6.4 Linux und Unix

Nachdem Sie im vorangegangenen Abschnitt das Wichtigste über Windows erfahren haben, geht es hier um eine andere gängige Systemplattform: *Linux*, inzwischen das am weitesten verbreitete System der Unix-Familie. Viele der Informationen in diesem Abschnitt gelten auch für andere Unix-Varianten, zum Beispiel die verschiedenen BSD-Systeme oder Apples Betriebssystem macOS, dessen Unterbau eine BSD-Unix-Version namens *Darwin* ist.

Der Begriff Linux bezeichnet eigentlich nur den Kernel, also die Funktionsgrundlage für sämtliche Anwendungssoftware. Das Gesamtsystem aus Kernel und Systemtools wird traditionell als *GNU/Linux* bezeichnet, da die Tools in der Regel aus dem GNU-Projekt der Free Software Foundation stammen. In aller Regel ist aber von einem lauffähigen Gesamtsystem die Rede, wenn jemand »Linux« sagt.

Die neueste Version des Linux-Kernels ist zurzeit (Mitte Mai 2023) 6.3.3. Seit Kernel 3.0 wurde das alte Versionierungsschema (gerade Unterversionen wie 2.4.x und 2.6.x für stabile und ungerade wie 2.5.x für Entwicklungsversionen) aufgegeben, und die Versionsnummern wurden um einen Block verkürzt. Ein neuer Hauptversionszweig (4.x, 5.x, 6.x) wird nun also dort erzeugt, wo früher ein Unterversionswechsel wie etwa 2.4 zu 2.6 genügte.

Das Betriebssystem Linux wird in verschiedenen *Distributionen* angeboten. Einige der wichtigsten sind:

- *openSUSE*, früher *SUSE Linux* genannt, wurde von der Nürnberger Firma SUSE AG entwickelt, die inzwischen zu Novell gehört. Novell wiederum wurde vor einigen Jahren von Attachmate

aufgekauft; die Unterstützung für openSUSE wurde seitdem vermindert, aber nicht eingestellt. Die Distribution wurde früher in zwei verschiedenen Versionen angeboten: der Personal Edition für Privatanwender*innen, die vorzugsweise mit Desktopanwendungen ausgestattet war, und der Professional Edition für Entwickler*innen, die Systemadministration oder Unternehmen, die zahlreiche Netzwerkanwendungen, Server, Entwicklungswerkzeuge und andere professionelle Programme enthielt. Inzwischen wird die Distribution als Community-Projekt unter dem Namen *openSUSE* weitergepflegt. Auch die von SUSE bereitgestellten Installations- und Verwaltungsprogramme wurden dafür unter der GPL freigegeben. Zusätzlich gibt es kommerzielle Produkte, etwa den SUSE Linux Enterprise Server.

- *Red Hat Linux* stammt von dem gleichnamigen amerikanischen Unternehmen und ist die beliebteste Distribution in den USA. Die Entwicklung verlief schon vor einigen Jahren wie bei openSUSE: Ursprünglich wurden eine Personal Edition und eine Professional Edition angeboten. Aus der stark erweiterten Personal-Variante ging das freie Community-Projekt *Fedora Linux* (früher Fedora Core Linux) hervor, während Unternehmenslösungen weiterhin unter dem Namen *Red Hat* verkauft werden.
- *Debian GNU/Linux* ist in den letzten Jahren immer wichtiger geworden. Die Distribution hat den besonderen Vorteil, dass alle Bestandteile voll und ganz aus freier Software unter der GPL bestehen, auch das Installationsprogramm. Dafür ist die Installation komplizierter als bei den anderen Distributionen, für den Linux-Einstieg ist diese Distribution daher nicht zu empfehlen. Fortgeschrittene Anwender*innen können die Distribution dagegen am stärksten an ihre eigenen Bedürfnisse anpassen. Es gibt allerdings besonders einsteigerfreundliche Distributionen wie *Ubuntu Linux*, die wiederum auf Debian

basieren. Auch das direkt von CD bootende Live-System *Knoppix* besitzt einen Debian-Unterbau.

Neben diesen am häufigsten vorkommenden Distributionen werden unzählige weitere angeboten, jede von ihnen besitzt ihre besonderen Eigenschaften, Vor- und Nachteile. Die Unterschiede treten nicht so sehr beim normalen Arbeiten mit dem System zutage, sondern nur bei der Installation neuer Programme und bei Konfigurationsarbeiten.

Einen Sonderweg im Vergleich zu allen anderen Distributionen geht etwa *Gentoo*: Hier werden die Installationspakete nicht wie üblich im Binärformat geliefert oder heruntergeladen, sondern als Quellcode, und mithilfe der Installationsskripte automatisch kompiliert. Dies macht Gentoo flexibler als andere Distributionen – sogar so flexibel, dass sich anstelle des Linux- ein BSD-Kernel installieren lässt – und bietet eine sehr große Auswahl an Software, da praktisch jede Open-Source-Software installiert werden kann.

Die meisten hier genannten Linux-Distributionen können Sie frei aus dem Internet herunterladen. Bei den Download-Dateien handelt es sich meist um die ISO-Images der Installations-DVDs oder -CDs. Jede handelsübliche Brennsoftware kann diese auf einen Datenträger brennen. In Tabelle 6.3 sehen Sie die URLs der wichtigsten Distributionen.

Distribution	Website
openSUSE	*de.opensuse.org*
Fedora Linux	*fedoraproject.org/wiki*
Debian GNU/Linux	*www.debian.org*
Ubuntu Linux	*www.ubuntulinux.org*

Distribution	Website
Knoppix	*www.knopper.net/knoppix*
Gentoo	*www.gentoo.org*

Tabelle 6.3 Wichtige Linux-Distributionen und ihre Websites

Eine andere Variante freier Unix-Derivate bilden die verschiedenen BSD-Systeme. Hier die wichtigsten, jeweils mit ihrer Projektwebsite für Informationen und Downloads:

- FreeBSD (*www.freebsd.org*)
- OpenBSD (*www.openbsd.org*)
- NetBSD (*www.netbsd.org*)

Viele der hier genannten Distributionen und Systeme bieten übrigens Live-CDs oder Live-DVDs. Sie können die entsprechenden Images herunterladen, auf einen Datenträger brennen und direkt davon booten, um die Betriebssysteme risikolos auszuprobieren. Eine andere interessante Lösung besteht darin, sie in einer Virtualisierungssoftware wie VirtualBox oder VMware zu starten; in diesem Fall kann das Image sogar ohne Brennvorgang als virtueller Datenträger eingestellt werden. Näheres dazu erfahren Sie in Kapitel 14, »Server für Webanwendungen«.

6.4.1 Arbeiten mit der Shell

Auch wenn so gut wie alle Distributionen inzwischen schon bei der Installation eine grafische Benutzeroberfläche einrichten, sollten Sie sich den Umgang mit der Konsole angewöhnen. Die mächtigsten Funktionen des Systems werden nach wie vor über die Kommandozeile aufgerufen; grafische Steuerprogramme dafür

stehen nicht flächendeckend und schon gar nicht durchgehend als Open Source zur Verfügung.

Öffnen Sie zunächst ein Terminalfenster (das konkrete Programm heißt in jedem System und jeder Distribution etwas anders – unter macOS und auf dem Linux-Desktop GNOME beispielsweise *Terminal*, auf dem Desktop KDE *Konsole*). Darin wird eine Eingabeaufforderung (Prompt) angezeigt, die je nach Konfiguration sehr unterschiedlich aussehen kann. In der Regel sehen Sie etwa Folgendes:

```
user@rechner: ~ $
```

Anstelle von `user` wird der Benutzername angezeigt, unter dem Sie sich angemeldet haben; hinter dem `@` steht der Name des Rechners, auf dem Sie gerade arbeiten. Auf diese Angaben folgt der Pfad des aktuellen Arbeitsverzeichnisses. Im zuvor gezeigten Beispiel ist das aktuelle Verzeichnis das individuelle Home-Verzeichnis (auf dieses Beispiel bezogen */home/user*), das durch die Tilde gekennzeichnet wird. Das Dollarzeichen bildet schließlich den Abschluss; dahinter blinkt der Cursor für die Befehlseingabe. Anstelle des Dollarzeichens erscheint bei manchen Shells `>` oder ein anderes Zeichen.

Wenn Sie als *root* angemeldet sind, bekommen Sie einen etwas anderen Prompt zu sehen, beispielsweise diesen:

```
rechner: ~ #
```

Es wird also kein Benutzername angezeigt, und hinter der Pfadangabe folgt eine Raute (`#`) anstelle des Dollarzeichens. Auch *root* befindet sich in diesem Beispiel in seinem Home-Verzeichnis, standardmäßig */root*.

In den folgenden Beispielen wird der Prompt einfach als Dollarzeichen dargestellt. Wenn für einen Befehl *root*-Rechte

erforderlich sind, wird dagegen die Raute verwendet. Benutzereingaben sind in den Beispielen jeweils fett gesetzt, um sie vom Prompt und von der Ausgabe des Systems abzusetzen.

Grundfunktionen der Shell

Das Programm, das Ihre Befehle entgegennimmt und zu interpretieren versucht, wird *Shell* genannt. Es gibt nicht *die eine* Linux-Shell, sondern eine Reihe verschiedener Shell-Programme, die sich bis zu einem gewissen Grad voneinander unterscheiden. Höchstwahrscheinlich läuft in Ihrem Linux-System eine Shell, die als *bash* bezeichnet wird. Geben Sie den folgenden Befehl ein, um herauszufinden, welche Shell Sie ausführen:

```
$ echo $0
```

`$0` ist eine spezielle Variable, die jeweils den Namen des aktuell laufenden Programms enthält. Die Ausgabe dürfte zum Beispiel */bin/bash* oder */bin/sh* lauten. Die gängigsten Shells werden in der folgenden Liste aufgeführt:

- *sh* oder *bsh*, die *Bourne Shell*, benannt nach ihrem Entwickler, war die ursprüngliche Shell des Bell-Labs-Unix. Sie beherrscht die kleinste gemeinsame Menge der Fähigkeiten aller anderen Shells.
- *csh*, die *C-Shell*, und ihre Erweiterung *tcsh* enthalten eine Reihe spezieller Funktionen, die von der Programmiersprache C beeinflusst wurden und besonders den Bedürfnissen der C-Programmierung entgegenkommen.
- *bash*, die *Bourne Again Shell* (ein nettes Wortspiel), ist die GNU-Weiterentwicklung der ursprünglichen Bourne Shell mit vielen interessanten Zusatzfunktionen. Diese Shell ist in allen Linux-Distributionen als Standard voreingestellt. Trotzdem werden alle hier genannten und meist noch weitere mitgeliefert.

- *zsh*, die *Z-Shell*, wurde 1990 an der Princeton University entwickelt und hat in Sachen Automatisierung mehr zu bieten als etwa die *bash*. Sie ist seit einigen Versionen die Standard-Shell unter macOS.
- *ksh*, die *Korn Shell*, ist der offizielle bei AT&T entwickelte Nachfolger der *bsh*. Sie vereint einige Vorteile von Bourne und C-Shell mit eigenen Erweiterungen. Die *ksh* selbst ist nicht frei verfügbar, es gibt aber eine freie Variante namens *pdksh* (*Public Domain Korn Shell*).
- *sash*, die *Stand-alone-Shell*, ist ein nützliches Hilfsmittel zur Fehlerbehebung: Viele Standard-POSIX-Dienstprogramme sind direkt in die Shell selbst eingebaut und brauchen nicht zusätzlich bereitgestellt zu werden. Ihre Namen beginnen normalerweise mit einem Minuszeichen, um sie von der voll ausgestatteten GNU-Version dieser Tools zu unterscheiden. Für Rettungssysteme, die vom USB-Stick starten, ist die *sash* ideal.

Um Missverständnissen vorzubeugen, sollten Sie zunächst verstehen, dass über 90 % der Eingaben, die Sie an der Kommandozeile vornehmen, unter allen Shells identisch sind: Es handelt sich nämlich bei diesen Eingaben überhaupt nicht um Shell-Kommandos. Die meisten »Unix-Befehle« sind separate Systemprogramme, die sich für gewöhnlich im Verzeichnis */bin* befinden und mit der Shell nichts zu tun haben. Die Shells unterscheiden sich insbesondere in der Art und Weise, wie die Funktionen der Systemprogramme durch intelligente Verknüpfungen erweitert werden können.

Die Konfiguration, mit der die Shell (und übrigens auch jedes andere Programm) ausgeführt wird, heißt *Umgebung* (*Environment*). Sie besteht aus der User- und der Group-ID, unter der das Programm läuft, aus dem aktuellen Arbeitsverzeichnis sowie aus einer Reihe

von *Umgebungsvariablen*, die von dem Programm ausgelesen werden. Die Shell bezieht ihre Umgebung aus diversen Konfigurationsdateien, insbesondere aus:

- */etc/profile*: zentrale Konfigurationsdatei für alle Shells und alle User. Diese Datei sollte nicht editiert werden; ändern Sie stattdessen *~/.bashrc* oder erstellen Sie eine benutzerspezifische *~/profile.local*.
- */etc/profile.d/**: zentrale Konfigurationsdateien für bestimmte Aspekte einzelner Shells.
- *~/.bashrc*: *bash*-spezifische Einstellungen für einen einzelnen User-Account im jeweiligen Home-Verzeichnis.

Die meisten Shells bieten heute die ursprünglich in der *csh* eingeführte Möglichkeit, Programme im Hintergrund zu starten: Wenn Sie ein `&`-Zeichen an einen Befehl anhängen, gelangt dessen Ausgabe nicht auf den Bildschirm, und Sie können sofort den nächsten Befehl eingeben. Es wird beim Aufruf des Befehls lediglich dessen Prozess-ID ausgegeben. Inzwischen bieten fast alle Shells diese Option an. Hier sehen Sie ein einfaches Beispiel, in dem die Suche nach Dateien, deren Name mit einem `a` beginnt, in den Hintergrund verbannt wird:

```
$ find . -name a* &
[1] 3125
$
```

In eckigen Klammern wird eine Job-Nummer angezeigt; dahinter erscheint die PID. Anstelle von 3125 werden Sie fast sicher eine andere zu sehen bekommen. Mithilfe des Befehls `fg` (für *foreground*) können Sie die Ausgabe des Befehls im Vordergrund fortsetzen. Falls sich mehrere Prozesse im Hintergrund befinden, müssen Sie die Job-Nummer angeben. Beispiel:

```
$ fg 1
```

Ebenso können Sie ein bereits laufendes Programm nachträglich in den Hintergrund stellen, indem Sie die Tastenkombination Strg + Z drücken. Auch in diesem Fall werden Job-Nummer und PID angezeigt, und Sie können das Programm mit `fg` zurückholen.

In der Regel bestehen die Befehle, die Sie eingeben, aus dem Namen des gewünschten Systemprogramms und einer durch Leerzeichen getrennten Liste von Parametern. Einige der Parameter sind *Optionen*, die bei den meisten Befehlen mit einem Minuszeichen beginnen, andere geben dagegen konkrete Werte wie Pfad- oder Dateinamen, Bezeichnungen und Ähnliches an.

Anweisungen werden durch das Drücken von ↵ abgeschlossen und unmittelbar ausgeführt. Zu lange Eingaben können Sie aber durch einen Backslash (\) und ↵ auf mehrere Zeilen aufteilen. Hier ein Beispiel, das in allen Dateien des aktuellen Verzeichnisses und allen Unterverzeichnissen nach dem Text `"in diesem Fall werden Job-Nummer und PID angezeigt"` sucht:

```
$ grep -r \
> "in diesem Fall werden Job-Nummer und PID angezeigt" \
> *
```

Die Shell sucht nach der Eingabe eines Kommandos in der folgenden Reihenfolge nach einer Möglichkeit, es auszuführen:

- `alias`-Definitionen (siehe Abschnitt 6.4.3, »Automatisierung«).
- Shell-Built-ins, das heißt Kommandos, die in das Shell-Binary selbst eingebaut sind.
- Externe Programme – die in der Umgebungsvariablen `PATH` angegebenen Verzeichnisse werden der Reihe nach durchsucht.

Falls der eingegebene Befehl an keinem der genannten Orte gefunden wird, erhalten Sie eine Fehlermeldung. Sollten Sie die *bash* verwenden und irrtümlich das Windows-Kommando `cls` zum

Bildschirmlöschen eingeben, erhalten Sie beispielsweise diese Ausgabe:

```
bash: cls: command not found
```

Möchten Sie wissen, ob es sich bei einem Kommando um einen Alias, ein Shell-Built-in oder ein Programm handelt, können Sie `type` *`Kommando`* eingeben. Hier für jeden Typ ein Beispiel:

```
$ type ls
ls is aliased to `/bin/ls $LS_OPTIONS'
$ type alias
alias is a shell builtin
$ type mkdir
mkdir is hashed (/bin/mkdir)
```

Wenn Sie den Namen eines Programms eingeben, sucht die Shell in ganz bestimmten Verzeichnissen nach diesem Programm. Die Verzeichnisse sind in einer Umgebungsvariablen namens `PATH` festgelegt. Möchten Sie diese Liste lesen, geben Sie Folgendes ein:

```
$ echo $PATH
/bin:/usr/bin:/usr/sbin:/usr/local/bin:/usr/share/bin
```

Der Befehl `echo` gibt sämtlichen folgenden Text in der nächsten Zeile aus. Das Dollarzeichen sorgt dafür, dass die Shell das folgende Wort als den Namen einer Variablen auffasst, deren Wert ausgegeben werden soll. Beachten Sie, dass Unix-Systeme, anders als Windows, auch bei Variablennamen zwischen Groß- und Kleinschreibung unterscheiden und dass diese Variable `PATH` heißt, nicht etwa `Path` oder `path`.

Der Wert der Variablen `PATH` besteht aus einer Liste von absoluten Pfadangaben (mit `/` beginnend), die durch Doppelpunkte voneinander getrennt werden. In der Praxis ist die Liste meist erheblich länger als im zuvor gezeigten Beispiel.

Im Folgenden soll ein Verweis auf das aktuelle Verzeichnis hinzugefügt werden. Üblicherweise wird ein Programm nämlich nicht einfach ausgeführt, wenn Sie sich in seinem Verzeichnis befinden, sondern nur, wenn dieses Verzeichnis auch in `PATH` steht. Um das zu ändern, können Sie die spezielle Verzeichnisangabe `.` (einen einzelnen Punkt) hinzufügen, da diese jeweils das aktuelle Verzeichnis repräsentiert.

Falls Sie den Inhalt der Variablen ändern möchten, funktioniert das in den verschiedenen Shells unterschiedlich. Hier sehen Sie Beispiele für die zuvor genannten Shells:

- *sh*, *bsh*, *bash*, *zsh* und *ksh*: `export PATH=$PATH:.`
- *csh* und *tcsh*: `set PATH=$PATH:.`

Der Wert, der `PATH` in den beiden Beispielen zugewiesen wird, nämlich `$PATH:.`, bedeutet: bisheriger Wert von `PATH`, Doppelpunkt, anschließender Punkt. Die vollständige Pfadliste aus dem zuvor gezeigten Beispiel sähe nach dieser Änderung folgendermaßen aus:

```
/bin:/usr/bin:/usr/sbin:/usr/local/bin:/usr/share/bin:.
```

In der Praxis sollten Sie sich gut überlegen, ob Sie diese Änderung durchführen möchten, da sie ein gewisses Sicherheitsrisiko darstellt. Wenn Sie den Punkt angeben möchten, gehört er auf jeden Fall ans Ende von `PATH`, weil Ihnen bei einem Angriff ansonsten leicht ein Programm untergejubelt werden könnte, das denselben Namen trägt wie ein Systemprogramm und deshalb stattdessen ausgeführt würde, falls Sie sich im entsprechenden Verzeichnis befinden.

Alle modernen Unix-Shells beherrschen die sehr bequeme Funktion der *Eingabevervollständigung*: Wenn Sie einen Befehl oder den Pfad einer Datei eintippen, können Sie zwischenzeitlich die ↹-Taste

drücken. Ist der Befehl oder Pfad zu diesem Zeitpunkt bereits eindeutig – lässt er also nur noch eine Interpretation zu –, wird er komplett ausgeschrieben. Bei Zweideutigkeiten wird er nur zum Teil ergänzt, und es ertönt ein Warnton. Das folgende Beispiel zeigt, wie Sie aus Ihrem Home-Verzeichnis schnell in das darunterliegende Verzeichnis *dokumente* wechseln:

```
user@rechner: ~ $ cd do⇥
user@rechner: ~/dokumente $
```

Angenommen, in Ihrem Home-Verzeichnis befände sich ein weiteres Verzeichnis namens *dokumente2*. In diesem Fall wird durch ⇥ zwar das Wort »dokumente« ergänzt, aber die Shell weiß noch nicht, ob Sie wirklich das Verzeichnis *dokumente* meinen oder *dokumente2*. Deshalb wird der besagte Warnton ausgegeben. Wenn Sie zweimal ⇥ drücken, wird in den meisten modernen Shells eine Liste der möglichen Alternativen angezeigt.

Ähnlich komfortabel ist die *History* aller bereits eingegebenen Befehle. Mit den Pfeiltasten auf der Tastatur können Sie darin nach oben oder nach unten blättern; die früheren beziehungsweise späteren Befehle werden dadurch wieder angezeigt. Wenn der gewünschte Befehl erscheint, können Sie ihn ändern und anschließend mithilfe von ↵ ausführen.

Neuere Versionen der *bash* speichern die History in einer Datei namens *.bash_history* in Ihrem Home-Verzeichnis, sodass sie beim nächsten Log-in wieder zur Verfügung steht.

In der *bash* stehen Ihnen außerdem viele praktische Tastenkürzel zur Verfügung. Es gibt zwei verschiedene Modi, die nach den beiden bekannten Unix-Texteditoren *Emacs* und *vi* benannt sind.[44] Im standardmäßig eingestellten *Emacs*-Modus können Sie unter anderem folgende Tastenkombinationen verwenden:

- ← oder Strg + B bewegt den Cursor ein Zeichen nach links; → oder Strg + F navigiert ein Zeichen nach rechts.
- Alt + B wandert um ein Wort nach links und Alt + F um eines nach rechts; als Grenze gilt jeweils ein Leerzeichen.
- Mit Strg + A oder Pos1 gelangen Sie zum Zeilenanfang, mit Strg + E oder Ende zum letzten Zeichen der Zeile.
- Entf oder Strg + D löscht das Zeichen unter dem Cursor, während ← oder Strg + H das links davon befindliche Zeichen entfernt.
- Mit Strg + W entfernen Sie ein Wort.
- Strg + K löscht den Text von der Cursorposition bis zum Zeilenende.
- Strg + U entfernt den gesamten Inhalt der Zeile.
- Strg + R startet die inkrementelle Suche nach einem History-Eintrag (inkrementell bedeutet, dass die Eingabe eines Zeichens jeweils sofort zum ersten infrage kommenden Text springt). Sie können mit ↑ und ↓ durch die Suchergebnisse navigieren und mit ↵ das gewünschte auswählen.
- Strg + Bild↑ beziehungsweise Strg + Bild↓ ermöglichen das Blättern im Puffer des Terminal(-Fenster)s.
- Strg + L löscht den Bildschirm.

In modernen Terminal-Emulationen können Sie einen beliebigen Text durch Ziehen mit gedrückter linker Maustaste markieren. Ein Klick mit der mittleren Maustaste oder dem Scrollrad fügt diesen Text dann an der Textcursorposition wieder ein. Mäuse mit zwei Tasten lassen sich dabei so konfigurieren, dass ein Klick auf beide Tasten gleichzeitig die mittlere Taste emuliert. Bei den meisten

modernen Systemen funktioniert das Verfahren auch distributionsweit; in GUI-Programmen wird aber zusätzlich auch die Variante über die Zwischenablage unterstützt – Strg + C zum Kopieren beziehungsweise Strg + X zum Ausschneiden und Strg + V zum Einfügen.

Unter macOS, das sich ansonsten in vielen Belangen wie Linux und andere Unix-Varianten verhält, wird übrigens auch im Terminal die normale Zwischenablage verwendet – es wird also mit Cmd + C kopiert oder mit Cmd + X ausgeschnitten und mit Cmd + V eingefügt.

Aus Sicherheitsgründen sollten Sie nicht permanent als *root* arbeiten. Mitunter müssen Sie aber zwischendurch eine Konfigurationsaufgabe erledigen, die nur dem Superuser-Account gestattet ist. Es ist sicherlich keine sehr bequeme Lösung, sich mithilfe von `logout` abzumelden und als *root* wieder anzumelden. Angenehmer ist zu diesem Zweck der Befehl `su`, der für *substitute user* oder auch *superuser* steht: Wenn Sie gerade keine *root*-Privilegien haben und `su` eingeben, werden Sie nach dem *root*-Passwort gefragt. Sofern Sie es korrekt angeben, können Sie nun einzelne Befehle als *root* ausführen. Mit `exit` oder Strg + D erhalten Sie Ihre normale Shell zurück.

Als *root* können Sie mit `su` auch im Namen eines anderen User-Accounts agieren, ohne dessen Passwort zu kennen. Dazu müssen Sie lediglich `su` *`Benutzername`* eingeben.

Wenn Sie nur einen einzigen Befehl als *root* ausführen möchten, können Sie einfach `sudo` *`Kommando`* eingeben – auch hier werden Sie nach dem Passwort gefragt.

In Ubuntu, macOS und einigen anderen Systemen wird standardmäßig kein echter *root*-Account angelegt. Deshalb müssen

Sie hier im Standard-User-Account (der bei der Systeminstallation angelegt wurde) stattdessen Ihr eigenes Passwort eingeben, um mit `su` oder `sudo` *root*-Rechte zu bekommen. Das einfache `su` wird an dieser Stelle konsequenterweise durch `sudo su` ersetzt, da Sie schon für `su` selbst *root*-Rechte benötigen. Der Account muss dafür einer bestimmten Gruppe angehören, die je nach konkretem System etwa *admin* oder *sudoers* heißt. Bei dem Account, der während der Systeminstallation angelegt wird, ist das automatisch der Fall.

Hilfefunktionen

Traditionell ist jedes Unix-System mit einem eingebauten Hilfesystem ausgestattet, den *Manpages* (kurz für *Manual Pages*, also Handbuchseiten). Bei Linux-Systemen kommt ein neueres, komfortableres System namens *GNU info* hinzu. Bei den GNU-Tools, also praktisch allen Systemprogrammen, sind die *info*-Seiten in der Regel aktueller und ausführlicher als die Manpages.

Eine Manpage liefert Informationen über einen bestimmten Befehl, ein Hilfsprogramm oder eine Konfigurationsdatei. Um sie anzuzeigen, wird das Programm `man` verwendet. Geben Sie beispielsweise Folgendes ein, falls Sie Hilfe zum Befehl `ls` benötigen, der Verzeichnisinhalte auflistet:

```
$ man ls
```

Zuerst formatiert `man` die Hilfeseite, was eine Weile dauern kann. Danach wird die Seite angezeigt, und Sie können mithilfe der folgenden Tasten, die vom Pager-Programm `less` bereitgestellt werden, darin blättern und navigieren:

- E, ↵ oder ↓ – eine Zeile weiter
- Y oder ↑ – eine Zeile zurück

- F, Leertaste oder Bild ↓ – eine Fensterseite weiter
- B oder Bild ↑ – eine Fensterseite zurück
- Pos1 – zum Textanfang
- Ende – zum Textende
- `/ Suchbegriff` ↵ – vorwärts nach dem angegebenen Begriff suchen
- `? Suchbegriff` ↵ – rückwärts suchen
- N – nächstes Vorkommen des Suchbegriffs
- ⇧ + N – nächstes Vorkommen des Suchbegriffs in der jeweils anderen Richtung
- H – Hilfeseite zur Bedienung von `less`
- Q – Programm beenden

Jede Manpage gehört zu einer bestimmten Kategorie, die jeweils durch eine der folgenden Nummern oder Buchstaben gekennzeichnet wird:

- 0: Include-Dateien für eigene C-Programme
- 1: Shell-Programme
- 2: Systemaufrufe (Kerneldienste)
- 3: Bibliotheksfunktionen (C-Standardbibliothek etc.)
- 4: Beschreibung der Gerätedateien (*/dev/**) und anderer Spezialdateien
- 5: Konfigurationsdateiformate
- 6: Spiele
- 7: Makros (kombinierte Programme)

- 8: Administrationsbefehle (in der Regel *root* vorbehalten)
- 9: Kernelroutinen
- n: (*new*) neue Tools
- l: (*local*) lokale Tools
- p: (*public*) öffentliche Tools
- o: (*old*) veraltete Tools

Die Buchstabensektionen sind veraltet und werden üblicherweise nicht mehr verwendet. Einige optionale Programme benutzen auch eigene Kategorien. Sie müssen die Kategorie immer dann angeben, wenn es mehrere Einträge mit dem gewünschten Namen gibt.

In diesem Fall lautet die Syntax `man Kategorie Eintrag`. Beispiel:

```
$ man 1 passwd
```

Der Befehl `whatis Eintrag` oder `man -f Eintrag` zeigt sämtliche Manpages mit dem angegebenen Namen an; die Sektionen stehen in Klammern:

```
$ whatis passwd
passwd (lssl)        - compute password hashes
passwd (1)           - change user password
passwd (5)           - password file
```

Die Option `man -k String` oder das gleichbedeutende Kommando `apropos` verwendet den eingegebenen Text dagegen als Teil-String:

```
$ apropos passwd
htpasswd2 (1)  - Manage user files for basic authentication
ldappasswd (1) - change the password of an LDAP entry
passwd (1)     - change user password
gpasswd (1)    - change group password
[...]
```

Das Hilfesystem *GNU info* wird durch den Befehl `info` aktiviert. Wenn Sie kein Stichwort eingeben, wird der Directory Node

angezeigt, in dem Sie eine Übersicht über die verschiedenen Themen erhalten. Hier die wichtigsten Tastenkürzel im Überblick:

- Strg + F (*forward*) – ein Zeichen weiter
- Strg + B (*backward*) – ein Zeichen zurück
- Strg + N (*next*) – eine Zeile weiter
- Strg + P (*previous*) – eine Zeile zurück
- Strg + A – zum Zeilenanfang
- Strg + E – zum Zeilenende
- Leertaste – eine Bildschirmseite weiter
- Entf oder ← – eine Bildschirmseite zurück
- M *Thema* ↵ (*menu*) – ruft die *info*-Seite zum angegebenen Thema auf (falls es zu dem Wort unter dem Cursor eine Seite gibt, wird diese automatisch eingetragen)
- N (*next*) – eine Seite im aktuellen Oberthema weiterblättern
- P (*previous*) – eine Seite zurückblättern
- U (*up*) – eine Ebene nach oben; die oberste Ebene ist der Directory Node
- L (*last*) – zurück zur vorher angezeigten Seite
- H (*help*) – Hilfe zu *info* selbst; zurück mit L
- ? – tabellarische Kurzübersicht über *info*; zurück mit L
- Q – *info* beenden

Pipes und Ein-/Ausgabeumleitung

Eine der praktischsten Eigenschaften der Unix-Shells besteht in der Umleitung von Ein- und Ausgabe sowie deren Verkettung. Mit der Ausgabe eines Befehls können Sie mehr tun, als sie einfach auf dem Bildschirm darzustellen, und die Eingabe muss nicht unbedingt von der Tastatur stammen: Sie können die Eingabe für einen Befehl aus einer Datei holen, die Ausgabe in eine Datei schreiben und schließlich die Ausgabe des einen Befehls als Eingabe für den nächsten verwenden. Auf diese Weise können Sie die einfachen Bausteine der Systembefehle zur Erledigung komplexer Aufgaben einsetzen.

Die Standard-I/O-Kanäle

Die Standardbibliothek der Programmiersprache C kennt drei Standardkanale (*Streams*) zur Ein- und Ausgabe (*Input/Output* oder kurz *I/O*):

- `stdin` ist die *Standardeingabe*. Sie ist normalerweise mit der Tastatur verknupft.
- `stdout`, die *Standardausgabe*, wird per Voreinstellung auf die Konsole geleitet.
- `stderr` schlie lich ist die *Standardfehlerausgabe*. Auch sie landet fur gewohnlich auf der Konsole. Vorteil: Wenn Sie `stdout` in eine Datei umleiten, werden Fehlermeldungen noch immer angezeigt.

Da Unix und andere Betriebssysteme in C geschrieben sind, besitzen auch sie diese Eigenschaften: Die Ein- und Ausgabeumleitung basiert auf einer Verknupfung von `stdin`, `stdout` beziehungsweise `stderr` mit anderen Dateien oder Geraten.

Der Befehl `ls` dient beispielsweise dazu, den Inhalt des aktuellen Verzeichnisses auszugeben. Möchten Sie diesen Inhalt lieber in eine andere Datei schreiben, können Sie folgendermaßen vorgehen:

```
$ ls >inhalt.txt
```

In diesem einfachen Beispiel wird der Inhalt des aktuellen Verzeichnisses nicht auf den Bildschirm geschrieben, sondern in die Datei *inhalt.txt*. Diese Datei wird automatisch neu angelegt, falls sie noch nicht existiert, ansonsten wird sie überschrieben. Wenn Sie die Ausgabe eines Befehls lieber an eine bestehende Datei anhängen möchten, können Sie anstelle des einen `>`-Zeichens zwei verwenden (sollte die Datei noch nicht existieren, wird sie dadurch dennoch angelegt):

```
$ ls >>inhalt.txt
```

Auf ähnliche Weise können Sie die Eingabe für einen Befehl aus einer Datei lesen. Zum Beispiel gibt der Befehl `grep` alle Zeilen eines eingegebenen Textes zurück, in denen ein Suchmuster vorkommt. Falls Sie alle Zeilen der Datei *inhalt.txt* suchen möchten, die mindestens ein `a` enthalten, funktioniert das folgendermaßen:

```
$ grep a <inhalt.txt
```

Für den Befehl `grep` ist diese Schreibweise eigentlich überflüssig, da auch `grep` *`Muster Dateiname`* unterstützt wird – im vorliegenden Beispiel also:

```
$ grep a inhalt.txt
```

Eine interessante Variante der Eingabeumleitung ist das *Hier-Dokument* (englisch: *Here Document*). Diese Art der Eingabe stammt nicht aus einer Datei, sondern nimmt alle eingegebenen Zeilen bis zu einer speziellen Markierung (»bis hierhin«, daher der Name)

entgegen. Das folgende Beispiel sucht mithilfe von `grep` nach allen Zeilen in der Eingabe, die mindestens ein `a` enthalten:

```
$ grep a <<ENDE
> Hallo
> liebe
> Welt
> ENDE
```

Die Ausgabe dieser eingegebenen Sequenz lautet folgendermaßen:

```
Hallo
```

Die Markierung `ENDE` bildet das Ende der Eingabe. Die `grep`-Suchmuster werden im nächsten Abschnitt behandelt.

Eine weitere Variante der Ein- und Ausgabeumleitung ist die sogenannte *Pipe* (Röhre). Es geht darum, die Ausgabe eines Befehls als Eingabe für den nächsten zu verwenden. Eine der gängigsten Kombinationen ist die Weiterleitung der umfangreichen Ausgabe bestimmter Befehle an einen Pager – ein Programm, das Inhalte seitenweise ausgibt. Der ursprüngliche Unix-Pager wird `more` genannt, die erheblich mächtigere Open-Source-Alternative heißt `less` (in Anspielung auf *less is more*, »weniger ist mehr«).

Angenommen, der Inhalt des aktuellen Verzeichnisses wäre länger als die Anzahl der Zeilen Ihres Terminals. In diesem Fall könnten Sie diesen Inhalt an `less` weiterleiten:

```
$ ls |less
```

Das Pipe-Zeichen `|` wird auf einer deutschen PC-Tastatur mit der Tastenkombination Alt + < erzeugt; auf dem Mac ist es die Tastenkombination Alt + 7.

Das Programm `less` kann auch den Inhalt einer Datei anzeigen und ist auf diese Weise ein komfortabler Ersatz für `cat`, das der Anzeige

einer oder mehrerer Dateien im Terminal dient. Im Grunde ist `ls |less` also eine Kurzfassung für die beiden folgenden Einzelbefehle:

```
$ ls >temp.txt
$ less temp.txt
```

Auf ähnliche Weise lässt sich jede Pipe durch zwei Einzelbefehle ersetzen, wobei der zweite Befehl oft nicht direkt mit einem Dateinamen als Argument aufgerufen wird, sondern mit einer Eingabeumleitung.

Eine Pipe hat allerdings zwei bedeutende Vorteile gegenüber der Verwendung einzelner Befehle: Erstens muss keine Zwischendatei erzeugt werden, und zweitens beginnt der zweite Befehl einer Pipe bereits zu arbeiten, wenn er die erste Zeile aus der Ausgabe des ersten Befehls erhält.

Eine weitere verbreitete Anwendung für Pipes besteht in der unmittelbaren Filterung einer Ausgabe mithilfe von `grep`. Das folgende Beispiel gibt nur diejenigen Dateien im aktuellen Verzeichnis aus, die die Zeichenfolge `txt` enthalten:

```
$ ls |grep txt
```

Hier noch ein Beispiel: Die Ausgabe von `ls` wird an das Kommando `wc` (Wortzähler) weitergereicht; die Option `-l` sorgt dafür, dass nur Zeilen gezählt werden. Das Ergebnis ist somit die Anzahl der Einträge im aktuellen Verzeichnis:

```
$ ls |wc -l
```

Sie können mehrere Kommandos auch durch ein Semikolon getrennt hintereinanderschreiben. Dadurch werden sie einfach nacheinander ausgeführt. Das folgende Beispiel kommt sehr häufig vor, denn die meisten Programme, die Sie selbst aus dem Sourcecode kompilieren können, verwenden diese Sequenz dafür:

```
# ./configure [Optionen]; make; make install
```

Ein Nachteil des Semikolons besteht darin, dass auch dann versucht wird, den nächsten Befehl auszuführen, wenn der vorherige fehlschlägt. Abhilfe schafft hier die Verknüpfung mithilfe von `&&` (logisches Und) – der zweite Befehl wird dann nur bei Erfolg des ersten ausgeführt. Schreiben Sie die Sequenz also am besten wie folgt, um ein Programm unbeaufsichtigt zu kompilieren:

```
# ./configure [Optionen] && make && make install
```

Das Gegenteil besorgt die Verknüpfung durch `||` (logisches Oder) – hier wird der zweite Befehl nur dann ausgeführt, wenn der erste fehlschlägt. Das folgende Beispiel versucht, mit `ls` eine Datei zu listen, die nicht existiert, und gibt ansonsten neben der offiziellen auch noch eine eigene Fehlermeldung aus:

```
$ ls does_not_exist || echo "Gibt's nicht."
ls: does_not_exist: No such file or directory
Gibt's nicht.
```

Eine letzte Möglichkeit besteht darin, ein Kommando in *Backticks* (``) einzuschließen, um seine Ausgabe in einem anderen Zusammenhang zu verwenden. Hier ein Beispiel, das die Ausgabe von `whoami` (Name des aktuell angemeldeten Accounts) in einen ganzen Satz integriert:

```
$ echo "Zurzeit ist `whoami` angemeldet."
Zurzeit ist sascha angemeldet.
```

6.4.2 Die wichtigsten Systembefehle

In Linux und andere Unix-Varianten wurden Unmengen von Systemprogrammen eingebaut. Es ist vollkommen aussichtslos, an dieser Stelle auch nur die Hälfte davon zu behandeln. In diesem

Abschnitt lernen Sie stattdessen die wichtigsten Kommandos mit ihren gängigsten Optionen kennen.

Bevor es losgeht, sollten Sie sich einige wichtige Fakten über die meisten Unix-Systemprogramme merken:

- Im Erfolgsfall erhalten Sie keinerlei Rückmeldung, sondern nur den nächsten Prompt.
- Dateien werden standardmäßig gelöscht oder überschrieben, ohne dass zuvor nachgefragt wird.
- Die meisten Kommandos können mit einer Vielzahl von Optionen aufgerufen werden, die in der Regel aus einem Minuszeichen und einem Buchstaben (mit Unterscheidung von Groß- und Kleinschreibung) bestehen. Die GNU-Versionen der Tools kennen auch Optionen im Langformat – zwei Minuszeichen, gefolgt von einem ganzen Wort (oder mehreren durch weitere Minuszeichen getrennten Wörtern).

Arbeiten mit Dateien und Verzeichnissen

Einige der grundlegenden Befehle in einem Betriebssystem dienen der Manipulation von Dateien und Verzeichnissen.

Alle Unix-Shells bieten die Möglichkeit, Datei- und Verzeichnisnamen in vielen Befehlen durch Muster anzugeben, die auf mehrere Dateien passen. In diesen Mustern gibt es die folgenden wichtigen Sonderzeichen (die in Dateinamen verboten oder zumindest problematisch sind):

- Das `*` ersetzt beliebig viele Zeichen. `h*o` steht beispielsweise für »hallo«, »hello« oder auch »ho«.
- Das `?` steht für genau ein Zeichen. Zum Beispiel bezeichnet `te?t` sowohl »test« als auch »text«.

- Mehrere Zeichen in eckigen Klammern wie `[abc]` bedeuten, dass genau eines dieser Zeichen gemeint ist. Durch einen Bindestrich können Bereiche wie `a-z` gebildet werden; mehrere Listen werden einfach hintereinandergeschrieben. Beispielsweise bedeutet die Liste `[a-zA-Z0-9]`, dass alle Kleinbuchstaben, alle Großbuchstaben und alle Ziffern zulässig sind.
- Ein Ausrufezeichen vor der Liste in den eckigen Klammern bedeutet, dass jedes Zeichen außer den nachfolgenden Zeichen in dieser Liste zulässig ist. `[!Bb]` bedeutet etwa, dass auf keinen Fall ein `B` erlaubt ist – weder ein groß- noch ein kleingeschriebenes.
- Eine durch Kommata getrennte Liste von Zeichenketten in geschweiften Klammern bedeutet, dass eine dieser Zeichenketten erwartet wird. Zum Beispiel bedeutet `{info,hinweis,hilfe}.txt`, dass eine der drei Dateien *info.txt*, *hilfe.txt* oder *hinweis.txt* gesucht wird.
- Durch ein Pipe-Zeichen (`|`) können Sie schließlich mehrere Muster angeben, die durch *oder* verknüpft werden. Trifft eines dieser Muster auf eine Datei zu, passt sie zum Gesamtmuster. Der Ausdruck `b*|info*` bedeutet beispielsweise: alle Dateien, die mit »b« oder mit »info« beginnen.

Beachten Sie bitte, dass die *Dateierweiterung* (die Abkürzung hinter dem letzten Punkt, wie etwa *txt*) unter Unix ein normaler Bestandteil des Dateinamens ist. Falls Sie hauptsächlich mit den Dateimustern unter Windows vertraut sind, erscheint dies im ersten Moment vielleicht fremdartig. In einem Unix-Befehl steht `*` für alle Dateien. Unter Windows ist ein `*` dagegen nur der Platzhalter für Dateien ohne Erweiterung, während `*.*` dort für alle Dateien steht (außer in der Windows PowerShell). Diese einfachen Suchmuster für Dateien werden übrigens *nicht* mit dem bereits

erwähnten Befehl `grep` verwendet. Die dort zulässigen Muster bieten noch erheblich mehr Möglichkeiten.

Die folgende Übersicht zeigt die gängigsten Linux-Datei- und Verzeichnisbefehle mit ihren wichtigsten Optionen:

- `cp` (steht für *copy*) kopiert eine oder mehrere Dateien an den angegebenen Ort. Die Syntax ist grundsätzlich folgende:

  ```
  cp Quelle Ziel
  ```

  Die Quelle kann eine einzelne Datei oder ein Muster sein; Sie können alternativ auch einen Pfad angeben. Das Ziel ist entweder ein einzelner Dateiname (falls Sie nur eine Datei kopieren) oder ein Verzeichnis, falls im Zielordner bereits ein Verzeichnis mit diesem Namen existiert oder falls Sie als Quelle keine einzelne Datei, sondern ein Muster angegeben haben. Das folgende Beispiel kopiert die Datei *hallo.txt* in eine neue Datei namens *hi.txt*:

  ```
  $ cp hallo.txt hi.txt
  ```

  Das nächste Beispiel kopiert alle Dateien aus dem Verzeichnis *briefe* in das Verzeichnis *dokumente*, das im selben Verzeichnis liegt wie *briefe*:

  ```
  $ cp briefe/* dokumente
  ```

  Die Option `-r` (*recursive*) kopiert das angegebene Verzeichnis mit allen Unterverzeichnissen und darin enthaltenen Dateien.

- `mv` (*move*) dient dazu, Dateien umzubenennen oder in ein anderes Verzeichnis zu verschieben. Die Syntax lautet folgendermaßen:

  ```
  mv Quelle Ziel
  ```

  Die Quelle ist wieder eine einzelne Datei oder ein Muster, das Ziel ist ein völlig neuer Name oder der Name eines bestehenden

Verzeichnisses.
Diese Anweisung benennt die Datei *vorher.txt* in *nachher.txt* um:

```
$ mv vorher.txt nachher.txt
```

Wenn Sie als Quelle ein Muster anstelle einer einzelnen Datei angeben, muss das Ziel ein bestehendes Verzeichnis sein. Sie können mehrere Dateien auf einmal nicht umbenennen, sondern nur verschieben.

- `rm` (*remove*) löscht die angegebene Datei, und zwar endgültig. Eine Einrichtung wie der Windows- oder Mac-Papierkorb ist nicht vorgesehen – lediglich einzelne Desktop-Manager wie KDE oder GNOME sind damit ausgestattet.
  Das folgende Beispiel löscht alle Dateien aus dem aktuellen Verzeichnis, deren Name nicht mit `a` beginnt:

  ```
  $ rm [!a]*
  ```

  `rm` löscht Dateien nur im aktuellen Verzeichnis, aber nicht in dessen Unterverzeichnissen. Wenn Sie auch die Inhalte der Unterverzeichnisse löschen möchten, müssen Sie die Option `-r` (*recurse*) einsetzen. Noch effizienter (und gefährlicher!) ist die zusätzliche Option `-f` (*force*), die das Löschen schreibgeschützter Dateien erzwingt. Der folgende Befehl löscht alle Dateien im aktuellen Verzeichnis und alle Unterverzeichnisse und sollte nur mit äußerster Vorsicht eingesetzt werden:

  ```
  $ rm -rf *
  ```

  Wie hier können Sie übrigens auch bei den meisten anderen Befehlen mehrere Optionen hinter einem einzelnen Minuszeichen platzieren.
  Beachten Sie, dass es sich beim Löschen, Umbenennen oder Verschieben um *Schreibzugriffe* handelt, die Sie nur ausführen

dürfen, wenn Sie Schreibrechte für die jeweiligen Verzeichnisse und Dateien besitzen.

- `ln` (*link*) erzeugt sowohl harte Links (Inode-Verzeichniseinträge) als auch Symlinks. Die Syntax lautet `ln [Optionen] Quellpfad Zielpfad`. Die wichtigste Option ist `-s` für einen Symlink. Wenn Sie einen harten Link erzeugen, müssen Quellpfad und Zielpfad Dateinamen im selben Verzeichnis sein, während Symlinks auf Verzeichniseinträge im gesamten Dateisystem verweisen können.
- `ls` (*list*) zeigt den Inhalt des aktuellen oder des angegebenen Verzeichnisses an, also alle enthaltenen Dateien und Unterverzeichnisse. Wenn Sie ein Muster angeben, wird es als Filter verwendet. Existieren Dateien, deren Namen zu diesem Muster passen, werden nur diese angezeigt. Andernfalls werden zusätzlich zum aktuellen Verzeichnis auch die Inhalte der Unterverzeichnisse angezeigt, auf deren Namen das Muster passt. Die folgende Anweisung zeigt beispielsweise alle Dateien an, die mit `b` beginnen. Falls es keine gibt, werden alternativ die Inhalte aller Verzeichnisse angezeigt, die mit `b` anfangen:

  ```
  $ ls b*
  ```

  Eine wichtige Option dieses Befehls ist `-l` (*long*), die anstelle der einfachen Namen ausführliche Informationen über jeden Verzeichniseintrag ausgibt. Auch `-a` (*all*) wird relativ häufig verwendet, weil es versteckte Dateien und Verzeichnisse einblendet, das heißt diejenigen, deren Namen mit einem Punkt beginnen. Interessant ist schließlich noch die Option `-h` (*human-readable*), die die Dateigrößen nicht in Byte anzeigt, sondern je nach Größenordnung in KiB oder MiB mit Einheiten wie `K` beziehungsweise `M`.
- `pwd` (*print working directory*) gibt den vollständigen Pfad des aktuellen Arbeitsverzeichnisses an. Dies ist beispielsweise

nützlich, um den tatsächlichen Pfad des eigenen Home-Verzeichnisses zu ermitteln, der im Prompt durch `~` abgekürzt wird.

- `cd` (*change directory*) wechselt in das angegebene Arbeitsverzeichnis. Sie können den gewünschten Pfad entweder relativ zum aktuellen Arbeitsverzeichnis oder absolut durch einen vorangestellten Slash (`/`) angeben. Die folgende Anweisung wechselt beispielsweise aus */home/user/dokumente* in das Verzeichnis */home/user/briefe*:

  ```
  user@rechner: ~/dokumente $ cd ../briefe
  user@rechner: ~/briefe $
  ```

  Das nächste Beispiel wechselt dagegen mithilfe einer absoluten Angabe von */home/user/dokumente* nach */etc*:

  ```
  user@rechner: ~/dokumente $ cd /etc
  user@rechner: /etc $
  ```

- `mkdir` (*make directory*) legt ein neues Verzeichnis mit dem angegebenen Pfad an. So richtet etwa die folgende Anweisung unterhalb des aktuellen Verzeichnisses das neue Verzeichnis *test* ein:

  ```
  $ mkdir test
  ```

  Beachten Sie, dass bei der Angabe eines mehrgliedrigen Pfads alle Verzeichnisse außer dem hintersten bereits existieren müssen. Die Option `-p` (*parents*) erzeugt dagegen auch verschachtelte Pfade. Das folgende Beispiel legt im aktuellen Verzeichnis die ineinander verschachtelten Verzeichnisse *neu*, *texte* und *briefe* an:

  ```
  $ mkdir -p neu/texte/briefe
  ```

  Wenn Sie ein Verzeichnis erstellen und im Erfolgsfall gleich hineinwechseln möchten, können Sie `mkdir` und `cd` wie folgt

kombinieren (hier mit einem Verzeichnis namens *new_directory* unterhalb des aktuellen Arbeitsverzeichnisses):

```
$ mkdir new_directory && cd $_
```

Die automatische Variable `$_` steht dabei für das zuletzt angegebene Argument.

- `rmdir` (*remove directory*) löscht Verzeichnisse, allerdings nur leere. Zum Löschen verschachtelter Verzeichnisbäume wird `rm` mit der Option `-r` verwendet.
- `chmod` (*change mode*) ändert die Zugriffsrechte für Dateien und Verzeichnisse. Das Konzept der Dateizugriffsrechte wurde in diesem Kapitel bereits angesprochen. Es gibt grundsätzlich zwei Möglichkeiten, Rechte für die gewünschten Dateien oder Verzeichnisse anzugeben: symbolisch oder numerisch.
  Die symbolische Schreibweise verwendet zunächst einen Buchstaben für die Benutzerart, für die ein Recht geändert werden soll: `u` für den Owner-Account (*user*), `g` für die Gruppe (*group*), `o` für andere Accounts (*others*) und `a` für alle genannten auf einmal. Darauf folgt ein `+`, um ein bestimmtes Recht einzuräumen, ein `-`, um es zu entfernen, oder ein `=`, um die angegebenen Rechte zu setzen und die anderen zu entfernen. Zum Schluss werden die eigentlichen Rechte selbst angegeben: `r` für Lesen (*read*), `w` für Schreiben (*write*) und `x` für Ausführen (*execute*). Beachten Sie, dass Verzeichnisse das *execute*-Recht benötigen, um ihren Inhalt anzuzeigen oder in sie hineinzuwechseln.
  Die folgende Anweisung erlaubt beispielsweise allen das Lesen der Datei *inhalt.txt*:

```
$ chmod a+r inhalt.txt
```

Numerische Angaben setzen dagegen den gesamten Rechteblock für die Datei auf einmal: Die Stellen einer dreistelligen Oktalzahl (gekennzeichnet durch eine vorangestellte Null) geben von links nach rechts die Zugriffsrechte für den Owner-Account, die Gruppe und alle anderen an. Der Wert jeder Stelle ist dabei die Summe der Rechte, die gewährt werden: 4 für Lesen, 2 für Schreiben und 1 für Ausführen.
Das folgende Beispiel erlaubt dem Owner-Account das Lesen, Schreiben und Ausführen, allen anderen nur das Lesen und Ausführen des Verzeichnisses *test*:

```
$ chmod 0755 test
```

Die Option `-R` (großgeschrieben!) führt die gewünschte Änderung nicht nur im aktuellen Verzeichnis durch, sondern auch in allen Unterverzeichnissen.

- `chown` (*change owner*) weist der angegebenen Datei einen neuen Owner-Account zu. Die Syntax des Befehls ist folgende:

  ```
  chown User Datei(-muster)
  ```

  `User` muss ein existierender User-Account sein; außerdem können Sie diese Änderung nur durchführen, wenn Sie selbst Schreibrechte an dieser Datei haben. Das folgende Beispiel teilt die Datei *info* dem Account `user` zu:

  ```
  $ chown user info
  ```

  Wenn Sie gleichzeitig die Gruppe ändern möchten, können Sie den Gruppennamen durch einen Doppelpunkt getrennt hinter den Usernamen schreiben:

  ```
  $ chown user:users info
  ```

- `chgrp` (*change group*) ändert die Gruppe, zu der eine Datei gehört, und funktioniert genau wie `chown`.

Textanzeige und Textmanipulation

Viele der Arbeiten, die Sie im Betriebssystem durchführen, haben in irgendeiner Weise mit der Manipulation von Textdateien zu tun. In diesem Abschnitt werden einige der wichtigsten Befehle vorgestellt, die Ihnen die Arbeit mit solchen Dateien ermöglichen.

- Der bereits erwähnte Befehl `echo` gibt sämtlichen folgenden Text auf der Konsole aus. Sie können den gesamten Text oder einen Teil davon in Anführungszeichen setzen, müssen es aber nicht. Wenn Sie doppelte (`""`) oder gar keine Anführungszeichen verwenden, werden Variablen mit führendem Dollarzeichen durch ihren aktuellen Wert substituiert oder Befehle in Backticks ausgeführt. Beispiele:

  ```
  $ echo Hallo, $USER!
  Hallo, sascha!
  $ echo Dateien im aktuellen Verzeichnis: `ls -m`
  Dateien im aktuellen Verzeichnis: test.txt, hallo.sh, ...
  ```

  Die `ls`-Option `-m` gibt übrigens nur die Dateinamen durch Kommata getrennt hintereinander aus.
  Einfache Anführungszeichen verhindern dagegen die Substitution:

  ```
  $ echo '`Backticks` liefern die Befehlsausgabe'
  `Backticks` liefern die Befehlsausgabe
  $ echo '$USER' ist zurzeit $USER
  $USER ist zurzeit sascha
  ```

  Die Option `-n` verhindert den Zeilenumbruch nach der Ausgabe:

  ```
  $ echo -n "Hier kommt der Prompt: "
  Hier kommt der Prompt: ...$
  ```

- `cat` (*concatenate* oder auch *catalog*) ist der wichtigste aller Textdateibefehle: Er zeigt einfach den Inhalt der Datei an. Wenn Sie durch Leerzeichen getrennt eine Liste von Dateien angeben (oder ein Muster), werden die Inhalte aller genannten Dateien

hintereinander angezeigt. Dies ist übrigens eine einfache Möglichkeit, mehrere Textdateien in einer einzigen zusammenzufassen. Die folgende Anweisung schreibt die Dateien *teil1* und *teil2* in eine neue Datei namens *kapitel*:

```
$ cat teil1 teil2 >kapitel
```

Sie können `cat` mithilfe der Ausgabeumleitung und mit einem Hier-Dokument sogar als einfachen Editor für eine neue Textdatei verwenden. Allerdings können Sie die einzelnen Zeilen nach dem Abschluss durch ↵ nicht mehr ändern. Die folgende Anweisung startet die Eingabe der Datei *neu.txt*, die Eingabe `ENDE` schließt sie ab:

```
$ cat > neu.txt << ENDE
> Neuer Text
> Noch mehr Text
> ENDE
```

- Der Befehl `head` *`Textdatei`* zeigt nur den Beginn einer Datei an, standardmäßig die ersten zehn Zeilen. Mit der Option *`-Anzahl`* können Sie die Zeilenzahl auch bestimmen. Das folgende Beispiel gibt die ersten sieben Zeilen der Datei *test.txt* aus:

  ```
  $ head -7 test.txt
  ```

- Das Kommando `tail` zeigt umgekehrt das Ende einer Datei an. Dies ist ideal, um in einer Log-Datei nach einem kürzlich aufgetretenen Fehler zu suchen. Hier ein Beispiel, das die letzten 20 Zeilen der Haupt-Log-Datei */var/log/messages* ausgibt:

  ```
  $ tail -20 /var/log/messages
  ```

  Die Option `-f` zeigt die letzten zehn Zeilen an und hält danach die Ausgabe offen, um jede neu hinzukommende Zeile automatisch auszugeben. Dies ermöglicht die Live-Überwachung von Log-Dateien, zum Beispiel bei der Fehlersuche. Besonders die

Möglichkeit, die Ausgabe mit `grep` zu filtern, macht dieses Verfahren sehr leistungsfähig.

- `less` ist die erweiterte GNU-Version des Unix-Pagers `more`. Das Programm gibt seine Eingabedaten bildschirmseitenweise aus. Das folgende Beispiel gibt die Datei *roman* auf diese Weise aus:

  ```
  $ less roman
  ```

  Wenn das untere Ende des Bildschirms beziehungsweise des Terminalfensters erreicht ist, erscheint ein entsprechender Hinweis. Die möglichen Navigationstasten wurden bereits für den Befehl `man` beschrieben, der die Manpages mithilfe von `less` anzeigt.
  Anstatt eine oder mehrere Dateien als Parameter anzugeben, wird `less` auch häufig über eine Pipe zur seitenweisen Ausgabe der Ergebnisse anderer Befehle eingesetzt. Das folgende Beispiel gibt den Inhalt der ausführlichen Verzeichnisanzeige `ls -l` seitenweise aus:

  ```
  $ ls -l |less
  ```

- `grep` (*general regular expression print*) sucht in Dateien oder in seiner Eingabe nach Mustern und gibt nur diejenigen Zeilen aus, die das entsprechende Muster enthalten. Bei den verwendeten Mustern handelt es sich um sogenannte *reguläre Ausdrücke* (*Regular Expressions*, *RegExp*). Diese mächtige Syntax für die Formulierung von Suchmustern wird in zahlreichen Programmiersprachen, Editoren und Tools verwendet. In Kapitel 9, »Weitere Konzepte der Programmierung«, werden die RegExp-Optionen der Programmiersprachen Python und Java ausführlich vorgestellt.
  Wenn Sie beispielsweise in der Datei *test* nach Zeilen suchen möchten, die das Wort »hallo« enthalten, funktioniert dies

folgendermaßen:

```
$ grep hallo test
```

Die Option `-r` erlaubt die rekursive Suche im aktuellen Verzeichnis und allen Unterverzeichnissen. Dabei wird jeweils der Pfad der Datei angezeigt, in der das gesuchte Muster gefunden wird. Das folgende Beispiel sucht in allen Dateien des aktuellen Verzeichnisbaums nach dem Wort »Linux«:

```
$ grep -r Linux *
```

Mit der Option `-i` sorgen Sie dafür, dass bei der Suche nicht zwischen Groß- und Kleinschreibung unterschieden wird; der vorangegangene Befehl lautet in diesem Fall so:

```
$ grep -ri linux *
```

Alternativ wird `grep` häufig über eine Pipe als Filter eingesetzt. Möchten Sie zum Beispiel alle Dateien im aktuellen Verzeichnis angezeigt bekommen, deren Name mit `a` beginnt, können Sie Folgendes eingeben:

```
$ ls |grep ^a
```

Beachten Sie, dass es bei den Mustern größere Unterschiede zu den Dateimustern der meisten Befehle gibt; RegExp-Muster sind erheblich vielseitiger als Letztere. Tabelle 6.4 zeigt eine Übersicht der wichtigsten.

Muster (Beispiel)	Erläuterung
`abc`	der Text abc
`[abc]`	eines der Zeichen a, b oder c

Muster (Beispiel)	Erläuterung
`[a-z]`	eines der Zeichen von a bis z
`[a-mz0-9]`	eines der Zeichen von a bis m oder z oder eine der Ziffern von 0 bis 9
`[^abc]`	keines der angegebenen Zeichen
`.`	ein beliebiges Zeichen
`?`	das davorstehende Muster oder nichts
`*`	das davorstehende Muster beliebig oft
`+`	das davorstehende Muster einmal oder ofter
`{2}`	das davorstehende Muster genau zweimal
`{2,5}`	das davorstehende Muster mindestens zweimal, hochstens funfmal
`^[Aa]`	am Zeilenbeginn (`^`) steht ein gro es A oder ein kleines a
`[0-9]$`	am Zeilenende (`$`) steht eine Ziffer

Tabelle 6.4 Die wichtigsten RegExp-Muster fur grep

Wenn Sie irgendeins der Sonderzeichen aus der Tabelle als Literal benötigen, also als tatsächliches Zeichen in einem Text, müssen Sie ihm einen Backslash (`\`) voranstellen. `\+` steht beispielsweise für ein Pluszeichen. Derartige Konstrukte werden in der Shell und in vielen Programmiersprachen als *Escape-Sequenzen* bezeichnet. Angenommen, Sie suchten in einem Text nach deutschen Postleitzahlen. Das passende Muster lautet `[0-9]{5}`, weil genau

fünf Ziffern (Zahlen zwischen 0 und 9) benötigt werden. Wenn Sie sicher sind, dass die Postleitzahl jeweils am Anfang der Zeile steht, können Sie präziser `^[0-9]{5}` schreiben.
Wichtig ist, dass `*` und `?` nicht dasselbe bedeuten wie bei den Dateimustern: Sie beziehen sich stets auf das links danebenstehende Zeichen oder Teilmuster und geben an, wie oft es vorkommen darf.
Das folgende Beispiel zeigt, wie Sie in der Datei *adressen.txt* nach Personen suchen können, die »Meier« heißen, und zwar in allen erdenklichen Schreibweisen:

```
grep M[ae][iy]e?r adressen.txt
```

Der reguläre Ausdruck bedeutet: Zuerst kommt ein M, dann ein a oder e, anschließend ein i oder y, dann ein e oder auch nicht und zum Schluss ein r. Dieses Suchmuster findet die Varianten »Maier«, »Mayer«, »Mayr«, »Meier« und »Meyer« sowie drei weitere, die aber wohl nicht häufig in einer Adressliste auftauchen (»Mair«, »Meir« und »Meyr«).

- `diff` vergleicht die Inhalte zweier Textinhalte miteinander. Als Argumente werden die Namen der beiden Dateien genannt; die Ausgabe besteht aus denjenigen Zeilen, die in den beiden Dateien unterschiedlich sind. Dies ermöglicht die Analyse der Unterschiede zwischen verschiedenen Versionen eines Dokuments. Außerdem können Sie eine `diff`-Datei als Update für eine Datei auf eine neuere Version (einen sogenannten *Patch*) verbreiten. Mit dem Kommando `patch` kann sie dann auf die alte Datei angewendet werden.
- `wc` (*word count*) zählt die Zeichen, Wörter und Zeilen in einer Textdatei. Standardmäßig werden alle drei Werte angezeigt; alternativ kann die Ausgabe mithilfe der Optionen `-c` (*characters*), `-w` (*words*) oder `-l` (*lines*) auf einen von ihnen beschränkt werden.

Sie können den Befehl auch über eine Pipe auf eine beliebige Ausgabe anwenden.

6.4.3 Automatisierung

Unix-Systeme enthalten zahlreiche Möglichkeiten, Routineaufgaben zu automatisieren. In diesem Abschnitt lernen Sie zunächst Shell-Skripte kennen, in denen Sie programmgesteuerte Kommandoabfolgen zusammenfassen können. Anschließend geht es um Hilfsmittel wie Aliasse, Cronjobs und Log-Dateien.

Shell-Skripte

Eine besondere Eigenschaft von Unix-Shells sind die eingebauten Befehle, mit deren Hilfe sich bestimmte Aufgaben automatisieren lassen. Im Grunde ist dieser Abschnitt ein Vorgriff auf das Thema des nächsten Kapitels, die Programmierung. Dennoch sollen hier einige Worte zum Shell-Scripting gesagt werden. Falls Sie noch nie programmiert haben, sollten Sie sich allerdings zuerst besagtes Kapitel 7, »Grundlagen der Programmierung«, vornehmen. Konzepte, die an dieser Stelle nur ganz kurz und knapp angesprochen werden können, werden dort ausführlich erläutert.

Ein Shell-Skript ist im Grunde eine einfache Textdatei, die eine Abfolge von Shell- und Systembefehlen enthält. Diese Befehle werden beim Aufruf dieses Skripts nacheinander abgearbeitet. Für gewöhnlich erhalten Shell-Skripte die Dateiendung *.sh*. Der Name spielt aber eigentlich keine Rolle; wichtig ist, dass Sie das Skript mithilfe von `chmod` ausführbar machen.

Wie jedes Konsolenprogramm kann ein Shell-Skript Kommandozeilenparameter entgegennehmen. Hinter dem Namen

des Skripts kann also beim Aufruf eine durch Leerzeichen getrennte Liste von Zeichenfolgen stehen, die das Skript verarbeiten kann.

Die erste Zeile eines Shell-Skripts enthält die sogenannte *Shebang*-Angabe – ein Kurzwort für die Zeichen `#` (*sharp*) und `!` (*bang*), mit denen sie beginnt. An dieser Stelle steht, welche Shell dieses Skript ausführen soll, da sich verschiedene Shells in ihrer Skriptsyntax voneinander unterscheiden. Hier sehen Sie ein Beispiel für eine Shebang-Zeile; das zugehörige Skript wurde für die *bash* geschrieben:

```
#!/bin/bash
```

Nach dieser Zeile können Sie einen Befehl nach dem anderen untereinanderschreiben. Alle bisher besprochenen Systembefehle sind zulässig, darüber hinaus werden einige spezielle programmiertechnische Erweiterungen verwendet.

Die wichtigsten zusätzlichen Befehle und Elemente für Shell-Skripte sind folgende (hier konkret für die *bash*, bei anderen Shells kann die Syntax leicht abweichen):

- *Fallentscheidungen*. Anweisungen, die zwischen `if` *`Bedingung`* `then` und `fi` stehen, werden nur ausgeführt, wenn die Bedingung zutrifft. In der Regel besteht die Bedingung aus dem Vergleich zwischen einer Variablen oder einem Dateinamen und einem bestimmten Wert. Ein solcher Ausdruck steht bei Text- und Dateivergleichen in eckigen Klammern. Dabei stehen unter anderem folgende Vergleichsmöglichkeiten zur Verfügung:
  - `[ -e` *`Pfad`* `]` – Verzeichniseintrag existiert.
  - `[ -f` *`Pfad`* `]` – Verzeichniseintrag ist eine Datei.
  - `[ -d` *`Pfad`* `]` – Verzeichniseintrag ist ein Verzeichnis.
  - `[ -s` *`Pfad`* `]` – Verzeichniseintrag ist ein Symlink.

- `[ str1 = str2 ]` – Strings sind identisch.
- `[ str1 != str2 ]` – Strings sind verschieden.
- `[ -z str ]` – String ist leer (*zero*).
- `[ -n str ]` – String hat Inhalt (*nonzero*).
- `[ expr1 -a expr2 ]` – Beide Ausdrücke sind wahr (*and*).
- `[ expr1 -o expr2 ]` – Mindestens ein Ausdruck ist wahr (*or*).

Für numerische Vergleiche werden doppelte runde Klammern verwendet: `((...))`. Darin stehen die bekannten Vergleichsoperatoren `==`, `!=`, `<`, `>`, `<=` und `>=` zur Verfügung.
Alternativ kann auch der Erfolg eines Kommandos getestet werden. Dazu wird der Befehl mit seinen Parametern und Argumenten ohne weitere Kennzeichnung zwischen `if` und `then` geschrieben. Befehle werden mit einem Exit-Code beendet. 0 bedeutet in der Regel, dass alles in Ordnung ist, andere Werte deuten auf Fehler hin. Die Anweisungen werden bearbeitet, wenn der Befehl korrekt ausgeführt wird, weil der Exit-Code 0 als wahr gilt und jeder andere Wert als falsch.[45]
Hinter `else` können Sie alternative Anweisungen definieren, die ausgeführt werden sollen, falls die Bedingung nicht zutrifft, mit `elif` (Abkürzung für `else if`) können sogar eine oder mehrere zusätzliche Bedingungsprüfungen eingesetzt werden.

- *Einzelfallentscheidungen*. Zwischen `case` und `esac` können Sie verschiedene Muster aufführen, denen eine angegebene Variable oder ein String mit Variablen entsprechen kann. Die vollständige Schreibweise ist `case String in`. Hinter jedem Muster, das überprüft wird, steht eine schließende Klammer `)`. Darauf folgen beliebig viele Anweisungen, die nur ausgeführt werden, wenn das angegebene Muster auf den überprüften String passt. Vor dem nächsten Muster muss eine Befehlssequenz durch `;;`

abgeschlossen werden. Am Schluss kann `*)` stehen, um sämtliche noch nicht anderweitig gefundenen Werte zu verarbeiten – etwa um ungültige Argumente per Fehlermeldung abzufangen.

- *Schleifen*. Mitunter müssen bestimmte Anweisungen mehrfach ausgeführt werden. Dafür sind Schleifen zuständig. Die *bash* definiert verschiedene Arten von Schleifen; die wichtigsten sind die `for`-Schleife und die `while`-Schleife.
  Eine `for`-Schleife geht automatisch alle Kommandozeilenparameter durch, die Parameter werden nacheinander der angegebenen Schleifenvariablen zugewiesen. Alternativ können Sie mithilfe von `for ... in` ein Dateimuster angeben. Die Variable nimmt dann nacheinander den Namen jeder Datei an, auf die dieses Muster passt.
  Die `while`-Schleife verwendet dagegen eine Bedingung wie `if`, jedoch mit dem Unterschied, dass die Anweisungen mehrmals ausgeführt werden, solange die Bedingung noch zutrifft.
  Die Anweisungen, die in jedem Durchlauf der Schleife ausgeführt werden sollen, stehen in beiden Fällen zwischen `do` und `done`.
- *Variablen*. Mithilfe der Anweisung *`var=Wert`* wird einer Variablen innerhalb eines Shell-Skripts ein Wert zugewiesen. Es kann sich dabei sowohl um eine der Umgebungsvariablen wie `PATH` handeln als auch um beliebige temporäre Variablen, die nur innerhalb des Skripts verwendet werden, um Ihnen die Arbeit zu erleichtern. Wenn die Variablendefinition in einer Shell und allen in ihr aufgerufenen Skripten gültig bleiben soll, müssen Sie in der *bash* übrigens das Kommando `export` *`var=Wert`* verwenden.
  Innerhalb von Befehlen, die Sie in einem Shell-Skript aufrufen, wird eine Variable durch ein vorangestelltes `$`-Zeichen vor der Ausführung des Befehls durch ihren aktuellen Wert ersetzt (substituiert). Innerhalb eines komplexen Ausdrucks steht der

Variablenname hinter dem Dollarzeichen zusätzlich in geschweiften Klammern `{...}`.
Spezielle Variablen sind `$0` bis `$9` für die einzelnen Kommandozeilenparameter, `$*` für die gesamte Liste dieser Parameter zur Verwendung in einer Schleife und `$#` für die Anzahl der übergebenen Parameter.

Es folgen zwei kleine Beispiele, die die praktische Verwendung der zuvor erläuterten Anweisungen demonstrieren. Zuerst sehen Sie hier ein Skript, das im nächsten Kapitel (ohne Datum und Uhrzeit) auch in den beiden ausführlicher vorgestellten Programmiersprachen gezeigt wird. Es gibt den klassischen Programmiereinstiegssatz »Hallo, Welt!« aus, fügt Datum und Uhrzeit hinzu, fragt nach dem Namen und verwendet diesen anschließend für einen persönlichen Gruß. Hier der Code:

```
#!/bin/bash
echo Hallo, Welt!
echo Es ist `date +"%d.%m.%Y, %H:%M"`
echo -n "Wie heißt du? "
read name
echo Hallo, $name!
```

Speichern Sie das Skript zum Beispiel unter dem Namen *hallo.sh* und machen Sie es wie folgt ausführbar:

```
$ chmod +x hallo.sh
```

Danach können Sie es ausführen. Hier ein komplettes Ein- und Ausgabebeispiel:

```
$ ./hallo.sh
Hallo, Welt!
Es ist 07.05.2023, 17:43
Wie heißt du? Sascha
Hallo, Sascha!
```

Die meisten in dem Skript verwendeten Anweisungen wurden bereits erläutert.

Das folgende kurze Beispiel definiert ein Skript namens *backup*, das alle Dateien, die auf ein angegebenes Dateimuster passen, in Dateien mit der zusätzlichen Endung *.tmp* sichert:

```
#!/bin/bash
if (( $# < 1 ))
then
   echo "Verwendung: backup Dateimuster"
   exit 1
fi
for i in $*
do
  echo "Verarbeite ${i}"
  if [ -f $i ]
  then
    cp $i ${i}.tmp
  fi
done
```

Wenn kein Dateimuster angegeben wird, erscheint eine Warnmeldung, und das Skript wird mit einem Fehlercode beendet. Andernfalls werden alle gewöhnlichen Dateien (Test `-f`), die zum angegebenen Dateimuster passen, in einer Schleife nacheinander in entsprechenden *.tmp*-Dateien gesichert. Der folgende Aufruf sichert alle Dateien mit der Endung *.txt* im aktuellen Verzeichnis:

```
$ ./backup *.txt
```

Im nächsten Kapitel wird übrigens die Skriptsprache Python vorgestellt, die mit einer eingängigeren Syntax und erheblich umfangreicheren Möglichkeiten eine hervorragende Ergänzung oder gar Alternative zu Shell-Skripten bietet.

Aliasse

Aliasse sind eine Möglichkeit der *bash*, längere Befehlseingaben, etwa mit zahlreichen Parametern, zu einer Art Makro zu verkürzen. Grundsätzlich geschieht dies mithilfe einer Eingabe nach dem Schema:

```
alias Aliasname='Kommando [Argumente]'
```

Hier ein Beispiel, das `ls` mit den Optionen `-l` (Detailinformationen) und `-a` (auch versteckte Dateien anzeigen) unter dem Alias `l` bereitstellt:

```
$ alias l='ls -la'
```

Aufgerufen wird ein solcher Alias wie ein gewöhnliches Kommando. Beispiel:

```
$ l
```

Auch Aliassen können Sie Optionen oder Argumente übergeben, solange der zugrunde liegende Befehl diese unterstützt. Das folgende Beispiel ruft den Alias `l` mit der zusätzlichen Option `-X` (nach Dateiendungen sortieren) sowie dem Dateimuster `a*` (mit `a` beginnende Dateien und Verzeichnisse) auf:

```
$ l -X a*
```

Die Definition eines Alias können Sie sich mit `alias` *`Name`* anschauen, also etwa:

```
$ alias l
```

`alias` ohne Argumente zeigt alle aktuellen Alias-Definitionen an. `unalias` *`Name`* löscht den angegebenen Alias. Genau wie Umgebungsvariablen sind auch Aliasse in vielen Distributionen in den Shell-Konfigurationsskripten vorkonfiguriert.

Cronjobs

Manche Administrationsaufgaben oder Aufräumarbeiten müssen regelmäßig erledigt werden. Mit *Cronjobs* bieten Unix-Systeme eine flexible Möglichkeit, solche Aufgaben zu automatisieren.

Die einfachere Methode besteht darin, einen Symlink auf das auszuführende Programm oder Shell-Skript in einem der folgenden Verzeichnisse anzulegen:

- */etc/cron.hourly* (stündlich)
- */etc/cron.daily* (täglich)
- */etc/cron.weekly* (wöchentlich)
- */etc/cron.monthly* (monatlich)

Das folgende Beispiel sorgt dafür, dass ein Skript namens *daily.sh* jeden Tag ausgeführt wird:

```
# ln -s /usr/bin/skripten/daily.sh \
> /etc/cron.daily/daily.sh
```

Erheblich flexibler, aber etwas komplizierter in der Handhabung ist die Arbeit mit sogenannten *Crontabs*, die für jeden User einzeln angelegt werden. Es handelt sich um Textdateien, in denen jede Zeile einen Eintrag in folgendem Format darstellt:

```
Min Std Tag Mon Wochentag Kommando
```

Für die Zeit-Felder gelten folgende Wertebereiche:

- Minute: 0–59
- Stunde: 0–23
- Tag im Monat: 1–31
- Monat: 1–12
- Wochentag: 0–6 (0 = Sonntag, 1 = Montag ... 6 = Samstag)

Ein `*` in einem der Felder sorgt dafür, dass der Cronjob für jeden Wert in diesem Feld gilt. Hier ein Beispiel, das ein Skript namens *cleanup.sh* jeden Freitag um 18:00 Uhr ausführt:

```
0 18 * * 5 /usr/bin/skripten/cleanup.sh
```

Das folgende Beispiel ruft zu jeder vollen Stunde das Skript *hourly.sh* auf:

```
0 * * * * /usr/bin/skripten/hourly.sh
```

Auch Bereiche in der Form *Start-Ende* sind für ein Feld möglich. Das folgende Beispiel führt montags bis samstags von 8 bis 18 Uhr jeweils stündlich ein Skript namens *work.sh* aus:

```
0 8-18 * * 1-6 /usr/bin/skripten/work.sh
```

Hinter `*` oder *Start-Ende* können Sie optional einen `/` und ein Intervall angeben. Das folgende Beispiel führt alle zwei Stunden `who -a` aus, um eine vollständige Liste der angemeldeten User-Accounts und virtuellen Terminals zu erzeugen:

```
0 */2 * * * who -a
```

Eventuelle Ausgaben der betreffenden Skripte werden mithilfe des Kommandos `mail` an den betreffenden User-Account geschickt. Optional fügen Sie als erste Zeile der Crontab eine Anweisung wie diese ein, um die Mails an einen anderen Account zu schicken (sinnvoll beispielsweise für die *root*-Crontab):

```
MAILTO=sascha
```

Geben Sie `mail` ein, um solche Nachrichten zu lesen. Die jeweilige Nummer zeigt eine bestimmte Nachricht an; Q beendet sowohl die einzelne Nachricht als auch das Programm.

Der Befehl `crontab` mit der Option `-e` ermöglicht die Bearbeitung der Crontab im Editor *vi*. Drücken Sie darin die Taste I, um mit der Eingabe zu beginnen. Drücken Sie danach Esc und geben Sie `:wq` ↵ ein, um Ihre Änderungen zu speichern und den Editor zu beenden. `crontab -l` liefert die aktuelle Crontab, und `crontab -r` entfernt sie. Als *root* können Sie auch die Option `-u` *Username*

hinzufügen, um die Crontab eines anderen Accounts zu lesen oder zu ändern.

Syslog und Log-Dateien

Das Betriebssystem selbst und viele Programme, vor allem Daemons, schreiben Statusinformationen und Fehlermeldungen in *Log-Dateien*, auch *Protokolldateien* genannt. Dafür ist der *Syslog-Daemon* (*syslogd*) zuständig. Er erhält Protokollinformationen vom System und von diversen Programmen und entscheidet darüber, ob er sie ignoriert, in bestimmte Log-Dateien schreibt oder per Mail versendet.

Syslog-Meldungen bestehen aus drei Komponenten:

- *Facility* (Fehlerquelle) gibt an, von welcher Art Programm oder welcher Systemfunktion der Eintrag stammt – zum Beispiel `auth` für die Authentifizierung, `cron` für den Cron-Daemon oder `kern` für den Kernel.
- *Priority* (Dringlichkeitsstufe) bestimmt die Wichtigkeit der jeweiligen Meldung – von `emerg` (Notfall, der ein unbrauchbares System hinterlässt) unter anderem über `error` (normaler Fehler), `warn` (Warnung) und `info` (reine Information) bis hinunter zu `debug` (willkürlich erzeugte Meldung zur Programmfehlersuche).
- *Message* (Warnmeldung) – ein beliebiger Beschreibungstext.

In eigenen Shell-Skripten können Sie den Befehl `logger` nutzen, um Syslog-Meldungen zu erzeugen. Die allgemeine Syntax lautet:

```
logger [-p [facility:]priority] message
```

Geben Sie Folgendes ein, um es zu testen:

```
$ logger -p info Nur Test, kein Fehler
```

Da Sie keine Facility angegeben haben, steht die Meldung in */var/log/messages* (in manchen Systemen auch in */var/log/syslog*):

```
$ tail -1 /var/log/messages
Apr 23 18:51:32 linuxbox sascha: Nur Test, kein Fehler
```

6.5 Übungsaufgaben

Im Folgenden ist jeweils genau eine Antwort richtig.

1. Wie hieß das Betriebssystem, das die Entwicklung von Unix inspirierte?
   - TIMICS
   - MULTICS
   - COMPLICS
   - ULTRICS
2. Welche Programmiersprache wurde 1971 zur Neuimplementierung von Unix entwickelt?
   - Pascal
   - BASIC
   - C
   - Smalltalk
3. An welcher Universität wurde Unix entscheidend weiterentwickelt?
   - Massachusetts Institute of Technology
   - University of California, Berkeley
   - University of Illinois
   - Harvard
4. Was ist POSIX?
   - eine Unix-Variante

- eine Programmiersprache
- der kleinste gemeinsame Nenner aller Unix-Systeme
- eine jährliche Unix-Entwicklungskonferenz in San Francisco

5. Welches Betriebssystem gilt als das erste echte PC-Betriebssystem?
   - Windows
   - Linux
   - CP/M
   - MS-DOS
6. Was ist ein GUI?
   - Graphical User Interface (grafische Benutzeroberfläche)
   - Generic Unix Interface (Unix-Standardsystemschnittstelle)
   - Globally Unique Identity (weltweit einmalige Registriernummer)
   - Gate Undefined Issue (Sicherheitslücke in Betriebssystemen)
7. Wofür steht die Abkürzung GNU?
   - Georgia Newton University
   - GNU's Not Unix
   - Graphical Network Usage
   - Global Networking Union
8. Welche Windows-Version war als erste ein echtes Betriebssystem?
   - Windows 3.0
   - Windows NT 3.0

- Windows 3.11
- Windows 95

9. Welche der folgenden Windows-Versionen basierte nicht auf MS-DOS, sondern auf der Windows-NT-Architektur?
   - Windows Me
   - Windows 98 SE
   - Windows 2000
   - Windows 98
10. Welche der folgenden Aussagen über den Kernel trifft zu?
    - Monolithische Kernels sind schneller als Mikrokernels.
    - Alle modernen Kernels sind Mikrokernels.
    - Der Kernel nimmt die Prozess- und Speicherverwaltung vor.
    - Der Kernel stellt die grafische Benutzeroberfläche bereit.
11. Welche modernere Alternative ergänzt die Prozesse?
    - Swap-Partitionen
    - Threads
    - Dienste
    - Gerätetreiber
12. Was bedeutet »Bootstrapping« beim Betriebssystem?
    - Kontrolle des Bootvorgangs eines PCs über das Netzwerk
    - Installation eines Gerätetreibers
    - im Hintergrund laufende Programme während des Bootvorgangs

- anderes Wort für das Booten des Betriebssystems

13. Welche Aussage über die Betriebsmodi eines Betriebssystems trifft zu?
    - Es gibt vier verschiedene Betriebsmodi.
    - Prozesse im Kernelmodus können jederzeit unterbrochen werden.
    - Prozesse im Benutzermodus genießen einen stärkeren Zugriffsschutz.
    - Prozesse im Kernelmodus haben Priorität.
14. Welche der folgenden Aussagen über Multitasking ist falsch?
    - Man unterscheidet präemptives und kooperatives Multitasking.
    - Mac OS 9 unterstützte gar kein Multitasking.
    - Windows 3.11 unterstützte nur kooperatives Multitasking.
    - Alle aktuellen Betriebssysteme sind multitaskingfähig.
15. Welches Signal beendet einen Unix-Prozess normal?
    - SIGHUP
    - SIGINT
    - SIGKILL
    - SIGTERM
16. Mit welchem Unix-Systemaufruf wird ein Signal an einen Prozess gesendet?
    - `signal()`
    - `kill()`

- `send()`
- `term()`

17. Welche Aussage über Unix-Prozesse ist zutreffend?
    - Der Prozess `init` hat die PID 0.
    - Neue Prozesse werden durch den Systemaufruf `CreateProcess()` erzeugt.
    - Der Child-Prozess ist eine identische Kopie des Parent-Prozesses.
    - Mit einem `fork()`-Aufruf lassen sich beliebig viele Child-Prozesse auf einmal ableiten.
18. Welche der folgenden Aussagen über die UID und die GID eines Unix-Prozesses ist falsch?
    - Um einem Prozess ein Signal zu senden, muss die UID des Absenders der PID des Prozesses entsprechen.
    - Um einem Prozess ein Signal zu senden, muss die UID des Absenders der UID des Prozesses entsprechen.
    - Um einem Prozess ein Signal zu senden, muss die GID des Absenders der GID des Prozesses entsprechen.
    - Der User-Account *root* (UID 0, GID 0) darf jedem Prozess Signale senden.
19. Was ist kein Bestandteil einer Speicheradresse der Memory Management Unit (MMU) eines Intel-Prozessors?
    - Offset
    - Page Directory
    - Page Table

- Page File

20. Was ist kein typisches Verzeichnis im Unix-Verzeichnisbaum?
    - */bin*
    - */usr*
    - */usw*
    - */etc*
21. Wie lautet der relative Pfad, um von */home/user1/test/* aus die Datei */home/user2/hallo.txt* anzusprechen?
    - *../user2/hallo.txt*
    - *../../hallo.txt*
    - *user2/hallo.txt*
    - *../../user2/hallo.txt*
22. Für welches Verzeichnis im Unix-Verzeichnisbaum steht die Abkürzung ~ ?
    - das Systemverzeichnis
    - das Home-Verzeichnis des aktuellen User-Accounts
    - das Wurzelverzeichnis
    - das Verzeichnis */usr/sbin*
23. Welche Aussage über Symlinks im Unix-Dateisystem ist zutreffend?
    - Wird der letzte Symlink gelöscht, der auf eine Inode zeigt, dann wird die entsprechende Datei gelöscht.
    - Ein Symlink verweist auf eine Inode.
    - Ein Symlink verweist auf einen Verzeichniseintrag.

- Symlinks können nur auf Dateien auf demselben physikalischen Datenträger verweisen.

24. Was bedeutet das Unix-Dateizugriffsrecht `-rwxr-xr--`?
    - Der Owner-Account darf lesen, schreiben und ausführen; die Gruppe darf lesen und ausführen; alle anderen dürfen nur lesen.
    - Der Owner-Account darf lesen, schreiben und ausführen; andere lokale Accounts dürfen lesen und ausführen; Netzwerk-Accounts dürfen nur lesen.
    - Der Owner-Account darf lesen, schreiben und ausführen; andere Accounts dürfen lesen und ausführen; Gäste dürfen nur lesen.
    - Der Owner-Account darf lokal lesen, schreiben und ausführen; meldet er sich über das Netzwerk an, darf er nur lesen und ausführen; alle anderen dürfen nur lesen.
25. Wie lautet das Dateizugriffsrecht `-rw-rw-r--` numerisch?
    - 0775
    - 0664
    - 0441
    - 0442
26. Welcher der folgenden Windows-Dateinamen ist ungültig?
    - *[test].txt*
    - *(test).txt*
    - *<test>.txt*
    - *_test_.txt*

27. Welches Windows war das erste 32-Bit-System?
    - Windows 3.11
    - Windows 95
    - Windows NT 3.x
    - Windows 2000
28. Welches der folgenden Dateisysteme wurde nie nativ von Windows unterstützt?
    - FAT32
    - NTFS
    - FAT16
    - ext4
29. Wie hieß die Benutzeroberfläche von Windows 8 und 10?
    - Metro
    - Luna
    - Aqua
    - Aero Glass
30. Welche der folgenden Editionen von Windows 11 gibt es nicht?
    - Windows 11 Pro
    - Windows 11 Enterprise
    - Windows 11 Small Business
    - Windows 11 SE
31. Welche der folgenden Aussagen über das Dateisystem NTFS ist falsch?

- Es wird von allen Systemen der Windows-NT-Familie unterstützt.
- Es ist voll abwärtskompatibel mit FAT32.
- FAT32 lässt sich nachträglich in NTFS konvertieren.
- NTFS-Partitionen können komprimiert werden.

32. Welcher Windows-Konsolenbefehl gibt die Dateien im aktuellen Verzeichnis und allen Unterverzeichnissen seitenweise aus?
    - `dir /p/s`
    - `dir /a`
    - `ls -R |less`
    - `subdir |more`
33. Wie wechseln Sie aus dem Windows-Verzeichnis *C:\Users\User\Music* per relativer Pfadangabe in das Verzeichnis *C:\Users\User\Documents\Letters*?
    - `cd ..\Letters`
    - `cd Documents\Letters`
    - `cd ..\Documents\Letters`
    - `cd \Documents\Letters`
34. Welche Einschränkung hat der Windows-Konsolenbefehl `rmdir` gegenüber `del /s`?
    - `rmdir` löscht nur leere Verzeichnisse.
    - `rmdir` löscht nur Dateien, keine Verzeichnisse.
    - `rmdir` löscht nur Dateien und Verzeichnisse, die dem aktuellen User-Account gehören.

- Keine; die beiden Kommandos sind synonym.

35. Wie benennen Sie die Datei *alter-name.txt* auf der Windows-Konsole in *neuer-name.txt* um?
    - `mv alter-name.txt neuer-name.txt`
    - `del alter-name.txt |create neuer-name.txt`
    - `rename alter-name.txt neuer-name.txt`
    - Umbenennen geht leider nur in der grafischen Oberfläche, nicht auf der Konsole.
36. Wie zeigen Sie in der Windows PowerShell eine Liste der Dateien im aktuellen Verzeichnis an?
    - `Get-Files`
    - `Get-Filenames`
    - `Get-ChildItem`
    - `Get-Children`
37. Welcher Ausdruck überprüft in der PowerShell, ob die Variable `$a` den Wert `"test"` enthält?
    - `$a = "test"`
    - `$a == "test"`
    - `$a -eq "test"`
    - `$a.equals("test")`
38. Welche der folgenden Windows-Server-Versionen gab es nie?
    - Windows 2000 Server
    - Windows NT Server 4.0
    - Windows Server 2012 R2

- Windows 95 Server

39. Welche Linux-Distribution installiert Pakete in der Regel durch Kompilieren des Quellcodes?
    - openSUSE
    - Gentoo
    - Debian
    - Red Hat
40. Wie heißt die Datei, in der bei einem Unix-System die User-Accounts gespeichert sind?
    - */etc/users*
    - */etc/accounts*
    - */etc/passwd*
    - Es handelt sich nicht um eine einfache Datei, sondern um eine Datenbank in verschiedenen Dateien.
41. Wie holen Sie ein im Hintergrund laufendes Unix-Programm zurück aufs Terminal?
    - durch Drücken von Strg + Z
    - durch Eingabe von `fg`
    - Der Vorgang ist leider irreversibel; das Programm muss mit `kill` beendet und dann neu gestartet werden.
    - durch Auswahl in der Prozessliste
42. Wie lassen sich in der *bash* alle Bilddateien mit dem Namensschema *bild1.jpg* ansprechen, wobei die 1 für eine beliebig lange Zahl stehen kann?
    - `bild[0-9].jpg`

- `bild\n.jpg`
- `Bild?.jpg`
- `bild*.jpg`

43. Mit welcher Unix-Shell-Anweisung werden alle Dateien im aktuellen Verzeichnis gelöscht, deren Name mit a beginnt?
    - `rm |grep ^a`
    - `rm [!a]`
    - `rm a*`
    - `rm -rf a`
44. Wie lassen sich in der Linux-Shell alle Dateien im aktuellen Verzeichnis (einschließlich der versteckten) mit ausführlichen Informationen ausgeben?
    - `ls -a`
    - `ls --show-hidden`
    - Dies geht nur in der grafischen Benutzeroberfläche, nicht auf der Konsole.
    - `ls -la`
45. Wie erteilen Sie allen User-Accounts unter Linux das Recht, die Datei *listdir.sh* auszuführen?
    - `chmod a+x listdir.sh`
    - `chmod o+x listdir.sh`
    - `chown 0 listdir.sh`
    - `cp listdir.sh /usr/bin`
46. Was ist keine Komponente von Syslog-Meldungen?

- Facility
- Message
- Error
- Priority

47. Wann wird der durch `...` symbolisierte Code in folgendem Ausschnitt eines *bash*-Shell-Skripts ausgeführt?

```
if [ -f ~/.lock ]
then
  ...
fi
```

- wenn im Home-Verzeichnis des Users der Verzeichniseintrag *.lock* existiert
- wenn im *root*-Verzeichnis die reguläre Datei *.lock* existiert
- wenn im Home-Verzeichnis des Users die reguläre Datei *.lock* existiert
- Gar nicht; der Code enthält einen Syntaxfehler.

48. Wie wird in der *bash* ein Alias namens `lh` definiert, das den Befehl `ls -lah` enthält?

- `alias lh 'ls -lah'`
- `alias lh='ls -lah'`
- `alias 'ls -lah':lh`
- `alias lh:'ls -lah'`

49. Zu welchen Zeiten wird ein Cronjob mit folgendem Crontab-Eintrag ausgeführt?

```
*/15 8-18 * * *
```

- werktags, einmal stündlich, von 8:15 bis 18:15 Uhr

- jeden Tag, einmal stündlich, von 8:15 bis 18:15 Uhr
- alle 15 Tage, stündlich, von 8:00 bis 18:15 Uhr
- jeden Tag, viertelstündlich, von 8:00 bis 18:45 Uhr