung für X-Clients. Toolkits stellen Programmierern Funktionalität zur Verfügung, um die Grafikausgabe eines Programms und dessen Umgang mit Ereignissen wie Mausclicks, Tastendrücken und ähnlichem zu beschreiben. Diese Funktionalität wird dann vom Toolkit auf X11-Protokollnachrichten abgebildet. Der Hauptvorteil von Toolkits ist, dass Sie als Programmierer sich nicht mit den sehr primitiven X11-Operationen abmühen müssen, sondern Code schreiben können, der in einer höheren Programmiersprache bequem ist. Es gibt diverse Toolkits, die teils für bestimmte Programmiersprachen (etwa C oder C++) optimiert sind und teils auch die Programmierung in anderen Sprachen (etwa Python) zulassen.

KDE
GNOME

Arbeitsumgebung (*desktop environment*) Als X11 neu war, ging es den Entwicklern vorrangig darum, die technischen Möglichkeiten zur Verfügung zu stellen, die für grafische Anwendungen gebraucht werden. Es war ihnen weniger wichtig, Spielregeln zu definieren, wie diese Anwendungen aussehen und sich benehmen sollten. Über die Jahre entwickelte sich das zu einem Problem, weil fast jedes größere Programm seine eigenen Konventionen hatte³. Arbeitsumgebungen wie KDE oder GNOME setzen auf X11 auf und versuchen, (jeweils) einheitliche Standards für Aussehen und Bedienung einer Vielzahl nützlicher Programme zu definieren und Programme anzubieten, die diese Standards tatsächlich umsetzen und eine angenehme und konsistente *user experience* zu schaffen.



Die meisten Linux-Distributionen bieten eine bestimmte Arbeitsumgebung als voreingestellten Standard an und erlauben oft, alternativ eine andere zu installieren. Sie können dann beim Anmelden wählen, in welcher Umgebung Sie arbeiten können.

³Und ja auch die Konkurrenz nicht schlief – Apple und Microsoft waren weitaus konsequenter darin, einheitliches *look and feel* für Programme auf ihren Plattformen zu fordern.



Die Arbeitsumgebungen schließen einander auch nicht gegenseitig aus. Wenn Sie in Umgebung X arbeiten, aber ein nettes Programm verwenden wollen, das für Umgebung Y geschrieben ist, dann steht dem in der Regel nichts im Weg – Sie müssen allenfalls in Kauf nehmen, dass Laufzeitbibliotheken von Y geladen werden müssen und das zusätzlichen Speicher kostet. Für diverse grundlegende Funktionalität wie zum Beispiel das Ausschneiden und Einkleben von Textstücken gibt es übergreifende Standards, die von allen Arbeitsumgebungen unterstützt werden und das »Mischen« erleichtern.



Die meisten Arbeitsumgebungen basieren auf bestimmten Toolkits – KDE zum Beispiel auf Qt, GNOME auf Gtk+. Das heißt, wenn Sie Software entwickeln wollen, die für eine bestimmte Arbeitsumgebung gedacht ist, dann sollten Sie das betreffende Toolkit verwenden, um eine optimale Integration sicherzustellen.

Übungen



11.1 [!1] Wenn Sie in einer grafischen Umgebung arbeiten: Wie lautet Ihr aktueller Displayname?



11.2 [1] Versuchen Sie, einen Client über TCP/IP mit Ihrem X-Server zu verbinden, indem Sie einen Displaynamen wie `red.example.com:0` verwenden. Funktioniert das?



11.3 [1] Wofür steht der Displayname `bla.example.com:1.1`? Geben Sie eine Kommandozeile an, die das Programm `xterm` mit Anzeige auf diesem Display startet.

11.2 X11 konfigurieren

Wenn Sie herausfinden wollen, ob Ihr Grafiksystem von Linux und X.org unterstützt wird, ist es das Einfachste, das System mit einem geeigneten (möglichst aktuellen) Live-Linux wie Knoppix zu starten und zu schauen, was passiert. Wenn Sie eine grafische Oberfläche bekommen, ist alles in Ordnung.



Früher konnte es an schwarze Magie grenzen, X11 auf einem Linux-Rechner zum Laufen zu bringen. Es war nicht ungewöhnlich, Details über die verwendete Grafikkarte und detaillierte Steuerungsparameter für den Bildschirm mit der Hand in eine Textdatei eintragen zu müssen (und zumindest bei den einst üblichen Festfrequenz-Monitoren bestand durchaus die Möglichkeit, bei Fehlern den Bildschirm irreparabel zu beschädigen). Zum Glück können aktuelle Versionen von X.org selbst herausfinden, mit welcher Hardware sie es sowohl im Rechner als auch als Bildschirm zu tun haben und wie die optimalen Parameter aussehen. Sie können zwar immer noch manuell eingreifen, wenn Sie möchten, aber das ist nur noch in Ausnahmefällen nötig.

Sollte der X-Server nicht korrekt gestartet werden, dann ist Ihre erste Anlaufstelle dessen Protokolldatei, die üblicherweise in `/var/log/Xorg.0.log` zu finden ist. Dort vermerkt der X-Server sehr ausführlich, welche Hardware er erkennt und welche Treiber und Einstellungen er ihr zuordnet.

Grundsätzlich können Sie als Systemadministrator den X-Server mit dem Kommando

```
# Xorg -configure
```

starten. Der X-Server nimmt dann eine Hardwareerkennung vor und schreibt deren Ergebnis als rudimentäre Konfiguration in die Datei `/etc/X11/xorg.conf`. Sie können diese Datei dann als Ausgangspunkt für eine eigene Konfiguration verwenden.



Statt `xorg` können Sie auch das Kommando `x` verwenden. Nach Konvention ist das eine Abkürzung für »den passenden X11-Server für dieses System«.

Syntax Abschnitte `xorg.conf` ist die zentrale Konfigurationsdatei für den X-Server. Sie besteht aus einzelnen Abschnitten, die jeweils durch das Schlüsselwort `Section` eingeleitet und mit `EndSection` abgeschlossen. Einige der Abschnitte, namentlich diejenigen, die Ein- und Ausgabegeräte definieren, können mehrfach auftreten, damit Sie zum Beispiel Maus und Touchpad parallel betreiben können. In der Konfigurationsdatei sind Groß- und Kleinschreibung außer in Dateinamen nicht relevant.



Wie üblich werden Leerzeilen in der Konfiguration ignoriert, genau wie Kommentarzeilen, die mit dem traditionellen Lattenzaun (`»#«`) anfangen. Zeilen mit Einstellungen dürfen eingerückt werden, um die Dateistruktur besser hervorzuheben.

Files Hier ist ein Überblick über die wichtigsten Abschnitte in `xorg.conf`:
Der `Files`-Abschnitt definiert Pfade, nämlich:

FontPath benennt Verzeichnisse, in denen der X-Server nach Schriften sucht, oder einen Schriftenserver (engl. *font server*, Abschnitt 11.5). Meist gibt es sehr viele Schriftenverzeichnisse und damit viele `FontPath`-Parameter. Die Reihenfolge ist wichtig!

ModulePath legt Verzeichnisse fest, in denen Erweiterungsmodule für X.Org liegen (typischerweise `/usr/lib/xorg/modules`).

RGBPath wurde früher benutzt, um eine Datei zu benennen, in der alle dem X-Server bekannten Farbnamen mit den zugehörigen RGB-Werten (Rot, Grün, Blau) stehen. Der gängige Name ist `/etc/X11/rgb.txt`, und verwirrenderweise ist die Datei in vielen Linux-Distributionen noch enthalten, vermutlich damit Sie die gültigen Farbnamen nachschlagen können. Die Farbnamen können Sie dort verwenden, wo X11-Clients Ihnen die Angabe von Farben erlauben:

```
$ xterm -fg GoldenRod -bg NavyBlue
```



Wenn Ihre gewünschte Farbe nicht namentlich in der Datei steht, können Sie sie auch explizit in Hexadezimalzahlen angeben. `GoldenRod` ist zum Beispiel `#daa520` (die Werte für Rot, Grün und Blau sind respektive 218, 165 und 32). Das vorgesetzte `»#«` zeigt an, dass es sich um eine RGB-Farbe handelt.



Die fantasievollen Namen wie `GoldenRod`, `PeachPuff` oder `MistyRose` stammen aus dem 72-Farben-Crayola-Wachsmalkreidensatz, den der frühe X11-Entwickler John C. Thomas zur Hand hatte.



Im Prinzip hält Sie niemand davon ab, in der Datei auch Ihre eigenen Lieblingsfarben zu verewigen. Sie sollten keine der vorhandenen Farben entfernen oder zu radikal ändern, da es sein kann, dass sich manche Programme auf ihre Existenz und ihr Aussehen verlassen. Außerdem ist es recht wahrscheinlich, dass Ihr X-Server die Datei sowieso nicht anschaut, da die Unterstützung für `RGBPath` standardmäßig in `X.org` nicht mehr enthalten ist.

Hier ein stark verkürztes Beispiel:

```
Section "Files"
  ModulePath  "/usr/lib/xorg/modules"
  # RGBPath   "/usr/share/X11/rgb.txt" # nicht mehr unterstützt
  FontPath    "/usr/share/fonts/X11/misc:unscaled"
  <<<<<<
EndSection
```

Sowohl FontPath als auch ModulePath dürfen in der Datei mehrmals auftreten; ihre Werte werden dann in der Reihenfolge des Auftretens aneinandergehängt.

Der Module-Abschnitt gibt an, welche grafikartenunabhängigen X-Server-Module geladen werden sollen, etwa zur Unterstützung bestimmter Schriftformate (Type1, TTF, ...) oder für besondere Eigenschaften wie direkte Videodarstellung. Module

```
Section "Module"
  Load "dri"
  Load "v4l"
EndSection
```

Welche Module zur Verfügung stehen, hängt vom X-Server ab; normalerweise muss die Liste nicht geändert werden. Sie kann auch ganz fehlen, dann holt sich der X-Server die Module, die er braucht.



Gesucht werden Module in den mit ModulePath angegebenen Dateien sowie deren Unterverzeichnissen drivers, extensions, input, internal, und multimedia (falls vorhanden).



Die Module extmod, dbe, dri, dri2, glx und record werden automatisch geladen, wenn sie existieren. Ist das nicht gewünscht, müssen Sie sie mit Direktiven wie

```
Disable "dbe"
```

abschalten. Zumindest extmod sollten Sie aber auf jeden Fall laden.

Im Extensions-Abschnitt können Sie angeben, welche X11-Protokollerweiterungen ein- oder ausgeschaltet werden sollen: Extensions

```
Section "Extensions"
  Option "MIT-SHM" "Enable"
  Option "Xinerama" "Disable"
EndSection
```

Die Namen von Erweiterungen müssen in der korrekten Groß- und Kleinschreibung angegeben werden. Eine Liste der Erweiterungen bekommen Sie mit einem Kommando wie

```
$ sudo Xorg -extension ?
```

Es gibt eigentlich keinen speziellen Grund, bestimmte Erweiterungen auszuschalten (außer möglicherweise Kompatibilitätstests). Der Server nutzt die, die von Clients verwendet werden, und lässt die anderen links liegen.

Der ServerFlags-Abschnitt beeinflusst das Verhalten des X-Servers. Einzelne Optionen werden mit dem Parameter Option gesetzt und können im ServerLayout-Abschnitt oder beim Start des Servers auf der Kommandozeile überschrieben werden. Das Ganze sieht ungefähr so aus: ServerFlags

```
Section "ServerFlags"
  Option "BlankTime" "10"
                                     Bildschirmschoner nach 10 Minuten
EndSection
```

Einige wichtige Server-Flags sind:

AutoAddDevices Gibt an, ob Tastaturen, Mäuse und ähnliche Eingabegeräte automatisch mithilfe von udev erkannt werden. Standardmäßig eingeschaltet.

DefaultServerLayout Gibt an, welche Anordnung von Grafikkarten, Bildschirmen, Tastaturen, Mäusen, ... standardmäßig verwendet wird. Verweist auf einen ServerLayout-Abschnitt. Andere Server-Layouts können über die Kommandozeile ausgewählt werden.

DontZap Wenn diese Option eingeschaltet ist (der Normalfall), kann der Server *nicht* mit der Tastenkombination  +  +  beendet werden.

Schalter Die meisten Optionen sind Schalter, die Werte wie 1, true, yes oder on bzw. 0, false, no oder off annehmen können. Wenn Sie keinen Wert angeben, wird true angenommen. Sie können vor den Optionsnamen auch ein No setzen, das heißt dann »nein«:

Option "AutoAddDevices" "1"	<i>Option einschalten</i>
Option "AutoAddDevices" "off"	<i>Option ausschalten</i>
Option "AutoAddDevices"	<i>Option einschalten</i>
Option "NoAutoAddDevices"	<i>Option ausschalten</i>

Andere Optionen haben Werte, die zum Beispiel ganzzahlig oder Zeichenketten sein dürfen. Alle Werte müssen in Anführungszeichen stehen.

InputDevice Der InputDevice-Abschnitt konfiguriert jeweils ein Eingabegerät wie Maus oder Tastatur und kann mehrmals vorkommen. Im Folgenden ein kommentiertes Beispiel:

```
Section "InputDevice"
  Identifier "Tastatur1"
  Driver "Keyboard"
  Option "XkbLayout" "de"
  Option "XkbModel" "pc105"
  Option "XkbVariant" "nodeadkeys"
EndSection
```

Ein typischer Eintrag für eine moderne PC-Tastatur. Die einzelnen Einträge haben die folgende Bedeutung:

Driver lädt ein Modul („Treiber“) aus dem ModulePath.

Identifier gibt dem Abschnitt einen Namen, damit er in einem ServerLayout-Abschnitt erwähnt werden kann.

XkbLayout legt das Tastaturlayout auf Deutsch fest.

XkbModel definiert eine Standard-PC-Tastatur mit 105 Tasten.

XkbVariant Der Wert deadkeys erlaubt es, Zeichen mit Akzenten aus mehreren Eingaben zusammensetzen, also etwa »ñ« als   einzugeben. Mit nodeadkeys liefert   »~n«.



Wenn das Server-Flag AutoAddDevices gesetzt ist (der Normalfall), brauchen Sie eigentlich gar keine InputDevice-Abschnitte, da der X-Server die Eingabegeräte selbstständig erkennt. Eine detailliertere Konfiguration ist nur nötig, wenn der X-Server zum Beispiel nicht alle verfügbaren Eingabegeräte tatsächlich benutzen soll.



In alten Konfigurationsdateien finden Sie manchmal noch die Abschnitte Keyboard oder Pointer. Diese Namen sind verpönt; benutzen Sie lieber InputDevice.

Der Monitor-Abschnitt beschreibt die Eigenschaften des verwendeten Monitors. Monitor
Auch dieser Abschnitt kann mehrmals auftreten.

```
Section "Monitor"
  Identifier "Monitor0"
  HorizSync 30-90
  VertRefresh 50-85
EndSection
```

Wie InputDevice muss dieser Abschnitt einen Identifier haben, mit dem er in Server-Layout-Abschnitten aufgerufen wird. Die horizontalen und vertikalen Frequenzen können Sie der Dokumentation des Monitors entnehmen.



Der optionale Modes-Abschnitt (den es auch mehrmals geben darf) erlaubt es Ihnen, detailliert die Ansteuerung Ihres Bildschirms zu konfigurieren. Modes
Unser Tipp ist es, mit der dafür nötigen Zeit und Energie lieber zielführende Dinge in Angriff zu nehmen, wenn Sie es irgendwie einrichten können, da heutige Hardware die erforderlichen Einstellungen auch ohne Weiteres alleine herausfinden kann. Trotzdem hier ein Beispiel:

```
Section "Monitor"
  <<<<<<
  UseModes "Modus1"
  <<<<<<
EndSection

Section "Modes"
  Identifier "Modus1"
  Modeline "800x600" 48.67 800 816 928 1072 600 600 610 626
  Modeline "640x480" <<<<<<
  <<<<<<
EndSection
```

(Mit UseModes in Monitor benennen Sie den zu verwendenden Modes-Abschnitt.)
Sie können Modeline-Einträge auch direkt im Monitor-Abschnitt platzieren oder dort oder im Modes-Abschnitt die etwas weniger kompakten Mode-Unterabschnitte verwenden.



Wenn Sie dringend wissen müssen, was die magischen Zahlen in den Videomodi bedeuten, dann lesen Sie in `xorg.conf(5)` nach.

Der Device-Abschnitt legt fest, welche Grafikkarte der X-Server verwenden soll. Device
Auch dieser Abschnitt kann mehrmals auftreten. Ein Beispiel für eine Minimal-konfiguration mit VGA-Treiber:

```
Section "Device"
  Identifier "Standard VGA"
  Driver "vga"
EndSection
```

Ein Beispiel für eine nVidia-Grafikkarte mit dem proprietären Treiber:

```
Section "Device"
  Identifier "Device0"
  Driver "nvidia"
  VendorName "NVIDIA Corporation"
  BoardName "NVS 3100M"
EndSection
```



Wenn das System mehrere Grafikkarten hat, sollten Sie, um Verwirrung zu vermeiden, mit der BusID-Direktive die PCI-Adresse der Grafikkarte angeben, für die der Abschnitt gilt. Die korrekte PCI-Adresse können Sie zum Beispiel mit `lspci` herausfinden:

```
# lspci | grep "VGA compatible"
01:00.0 VGA compatible controller: NVIDIA Corporation GT218M
```

Screen Der Screen-Abschnitt verknüpft jeweils einen Monitor und eine Grafikkarte:

```
Section "Screen"
  Identifier   "Screen0"
  Device      "Device0"
  Monitor     "Monitor0"
  DefaultDepth 24
  SubSection "Display"
    Depth     24
    Modes     "1280x720"
  EndSubSection
  SubSection "Display"
    Depth     24
    Modes     "1600x900"
  EndSubSection
EndSection
```

Die Unterabschnitte namens `Display` legen verschiedene Kombinationen aus Farbtiefe und Auflösung fest, zwischen denen im laufenden System mit `Strg` + `Alt` + `+` bzw. `Strg` + `Alt` + `-` hin- und hergeschaltet werden kann.

Möglicherweise haben Sie auch einen `DRI`-Abschnitt, in dem Einstellungen für den direkten Zugriff des X-Servers auf die Grafikhardware gemacht werden.

ServerLayout Der `ServerLayout`-Abschnitt beschreibt die Gesamtkonfiguration des Servers mit Ein- und Ausgabegeräten. Hier würde zum Beispiel auch die Anordnung mehrerer Bildschirme angegeben werden:

```
Section "ServerLayout"
  Identifier   "Layout0"
  Screen      0 "Screen0" 0 0
  InputDevice "Keyboard0" "CoreKeyboard"
  InputDevice "Maus0" "CorePointer"
  Option      "Xinerama" "0"
EndSection
```



Sie brauchen eine `Screen`-Direktive für jeden Bildschirm, den Sie verwenden. Die erste Null ist die Bildschirmnummer, die von Null an fortlaufend vergeben werden muss (eigentlich kann sie auch wegfallen, dann werden die Bildschirme in der Reihenfolge des Auftretens nummeriert). Nach dem Namen eines `Screen`-Abschnitts, der anderswo in der Datei stehen muss, kommt noch eine Positionsangabe, wobei `>0 0<` nur bedeutet, dass die linke obere Ecke des Bildschirms der Koordinate (0, 0) entsprechen soll. X11-Koordinaten zählen nach rechts und unten.



Ist die Option `Xinerama` eingeschaltet, werden alle Screens als Ausschnitte eines logischen Bildschirms betrachtet, dessen Gesamtausdehnung so groß ist, dass die Screens hineinpassen. Die einzelnen Screens müssen so konfiguriert sein, dass sie dieselbe Farbtiefe haben (heute typischerweise 24 Bit); die Auflösung muss nicht dieselbe sein. Zum Beispiel könnten Sie einen primären Bildschirm mit 1920 mal 1080 Bildpunkten haben und außerdem einen Beamer mit 1024 mal 768 Bildpunkten anschließen. Mit etwas wie

```
Screen 0 "LaptopDisplay" 0 0
Screen 1 "Beamer" RightOf "LaptopDisplay"
Option "Xinerama" "Enable"
```

haben Sie anschließend einen »logischen« Bildschirm mit 2944 Bildpunkten Breite und 1080 Bildpunkten Höhe, wobei die beiden Bildschirme an der Oberkante aneinander ausgerichtet sind.



Im Beispiel aus dem vorigen Absatz gibt es einen »toten« Streifen von 1024 mal 312 Bildpunkten, der zwar theoretisch vorhanden ist (X11 kann nur mit rechteckigen Bildschirmen umgehen, egal ob physikalisch oder logisch), aber nicht wirklich genutzt werden kann. Insbesondere dürfen neue Fenster nicht automatisch komplett im toten Streifen platziert werden, weil Sie sie von dort nicht in einen sichtbaren Teil des Bildschirms schleppen können. In so einer Situation sollten Sie daher unbedingt einen Fenster-Manager verwenden, der sich mit `Xinerama` verträgt und darauf achtet, dass so etwas nicht vorkommt.



Natürlich können sich die Bildschirme aus den `Screen`-Zeilen einander auch überlappen (es ist zum Beispiel manchmal nützlich, wenn ein Beamer für eine Präsentation die linke obere Ecke des Notebook-Bildschirms zeigt). Geben Sie dafür die Position des Zusatzbildschirms am besten in absoluten Zahlen an:

```
Screen 0 "LaptopDisplay" 0 0
Screen 1 "Beamer" 64 0
Option "Xinerama" "Enable"
```

Platz lassen für Kontrollleiste links



Sie können mehrere `ServerLayout`-Abschnitte in Ihrer Konfigurationsdatei haben und sich über die Kommandozeilenoption `-layout` beim (direkten) Aufruf des X-Servers eine davon aussuchen. Das ist beispielsweise nützlich, um auf einem Notebook-Rechner verschiedene Konfigurationen für den externen Videoausgang vorzuhalten.

In `ServerLayout`-Abschnitten können Sie auch Optionen aus dem `ServerFlags`-Abschnitt unterbringen. Diese Optionen gelten dann nur für die betreffende Konfiguration.

Hier eine kurze Zusammenfassung der Konfigurationsdatei: Die `ServerLayout`-Abschnitte stehen auf der höchsten Ebene. In ihnen werden die Eingabe- und Ausgabegeräte benannt, die zu einer Konfiguration gehören; sie beziehen sich auf `InputDevice`- und `Screen`-Abschnitte anderswo in der Datei. Ein `Screen` besteht dabei aus einer Grafikkarte (`Device`) und einem zugeordneten Monitor (`Monitor`). Auf den X-Server im Ganzen beziehen sich die Abschnitte `Files`, `ServerFlags`, `Module` und `DRI`.

Zusammenfassung

Übungen



11.4 [!1] Schauen Sie in die `X.org`-Konfigurationsdatei auf Ihrem System (falls es überhaupt eine gibt). Wurde sie manuell angelegt oder von `X.org` erzeugt? Welche Geräte sind dort definiert? Welche `Server-Flags` werden gesetzt?



11.5 [2] Über das `X`-Protokoll werden Grafikbefehle und Ereignisse transportiert, die eine Bildschirmdarstellung auf einem beliebigen über das Netz mit dem `X-Client`programm verbundenen `X-Server` gestatten. Vergleichen Sie diesen Ansatz mit dem ebenfalls populären Verfahren, direkt Bildschirmhalte hin und her zu kopieren, wie das zum Beispiel bei `VNC` und ähnlichen Produkten gemacht wird. Wo liegen die Vorteile, wo die Nachteile der beiden Methoden?

11.3 Displaymanager

11.3.1 Grundlegendes zum Starten von X

Grundsätzlich können Sie den X-Server auf Ihrem Rechner starten, indem Sie einfach (auf einer Textkonsole) das Kommando

```
$ X
```

ausführen. (Der X-Server muss meistens, o Graus, als root laufen, aber X kümmert sich darum.) Anschließend können Sie zum Beispiel – ebenfalls auf einer Textkonsole – mit einem Kommando wie

```
$ xterm -display :1
```

einen grafischen Terminal-Emulator starten. Von diesem xterm aus ist es möglich, weitere X-Clients ohne expliziten Displaynamen zu starten, da die im xterm laufende Shell eine passende DISPLAY-Variable hat.



Im wirklichen Leben sollten Sie diese Methode nicht verwenden – einerseits ist sie furchtbar unbequem, und andererseits nehmen die im folgenden gezeigten Ansätze Ihnen einige Arbeit ab, was die Konfiguration und Sicherheit des ganzen angeht.

Eine bequemere Methode zum Start von X aus einer Textsitzung ist die Verwendung des Kommandos `startx`. `startx` macht einige Initialisierungen und ruft dann ein anderes Programm namens `xinit` auf, das die eigentliche Arbeit erledigt – es kümmert sich um den Start des X-Servers und der ersten X-Clients.

Über die Dateien `~/.xinitrc` und `/etc/X11/xinit/xinitrc` können Sie X-Clients starten, beispielsweise eine Uhr, einen Terminalemulator und einen Fenstermanager. Alle X-Clients müssen mit einem `>&<` am Zeilenende im Hintergrund gestartet werden, lediglich der letzte X-Client – in aller Regel der Fenstermanager – muss im Vordergrund gestartet werden. Wird dieser letzte Prozess beendet, beendet `xinit` den X-Server. Hier ein einfaches Beispiel für eine `~/.xinitrc`-Datei:

```
# Ein Terminal
xterm -geometry 80x40+10+10 &
# Eine Uhr
xclock -update 1 -geometry -5-5 &
# Der Fenstermanager
fvwm2
```

Mit `-geometry` können Sie die Position der sich öffnenden Fenster im Vorhinein festlegen.



Der Wert von `-geometry` besteht aus einer optionalen Größenangabe (zumeist in Pixeln, bei manchen Programmen, etwa `xterm`, aber in Zeichen) gefolgt von einer ebenfalls optionalen Positionsangabe (aber eine von beiden sollten Sie schon haben). Die Positionsangabe besteht aus einer `x`- und einer `y`-Koordinate, wobei positive Zahlen vom linken bzw. oberen Bildschirmrand und negative Zahlen vom rechten bzw. unteren Bildschirmrand aus zählen.

Sie können beim Aufruf von `startx` auch eine Servernummer angeben:

```
$ startx -- :1
```

Damit können Sie zum Beispiel einen zweiten X-Server starten.

Übungen



11.6 [2] Wie würden Sie eine `xclock` so starten, dass sie 150 auf 150 Pixel groß ist und 50 Pixel vom Bildschirmrand entfernt in der linken unteren Ecke des Schirms erscheint?



11.7 [2] Versuchen Sie, mit dem Kommando »`startx -- :1`« einen weiteren X-Server zu starten. (Dieser sollte sich auf der virtuellen Konsole `tty8` manifestieren, also über `Strg` + `Alt` + `F8` zu erreichen sein.)

11.3.2 Der Displaymanager LightDM

Auf modernen Arbeitsplatzrechnern ist es üblich, die grafische Oberfläche über einen Displaymanager schon beim Systemstart hochzufahren. Gängige Distributionen bieten mehrere Displaymanager an; welcher davon gestartet wird, hängt von einem distributionsspezifischen Auswahlmechanismus ab.



Seit Version 4.0 der LPIC-1-Zertifizierung wird besonderen Wert auf den Displaymanager LightDM gelegt, den wir im Folgenden etwas ausführlicher erklären. Die anderen Displaymanager (`xdm`, `kdm`, `gdm`) müssen Sie nur noch dem Namen nach kennen.

LightDM ist ein populärer Displaymanager, der weitgehend von konkreten Arbeitsumgebungen unabhängig ist. Wie sein Name andeutet, ist er relativ ressourcenschonend, kann aber alles, was von einem Displaymanager erwartet wird, und ist zumindest optional bei allen namhaften Linux-Distributionen installierbar.

Eine wichtige Funktion eines Displaymanagers ist die grafische Anmeldung von Benutzern. LightDM erledigt das nicht selbst, sondern greift dazu auf sogenannte *Greeter* zurück. Es gibt diverse Greeter, die typischerweise mit verschiedenen Arbeitsumgebungen harmonisieren. Die Greeter sind für das Aussehen des Anmeldebildschirms verantwortlich.

Konfiguration Die Konfiguration von LightDM steht in Dateien mit der Endung `.conf` in den Verzeichnissen `/usr/share/lightdm/lightdm.conf.d` und `/etc/lightdm/lightdm.conf.d` sowie in der Datei `/etc/lightdm/lightdm.conf`. Die Konfiguration wird in dieser Reihenfolge gelesen. Als Systemadministrator sollten Sie Änderungen idealerweise machen, indem Sie Dateien in `/etc/lightdm/lightdm.conf.d` platzieren. Die Konfigurationsdateien sind in Abschnitte eingeteilt, die Überschriften in eckigen Klammern haben und Schlüssel-Wert-Paare enthalten. Sie könnten zum Beispiel das X-Server-Layout ändern, indem Sie eine Datei `/etc/lightdm/lightdm.conf.d/99local.conf` anlegen, die die Zeilen

```
[Seat:*]
xserver-layout=presentation
```

enthält.

Die wesentlichen Abschnitte in der LightDM-Konfiguration sind:

[LightDM] Voreinstellungen für LightDM insgesamt. Hier werden Parameter festgelegt wie der Name des Benutzers, der Greeter ausführt, die Verzeichnisse für Protokolldateien, Laufzeitdaten und Sitzungsinformationen und ähnliches.

[Seat:*] (oder bei älteren Versionen `[SeatDefaults]`) Voreinstellungen für einen einzelnen Arbeitsplatz (Kombination aus Grafikkarte, Bildschirm(en), Tastatur, Maus, ..., die von einem einzigen X-Server verwaltet wird). Hier können Sie angeben, ob der Arbeitsplatz lokal angeschlossen ist oder entfernt (»X-Terminal«), wie der X-Server aufgerufen wird, wie der Greeter funktionieren soll und aufgerufen wird, wie die Sitzung aufgebaut werden soll und

ob ein Benutzer automatisch angemeldet wird. Alle diese Einstellungen gelten für alle Arbeitsplätze, die an diesen Rechner angeschlossen sind, sofern sie nicht gezielt überschrieben werden.

Einige gängige Einstellungen sind zum Beispiel die folgenden:

```
[Seat:*]
greeter-hide-users=true
greeter-show-manual-login=true
```

Versteckt im Greeter die anklickbare Liste von Benutzern und erlaubt die textuelle Eingabe von Benutzernamen. Dies kann aus Sicherheitsgründen nützlich sein oder weil Sie zu viele Benutzer haben und die Liste darum unhandlich wäre.

```
[Seat:*]
autologin-user=hugo
autologin-user-timeout=10
```

Wenn der Greeter ausgeführt wird, wartet er 10 Sekunden auf Benutzerinteraktion und meldet sonst automatisch den Benutzer hugo an. So etwas sollten Sie natürlich nur benutzen, wenn es ausgeschlossen ist, dass irgendwelche hergelaufenen Leute Ihren Rechner einschalten.

```
[Seat:*]
user-session=mysession
```

Legt `mysession` als Sitzungsname fest. Dies setzt voraus, dass es eine Datei namens `/usr/share/xsessions/mysession.desktop` gibt, die die aufzubauende Sitzung beschreibt. Diese könnte ungefähr aussehen wie

```
[Desktop Entry]
Name=My Session
Comment=My very own graphical session
Exec=/usr/local/bin/startmysession
Type=Application
```

und `/usr/local/bin/startmysession` wäre typischerweise ein Shellskript, das die Sitzung aufbaut.



Die Details dafür würden hier etwas weit führen; lassen Sie sich von einer Datei namens `/etc/X11/Xsession` oder `/usr/bin/startlxde` (je nachdem, welche Arbeitsumgebung Sie installiert haben) inspirieren.

[Seat:0] (und **[Seat:1]** und so weiter) Voreinstellungen für einzelne Arbeitsplätze, die von den Vorgaben in **[Seat:*]** abweichen.



Wenn Sie mehr als einen Arbeitsplatz verwalten wollen, müssen Sie die gewünschten Arbeitsplätze im **[LightDM]**-Abschnitt aufzählen:

```
[LightDM]
seats = Seat:0, Seat:1, Seat:2
```

[XDMCPServer] Entfernte Arbeitsplätze (»X-Terminals«) nehmen über XDMCP (das *X Display Manager Control Protocol*) Kontakt mit dem Displaymanager auf. In diesem Abschnitt stehen Voreinstellungen für XDMCP, insbesondere standardmäßig

```
[XDMCPServer]
enabled = false
```

[VNCServer] Mit diesem Abschnitt können Sie einen X-Server konfigurieren, der über VNC zugänglich ist (das Programm heißt `xvnc`). Damit ist ein grafischer Fernzugriff von Rechnern möglich, die selber kein X unterstützen – VNC-Clients gibt es für diverse Betriebssysteme.

 Die Prüfungsziele des LPI reden davon, dass Sie die Begrüßungsnachricht des Displaymanagers ändern können sollten. Es zeigt sich, dass der Standard-Greeter von LightDM überhaupt keine Begrüßungsnachricht unterstützt, also müssen wir an dieser Stelle leider passen. Bei anderen, weniger üblichen Greetern kann das natürlich anders aussehen – prüfen Sie, ob in `/etc/lightdm` eine Konfigurationsdatei für den betreffenden Greeter steht, und schauen Sie, was man da gegebenenfalls so alles einstellen kann.

 Was Sie auch beim Standard-Greeter einstellen können, ist das Hintergrundbild (typischerweise im SVG-Format). Mit einem SVG-Editor wie Inkscape können Sie sich da nach Belieben austoben, und Sie müssen nur dafür sorgen, dass in der Datei `/etc/lightdm/lightdm-gtk-greeter` etwas steht wie

```
[greeter]
background=/usr/local/share/images/lightdm/my-greeter-bg.svg
```

Starten und Stoppen Gestartet wird der Displaymanager vom Init-System. Das heißt, mit

```
# service lightdm start
```

sollten Sie LightDM starten können, egal ob Ihr System auf System-V-Init oder systemd basiert. Entsprechend funktioniert natürlich auch

```
# service lightdm stop
```

 Alternativ können Sie natürlich auch das Init-Skript direkt aufrufen (bei System-V-Init) oder

```
# systemctl start lightdm oder stop
```

sagen.

Um den Displaymanager zu aktivieren, müssen Sie bei System-V-Init dafür sorgen, dass das Init-Skript für LightDM im gewünschten Runlevel (typischerweise 5) aktiv ist. (Natürlich müssen Sie gleichzeitig einen allfälligen anderen Displaymanager deaktivieren. Zumindest pro X-Server gilt für Displaymanager das Highlander-Prinzip.) Bei systemd-basierten Systemen reicht ein

```
# systemctl enable lightdm
```

zum Aktivieren bzw. ein

```
# systemctl disable lightdm
```

Von Rechts wegen sollte sich natürlich Ihre Linux-Distribution um solche Details kümmern.

11.3.3 Andere Displaymanager

Hier noch kurz ein paar Anmerkungen zu den traditionelleren Displaymanagern, die in den LPI-Prüfungszielen erwähnt werden (es gibt noch mehr).

xdm Der `xdm` ist der Standarddisplaymanager des X11-Systems. Wie viele Beispielprogramme aus X11 ist er sehr schlicht gehalten – er bietet lediglich ein simples grafisches Login-Fenster an. Seine Konfiguration erfolgt über die Dateien im Verzeichnis `/etc/X11/xdm/`.

Xresources Hier können u. a. die Begrüßungsmeldung (`xlogin*greeting`), die Schrift dafür (`xlogin*login.greetFont`) und das vom `xdm` angezeigte Logo (`xlogin*logo FileName`) eingestellt werden.

Xsetup Dies ist ein Shellskript, das beim Start von `xdm` abgearbeitet wird. Hier können Sie unter anderem ein Programm starten, das ein Hintergrundbild auf dem Anmeldebildschirm platziert.

Xservers Hier wird festgelegt, welche X-Server auf welchen Displays starten.

Xsession Spielt eine ähnliche Rolle für `xdm` wie `~/.xinitrc` für `startx` bzw. `xinit`, was die Initialisierung einer Sitzung für Benutzer angeht; auch hier gibt es ein benutzerspezifisches Analogon, nämlich `~/.xsession`.

kdm `kdm` kommt aus dem KDE-Projekt und ist im Wesentlichen eine Erweiterung von `xdm`. Seine Konfiguration entspricht der des `xdm` (die Konfigurationsdateien stehen eventuell distributionsabhängig woanders). `kdm` können Sie außerdem über das KDE-Kontrollzentrum `kcontrol` konfigurieren, dessen Einstellungen zum Beispiel in `/etc/X11/kdm/kdmrc` abgelegt werden.

gdm Der GNOME-Displaymanager `gdm` ist Bestandteil der Desktopumgebung GNOME. Er ist eine komplett neue Entwicklung, bietet aber ungefähr die gleichen Möglichkeiten wie `kdm`. Er wird in der Datei `gdm.conf` konfiguriert, die oft im Verzeichnis `/etc/X11/gdm` liegt. Auch für `gdm` gibt es mit `gdmconfig` ein komfortables Konfigurationsprogramm.

Übungen



11.8 [3] Erlaubt der Displaymanager auf Ihrem System (sofern Sie einen haben) die Auswahl zwischen verschiedenen Arbeitsumgebungen bzw. »Sitzungsarten«? Wenn ja, welche? Probieren Sie einige davon aus (vor allem, falls vorhanden, die *failsafe*-Sitzung).

11.4 Informationen anzeigen

Wenn Ihre X-Sitzung läuft, können Sie einige Programme verwenden, um sich interessante Informationen anzeigen zu lassen.

`xdpyinfo` `xdpyinfo` liefert Ihnen Informationen über Ihr aktuelles X-Display. Wir greifen hier nur ein paar interessante Elemente heraus:

```
$ xdpyinfo
name of display:      :0
version number:      11.0
vendor string:       The X.Org Foundation
vendor release number: 11702000
X.Org version:       1.17.2
<<<<<<
number of extensions: 30
BIG-REQUESTS
Composite
DAMAGE
<<<<<<
XVideo
```

*Auf dem lokalen Rechner
Nicht überraschend*

Versionsnummer des Servers

Eingebundene Erweiterungen

```

default screen number:  0
number of screens:     1

screen #0:
  dimensions:  1680x1050 pixels (442x276 millimeters)
  resolution:  97x97 dots per inch
  depths (7):  24, 1, 4, 8, 15, 16, 32
  root window id:  0x8f
  depth of root window:  24 planes
  number of colormaps:  minimum 1, maximum 1
  <<<<<<
  options:  backing-store WHEN MAPPED, save-unders NO
  largest cursor:  64x64
  current input event mask:  0xfac033
    KeyPressMask EnterWindowMask LeaveWindowMask
    <<<<<<
  number of visuals:  204
  default visual id:  0x21
  visual:
    visual id:  0x21
    class:  TrueColor
    depth:  24 planes
    available colormap entries:  256 per subfield
    red, green, blue masks:  0xff0000, 0xff00, 0xff
    significant bits in color specification:  8 bits
  <<<<<<

```

*Standard-Bildschirm ...
... nur einer da!*

203 andere Visuals überspringen wir hier

Hier ist ziemlich genau beschrieben, welche grafischen Möglichkeiten dieser X-Server anbietet. 24 Bit Farbtiefe ist das, was man sich heutzutage wünscht – potentiell können damit über 16 Millionen Farben dargestellt werden, wobei der optische Apparat normaler Menschen nur so um die gut 100.000 überhaupt unterscheiden kann. Ein »Visual« beschreibt, wie man Pixel in bestimmten Farben auf das Display bekommt, wobei auch hier TrueColor das Nonplusultra darstellt⁴. Es gibt andere Typen von Visuals, die aber nur noch für Hardcore-X-Entwickler von Bedeutung sind.

Visual



Mehr über `xdpinfo` können Sie der `xdpinfo(1)` entnehmen, die das Programm im Detail beschreibt.

Informationen über einzelne Fenster bekommen Sie mit dem Kommando `xwininfo`. Nachdem Sie es in einem Terminalemulator gestartet haben, fordert es Sie auf, mit der Maus in das gewünschte Fenster zu klicken:

xwininfo

```

$ xwininfo
xwininfo: Please select the window about which you
           would like information by clicking the
           mouse in that window.

xwininfo: Window id: 0x500007d "emacs@red.example.com"

Absolute upper-left X: 1071
Absolute upper-left Y: 27
Relative upper-left X: 0
Relative upper-left Y: 0
Width: 604

```

⁴Benutzer älterer Grafikkarten erinnern sich möglicherweise prinzipiell an Visuals vom Typ PseudoColor – typischerweise konnte man 256 Farben aus den bewussten 16 Millionen aussuchen. Nicht schön.

```

Height: 980
Depth: 24
Visual Class: TrueColor
Border width: 0
Class: InputOutput
Colormap: 0x20 (installed)
Bit Gravity State: NorthWestGravity
Window Gravity State: NorthWestGravity
Backing Store State: NotUseful
Save Under State: no
Map State: IsViewable
Override Redirect State: no
Corners: +1071+27 -5+27 -5-43 +1071-43
-geometry 80x73-1+0

```

Hier sehen Sie Sachen wie die Koordinaten des Fensters auf dem Bildschirm, das verwendete Visual und Ähnliches. Der *backing store state* und der *save under state* geben an, was mit Fensterinhalten passiert, die von anderen Fenstern (typischerweise Menüs) überdeckt werden – in unserem Fall werden sie nicht gespeichert, sondern nach Bedarf neu gemalt, was auf heutigen Rechnern oft schneller ist –, und der *map state* gibt an, ob das Fenster tatsächlich auf dem Bildschirm zu sehen ist.

Sie können die Fensterinformationen auch ohne Mausclick abfragen, wenn Sie die Fenster-ID (*window id*) oder den Fensternamen wissen. Die oben gezeigte Beispielausgabe nennt beide – der Fensternamen steht hinter der Fenster-ID auf derselben Zeile.

`xwininfo` kennt diverse Optionen, mit denen Sie Art und Umfang der auszugebenden Daten beeinflussen können. Das Meiste ist nur interessant, wenn Sie sich mit den Innereien von X etwas auskennen. Spielen Sie ein bisschen herum. Die Details finden Sie in `xwininfo(1)`.

Übungen



11.9 [!1] Welche Auflösung in Pixeln hat Ihr Bildschirm? Stimmen die metrischen Abmessungen, die X11 angibt (und damit die Auflösung in Punkten pro Zoll)?



11.10 [2] Verwenden Sie `xwininfo`, um sich Informationen über ein Fenster auf Ihrem Bildschirm ausgeben zu lassen. Verschieben Sie das Fenster und/oder ändern Sie seine Größe, und überzeugen Sie sich, dass `xwininfo` andere Koordinaten ausgibt.

11.5 Der Schriftenserver

Das X11-Protokoll enthält nicht nur Operationen zum Zeichnen von Punkten, Linien und anderen geometrischen Formen, sondern auch zum Darstellen von Text. Traditionell bietet der X-Server eine Reihe von Schriften an; der Client kann abfragen, welche Schriften es gibt, und signalisiert dann, in welcher Schrift und an welcher Position auf dem Bildschirm ein Text ausgegeben werden soll. Das Besorgen der Schriftdateien und die tatsächliche Darstellung sind Aufgabe des Servers.



Es zeigt sich, dass heutzutage praktisch kein moderner X-Client mehr diesen Mechanismus benutzt. X.org und die gängigen Toolkits unterstützen die `XRENDER`-Erweiterung, die mit Transparenz umgehen kann, um zum Beispiel »Antialiasing« zu ermöglichen. Dies wiederum gestattet die Darstellung skalierbarer Schriften mit glatten Kanten und ist nötig für Textausgabe

mit hoher Qualität. Bei XRENDER werden die X11-Operationen zur Textdarstellung völlig umgangen; statt dessen kann der Client »Glyphen« (Buchstaben, Ziffern und andere Zeichen) in den Server hochladen und diese für die Textdarstellung nutzen. Welche Schriften der X-Server anbietet, ist für XRENDER bedeutungslos, da die Schriften auf dem Client vorliegen und dem X-Server als Glyphen zur Verfügung gestellt werden [Pac01].

 Der Rest dieses Abschnitts ist nur interessant, wenn Sie die LPI-102-Prüfung ablegen wollen. Für das wirkliche Leben hat er keine wie auch immer geartete Relevanz mehr. Sparen Sie sich also die Zeit und tun Sie lieber etwas Nützliches – putzen Sie Ihr Bad oder führen Sie Ihren Hund aus.

Lokale Schriften auf dem Server Wenn Sie doch die X11-Textoperationen und damit die Schriften auf dem Server nutzen wollen (oder müssen), müssen Sie zunächst dafür sorgen, dass der X-Server die gewünschten Schriften finden kann. Erste Anlaufstelle für die Konfiguration von Schriften für X.org ist der Files-Abschnitt in der Konfigurationsdatei mit den dort zu hinterlegenden Verzeichnissen (Einträge `FontPath`), in denen der X-Server nach Schriften sucht. Üblicherweise wird pro Verzeichnis ein `FontPath`-Eintrag gesetzt.

 In der Praxis bedeutet das für die Schrifteninstallation: Sie müssen die Schriften in einem passenden Verzeichnis unterbringen und das entsprechende Verzeichnis über einen `FontPath`-Eintrag dem X-Server bekannt machen. Die Reihenfolge der Einträge in der Datei spielt dabei eine wichtige Rolle, da sie der Suchreihenfolge des X-Servers entspricht. Es wird immer die erste passende Schrift verwendet.

 Anstatt ein Schriftenverzeichnis dauerhaft in die Datei einzutragen, können Sie es dem X-Server auch temporär mit dem Kommando `xset` bekannt machen. Das Kommando

```
$ xset +fp /usr/share/fonts/X11/truetype
```

meldet ein Schriftenverzeichnis temporär an (bis zum nächsten Server-Neustart).

```
$ xset -fp /usr/share/fonts/X11/truetype
```

lässt den X-Server das Ganze wieder vergessen.

 Mit dem Kommando »`xset q`« können Sie die aktuelle Konfiguration des X-Servers abfragen und sich so überzeugen, dass er über die richtigen Schriftenverzeichnisse Bescheid weiß.

Mit dem Kopieren und Bekanntmachen des Schriftenverzeichnisses ist es noch nicht getan. Das Verzeichnis kann außer den Schriften selber noch die folgenden Dateien enthalten:

Die Datei `fonts.dir` enthält eine Liste aller im Verzeichnis befindlichen Schriften inklusive Dateiname, Hersteller, Schriftname, Gewicht, Neigung, Breite, Stil, Pixel, Punkte, *x*-Auflösung, *y*-Auflösung, Zeichensatzcodierung und noch einiger anderer Daten. Ein (einzeiliger) Eintrag für eine Schrift könnte beispielsweise so aussehen:

```
luBIS12-IS08859-4.pcf.gz-b&h-lucida-bold-i-normal-sans-><12-120-75-75-p-79-iso8859-4
```

In der ersten Zeile der Datei steht die Gesamtzahl der im Verzeichnis enthaltenen Schriften.

Die Datei `fonts.dir` muss existieren. Sie müssen sie aber selbstverständlich nicht von Hand pflegen, sondern können das Programm `mkfontdir` dazu verwenden. Angenommen, Sie haben im Verzeichnis `/usr/local/share/X11/fonts/truetype` eine Schrift hinzugefügt, reicht ein Aufruf von

```
# mkfontdir /usr/local/share/X11/fonts/truetype
```

um die entsprechende `fonts.dir`-Datei zu aktualisieren. Wenn Sie zu einem existierenden Schriftenverzeichnis Schriften hinzufügen, dann müssen Sie in Ihrer laufenden Sitzung

```
$ xset fp rehash
```

aufrufen, damit der X-Server diese Schriften findet.

`fonts.scale` Die Datei `fonts.scale` enthält bei Verzeichnissen mit skalierbaren (vektorbasierten statt bitmapbasierten) Schriften eine Liste der entsprechenden Schriften, und
`fonts.alias` die Datei `fonts.alias` ermöglicht die Vergabe von Aliasnamen für einzelne Schriften.

Der Schriftenserver Mit dem Schriftenserver (engl. *font server*) `xfs` (nicht zu verwechseln mit dem XFS-Dateisystem) ist es möglich, Schriften im Netz zentral auf einem Server zu verwalten. Bei den heutigen Plattenpreisen ist eine komplette Zentralisierung der X11-Schriften nicht mehr notwendig (oder wünschenswert), aber für Spezialschriften, die nicht Teil der X11- oder Linux-Distribution sind und im lokalen Netz verfügbar sein sollen, kann das die Wartung deutlich vereinfachen.



Der Schriftenserver hat noch einen anderen Vorteil: Ohne Schriftenserver kann es zu Problemen kommen, wenn ein Client den X-Server um eine Liste der verfügbaren Schriften bittet. Der X-Server läßt dann nämlich alles stehen und liegen und sucht das System nach Schriften ab, was zu erheblichen Verzögerungen führen kann, da in dieser Zeit keine X11-Operationen für Grafikdarstellungen bearbeitet werden.

Konfiguration des Servers
 Neuladen der Konfiguration `xfs` ist ein freistehender Daemon. Er bietet seine Dienste auf dem TCP-Port 7100 an und wird über `/etc/X11/fs/config` oder `/etc/X11/xfs.conf` konfiguriert. Nach Änderungen an dieser Datei muss der Server mit dem Signal `SIGHUP` von der neuen Situation in Kenntnis gesetzt werden:

```
# pkill -1 xfs
```

`xfs` greift auf Schriften zurück, die wie für den X-Server beschrieben installiert werden müssen und damit übrigens auch vom lokalen X-Server verwendet werden können. Die Schriftenverzeichnisse werden mittels des `catalogue`-Parameters in die Konfigurationsdatei eingetragen, beispielsweise so:

```
catalogue = /usr/share/fonts/X11/misc:unscald,  
            /usr/share/fonts/X11/75dpi:unscald,  
            /usr/share/fonts/X11/100dpi:unscald  
            <<<<<<
```

Die einzelnen Pfade müssen jeweils durch Kommas voneinander getrennt werden. Weitere Parameter können Sie der Dokumentation entnehmen.

Konfiguration des Clients Um einen X-Server an den Schriftenserver anzubinden, müssen Sie lediglich einen Eintrag im `Files`-Abschnitt der `xorg.conf`-Datei hinzufügen:

```
FontPath "tcp/<Rechnername>:<Portnummer>"
```

Wenn Sie wollen, dass standardmäßig die Schriften des Schriftenservers bevorzugt werden, sollten Sie den beschriebenen Eintrag vor allen anderen `FontPath`-Einträgen einfügen. (Zum Experimentieren geht das auch temporär mit `xset`.)

Übungen



11.11 [!1] Verwenden Sie das Kommando `xlsfonts`, um sich die Schriften auflisten zu lassen, die Ihr X-Server kennt.



11.12 [1] Benutzen Sie das Kommando `xfontsel`, um die Auswahl an Schriften bequem zu erforschen. Finden Sie eine Schrift, die Sie mögen, und merken Sie sie sich für die nächste Aufgabe.



11.13 [2] Rufen Sie einen hinreichend antiken X-Client (`xman`, `xterm` oder `xedit` drängen sich auf) mit der Schrift aus der vorigen Aufgabe auf. Die kanonische Kommandozeilenoption dafür ist `>>-fn<<` (wie *font*). Was passiert?

11.6 Fernzugriff und Zugriffskontrolle

Grundsätzlich ist der X-Server, wie erwähnt, über einen TCP-Port (6000 + »Servernummer«) für Clients von außen erreichbar. Diese müssen nur mit der korrekten Display-Einstellung – über eine Kommandooption à la `-display` oder die Umgebungsvariable `DISPLAY` – aufgerufen werden und können ihre Ausgabe auf dem Server anzeigen und Eingaben akzeptieren, aber theoretisch auch die Sitzung beliebig stören oder ausspähen.

Prinzipiell gibt es zwei X-interne Möglichkeiten, um Zugriffe auf den X-Server zu kontrollieren, nämlich `xhost` und `xauth`.

`xhost` bietet rechnerbasierten Zugriffsschutz. Mit

`xhost`

```
$ xhost red.example.com
```

oder

```
$ xhost +red.example.com
```

erlauben Sie den Zugriff auf Ihren Server für Clients, die auf `red.example.com` laufen. Ein

```
$ xhost -red.example.com
```

sperrt den Zugriff wieder. »`xhost`« allein gibt eine Liste aller zugelassenen Rechner aus. Da hiermit *jeder* Benutzer auf dem entfernten Rechner direkten Zugang zu Ihrer X-Sitzung bekommt, sollten Sie `xhost` *nicht* einsetzen!!



Manchmal werden Sie – etwa in Anleitungen zur Installation proprietärer Softwarepakete – aufgefordert, das Kommando »`xhost` +« auszuführen. Dies schaltet Ihren X-Server für Zugriffe von Clients auf beliebigen Rechnern (potentiell das komplette Internet) frei. Lassen Sie sich auf sowas nicht ein.

Bei `xauth` wird beim Start (meist vom Displaymanager oder `startx`) ein zufälliger Schlüssel oder *magic cookie* erzeugt und dem X-Server mitgeteilt. Außerdem wird der Schlüssel in der Datei `~/.Xauthority` des angemeldeten Benutzers abgelegt, die für andere Benutzer nicht lesbar ist. Der Server nimmt nur Verbindungen von Clients an, die den korrekten *magic cookie* vorweisen können. Die *magic cookies* lassen sich mit `xauth` auch auf andere Rechner übertragen oder von diesen wieder entfernen. Näheres steht in `xauth(1)`.

`xauth`

`~/.Xauthority`



Eine wesentlich sicherere Methode zum Starten von X-Clients auf entfernten Rechnern besteht darin, die X-Weiterleitung der Secure Shell zu verwenden. Wir beschreiben das in *Linux-Administration II*.

Mit der Option »`-nolisten tcp`« können Sie verhindern, dass Ihr X-Server überhaupt über TCP von außen zugänglich ist. Dies ist eine empfehlenswerte Einstellung und bei vielen Linux-Distributionen heute Standard.

Übungen



11.14 [2] Sorgen Sie dafür, dass Ihr X-Server *nicht* mit »-nolisten tcp« gestartet wird (oder starten Sie einen zweiten X-Server mit etwas wie »startx -- :1«) und versuchen Sie, einen Client über eine geeignete DISPLAY-Einstellung mit dem Server zu verbinden (Sonderpunkte, wenn Sie den Client auf einem anderen Rechner laufen lassen). (*Tipp*: Wenn es bei einer Verbindung von einem anderen Rechner Schwierigkeiten gibt, dann prüfen Sie, ob Ihr Rechner einen überaktiven Paketfilter verwendet, der Ihren X-Server von der Außenwelt abschottet. Vor allem die SUSE-Distributionen machen das ganz gerne.)

Kommandos in diesem Kapitel

X	Startet den passenden X-Server für das System	X(1)	182
lspci	Gibt Informationen über Geräte auf dem PCI-Bus aus	lspci(8)	179
xauth	Verwaltet den Zugriff auf den X-Server über <i>magic cookies</i>	xauth(1)	191
xdpyinfo	Zeigt Informationen über das aktive X-Display	xdpyinfo(1)	186
xhost	Erlaubt Clients auf anderen Rechnern den Zugriff auf den X-Server über TCP	xhost(1)	191
xwininfo	Zeigt Informationen über ein X-Fenster an	xwininfo(1)	187

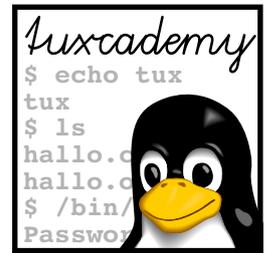
Zusammenfassung

- X11 ist ein client-server-orientiertes, netzwerktransparentes Grafiksystem.
- Der X-Server verwaltet den Grafikbildschirm, Tastatur und Maus eines Rechners; X-Clients greifen über das X-Protokoll auf den X-Server zu.
- Auf Arbeitsplatzrechnern wird X11 gerne so installiert, dass eine grafische Anmeldung möglich ist. Bei anderen Rechnern kann die Grafikoberfläche gegebenenfalls manuell gestartet werden.
- Neben dem einfachen Displaymanager *xdm* liefern die meisten Arbeitsumgebungen einen eigenen Displaymanager mit.

Literaturverzeichnis

Pac01 Keith Packard. »Design and Implementation of the X Rendering Extension«. *Proc. FREENIX Track, 2001 Usenix Annual Technical Conference*. The USENIX Association, 2001 S. 213–224.

<http://keithp.com/~keithp/talks/usenix2001/>



12

Linux für Behinderte

Inhalt

12.1 Einführung	194
12.2 Tastatur, Maus und Joystick	194
12.3 Die Bildschirmdarstellung	195

Lernziele

- Wissen, welche Hilfsmittel Linux für Behinderte zur Verfügung stellt

Vorkenntnisse

- Grundkenntnisse von Linux, X11 und Arbeitsumgebungen wie KDE oder GNOME

12.1 Einführung

Der Computer und das Internet haben die Möglichkeiten, die Menschen mit Behinderungen zur Verfügung stehen, revolutioniert. Blinde und Sehbehinderte können auf weitaus mehr Informationen zugreifen, als ihnen früher zur Verfügung standen, und auch Menschen mit anderen Einschränkungen profitieren von den Arbeits-, Kontakt-, Informations- und Ausdrucksmöglichkeiten, die Computer ihnen geben.

Hilfsmittel In diesem Kapitel geben wir einen kurzen Überblick über verschiedene Hilfsmittel, die Linux und die gängigen Softwarepakete Behinderten zur Verfügung stellen. Tiefschürfende technische Diskussionen sind hier nicht möglich; es geht uns mehr darum, Ihnen Ansatzpunkte zu geben, wo Sie bei Bedarf mehr Informationen erhalten können.

12.2 Tastatur, Maus und Joystick

Wer nicht in der Lage ist, eine herkömmliche Tastatur oder Maus zu bedienen – sei es, weil die nötigen Gliedmaßen fehlen oder weil sie nicht genau genug kontrolliert werden können –, kann auf verschiedene Hilfsmittel zurückgreifen, die unter Linux zur Verfügung stehen. Typische Hilfen sind:

Klebende Tasten (*sticky keys*) sorgen dafür, dass Tasten wie die Umschalt- oder -Taste nicht mehr festgehalten werden müssen, während Sie eine andere Taste drücken. Ein Druck auf die Umschalttaste (mit anschließendem Loslassen) genügt, damit das nächste Zeichen so ausgewertet wird, als wäre die Umschalttaste noch gedrückt. Dies hilft zum Beispiel Querschnittsgelähmten, die nur den Kopf bewegen können und mit einer Stange tippen.

Langsame Tasten (*slow keys*) lassen das System ungewollte Tastendrücke ignorieren, die entstehen, wenn Sie auf dem Weg zu der Taste, die Sie wirklich meinen, andere Tasten drücken.

Zurückschnellende Tasten (*bounce keys*) sorgen dafür, dass das System überzählige Drücke derselben Taste ignoriert.

Wiederholungstasten (*repeat keys*) erlauben es, anzugeben, ob festgehaltene Tasten wiederholt oder nur einmal weitergemeldet werden sollen.

Maustasten (*mouse keys*) gestatten die Steuerung der Maus über den numerischen Tastenblock der Tastatur.

XKEYBOARD Bei X11 unter Linux werden diese Hilfen über die XKEYBOARD-Erweiterung zur Verfügung gestellt, die normaler Bestandteil des X-Servers ist. Die Herausforderung ist also lediglich, sie zu aktivieren. Dafür gibt es ein Programm namens `xkbs`, und auch die Arbeitsumgebungen stellen Oberflächen zur Verfügung, etwa das KDE-Kontrollzentrum (Dialog »Zugangshilfen« unter »Regionaleinstellungen und Zugangshilfen«).

Bildschirmtastatur Zumindes für GNOME gibt es auch eine »Bildschirmtastatur« namens GOK, die es möglich macht, mit der Maus, einem Joystick oder gar nur einer einzelnen Taste zu »tippen«. Dies ist eine Hilfe für Menschen, die mit einer Tastatur nicht zurecht kommen, aber die Maus verwenden können. KDE hat derzeit nichts dergleichen anzubieten.

RSI Die Maus kann in vielen Fällen durch die Tastatur ersetzt werden. Zumindest theoretisch sollten in den Arbeitsumgebungen alle Funktionen auch über die Tastatur ansprechbar sein. Wer wegen Überanstrengung (RSI) Schwierigkeiten hat, eine Maus zu verwenden, kommt vielleicht auch mit einem stationären »Trackball« besser zurecht. Für Benutzer mit motorischen Schwierigkeiten kann es auch nützlich sein, den Zeitabstand für Doppelklicks zu verlängern.



Im KDE-Kontrollzentrum finden Sie die Einstellungen dafür, den numerischen Tastenblock als »Mausersatz« nutzen zu können, unter »Angeschlossene Geräte/Maus« in der Registerkarte »Mausnavigation«. Bei GNOME steht das Äquivalent in der Tastaturkonfiguration.

12.3 Die Bildschirmdarstellung

Linux hat für Sehbehinderte und Blinde den großen Vorteil, dass die Bildschirmdarstellung sehr flexibel gesteuert werden kann. Da das System nicht auf einer grafischen Oberfläche besteht, ist es wesentlich einfacher für Blinde zu bedienen, die eine Braille-Zeile oder einen Screenreader verwenden, als rein grafisch aufgebaute Systeme.

Menschen, die eine gewisse Sehkraft haben, hilft Linux zum Beispiel dadurch, den Mauszeiger oder Teile der Bildschirmdarstellung zu vergrößern, damit sie leichter ausgemacht werden können. Unter KDE zum Beispiel können Sie das Aussehen des Mauszeigers in der Registerkarte »Zeigerdesign« unter »Angeschlossene Geräte/Maus« ändern. (Es kann sein, dass Sie erst ein behindertengerechtes Mauszeiger-Thema installieren müssen.) Auch bei GNOME ist die Konfiguration des Mauszeigers Bestandteil der Maus-Konfiguration.

Zeigerdesign

KDE und GNOME erlauben ebenso die Anpassung der Schriftgrößen, die die Arbeitsumgebung verwendet. Auf einem hochauflösenden Bildschirm können die Standardschriften schon für normalsichtige Benutzer unbequem klein sein; Sehbehinderte profitieren erst recht von einer großzügigen Zugabe. Dasselbe gilt für die Farbschemata der Oberfläche, wo es ebenfalls Themen gibt, die Details über eine kontrastreichere Darstellung besser hervorheben.

Anpassung der Schriftgrößen

Farbschemata

Ein weiteres gängiges Hilfsmittel sind »Bildschirm Lupen«, die das Areal rund um den Mauszeiger in einem anderen Fenster stark vergrößert anzeigen. KMagnifier für KDE und das unter »Hilfsmittel« in den Systemeinstellungen zugängliche Äquivalent unter GNOME ermöglichen das bequem.

Bildschirm Lupen

Für Blinde unterstützt Linux verschiedene Braille-Zeilen oder auch Sprachausgabe. Eine Braille-Zeile kann eine Zeile Text in »Blindenschrift« darstellen, die ein Blinder ertasten kann; Braille-Zeilen sind allerdings ziemlich teuer und mechanisch aufwendig. BrlTTY ist ein Programm, das auf der Linux-Konsole läuft und eine Braille-Zeile ansteuert; Orca ist dasselbe für GNOME. Ferner gibt es noch Emacspeak, das im Grunde ein Screenreader für GNU Emacs ist, der Bildschirmhalte vorliest; da man aber viele andere Programme innerhalb von Emacs starten kann, ist das fast so gut wie eine grafische Oberfläche.

Braille-Zeilen

Sprachausgabe

BrlTTY

Orca

Emacspeak



KDE tut sich mit einigen dieser technischen Hilfsmittel schwer. Es gibt ein einigermaßen etabliertes Protokoll namens AT-SPI (*Assistive Technologies Service Provider Interface*), das unter Unix bzw. Linux zur Kommunikation zwischen behindertengerechter Software und technischen Hilfsmitteln wie Braille-Zeilen verwendet wird. Ärgerlicherweise (aus der Sicht von KDE) wurde AT-SPI von GNOME-Entwicklern erfunden und setzt GTK2+ voraus (eine Grafikbibliothek, die auch GNOME zugrunde liegt, mit der KDE aber nichts anfangen kann). Ferner benutzt es CORBA zur Kommunikation, was ebenfalls nicht zu KDE passt. Wie man diese Probleme lösen wird, ist noch unklar.

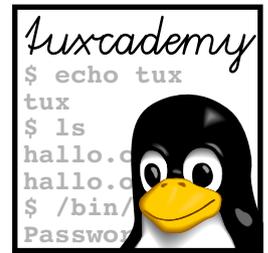
Kommandos in diesem Kapitel

xkbset Erlaubt die Verwaltung von Tastatureinstellungen für X11

xkbset(1) 194

Zusammenfassung

- Die Verwendung von Computern ist gerade für behinderte Menschen ein wichtiges Stück Lebensqualität.
- Linux bietet diverse Zugangshilfen für Menschen, die mit einer normalen Tastatur oder einer normalen Maus nicht umgehen können.
- Blinde und Sehbehinderte profitieren von vergrößerten und kontrastverstärkten Bildschirmdarstellungen, Bildschirmlupen, Screenreader-Programmen mit Sprachausgabe und Braille-Zeilen, die Linux ansteuert.



A

Musterlösungen

Dieser Anhang enthält Musterlösungen für ausgewählte Aufgaben.

1.1 Einige denkbare Vorteile könnten sein: Shellskripte sind in der Regel wesentlich schneller zu erstellen, sind nicht abhängig von der Rechnerarchitektur (Intel, PowerPC, SPARC, ...) und kürzer als äquivalente Programme zum Beispiel in C. Shellprogrammierung ist zumindest auf einer elementaren Ebene viel leichter zu lernen als die Programmierung in einer Sprache wie C. Als Nachteile kommen in Frage, dass Shellskripte oft weniger effizient laufen, man in der Auswahl der möglichen Datenstrukturen sehr eingeschränkt ist und Shellskripte auch nicht für Programme taugen, die erhöhten Sicherheitsanforderungen genügen müssen. – Shellskripte eignen sich am besten für »Ad-hoc-« oder »Wegwerfprogrammierung«, wo man nicht viel Zeit investieren möchte, für die Erstellung von Prototypen (wobei es durchaus Chancen gibt, dass der Prototyp sich anschließend als »gut genug« herausstellt) und für die Automatisierung von Aufgaben, die man sonst über die Kommandozeile erledigen würde. Programmiersprachen wie C bedeuten immer einen höheren Entwicklungsaufwand, der sich natürlich auch rechnen muss; ein gängiger Ansatz besteht deshalb darin, ein Programm zuerst als Shellskript zu realisieren und später zum Beispiel laufzeitkritische Programmteile durch C-Programme zu ersetzen. Das Optimum liegt oft in einer Mischung aus Shell- und Binärcode.

1.2 Ein möglicher Ansatz wäre:

```
find /bin /usr/bin -type f -exec file {} \; \
| grep "shell script" | wc -l
```

Das `find`-Kommando zählt alle Dateien in `/bin` und `/usr/bin` auf und wendet jeweils `file` auf jeden gefundenen Dateinamen an. `grep` sucht diejenigen Zeilen heraus, die »shell script« enthalten, und `wc` bestimmt deren Anzahl. – Wie können Sie die Ausführung dieser Pipeline beschleunigen?

1.3 Eine *quick*"=*and*"=*dirty*-Methode hierfür ist das Kommando

```
ls $(cat /etc/shells) 2>/dev/null
```

das Ihnen zumindest die als Login-Shell zugelassenen Shells auflistet (Fehlermeldungen über die Shells, die zwar in der Datei stehen, aber auf dem System als

Programm nicht existieren, werden durch die Umleitung der Standardfehlerausgabe nach `/dev/null` unterdrückt). Beachten Sie auch, dass in `/etc/shells` Kommentare mit Lattenzaun am Anfang erlaubt sind, was zusätzliche Fehlermeldungen provoziert, die ebenfalls unterdrückt werden.

1.4 Die `tcsh` versucht hilfreicherweise, Ihr fehlerhaftes Kommando zu korrigieren und Ihnen eine »richtige« Version anzubieten, die Sie akzeptieren, ablehnen oder editieren oder das Kommando als ganzes abbrechen können. – Die `tcsh` können Sie mit `exit` verlassen; in fast allen Shells funktioniert auch `[Strg]+[d]` als »Dateiende auf der Eingabe«.

1.5 Verwenden Sie `chsh`, um die Shell zu ändern. Auf eine andere virtuelle Konsole kommen Sie zum Beispiel mit `[Alt]+[F2]` (`[Strg]+[Alt]+[F2]`), wenn Sie in der grafischen Oberfläche arbeiten).

1.6 Verwenden Sie im einfachsten Fall `vipw`, um den Eintrag für `root` in `/etc/passwd` zu kopieren und die Login-Shell (letzte Spalte) auf `/bin/sash` zu ändern. Denken Sie auch daran, `/etc/shadow` entsprechend anzupassen (`>>vipw -s<<`).

1.8 Als erstes Zeichen des Programmnamens wird ein `>>-<<` übergeben.

1.9 Sorgen Sie dafür, dass diese Shell glaubt, sie würde als Login-Shell aufgerufen: Verwenden Sie die Bash als »echte« Login-Shell mit einer `.profile`-Datei wie der folgenden:

```
PATH=$HOME/bin:$PATH
exec -l myshell
```

Dies nutzt eine spezielle Eigenschaft des `exec`-Kommandos der Bash aus. Steht Ihnen ausgerechnet die Bash nicht als Login-Shell zur Verfügung, müssen Sie zu schmutzigeren Tricks greifen: Verwenden Sie eine `.profile`-Datei wie

```
PATH=$HOME/bin:$PATH
exec -myshell
```

Dabei ist `\$HOME/bin/-myshell` ein (symbolisches) Link auf `/usr/local/bin/myshell` (oder wo auch immer die von Ihnen gewünschte Shell liegt). Diese Shell sieht dann als ihren Programmnamen `>>-myshell<<` und initialisiert sich als Login-Shell – jedenfalls wenn sie nach den Regeln spielt. Das `>>exec<<` ist nicht unbedingt nötig; es sorgt dafür, dass Ihr Login-bash-Prozess sich durch `myshell` ersetzt, statt `myshell` als Kindprozess aufzurufen. Ein einfaches `>>-myshell; exit<<` würde es auch tun (wofür wird das `exit` gebraucht?).

Manche Shells haben auch eine Kommandozeilen-Option (bei der Bash ist es `-l`), mit der Sie signalisieren können, dass eine Shell sich als Login-Shell fühlen soll. In diesem Fall kann der schmutzige Trick mit dem Link natürlich wegfallen, und Sie schreiben einfach etwas wie

```
exec /usr/local/bin/bash -l
```

2.2 Da in diesen Dateien zum Beispiel Umgebungsvariable gesetzt werden, muss wohl `source` verwendet werden ...

2.3 Statt des Benutzerskripts wird das Programm `/bin/test` (oder das interne Shellkommando `test`, etwa bei der Bash) gestartet. Das kann leicht passieren, wenn der Benutzer das Verzeichnis mit dem eigenen Skript nicht im `PATH` hat oder `/bin` vor diesem Verzeichnis auftaucht. Perfiderweise produziert das Systemkommando `test`, ohne Parameter aufgerufen, keine Fehlermeldung – in unseren Augen ein schweres Versäumnis sowohl der externen als auch der Shell-Implementierung.

2.4 Das Shellskript muss das gewünschte Zielverzeichnis auf seiner Standardausgabe liefern und von der aufrufenden Shell mit etwas wie

```
$ cd `meinskript.sh`
```

gestartet werden. Diesen etwas umständlichen Aufruf verstecken Sie am besten in einem Aliasnamen:

```
$ alias meinskript='cd `meinskript.sh`'
```

(Zusatzfrage: Warum sind hier die einfachen Anführungszeichen wichtig?)

2.5 Das Aufrufkommando `»./blubb«` wird an die erste Zeile angehängt und diese ausgeführt. Darum sollte als Ausgabe

```
bla fasel ./blubb
```

erscheinen. Da das Programm `echo` seine Parameter nicht als Dateinamen interpretiert, ist der eigentliche Inhalt der Datei irrelevant; das `»echo Hallo Welt«` ist also reine Nebelwerferei.

2.6 Es kommt auf das Skript an. Wie bereits anderswo in dieser Schulungsunterlage angesprochen ist `»#!/bin/sh«` sowas wie ein Versprechen des Shellskripts `»Ich kann mit jeder Bourne-artigen Shell ausgeführt werden«`. In so einem Skript sollten also keine Bash-spezifischen Konstruktionen vorkommen. Entsprechend bedeutet die erste Zeile `»#!/bin/bash«` soviel wie `»Ich brauche wirklich die Bash«`. Wer so ein Skript in sein eigenes Softwaresystem integrieren möchte, wird dadurch gewarnt, dass er vermutlich die ziemlich dicke und fette Bash dazunehmen muss, wenn er sonst eine der schlankeren Shells (`dash`, `busybox`, ...) verwenden könnte. Es hat also keinen Sinn, immer `»#!/bin/bash«` zu schreiben; immer `»#!/bin/sh«` zu schreiben ist hingegen gefährlicher.

2.7 Eine gangbare Vorgehensweise wäre:

1. Eine Liste aller »echten« Benutzer aufstellen
2. Wiederhole die folgenden Schritte für jeden Benutzer *u* in der Liste
3. Den Zeitpunkt *t* des letzten Einloggens von *u* bestimmen
4. Das Heimatverzeichnis *v* von *u* bestimmen
5. Den durch *v* belegten Plattenplatz *p* bestimmen
6. *u*, *t* und *p* ausgeben
7. Ende der Wiederholung

(Es gibt natürlich andere Möglichkeiten.)

2.8 Ein Ansatz wäre:

1. Bestimme eine sortierte Liste aller Heimatverzeichnisse von Benutzern
2. Bestimme daraus eine Liste aller Verzeichnisse, die Heimatverzeichnisse enthalten (/home/entwick und /home/market in unserem Beispiel) – dabei sollen allfällige Dubletten entfernt werden
3. Prüfe für jedes dieser Verzeichnisse d den belegten Platz b_d :
4. Wenn $b_d > 95\%$, dann nimm d in die Warnliste auf
5. Schicke die Warnliste an den Systemverwalter

Auch hier gibt es natürlich zahllose andere Möglichkeiten.

3.1 `*` und `@` verhalten sich gleich, bis auf einen Spezialfall – die Expansion von `»$@«`. Bei letzterer ergibt jeder Positionsparameter ein einzelnes »Wort«, während der Ausdruck `»$*«` ein einziges Wort liefert.

3.2 Die Bash rundet immer zu der betragsmäßig nächstniedrigeren Ganzzahl hin. Da sie keine Gleitkommazahlen kennt, ist das eine naheliegende, wenn auch nicht notwendigerweise die optimale Vorgehensweise.

3.3 Auf gängigen Linux-Systemen benutzt die Bash 64-Bit-Arithmetik, die größte darstellbare Zahl ist also $2^{63} - 1$ oder 9.223.372.036.854.775.807. Wenn Sie nicht im Quellcode nachlesen wollen, können Sie ein Skript wie

```
#!/bin/bash
a=0
while [ $a -ge 0 ]
do
    a=$((2*a+1))
    echo $a
done
```

ausführen und das Ergebnis interpretieren.

3.4 Das geht mit etwas wie

```
#!/bin/sh
echo $1 | grep -i '^[aeiou][aeiou]*$'
```

(Parameterprüfung *ad libitum*). Der Rückgabewert eines Skripts ist der Rückgabewert des letzten Kommandos, und der Rückgabewert einer Pipeline ist der Rückgabewert des letzten Kommandos der Pipeline. Was `grep` macht, passt gerade – bingo!

3.5 Versuchen Sie etwas wie

```
#!/bin/sh
# absreldpath -- teste Pfadnamen auf absolut/relativ

if [ "${1:0:1}" = "/" ]
then
    echo absolut
else
    echo relativ
fi
```

Andere Schreibweisen – etwa

```
[ "${1:0:1}" = "/" ] && echo absolut || echo relativ
```

sind denkbar, aber für den ernsthaften Gebrauch vielleicht zu kryptisch.

3.6 Die anerkanntermaßen einfachste Methode ist eine weitere case-Alternative der Form

```
<<<<<<
  restart)
    $0 stop
    $0 start
    ;;
<<<<<<
```

3.7 Dies ist zwar völlig trivial, wenn Sie die Unterlage gelesen haben, aber weil Sie's sind ...

```
#!/bin/bash
# Name: multi3
i=3
until test $i -gt $1
do
  echo $i
  i=$((i+3))
done
```

3.8 Wir erlauben uns hier eine Bash-spezifische Schreibvereinfachung im Interesse der Übersichtlichkeit: Das »Kommando« ((...)) (ohne Dollarzeichen!) bewertet den Ausdruck zwischen den Klammern und liefert einen Rückgabewert von 0, wenn der Wert des Ausdrucks ungleich 0 ist, 1 sonst.

```
#!/bin/bash
# prim -- Bestimmt Primzahlen bis zu einer Obergrenze
#       Gruselig ineffizientes Verfahren.

echo 2
i=3
while ((i < $1))
do
  prim=1
  j=2
  while ((prim && j*j <= i))
  do
    if ((i % j == 0))
    then
      prim=0
    fi
    j=$((j+1))
  done
  ((prim)) && echo $i
  i=$((i+2))
done
```

(Die Mathematiker im Publikum werden aufheulen; diese Methode ist gegenüber Verfahren wie dem »Sieb des Eratosthenes« viel zu umständlich. Natürlich gibt es geschicktere Methoden, aber wir sind ja hier, um Shellprogrammierung zu lernen, und nicht Zahlentheorie.)

3.9 Zunächst könnten Sie das `if` direkt vom `fgrep` abhängig machen und nicht auf dem Umweg über `?:`:

```
if fgrep -q $pattern $f
then
    cp $f $HOME/backups
fi
```

Statt den Rückgabewert des Kommandos mit `!` zu negieren und dann gegebenenfalls `continue` aufzurufen, ziehen wir das `cp` in den `then`-Teil hinein und werden dadurch das `continue` ganz los. Die andere Beobachtung ist, dass in unserer neuen Fassung das Kopieren auch dann übersprungen wird, wenn beim `fgrep` etwas anderes Unvorhergesehenes passiert ist (etwa die angegebene Datei nicht gefunden wurde). Das ursprüngliche Skript hätte in diesem Fall das Kopieren trotzdem versucht.

In diesem einfachen Fall können Sie aber auch die bedingte Kommandoausführung verwenden und etwas schreiben wie

```
fgrep -q $pattern $f && cp $f $HOME/backups
```

Das ist dann die kürzeste mögliche Form.

3.10 Ersetzen Sie hierzu das `break` durch ein `continue`.

3.12 Eine Möglichkeit:

```
#!/bin/bash
# tclock -- Zeige eine Uhr in einem Textterminal
trap "clear; exit" INT
while true
do
    clear
    banner $(date +%X)
    sleep 1
done
```

3.13 Zum Beispiel:

```
function toupper () {
    echo $* | tr '[:lower:]' '[:upper:]'
}
```

3.14 Mit dem `exec` endet die Ausführung von `test1` unwiderruflich. Die Ausgabe ist also

```
Hallo
Huhu
```

4.1 In Analogie zu `grep` sollte das `exit` hinter der ersten Fehlermeldung wohl eine 2 als Argument haben. Der Rückgabewert 1 signalisiert dann »Gruppe existiert nicht«.

4.2 Ein möglicher Ansatz:

```
#!/bin/bash
# hierarchy -- Dateinamenhierarchie nach oben verfolgen

name="$1"
until [ "$name" = "/" ]
do
    echo $name
    name="$(dirname $name)"
done
```

4.3 Das hierarchy-Skript liefert uns die richtigen Namen, aber in der verkehrten Reihenfolge. Wir müssen nur noch prüfen, ob das betreffende Verzeichnis schon existiert:

```
#!/bin/bash
# mkdirp -- "mkdir -p" für Arme

for dir in $(hierarchy "$1" | tac)
do
    [ -d "$dir" ] || mkdir "$dir"
done
```

4.4 Ersetzen Sie die Zeile

```
conffile=/etc/multichecklog.conf
```

durch etwas wie

```
conffile=${MULTICHECKLOG_CONF:-/etc/multichecklog.conf}
```

4.5 Eine Möglichkeit (in der checklonger-Funktion):

```
function checklonger () {
    case "$1" in
        *k) max=$(( ${1%k} * 1000 )) ;;
        *M) max=$(( ${1%M} * 1000000 )) ;;
        *) max=$1 ;;
    esac
    test ...
}
```

4.6 Der Trick besteht darin, seq rückwärts zählen zu lassen:

```
function rotate () {
    rm -f "$1.9"
    for i in $(seq 9 -1 1)
    do
        mv -f "$1.$((i-1))" "$1.$i"
    done
    mv "$1" "$1.0"
    > "$1"
}
```

Statt der hartcodierten 9 können Sie natürlich auch eine Variable einsetzen.

4.7 Die bequemste Möglichkeit verwendet die `--reference`-Option von `chmod` und `chown`, die die betreffenden Daten von einer anderen, existierenden Datei kopiert (hier der alten Protokolldatei). Zum Beispiel:

```
<<<<<<
mv "$1" "$1.0"
> "$1"
chmod --reference="$1.0" "$1"
chown --reference="$1.0" "$1"
<<<<<<
```

4.8 `grep` unterstützt die Option `-e`, mit der ein Suchausdruck eingeleitet werden kann. Diese Option darf mehrmals auf der Kommandozeile stehen, und `grep` sucht dann nach allen so angegebenen regulären Ausdrücken auf einmal.

4.9 `»"$@"«` sorgt dafür, dass die Argumente von `gdf` an `df` übergeben werden. Damit funktionieren Aufrufe wie `»gdf / /home«`. Optionen, die die Ausgabe von `df` radikal ändern, sollten Sie sich allerdings besser verkneifen.

5.1 Die Technik dafür entspricht haargenau der für die Kontrolle des neuen Benutzernamens.

5.2 Das ginge mit etwas wie

```
read -p "Login-Shell: " shell
if ! [ grep "^$shell$" /etc/shells ]
then
    echo >&2 "$shell ist keine gültige Login-Shell"
    exit 1
fi
```

5.3 Sie können zum Beispiel die `read`-Anweisung durch etwas wie

```
prompt=${1:-"Bitte bestätigen"}
read -p "$prompt (j/n): " answer
```

ersetzen.

5.4 Ein mögliches Beispiel:

```
#!/bin/sh
# zahlenraten -- einfaches Ratespiel

max=100
zahl=$(( RANDOM % max ))

echo Raten Sie eine Zahl zwischen 0 und $max.
while true; do
    read -p "Zahl? " eingabe
    d=$(( eingabe - zahl ))
    if [ $d = 0 ]; then
        echo "Herzlichen Glückwunsch, das war richtig"
        break
    elif [ $d -gt 0 ]; then
        echo "Zu groß"
    else

```

```

    echo "Zu klein"
  fi
done

```

5.5 Die Variable der select-Schleife bekommt eine leere Zeichenkette als Wert. Aus diesem Grund heißt es im `newuser`-Skript `»[-n "$type"] && break«`, damit die Schleife erst verlassen wird, wenn der Aufrufer einen gültigen Wert eingegeben hat.

5.6 Die naheliegende Lösung (`»score«` ausgeben) ist nicht richtig, da diese Variable ja den *aktuellen* Punktestand enthält. Den künftigen Punktestand können Sie mit der `next`-Option von `question` abfragen. Also in present:

```

# Finde und zeige die Frage
echo "Für $(question $1 next) Punkte:"
question $1 display

```

5.7 Naheliegende Erweiterungen wären zum Beispiel (in ungefährender Reihenfolge des Aufwands):

- Mehr als eine Frage pro Punktstufe (mit zufälliger Auswahl?)
- `»Aussteigen«`: Wer nicht mehr weiter weiß oder kann, kann mit seiner aktuellen Punktzahl das Spiel beenden
- Feste Gewinnstufen (wer zwischen 500 und 16000 Punkten hat und eine falsche Antwort gibt, fällt auf 500 zurück, wer 16000 Punkte hat oder mehr, behält 16000)
- `»50-50-Joker«`: Der Teilnehmer bekommt einen weiteren Menüpunkt zur Auswahl, mit dem er (einmalig) zwei falsche Antworten verschwinden lassen kann

Was fällt Ihnen noch ein?

5.8 Eigentlich sollte ein `grep`-Aufruf pro Frage reichen, wenn Sie die verschiedenen Zeilen in Shellvariablen abspeichern und bei Bedarf ausgeben. Da immer nur eine Frage im Speicher sein muss, gibt es keine Schwierigkeiten mit aufwendigen Datenstrukturen.

6.1 Die jeweils passenden Zeilennummern sind: (a) 4 (wer hätte es gedacht?); (b) 2–4 (das `ABC` in Zeile 2 zählt nicht); (c) 2–3 und 4–5 (Adressbereiche mit einem regulären Ausdruck als erster Adresse können mehrmals passen); (d) 6; (e) 3 (die Zeile 2 ist schon `»durch«`); (f) 3 und 5–6 (die Zeilen, die kein `ABC` enthalten).

6.2 Reguläre Ausdrücke als zweite Adresse in einem Bereich passen frühestens auf die erste Zeile nach dem Bereichsanfang. Beim Adressbereich `»1,/⟨Ausdruck⟩/«` könnte `⟨Ausdruck⟩` also niemals auf die erste Zeile der Eingabe passen. Mit `»0,/⟨Ausdruck⟩/«` wird genau das ermöglicht.

6.3 Die vermutlich einfachste Methode ist

```
sed '/^$/d'
```

6.4 Zum Beispiel:

```
sed -ne '/<Directory>/,/<\/Directory>/p' httpd.conf
```

6.5 Sie werden staunen – es geht, aber es ist bei weitem nicht so simpel wie head. Lesen Sie die Info-Dokumentation zu GNU sed für die Details.

6.6 Versuchen Sie etwas wie

```
sed '/^[ A-Z]\+$/a\  
'
```

6.7 Hierfür ist die *i-j*-Adressierung nützlich:

```
sed '1-2y/abcdefghijklmnopqrstuvwxyz/ABCDEFGHIJKLMNOPQRSTUVWXYZ/'
```

6.8 Zum Beispiel: »sed 's/\<gelb\>/blau/g'«. Denken Sie an den *g*-Modifikator, damit alle gelbs in jeder Zeile ersetzt werden, und an die Wortklammern, damit »Eigelb« nicht zu »Eiblau« wird.

6.9 Versuchen Sie mal »sed -ne '/^[^A]/p; /^A/s/[A-Za-z]\+//p'«.

6.10 Ersetzen Sie zum Beispiel die Zeile

```
mv $out "$f"
```

durch etwas wie

```
mv "$f" "$f".bak  
mv $out "$f"
```

(Wenn schon eine .bak-Datei existiert: Pech!)

6.11 Für die »langen« Optionen müssen Sie erfreulicherweise nichts Besonderes machen, da diese schon durch den existierenden Code abgedeckt sind (wie?). Was »-« angeht, müssen Sie in der case-Fallunterscheidung dafür einen Fall vorsehen, der die Schleife beendet (analog zum Fall »*«). Warum reicht hierfür die »*«-Klausel nicht aus?

6.12 Eine Möglichkeit, die sonst nicht viel ändert, wäre zum Beispiel

```
function question () {  
    if [ "$1" = "get" ]  
    then  
        echo $2  
        return  
    fi  
    case "$2" in  
        display) re='?' ;;  
        correct) re='+' ;;  
        answers) re='[-+]';;  
        next)    re='>' ;;  
        *)      echo >&2 "$0: get: ungültiger Feldtyp $2"; exit 1 ;;  
    esac  
    sed -ne "/question $1/,/end/p" $qfile | sed -ne "/^$re/s/^.//p"  
}
```

7.1 Das genaue Format der Datumsangabe im »ls -l« kann je nach der Sprachumgebung variieren. Das führt mitunter dazu, dass das Datum nur ein Feld ergibt und nicht, wie in diesem Beispiel vorausgesetzt, zwei, so dass der Test auf 9 Felder in der Zeile nicht das gewünschte Resultat liefert (es sind dann ja nur 8). Aus diesem Grund haben wir das Beispiel so geändert, dass wir uns mit »mehr als 2 Felder« zufrieden geben – es geht ja nur darum, die erste Zeile auszusondern. (Was man natürlich auch auf einige Dutzend andere Arten tun könnte; überlegen Sie sich ein paar.)

7.2 Mit dem Programm histcount könnte das so aussehen:

```
$ histcount ~/.bash_history | sort -nr
```

(Natürlich kommen Sie notfalls auch ohne awk aus, etwa mit

```
$ sort ~/.bash_history | uniq -c | sort -nr
```

Warum können Sie hier sort und uniq verwenden und im Falle von histcount nicht?)

7.3 Immer dann, wenn das erste Wort der Kommandozeile nicht das tatsächliche Kommando ist. Zum einen ist es erlaubt, vor das eigentliche Kommando Zuweisungen an Umgebungsvariable zu stellen – denken Sie an etwas wie

```
$ TZ=Asia/Tokyo date
```

– und zum anderen erlaubt die Shell Sachen wie Ein- und Ausgabeumleitung in durchaus ungewöhnlichen Positionen:

```
$ </dev/tty >/tmp/output blafasel
```

7.4 Im Grunde müssen Sie hier Techniken aus dem Shell-History- und dem Doppelte-Wörter-Beispiel kombinieren. Im einfachsten Fall reicht etwas wie

```
#!/usr/bin/awk -f
# countwords -- Wörter zählen

{
    for (i = 1; i <= NF; i++) {
        count[$i]++
    }
}
END {
    for (w in count) {
        print count[w], w
    }
}
```

Dieser simple Ansatz ignoriert allerdings einerseits Groß- und Kleinschreibung, andererseits werden Wörter durch Freiplatz (die FS-Grundeinstellung) getrennt, so dass Interpunktion als Teil eines Worts betrachtet wird. Um das zu vermeiden, können Sie eine tr-Pipeline vorschalten wie im Doppelte-Wörter-Beispiel oder die GNU-awk-Funktionen »tolower« und »gsub« verwenden (siehe hierzu die GNU-awk-Dokumentation).

7.5 Das zweite Feld jeder Zeile ist die Punktzahl des Vereins, das dritte die Tordifferenz. Da die Punktzahl wichtiger ist als die Tordifferenz, empfiehlt sich ein sort-Aufruf der Form

```
sort -t: -k2,2nr -k3,3nr
```

(die Einträge sollen als Zahlen betrachtet werden, und der jeweils höchste Wert sollte vorne stehen). Details siehe sort(1). – Sie könnten das natürlich auch in awk abhandeln (eine Sortierfunktion haben Sie schon gesehen, und zumindest GNU-awk hat auch eine effiziente Sortierfunktion asort eingebaut), aber sort ist oft bequemer, gerade wenn es um kompliziertere Kriterien geht.

7.6 Hier ist ein Lösungsvorschlag (aber Sie hätten selber darauf kommen sollen):

```
#!/usr/bin/awk -f

BEGIN { FS = ":"; OFS = ":" }

{
    zuschauer[$2] += $6
    zuschauer[$3] += $6
}

END {
    for (team in zuschauer) {
        print team, zuschauer[team]
    }
}
```

7.7 Hier empfiehlt sich ein dreistufiger Ansatz. Ein awk-Programm ähnlich dem schon gezeigten bestimmt die unsortierte Tabelle, sort sortiert sie und ein weiteres awk-Programm formatiert sie schön. Das erste awk-Programm, bltab2, unterscheidet sich von dem vorigen hauptsächlich darin, dass es die gewonnenen und verlorenen Spiele zählt:

```
$1 <= tag {
    spiele[$2]++; spiele[$3]++
    tore[$2] += $4 - $5; tore[$3] += $5 - $4
    if ($4 > $5) {
        gewonnen[$2]++; verloren[$3]++
    } else if ($4 < $5) {
        verloren[\$2]++; gewonnen[\$3]++
    }
}
```

Die Punkte lassen sich dann leicht bei der Ausgabe berechnen:

```
END {
    for (team in spiele) {
        u = spiele[team] - gewonnen[team] - verloren[team]
        punkte = 3*gewonnen[team] + u
        print team, spiele[team], gewonnen[team], u,
            verloren[team], punkte, tore[team]
    }
}
```

Das awk-Skript für die Ausgabe – nennen wir es blfmt – könnte dann ungefähr so aussehen:

```
#!/usr/bin/awk -f
# blfmt -- Bundesliga-Tabelle formatiert ausgeben

BEGIN {
    FS = ":"
    print "PL MANNSCHAFT          SP  G  U  V  PUNKTE TORE"
    print "-----"
}

{
    printf "%2d %-25.25s %2d %2d %2d %2d    %3d  %3d\n",
        ++i, $1, $2, $3, $4, $5, $6, $7
}
}
```

Aufgerufen wird das Ganze über eine Pipeline der Form

```
bltab2 tag=3 bl03.txt | sort -t: -k6,6rn -k7,7rn | blfmt
```

die Sie natürlich in ein Shellskript packen können, um die Aufgabe perfekt zu lösen.

7.8 Sorgen Sie dafür, dass der 1. FCK zwei Punkte abgezogen bekommt, sinnvollerweise zwischen der Berechnung und dem Sortieren:

```
bltab2 tag=3 bl03.txt | awk -f '{
    if ($1 == "1.FC Kaiserslautern") {
        $6 -= 2
    }
    print
}' | sort -t: -k6,6rn -k7,7rn | blfmt
```

7.9 Wie viele der hier gezeigten Programme besteht auch `gdu` aus einem »Datensammelteil« und einem »Ausgabeteil«. Hier ist der Datensammelteil; wir lesen die Eingabe ein und merken uns den Platzverbrauch pro Benutzer sowie den bisher gesehenen größten Platzverbrauch – letzteren brauchen wir zur Skalierung der Ausgabe.

```
{
    sub(/^\.*/ //, "", \ $2)
    space[$2] = $1
    if (\ $1 > max) {
        max = $1
    }
}
}
```

Zum Testen tut es vielleicht zuerst ein Ausgabeteil, der die Ergebnisse in numerischer Form liefert:

```
END {
    for (u in space) {
        printf "%-10.10s %f\n", u, 60*space[u]/max
    }
}
```

Wenn Sie sich überzeugt haben, dass die Zahlenwerte vernünftig sind, können Sie es mit der grafischen Darstellung versuchen:

```

END {
  stars = "*****"
  stars = stars stars
  for (u in space) {
    n = int(60*space[u]/max + 0.5)
    printf "%-10.10s %-60s\n", u, substr(stars, 0, n)
  }
}

```

8.1 Andere Besatzungsmitglieder könnten Sie ebenfalls in die Tabelle »Person« aufnehmen und eine Spalte »Position« (o. ä.) hinzufügen, die Einträge wie »Kommandant«, »1. Offizier« und so weiter enthält. Wenn Sie sauber arbeiten wollen, dann ist diese Spalte in der Tabelle »Person« natürlich ein Fremdschlüssel, der auf eine Tabelle »Position« verweist, damit Sie später leicht alle 1. Offiziere finden können.

8.2 Den Regisseur können Sie analog behandeln wie in der vorigen Aufgabe, wobei Sie sinnvollerweise eine neue Tabelle einführen, um die Rollen im Film und die tatsächlich existierenden Menschen nicht durcheinander zu bringen. Auch hier tun Sie wahrscheinlich gut daran, sich nicht nur auf die Regisseure zu beschränken, sondern gleich eine Tabelle »Funktion« (oder so) vorzusehen, um außer den Regisseuren auch noch für Drehbuchautoren, Oberelektriker und wen man sonst noch so in einer Filmcrew antrifft sorgen zu können. Wenn Sie besonders trickreich sind, denken Sie an die Funktion »Darsteller« und sehen einen Fremdschlüssel vor, der auf die Tabelle »Person« verweist und bei Nicht-Schauspielern NULL ist.

8.8 Versuchen Sie es mit etwas wie

```

sqlite> SELECT title FROM film
...> WHERE jahr < 1985 AND budget < 40

```

(Sie können in einer WHERE-Klausel mehrere Ausdrücke mit AND, OR oder NOT zusammenschließen.)

8.9 Wenn Sie keine WHERE-Klausel und kein explizites JOIN haben, dann produziert ein SELECT über mehrere Tabellen das kartesische Produkt aller Tupel in diesen Tabellen, also etwas wie

```

1|James T.|Kirk|1|1|USS Enterprise
1|James T.|Kirk|1|2|USS Enterprise
1|James T.|Kirk|1|3|Millennium Falcon
<<<<<<
2|Willard|Decker|1|1|USS Enterprise
2|Willard|Decker|1|2|USS Enterprise
2|Willard|Decker|1|3|Millennium Falcon
<<<<<<

```

9.1 (a) Am 1. März um 17 Uhr; (b) Am 2. März um 14 Uhr; (c) Am 2. März um 16 Uhr; (d) Am 2. März um 1 Uhr.

9.2 Etwa mit »at now + 3 minutes«.

9.4 Eine Möglichkeit wäre »atq | sort -bk 2«.

9.6 Die Aufgabenlisten-Datei gehört Ihnen, aber Sie haben kein Zugriffsrecht auf das crontabs-Verzeichnis. Bei Debian GNU/Linux gilt zum Beispiel

```
$ ls -ld /var/spool/cron/crontabs
drwx-wx--T 2 root crontab 4096 Aug 31 01:03 /var/spool/cron/crontabs
```

Das heißt, root hat (wie üblich) vollen Zugriff (hätte er sowieso), und Mitglieder der Gruppe crontab dürfen zumindest auf Dateien zugreifen, von denen sie wissen, wie sie heißen (ls ist nicht erlaubt). Das Programm crontab ist Set-GID crontab:

```
$ ls -l $(which crontab)
-rwxr-sr-x 1 root crontab 27724 Sep 28 11:33 /usr/bin/crontab
```

und wird damit mit den Rechten der Gruppe crontab ausgeführt, egal welcher Benutzer es aufruft. (Der Set-GID-Mechanismus ist in der Schulungsunterlage *Linux-Administration I* genauer erklärt.)

9.7 Registrieren Sie die Aufgabe für den Monats-Dreizehnten und fragen Sie im aufgerufenen Skript ab (etwa mit »date +%u«), ob gerade Freitag ist.

9.8 Die Details sind distributionsabhängig.

9.9 Die passende Zeile für das minütliche Protokoll ist etwas wie

```
* * * * * /bin/date >>/tmp/date.log
```

Für den Zwei-Minuten-Abstand können Sie natürlich etwas schreiben wie

```
0,2,4,<KKKK,56,58 * * * * /bin/date >>/tmp/date.log
```

aber

```
*/2 * * * * /bin/date >>/tmp/date.log
```

ist bequemer.

9.10 Die Kommandos dafür sind »crontab -l« bzw. »crontab -r«.

9.11 Sie sollten hugo in /etc/cron.deny (bei SUSE-Distributionen /var/spool/cron/deny) eintragen bzw. ihn aus /etc/cron.allow löschen.

9.13 In /etc/cron.daily steht ein Skript namens 0anacron, das als erster Job ausgeführt wird. Dieses Skript ruft »anacron -u« auf; mit dieser Option aktualisiert anacron die Zeitstempel, ohne tatsächlich Jobs auszuführen (denn das macht cron dann als nächstes). Wenn das System neu gestartet wird, vermeidet anacron so überflüssige Job-Ausführungen, jedenfalls sofern der Neustart erst erfolgt, nachdem cron sein Ding getan hat.

10.4 Es gibt Systeme, bei denen /usr auf einer anderen Partition oder (im Falle von Thin Clients) auf einem anderen Rechner liegt. Die Systemzeit ist aber so wichtig, dass sie schon früh beim Systemstart korrekt zur Verfügung stehen sollte, selbst wenn es gruselige Probleme gibt. Eine Kopie der betreffenden Datei (groß sind die Zeitzeileninformationen in aller Regel nicht) ist also der sichere Ansatz.

10.5 Die einfachste Möglichkeit ist wahrscheinlich `zdump` in Kombination mit `watch`: Probieren Sie etwas wie

```
$ ZONES=Asia/Tokyo Europe/Berlin America/New_York
$ watch -t zdump $ZONES
```

(brechen Sie das Programm mit `Strg`+`C` ab, wenn Sie genug davon haben). In einer grafischen Umgebung bietet sich natürlich auch etwas wie

```
$ for z in $ZONES; do
> TZ=$z xclock -title $z#/ -update 1 &
> done
```

an.

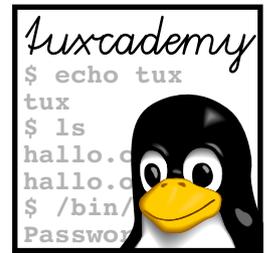
11.3 Der Displayname adressiert den X-Server Nr. 1 (mutmaßlich der zweite) auf dem Rechner `bla.example.com`, und zwar insbesondere den zweiten Bildschirm, den dieser Server kontrolliert. Eine mögliche Kommandozeile wäre

```
# xterm -display bla.example.com:1.1
```

11.5 Die VNC-Methode ist wesentlich einfacher zu implementieren, da das X-Protokoll ziemlich umfangreich und aufwendig ist. Ein Protokoll für »entfernte Framebuffer«, wie VNC es verwendet, hat im Vergleich sehr wenige Operationen und ist deutlich überschaubarer, wie die Existenz von VNC-Clients belegt, die in ein paar hundert Zeilen einer Programmiersprache wie Tcl geschrieben sind. Daraus resultiert eine *de facto* größere Anzahl von Plattformen, auf denen VNC zur Verfügung steht, bis hin zu PDAs und ähnlichen Geräten mit sehr wenigen Ressourcen. – Dafür erlaubt das X-Protokoll natürlich verschiedene Optimierungen, die man nur ausnutzen kann, wenn man weiß, *was* da gerade gezeichnet wird. Allgemein läßt die Effizienzfrage sich nur schwer beantworten, da es sehr darauf ankommt, was man gerade macht: Elektronische Bildverarbeitung zum Beispiel ist mit X11 ziemlich aufwendig, da sie größtenteils im X-Client stattfindet und große Datenmengen zum Server transportiert werden müssen (spezielle Kommunikationsmechanismen machen das für den Fall erträglich, wo Server und Client auf demselben Rechner laufen). Für VNC gibt es da *a priori* keinen großen Unterschied zum normalen Systembetrieb, da in jedem Fall nur Pixeldaten übertragen werden. X11 ist da im Vorteil, wo sich große Änderungen der Anzeige durch wenige X-Protokollkommandos beschreiben lassen.

11.6 Versuchen Sie etwas wie

```
$ xclock -geometry 150x150+50-50
```



B

Reguläre Ausdrücke

B.1 Überblick

Reguläre Ausdrücke sind ein zentrales Konzept für Shellprogrammierung und den Umgang mit Programmen wie `sed` und `awk`. Zur Illustration hier eine angepasste Einführung zum Thema aus *Linux-Grundlagen für Anwender und Administratoren*:

Reguläre Ausdrücke werden gerne »rekursiv« aus Bausteinen aufgebaut, die selbst als reguläre Ausdrücke gelten. Die einfachsten regulären Ausdrücke sind Buchstaben, Ziffern und viele andere Zeichen des üblichen Zeichenvorrats, die für sich selber stehen. »a« wäre also ein regulärer Ausdruck, der auf das Zeichen »a« passt; der reguläre Ausdruck »abc« passt auf die Zeichenkette »abc«. Ähnlich wie bei Shell-Suchmustern gibt es die Möglichkeit, Zeichenklassen zu definieren; der reguläre Ausdruck »[a-e]« passt also auf genau eines der Zeichen »a« bis »e«, und »a[xy]b« passt entweder auf »axb« oder »ayb«. Wie bei der Shell können Bereiche gebildet und zusammengefasst werden – »[A-Za-z]« passt auf alle Groß- und Kleinbuchstaben (ohne Umlaute) –, nur die Komplementbildung funktioniert etwas anders: »[^abc]« passt auf alle Zeichen *außer* »a«, »b« und »c«. (Bei der Shell hieß das »[!abc]«.) Der Punkt ».« entspricht dem Fragezeichen in Shellsuchmustern, steht also für ein einziges beliebiges Zeichen – ausgenommen davon ist nur der Zeilentrenner »\n«: »a.c« passt also auf »abc«, »a/c« und so weiter, aber nicht auf die mehrzeilige Konstruktion

Zeichen

Zeichenklassen

Komplement

```
a
c
```

Der Grund dafür besteht darin, dass die meisten Programme zeilenorientiert vorgehen und »zeilenübergreifende« Konstruktionen schwieriger zu verarbeiten wären. (Was nicht heißen soll, dass es nicht manchmal nett wäre, das zu können.)

Während Shellsuchmuster immer vom Anfang eines Pfadnamens aus passen müssen, reicht es bei Programmen, die Zeilen aufgrund von regulären Ausdrücken auswählen, meist aus, wenn der reguläre Ausdruck irgendwo in einer Zeile passt. Allerdings können Sie diese Freizügigkeit einschränken: Ein regulärer Ausdruck, der mit dem Zirkonflex (»^«) anfängt, passt nur am Zeilenanfang, und ein regulärer Ausdruck, der mit dem Dollarzeichen (»\$«) aufhört, entsprechend nur am Zeilenende. Der Zeilentrenner am Ende jeder Zeile wird ignoriert, so dass Sie »xyz\$« schreiben können, um alle Zeilen auszuwählen, die auf »xyz« enden, und nicht »xyz\n\$« angeben müssen.

Zeilenanfang

Zeilenende



Strenggenommen passen »^« und »\$« auf gedachte »unsichtbare« Zeichen am Zeilenanfang bzw. unmittelbar vor dem Zeilentrenner am Zeilenende.

- Wiederholung Schließlich können Sie mit dem Stern (`*`) angeben, dass der davorstehende Ausdruck beliebig oft wiederholt werden kann (auch gar nicht). Der Stern selbst steht nicht für irgendwelche Zeichen in der Eingabe, sondern modifiziert nur das Davorstehende – das Shellsuchmuster `a*.txt` entspricht also dem regulären Ausdruck `^a.*\.txt` (denken Sie an die »Verankerung« des Ausdrucks am Anfang der Eingabe und daran, dass ein einzelner Punkt auf alle Zeichen passt).
- Vorrang Wiederholung hat Vorrang vor Aneinanderreihung; `ab*` ist ein einzelnes `a` gefolgt von beliebig vielen `b` (auch keinem), nicht eine beliebig häufige Wiederholung von `ab`.

B.2 Extras

- Erweiterungen Die Beschreibung aus dem vorigen Abschnitt gilt für praktisch alle Linux-Programme, die reguläre Ausdrücke verarbeiten. Diverse Programme unterstützen aber auch verschiedene Erweiterungen, die entweder Schreibvereinfachungen bieten oder tatsächlich zusätzliche Dinge erlauben. Die »Krone der Schöpfung« markieren hier die modernen Skriptsprachen wie Tcl, Perl oder Python, deren Implementierungen inzwischen weit über das hinausgehen, was reguläre Ausdrücke im Sinne der theoretischen Informatik können.

Einige gängige Erweiterungen sind:

Wortklammern Die Sequenz `\<` passt am Anfang eines Worts (soll heißen, an einer Stelle, wo ein Nichtbuchstabe auf einen Buchstaben stößt). Analog passt `\>` am Ende eines Worts (da, wo ein Buchstabe von einem Nichtbuchstaben gefolgt wird).

Gruppierung Runde Klammern `(...)` erlauben es, Aneinanderreihungen von regulären Ausdrücken zu wiederholen: `a(bc)*` passt auf ein `a` gefolgt von beliebig vielen Wiederholungen von `bc`.

Alternative Mit dem vertikalen Balken `|` können Sie eine Auswahl zwischen mehreren regulären Ausdrücken treffen: Der reguläre Ausdruck `Affen(brot|schwanz|kletter)baum` passt genau auf eine der Zeichenketten `Affenbrotbaum1`, `Affenschwanzbaum2` oder `Affenkletterbaum`.

Optionales Das Fragezeichen `?` macht den vorstehenden regulären Ausdruck optional, das heißt, er tritt entweder einmal auf oder gar nicht. `Boot(smnn)?` passt auf `Boot` oder `Bootsmann`.

Mindestens einmal Vorhandenes Das Pluszeichen `+` entspricht dem Wiederholungsoperator `*`, bis darauf, dass der vorstehende Ausdruck mindestens einmal auftreten muss, also nicht ganz entfallen darf.

Bestimmte Anzahl von Wiederholungen Sie können in geschweiften Klammern eine Mindest- und eine Maximalanzahl von Wiederholungen angegeben werden: `ab{2,4}` passt auf `abb`, `abbb` und `abbbb`, aber nicht auf `ab` oder `abbbbb`. Sie können sowohl die Mindest- als auch die Maximalanzahl weglassen; fehlt die Mindestanzahl, wird 0, fehlt die Maximalanzahl, so wird »Unendlich« angenommen.

Rückbezug Mit einem Ausdruck der Form `\n` können Sie eine Wiederholung desjenigen Teils in der Eingabe verlangen, der auf den Klammerausdruck Nr. *n* im regulären Ausdruck gepasst hat. `(ab)\1` zum Beispiel passt auf `abab`. Weitere Details finden Sie in der Dokumentation zu GNU `grep`.

»Bescheidene« Vorgehensweise Die Operatoren `*`, `+` und `?` sind normalerweise »gierig«, das heißt, sie versuchen auf so viel der Eingabe zu passen wie möglich: `^a.*` bezogen auf die Eingabe `abacada` passt auf `abacada`,

¹*Adansonia digitata*

²Volkstümlicher Name für die Araukarie (*Araucaria araucana*)

Tabelle B.1: Unterstützung von regulären Ausdrücken

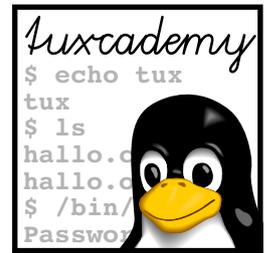
Erweiterung	GNU grep	GNU egrep	trad. egrep	sed	awk	Perl	Tcl
Wortklammern	•	•	•	•1	•1	•2	•2
Gruppierung	•1	•	•	•1	•	•	•
Alternative	•1	•	•	•	•	•	•
Optionales	•1	•	•	•1	•	•	•
mind. einmal	•1	•	•	•1	•	•	•
Anzahl	•1	•	◦	•1	•1	•	•
Rückbezug	◦	•	•	•	◦	•	•
Bescheiden	◦	◦	◦	◦	◦	•	•

•: wird unterstützt; ◦: wird nicht unterstützt

Anmerkungen: 1. Wird durch einen vorgesetzten Rückstrich (»\«) angesprochen, also z. B. »ab\+« statt »ab+«. 2. Ganz andere Syntax (siehe Dokumentation).

nicht »aba« oder »abaca«. Es gibt aber entsprechende »bescheidene« Versionen »*?«, »+?« und »??«, die auf so wenig der Eingabe passen wie möglich. In unserem Beispiel würde »^a.*?a« auf »aba« passen. Auch die geschweiften Klammern haben möglicherweise eine bescheidene Version.

Nicht jedes Programm unterstützt jede Erweiterung. Tabelle B.1 zeigt eine Übersicht der wichtigsten Programme. Insbesondere Perl und Tcl unterstützen noch jede Menge hier nicht diskutierte Erweiterungen.



C

LPIC-1-Zertifizierung

C.1 Überblick

Das *Linux Professional Institute* (LPI) ist eine herstellerunabhängige, nicht profitorientierte Organisation, die sich der Förderung des professionellen Einsatzes von Linux widmet. Ein Aspekt der Arbeit des LPI ist die Erstellung und Durchführung weltweit anerkannter, distributionsunabhängiger Zertifizierungsprüfungen beispielsweise für Linux-Systemadministratoren.

Mit der „LPIC-1“-Zertifizierung des LPI können Sie nachweisen, dass Sie über grundlegende Linux-Kenntnisse verfügen, wie sie etwa für Systemadministratoren, Entwickler, Berater oder Mitarbeiter bei der Anwenderunterstützung sinnvoll sind. Die Zertifizierung richtet sich an Kandidaten mit etwa 1 bis 3 Jahren Erfahrung und besteht aus zwei Prüfungen, LPI-101 und LPI-102. Diese werden in Form von computerorientierten Multiple-Choice- und Kurzantworttests über die Prüfungszentren von Pearson VUE und Thomson Prometric angeboten oder können auf Veranstaltungen wie dem LinuxTag oder der CeBIT zu vergünstigten Preisen auf Papier abgelegt werden. Das LPI veröffentlicht auf seinen Web-Seiten unter <http://www.lpi.org/> die **Prüfungsziele**, die den Inhalt der Prüfungen umreißen.

Prüfungsziele

Die vorliegende Unterlage ist Teil eines Kurskonzepts der Linup Front GmbH zur Vorbereitung auf die Prüfung LPI-101 und deckt damit einen Teil der offiziellen Prüfungsziele ab. Details können Sie den folgenden Tabellen entnehmen. Eine wichtige Beobachtung in diesem Zusammenhang ist, dass die LPIC-1-Prüfungsziele nicht dazu geeignet oder vorgesehen sind, einen Einführungskurs in Linux didaktisch zu strukturieren. Aus diesem Grund verfolgt unser Kurskonzept keine strikte Ausrichtung auf die Prüfungen oder Prüfungsziele in der Form „Belegen Sie Kurs x und y , machen Sie Prüfung p , dann belegen Sie Kurs a und b und machen Sie Prüfung q “. Ein solcher Ansatz verleitet viele Kurs-Interessenten zu der Annahme, sie könnten als absolute Linux-Einsteiger n Kurstage absolvieren (mit möglichst minimalem n) und wären anschließend fit für die LPIC-1-Prüfungen. Die Erfahrung lehrt, dass das in der Praxis nicht funktioniert, da die LPI-Prüfungen geschickt so angelegt sind, dass Intensivkurse und prüfungsorientiertes „Büffeln“ nicht wirklich helfen.

Entsprechend ist unser Kurskonzept darauf ausgerichtet, Ihnen in didaktisch sinnvoller Form ein solides Linux-Basiswissen zu vermitteln und Sie als Teilnehmer in die Lage zu versetzen, selbständig mit dem System zu arbeiten. Die LPIC-1-Zertifizierung ist nicht primäres Ziel oder Selbstzweck, sondern natürliche Folge aus Ihren neuerworbenen Kenntnissen und Ihrer Erfahrung.

C.2 Prüfung LPI-102

Die folgende Tabelle zeigt die Prüfungsziele der Prüfung LPI-102 (Version 4.0) und die Unterlagen, die diese Prüfungsziele abdecken. Die Zahlen in den Spalten für die einzelnen Unterlagen verweisen auf die Kapitel, die das entsprechende Material enthalten.

Nr	Gew	Titel	ADM1	GRD2	ADM2
105.1	4	Die Shell-Umgebung anpassen und verwenden	–	1–2	–
105.2	4	Einfache Skripte anpassen oder schreiben	–	2–5	–
105.3	2	SQL-Datenverwaltung	–	8	–
106.1	2	X11 installieren und konfigurieren	–	11	–
106.2	1	Einen Display-Manager einrichten	–	11	–
106.3	1	Hilfen für Behinderte	–	12	–
107.1	5	Benutzer- und Gruppenkonten und dazugehörige Systemdateien verwalten	2	–	–
107.2	4	Systemadministrationsaufgaben durch Einplanen von Jobs automatisieren	–	9	–
107.3	3	Lokalisierung und Internationalisierung	–	10	–
108.1	3	Die Systemzeit verwalten	–	–	8
108.2	3	Systemprotokollierung	–	–	1–2
108.3	3	Grundlagen von Mail Transfer Agents (MTAs)	–	–	11
108.4	2	Drucker und Druckvorgänge verwalten	–	–	9
109.1	4	Grundlagen von Internet-Protokollen	–	–	3–4
109.2	4	Grundlegende Netz-Konfiguration	–	–	4–5, 7
109.3	4	Grundlegende Netz-Fehlersuche	–	–	4–5, 7
109.4	2	Clientseitiges DNS konfigurieren	–	–	4
110.1	3	Administrationsaufgaben für Sicherheit durchführen	2	–	4–5, 13
110.2	3	Einen Rechner absichern	2	–	4, 6–7, 13
110.3	3	Daten durch Verschlüsselung schützen	–	–	10, 12

C.3 LPI-Prüfungsziele in dieser Schulungsunterlage

105.1 Die Shell-Umgebung anpassen und verwenden

Gewicht 4

Beschreibung Kandidaten sollten in der Lage sein, Shellumgebungen gemäß der Anforderungen von Benutzern anzupassen. Kandidaten sollten in der Lage sein, globale Voreinstellungen und die von Benutzern zu ändern.

Wichtigste Wissensgebiete

- Umgebungsvariable (etwa PATH) beim Anmelden oder Erzeugen einer neuen Shell setzen
- Bash-Funktionen für häufig gebrauchte Kommandofolgen erstellen
- Skelett-Verzeichnisse für neue Benutzerkonten warten
- Den Kommando-Suchpfad mit den richtigen Verzeichnissen setzen

Hier ist eine auszugsweise Liste der verwendeten Dateien, Begriffe und Hilfsprogramme:

- | | | |
|--------------------|-------------------|------------------|
| • . | • set | • ~/.bash_logout |
| • source | • unset | • function |
| • /etc/bash.bashrc | • ~/.bash_profile | • alias |
| • /etc/profile | • ~/.bash_login | • lists |
| • env | • ~/.profile | |
| • export | • ~/.bashrc | |

105.2 Einfache Skripte anpassen oder schreiben

Gewicht 4

Beschreibung Kandidaten sollten in der Lage sein, existierende Skripte anzupassen oder einfache neue Bash-Skripte zu schreiben.

Wichtigste Wissensgebiete

- Standard-sh-Syntax verwenden (Schleifen, Fallunterscheidungen)
- Kommandosubstitution verwenden
- Rückgabewerte auf Erfolg, Misserfolg oder andere von einem Programm gelieferte Informationen prüfen
- Bedingt Mail an den Systemadministrator schicken
- Den richtigen Skript-Interpreter über die Shebang-Zeile (!) wählen
- Den Ort, die Eigentümerschaft, die Ausführungs- und SUID-Rechte von Skripten verwalten

Hier ist eine auszugsweise Liste der verwendeten Dateien, Begriffe und Hilfsprogramme:

- for
- while
- test
- if
- read
- seq
- exec

105.3 SQL-Datenverwaltung

Gewicht 2

Beschreibung Kandidaten sollten in der Lage sein, mit einfachen SQL-Kommandos Datenbanken abzufragen und Daten zu manipulieren. Dieses Lernziel umfasst auch Anfragen, die 2 Tabellen verbinden und/oder Subselects verwenden.

Wichtigste Wissensgebiete

- Gebrauch einfacher SQL-Kommandos
- Einfache Datenmanipulation

Hier ist eine auszugsweise Liste der verwendeten Dateien, Begriffe und Hilfsprogramme:

- insert
- update
- select
- delete
- from
- where
- group by
- order by
- join

106.1 X11 installieren und konfigurieren

Gewicht 2

Beschreibung Kandidaten sollten in der Lage sein, X11 zu installieren und zu konfigurieren.

Wichtigste Wissensgebiete

- Sicherstellen, dass die Videokarte und der Monitor von einem X-Server unterstützt werden
- Wissen von der Existenz des X-Fontservers
- Die X-Window-Konfigurationsdatei prinzipiell verstehen und kennen

Hier ist eine auszugsweise Liste der verwendeten Dateien, Begriffe und Hilfsprogramme:

- /etc/X11/xorg.conf
- DISPLAY
- xdpinfo
- xhost
- xwininfo
- X

106.2 Einen Display-Manager einrichten

Gewicht 1

Beschreibung Kandidaten sollten in der Lage sein, die grundlegenden Eigenschaften und die Konfiguration des LightDM-Display-Managers zu beschreiben. Dieses Prüfungsziel umfasst Wissen von der Existenz der Display-Manager XDM (X Display Manager), GDM (Gnome Display Manager) und KDM (KDE Display Manager).

Wichtigste Wissensgebiete

- Grundlegende Konfiguration von LightDM
- Den Display-Manager ein- und ausschalten
- Die Begrüßung des Display-Managers ändern
- Display-Manager für den Gebrauch durch X-Terminals konfigurieren

Hier ist eine auszugsweise Liste der verwendeten Dateien, Begriffe und Hilfsprogramme:

- lightdm
- /etc/lightdm/

106.3 Hilfen für Behinderte

Gewicht 1

Beschreibung Wissen um die Existenz von Hilfen für Behinderte.

Wichtigste Wissensgebiete

- Tastatureinstellungen für Behinderte (AccessX)
- Visuelle Einstellungen und Themen
- Assistive Techniken (ATs)

Hier ist eine auszugsweise Liste der verwendeten Dateien, Begriffe und Hilfsprogramme:

- »Klebrige« und mit hohem Kontrast oder großer Bildschirm
- Wiederholungstasten
- Schrift
- Gesten (beim Anmelden, etwa bei gdm)
- Langsame/Bounce/Umschalt-
- Screen-Reader
- Braille-Anzeige
- Orca
- Tasten
- Bildschirmvergrößerung
- GOK
- Maustasten
- Tastatur auf dem
- emacspeak
- Desktop-Themen

107.2 Systemadministrationsaufgaben durch Einplanen von Jobs automatisieren

Gewicht 4

Beschreibung Kandidaten sollten in der Lage sein, cron oder anacron zu verwenden, um Jobs in regelmäßigen Abständen auszuführen, und at, um Jobs zu einem bestimmten Zeitpunkt auszuführen.

Wichtigste Wissensgebiete

- Cron- und At-Jobs verwalten
- Benutzerzugang zu den Diensten cron und at konfigurieren
- anacron konfigurieren

Hier ist eine auszugsweise Liste der verwendeten Dateien, Begriffe und Hilfsprogramme:

- /etc/cron.{d,daily,hourly,monthly,weekly,yearly}/deny
- /etc/at.deny
- /etc/at.allow
- /etc/crontab
- /etc/cron.allow
- /var/spool/cron/
- crontab
- at
- atq
- atrm
- anacron
- /etc/anacrontab

107.3 Lokalisierung und Internationalisierung

Gewicht 3

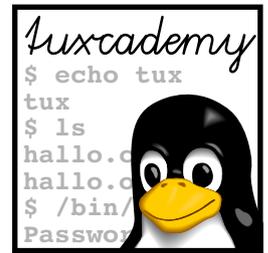
Beschreibung Kandidaten sollten in der Lage sein, ein System in einer anderen Sprache als Englisch zu lokalisieren. Dazu gehört auch ein Verständnis dafür, warum LANG=C in Shellskripten nützlich ist.

Wichtigste Wissensgebiete

- Locale-Einstellungen und -Umgebungsvariable konfigurieren
- Zeitzonen-Einstellungen und -Umgebungsvariable konfigurieren

Hier ist eine auszugsweise Liste der verwendeten Dateien, Begriffe und Hilfsprogramme:

- /etc/timezone
- /etc/localtime
- /usr/share/zoneinfo/
- LC_*
- LC_ALL
- LANG
- TZ
- /usr/bin/locale
- tzselect
- tzconfig
- date
- iconv
- UTF-8
- ISO-8859
- ASCII
- Unicode



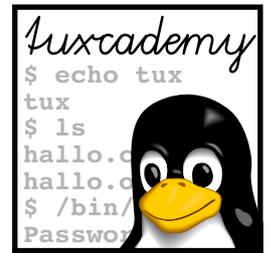
D

Kommando-Index

Dieser Anhang fasst alle im Text erklärten Kommandos zusammen und verweist auf deren Dokumentation sowie die Stellen im Text, wo die Kommandos eingeführt werden.

X	Startet den passenden X-Server für das System	x(1)	182
anacron	Führt periodische Jobs aus, wenn der Computer nicht immer läuft	anacron(8)	148
ash	Eine leichtgewichtige POSIX-konforme Shell	ash(1)	16
at	Registriert Kommandos zur zeitversetzten Ausführung	at(1)	142
atd	Daemon für die zeitversetzte Ausführung von Kommandos über at	atd(8)	144
atq	Programm zur Abfrage der Warteschlangen für zeitversetzte Kommandoausführung	atq(1)	144
atrm	Storniert zeitversetzt auszuführende Kommandos	atrm(1)	144
awk	Programmiersprache für Textverarbeitung und Systemverwaltung	awk(1)	106
busybox	Eine Shell, die Versionen vieler Unix-Werkzeuge integriert hat	busybox(1)	16
chmod	Setzt Rechte für Dateien und Verzeichnisse	chmod(1)	24
chsh	Ändert die Login-Shell eines Benutzers	chsh(1)	14
cmp	Vergleicht zwei Dateien byteweise	cmp(1)	101
crontab	Programm zur Verwaltung von regelmäßig auszuführenden Kommandos	crontab(1)	148
dash	Eine leichtgewichtige POSIX-konforme Shell	dash(1)	16
dialog	Erlaubt „grafische“ Interaktionselemente auf einem Textbildschirm	dialog(1)	84
env	Gibt die Prozessumgebung aus oder startet Programme mit veränderter Umgebung	env(1)	35
exec	Startet ein neues Programm im aktuellen Shell-Prozess	bash(1)	55
export	Definiert und verwaltet Umgebungsvariable	bash(1)	35
file	Rät den Typ einer Datei anhand des Inhalts	file(1)	14
find	Sucht nach Dateien, die bestimmte Kriterien erfüllen	find(1), Info: find	14
for	Shell-Kommando für Schleifen über eine Liste von Elementen	bash(1)	49
iconv	Konvertiert zwischen Zeichencodierungen	iconv(1)	157
kdialog	Erlaubt Benutzung von KDE-Interaktionselementen von Shellskripten aus	kdialog(1)	90
locale	Zeigt Informationen über die Lokalisierung an	locale(1)	161, 162
localedef	Übersetzt Locale-Definitionsdateien	localedef(1)	162

logrotate	Verwaltet, kürzt und „rotiert“ Protokolldateien	logrotate(8)	66
lspci	Gibt Informationen über Geräte auf dem PCI-Bus aus	lspci(8)	179
mkfifo	Legt FIFOs (benannte Pipes) an	mkfifo(1)	71
mktemp	Erzeugt einen eindeutigen temporären Dateinamen (sicher)	mktemp(1)	100
printf	Gibt Zahlen und Zeichenketten formatiert aus	printf(1), bash(1)	73
sash	„Stand-Alone Shell“ mit eingebauten Kommandos, zur Problembehebung	sash(8)	16
seq	Erzeugt Folgen von Zahlen auf der Standardausgabe	seq(1)	71
set	Verwaltet Shellvariable	bash(1)	34
strace	Protokolliert die Systemaufrufe eines Programms	strace(1)	19
test	Wertet logische Ausdrücke auf der Kommandozeile aus	test(1), bash(1)	44
timeconfig	[Red Hat] Erlaubt die bequeme Festlegung der systemweiten Zeitzone	timeconfig(8)	166
tr	Tauscht Zeichen in der Standardeingabe gegen andere aus oder löscht sie	tr(1)	73
tzselect	Erlaubt eine bequeme interaktive Wahl der Zeitzone	tzselect(1)	166
unbuffer	Unterdrückt Ausgabepufferung eines Programms (Bestandteil des expect-Pakets)	unbuffer(1)	72
uniq	Ersetzt Folgen von gleichen Zeilen in der Eingabe durch die erste solche	uniq(1)	113
unset	Löscht Shell- oder Umgebungsvariable	bash(1)	36
xauth	Verwaltet den Zugriff auf den X-Server über <i>magic cookies</i>	xauth(1)	191
xdpyinfo	Zeigt Informationen über das aktive X-Display	xdpyinfo(1)	186
xhost	Erlaubt Clients auf anderen Rechnern den Zugriff auf den X-Server über TCP	xhost(1)	191
xkbset	Erlaubt die Verwaltung von Tastatureinstellungen für X11	xkbset(1)	194
xmessage	Zeigt eine Nachricht oder Anfrage in einem X11-Fenster an	xmessage(1)	90
xwininfo	Zeigt Informationen über ein X-Fenster an	xwininfo(1)	187
zdump	Gibt die aktuelle Zeit oder die Zeitzonendefinitionen für verschiedene Zeitzonen aus	zdump(1)	165
zic	Compiler für Zeitzonen-Dateien	zic(8)	165



Index

Dieser Index verweist auf die wichtigsten Stichwörter in der Schulungsunterlage. Besonders wichtige Stellen für die einzelnen Stichwörter sind durch **fette** Seitenzahlen gekennzeichnet. Sortiert wird nur nach den Buchstaben im Indexeintrag; „~/ .bashrc“ wird also unter „B“ eingeordnet.

- != (awk-Operator), 110
- !~ (awk-Operator), 110
- # (Shellvariable), 61
- \$ (awk-Operator), 108, 110
- \$ (Shellvariable), 64, 100
- && (awk-Operator), 109
- * (awk-Operator), 110
- * (Shellvariable), 39–40, 87, 200
- + (awk-Operator), 110
- (awk-Operator), 110
- . (Shellkommando), 25
- / (awk-Operator), 110
- < (awk-Operator), 110
- <= (awk-Operator), 110
- = (awk-Operator), 109
- == (awk-Operator), 109–110
- > (awk-Operator), 110
- >= (awk-Operator), 110
- ? (Shellvariable), 37, 43, 201
- @ (Shellvariable), 39–40, 42, 87, 200
- ^ (awk-Operator), 110
- _ (Umgebungsvariable), 143
- ~ (awk-Operator), 110
- Ø (Shellvariable), 61–62
- 1 (Shellvariable), 39, 50
- 2 (Shellvariable), 39, 50

- Abstraktion, **82**
- Aho, Alfred V., 106
- Alias, **17**
- alias (Shellkommando), 17
- Alk, 106
- anacron, 141, 148–150, 211
 - s (Option), 150
 - u (Option), 211
- Arithmetische Expansion, 40
- ash, 16
- asort (awk-Funktion), 207
- at, 11, 141–145, 147
 - c (Option), 144
 - f (Option), 143
 - q (Option), 144
- AT&T, 106
- atd, 144–145
 - b (Option), 144
 - d (Option), 144
 - l (Option), 144
- atq, 144
 - q (Option), 144
- atrm, 144
- awk, 11, 26, 28, 40, 94, 105–120, 139, 163, 207–208, 213, 215
 - F (Option), 108, 110
 - f (Option), 107
- awk-Funktion
 - asort, 207
 - close, 119
 - getline, 119
 - gsub, 207
 - int, 111
 - length, 111
 - log, 111
 - sqrt, 111
 - sub, 111, 118
 - substr, 111
 - tolower, 207
- awk-Kommando
 - for, 111–113
 - if, 114
 - print, 114, 117
 - printf, 115, 120
 - return, 112
- awk-Operator
 - !=, 110
 - !~, 110
 - \$, 108, 110
 - &&, 109
 - *, 110
 - +, 110
 - , 110
 - /, 110
 - <, 110

- <=, 110
- =, 109
- ==, 109–110
- >, 110
- >=, 110
- ^, 110
- ~, 110
- awk-Variable
 - FILENAME, 115
 - FNR, 115
 - FS, 110–111, 207
 - NF, 109–110
 - OFS, 117
 - RS, 110–111
- banner, 54
- basename, 39, 66
- BASH (Umgebungsvariable), 18
- bash, 14, 37, 56, 198
 - l (Option), 18, 198
- BASH_ENV (Umgebungsvariable), 19
- ~/.bash_history, 113
- .bash_login, 18, 22
- .bash_logout, 18
- .bash_profile, 18, 22, 162
- .bashrc, 19–20, 22
- batch, 143–145
- /bin/sh, 146
- Bourne, Stephen L., 42
- Boyce, Raymond F., 125
- break (Shellkommando), 51–53, 80, 202
- busybox, 16
- C, 50
- cal, 17
 - m (Option), 17
- case (Shellkommando), 44, 47, 71, 201
- cat, 16, 51, 97
- cd (Shellkommando), 35
- Chamberlin, Donald D., 125
- chmod, 24, 26, 203
 - reference (Option), 203
- chown, 203
 - reference (Option), 203
- chsh, 14–15, 198
 - s (Option), 15
- Clancy, Tom, 165
- close (awk-Funktion), 119
- cmp, 101
 - s (Option), 101
- Codd, Edgar F., 125
- continue (Shellkommando), 51–53, 80, 202
- cp, 202
- cron, 11, 141–142, 145–150, 211
- crontab, 145, 147–148, 211
 - e (Option), 148
 - l (Option), 148, 211
 - r (Option), 148, 211
 - u (Option), 148
- csh, 14
- cut, 40, 60–62, 73, 79, 94, 106–108, 163
 - d (Option), 163
- dash, 16, 199
- date, 17, 159, 163, 211
- debconf, 166
- Definitionen, 12
- /dev/tty, 86
- df, 73–74
- dialog, 84, 86–90
 - clear (Option), 86
 - menu (Option), 86
 - msgbox (Option), 90
 - no-cancel (Option), 89
 - title (Option), 86
- diff, 29
 - r (Option), 29
- Dijkstra, Edsger, 42
- dirname, 66
- DISPLAY (Umgebungsvariable), 143, 172–173, 182, 191
- Displaynamen, 172
- done (Shellkommando), 51
- du, 118, 120
 - s (Option), 120
- echo (Shellkommando), 25, 30, 34, 43–44, 61, 72, 199
- EDITOR (Umgebungsvariable), 148
- egrep, 46, 215
- elif (Shellkommando), 46–47
- else (Shellkommando), 45, 47
- env, 35
- Eratosthenes, 201
- /etc/anacrontab, 149
- /etc/at.allow, 145
- /etc/at.deny, 145
- /etc/at.deny, 145
- /etc/bash.bashrc, 19–20
- /etc/bash.bashrc.local, 20
- /etc/bash.local, 20
- /etc/cron.allow, 147, 211
- /etc/cron.d, 147
- /etc/cron.daily, 147
- /etc/cron.deny, 147, 211
- /etc/cron.hourly, 147
- /etc/crontab, 147–148
- /etc/default, 70
- /etc/group, 60, 62–63, 119
- /etc/gshadow, 63
- /etc/init.d, 14
- /etc/inputrc, 21
- /etc/lightdm/lightdm.conf, 183
- /etc/lightdm/lightdm.conf.d, 183
- /etc/localtime, 165–167
- /etc/motd, 90

- /etc/passwd, 51, 60, 90, 111, 119, 146–147, 198
- /etc/profile, 18–20, 25
- /etc/profile.local, 20
- /etc/shadow, 198
- /etc/shells, 15, 19, 80, 197
- /etc/skel, 20
- /etc/sysconfig, 70
- /etc/sysconfig/clock, 166
- /etc/timezone, 165–166
- /etc/X11/gdm, 186
- /etc/X11/rgb.txt, 176
- /etc/X11/xdm/, 185
- /etc/X11/xinit/xinitrc, 182
- /etc/X11/xorg.conf, 175
- exec, 55, 202
- exec (Shellkommando), 198
- exit, 42–43
- exit (Shellkommando), 53, 61, 198, 202
- expect, 72
- export (Shellkommando), 35
 - n (Option), 35
- Felder, 87
 - assoziative, 111
- fgrep, 201–202
- fi (Shellkommando), 45
- file, 14, 197
- FILENAME (awk-Variable), 115
- filesum2, 113
- find, 14, 19, 197
- FNR (awk-Variable), 115
- fonts.alias, 190
- fonts.dir, 189
- fonts.scale, 190
- for (awk-Kommando), 111–113
- for (Shellkommando), 40, 49–50, 52, 78, 80
- Fox, Brian, 16
- FS (awk-Variable), 110–111, 207
- FUNCNAME (Shellvariable), 55
- gawk, 106
- gdm, 183, 186
- gdm.conf, 186
- gdmconfig, 186
- getline (awk-Funktion), 119
- grep, 14, 16, 43, 60–63, 71–72, 84, 94, 96, 106, 197, 200, 202, 204–205, 214–215
 - e (Option), 204
 - f (Option), 72
 - line-buffered (Option), 72
- groups, 62
- gsub (awk-Funktion), 207
- Hakim, Pascal, 148
- head, 96–97, 205
- HISTSIZE (Shellvariable), 20
- HOME (Umgebungsvariable), 36, 41, 147
- \$HOME/.bash_profile, 25
- httpd.conf, 97
- iconv, 157
 - c (Option), 158
 - l (Option), 157
 - o (Option), 157
 - output (Option), 157
- if (awk-Kommando), 114
- if (Shellkommando), 44–45, 47, 53, 110, 201
- IFS (Shellvariable), 41, 78, 83
- Inkscape, 185
- INPUTRC (Umgebungsvariable), 21
 - .inputrc, 21
- int (awk-Funktion), 111
- Iteration, 49
- join, 94
- kcontrol, 186
- kdiallog, 90
- kdm, 183, 186
- Kernighan, Brian W., 27, 106
- kill, 43
 - l (Option), 43
- killall, 68
- ksh, 14
- LANG (Umgebungsvariable), 158–159, 161–163
- LANGUAGE (Umgebungsvariable), 159
- LC_* (Umgebungsvariable), 162
- LC_ADDRESS (Umgebungsvariable), 161
- LC_ALL (Umgebungsvariable), 161–162
- LC_COLLATE (Umgebungsvariable), 161–162
- LC_CTYPE (Umgebungsvariable), 161
- LC_MEASUREMENT (Umgebungsvariable), 161
- LC_MESSAGES (Umgebungsvariable), 161
- LC_MONETARY (Umgebungsvariable), 161
- LC_NAME (Umgebungsvariable), 161
- LC_NUMERIC (Umgebungsvariable), 161, 163
- LC_PAPER (Umgebungsvariable), 161
- LC_TELEPHONE (Umgebungsvariable), 161
- LC_TIME (Umgebungsvariable), 161
- length (awk-Funktion), 111
- Libes, Don, 72
- ll, 19
- locale, 161–162, 168
 - a (Option), 162
- localedef, 162
- log (awk-Funktion), 111
- logger, 144
- .login, 18
- login, 18, 45

- LOGNAME (Umgebungsvariable), 45, 146
- logrotate, 66
- ls, 16, 43, 67, 162–163
- lspci, 179
- mail, 72
- MAILTO (Umgebungsvariable), 147
- man, 34, 106
- mawk, 106
- mkdir, 44, 66
 - p (Option), 66
- mkfifo, 71
- mkfontdir, 189
- mktemp, 100–101
 - p (Option), 101
 - t (Option), 101
- mv, 65–66
- n* (Shellvariable), 37
- name (Shellvariable), 37
- newuser (Shellkommando), 79, 205
- NF (awk-Variable), 109–110
- nice, 144
- OFS (awk-Variable), 117
- Open Group*, 170
- oversed, 102
- PAGER (Umgebungsvariable), 34
- PAGER (Shellvariable), 34
- paste, 94, 106
- PATH (Umgebungsvariable), 17, 20, 24, 36, 41, 198
- Perl, 214
- Pinguin, 106
- Positionsparameter, 65
- present (Shellkommando), 86
- print (awk-Kommando), 114, 117
- printf (awk-Kommando), 115, 120
- printf, 163
- printf (Shellkommando), 73–74
- .profile, 18–20, 22, 198
- Prüfungsziele, **217**
- PS1 (Shellvariable), 17, 20, 36
- PS3 (Shellvariable), 81
- Python, 214
- Ramey, Chet, 16
- RANDOM (Shellvariable), 80
- read (Shellkommando), 51, 72–73, 75, 78–79, 204
- return (awk-Kommando), 112
- return (Shellkommando), 55, 83
- rm, 17
- ROOT (Umgebungsvariable), 29
- RS (awk-Variable), 110–111
- Rückgabewert, **42**
- sash, 16
- sed, 11, 16, 40, 66, 93–103, 106, 111, 205, 213, 215
 - e (Option), 94, 101–102
 - expression= (Option), 102
 - f (Option), 94
 - i (Option), 102
 - i.bak (Option), 102
 - n (Option), 96
 - s (Option), 94–95
- select (Shellkommando), 80–81, 83, 88, 205
- seq, 71, 203
- set (Shellkommando), 17, 30, 34
 - C (Option), 17
 - n (Option), 30
 - o noclobber (Option), 17
 - o xtrace (Option), 17
 - u (Option), 30
 - v (Option), 30
 - x (Option), 17, 30
- sh, 14
- Shell, **14**
- SHELL (Umgebungsvariable), 146
- Shellkommando
 - ., 25
 - alias, 17
 - break, 51–53, 80, 202
 - case, 44, 47, 71, 201
 - cd, 35
 - continue, 51–53, 80, 202
 - done, 51
 - echo, 25, 30, 34, 43–44, 61, 72, 199
 - elif, 46–47
 - else, 45, 47
 - exec, 198
 - exit, 53, 61, 198, 202
 - export, 35
 - fi, 45
 - for, 40, 49–50, 52, 78, 80
 - if, 44–45, 47, 53, 110, 201
 - newuser, 79, 205
 - present, 86
 - printf, 73–74
 - read, 51, 72–73, 75, 78–79, 204
 - return, 55, 83
 - select, 80–81, 83, 88, 205
 - set, 17, 30, 34
 - shift, 37, 65, 102
 - source, 25, 45, 55, 198
 - test, 44–46, 50, 79, 83, 198
 - then, 45, 202
 - trap, 53–54, 86
 - typeset, 55
 - unalias, 17
 - unset, 36
 - until, 50–53
 - while, 42, 50–53, 72, 78, 110
- Shellskripte, **14**

- Shellvariable
 - #, 61
 - \$, 64, 100
 - *, 39–40, 87, 200
 - ?, 37, 43, 201
 - @, 39–40, 42, 87, 200
 - 0, 61–62
 - 1, 39, 50
 - 2, 39, 50
 - FUNCNAME, 55
 - HISTSIZE, 20
 - IFS, 41, 78, 83
 - n, 37
 - name, 37
 - PAGER, 34
 - PS1, 17, 20, 36
 - PS3, 81
 - RANDOM, 80
 - suffix, 66
- shift (Shellkommando), 37, 65, 102
- Signale, 53
- sort, 64, 94, 106–107, 114, 119, 207–208
- source (Shellkommando), 25, 45, 55, 198
- sqlite3, 128
- sqrt (awk-Funktion), 111
- ssh, 18
- ssh-agent, 35
- startx, 182, 186, 191
- strace, 19
- su, 143, 148
- sub (awk-Funktion), 111, 118
- substr (awk-Funktion), 111
- suffix (Shellvariable), 66
- syslogd, 67, 71, 144, 146
- Systemlast, 143

- tail, 97
- Tcl, 212, 214
- tcpdump, 47
- tcsh, 14, 16, 198
- TERM (Umgebungsvariable), 143
- test (Shellkommando), 44–46, 50, 79, 83, 198
 - eq (Option), 44
 - ge (Option), 44
 - gt (Option), 44
 - le (Option), 44
 - lt (Option), 44
 - ne (Option), 44
 - s (Option), 101
 - z (Option), 79
- Testschleife, 49
- then (Shellkommando), 45, 202
- Thomas, John C., 176
- timeconfig, 166
- /tmp, 101
- /tmp/oversed.z19516, 101
- TMPDIR (Umgebungsvariable), 101
- tolower (awk-Funktion), 207
- tr, 73, 94, 98, 207
- trap (Shellkommando), 53–54, 86
- typeset (Shellkommando), 55
 - F (Option), 55
 - f (Option), 55
- TZ (Umgebungsvariable), 166–167
- tzconfig, 166
- tzselect, 166
- Tzur, Itai, 148

- Umgebungsvariable, 34
 - _ , 143
 - BASH, 18
 - BASH_ENV, 19
 - DISPLAY, 143, 172–173, 182, 191
 - EDITOR, 148
 - HOME, 36, 41, 147
 - INPUTRC, 21
 - LANG, 158–159, 161–163
 - LANGUAGE, 159
 - LC_*, 162
 - LC_ADDRESS, 161
 - LC_ALL, 161–162
 - LC_COLLATE, 161–162
 - LC_CTYPE, 161
 - LC_MEASUREMENT, 161
 - LC_MESSAGES, 161
 - LC_MONETARY, 161
 - LC_NAME, 161
 - LC_NUMERIC, 161, 163
 - LC_PAPER, 161
 - LC_TELEPHONE, 161
 - LC_TIME, 161
 - LOGNAME, 45, 146
 - MAILTO, 147
 - PAGER, 34
 - PATH, 17, 20, 24, 36, 41, 198
 - ROOT, 29
 - SHELL, 146
 - TERM, 143
 - TMPDIR, 101
 - TZ, 166–167
 - VISUAL, 148
- unalias (Shellkommando), 17
- unbuffer, 72
- uniq, 113, 207
- unset, 162
- unset (Shellkommando), 36
- until (Shellkommando), 50–53
- useradd, 20
 - m (Option), 20
- /usr/lib/xorg/modules, 176
- /usr/share/i18n/locales, 162
- /usr/share/lightdm/lightdm.conf.d, 183
- /usr/share/zoneinfo, 165–167
- /usr/share/zoneinfo/Europe/Berlin, 165

- /var/log/messages, 71
- /var/spool/atjobs, 144

/var/spool/atspool, 144
/var/spool/cron/allow, 147
/var/spool/cron/crontabs, 145, 147
/var/spool/cron/deny, 147, 211
Variable, 108
 Bezüge auf, 34
Variablensubstitution, 34
vi, 148
vipw, 198
 -s (Option), 198
VISUAL (Umgebungsvariable), 148
VNC, 184

watch, 211
wc, 14, 197
Weinberger, Peter J., 106
while (Shellkommando), 42, 50–53, 72,
 78, 110
Wörter, 41

X, 175, 182
 -layout (Option), 181
X-Clients, 170
X-Protokoll, 170
X-Server, 170
X.org, 170
xauth, 191
~/.Xauthority, 191
xclock, 182
xdm, 183, 185–186, 192
xdpyinfo, 186–187
xedit, 191
xfontsel, 190
xfs, 190
xhost, 191
xinit, 182, 186
~/.xinitrc, 182, 186
xkbset, 194
xlsfonts, 190
xman, 191
xmessage, 90
Xorg, 175
 -nolisten tcp (Option), 191
xorg.conf, 176, 190
Xresources, 186
Xservers, 186
Xsession, 186
~/.xsession, 20, 186
xset, 189–190
 q (Option), 189
Xsetup, 186
xterm, 19, 156, 175, 182, 191
Xvnc, 184
xwininfo, 187–188

zdump, 165, 211
zic, 165