

# Grundlagen der Bash- Programmierung

## Begleitmaterial zu den Übungen im Kurs Wirtschaftsinformatik II

Version 4.0.0

Bachelorstudium Wirtschaftsinformatik  
Sommersemester 2019

Prof. Dr. Peter Mandl, LBA Björn Rottmüller, M. Sc.

## Vorwort

Das vorliegende Skriptum befasst sich mit der heute in Linux-Derivaten standardmäßig verwendeten Bash-Kommandosprache. Anhand dieses Skriptums sollen Grundlagen zu Bash sowie erste Programmiertechniken zur Entwicklung von Bash-Programmen vermittelt werden.

Der Schwerpunkt liegt im ersten Teil im Kennenlernen wichtiger Konzepte der Bash sowie im Erlernen der wichtigsten Kommandos, wobei für das Verständnis einiger grundlegender Aspekte auch auf Linux-Basiswissen eingegangen wird. Im zweiten Teil werden die einführenden Konzepte zur Entwicklung von kleineren Bash-Programmen vorgestellt.

Das Skriptum enthält einige Übungen, die während der Laborstunden zu bearbeiten sind.

Lernziele sind:

- Linux-Grundkonzepte, die für die Bash-Programmierung nützlich sind, kennenlernen. Hierzu gehören die Konzepte des Linux-Dateisystems und von Linux-Prozessen sowie das Linux-Berechtigungskonzept.
- Die wichtigsten Kommandos kennenlernen und anwenden können.
- Grundlagen der Bash-Programmierung kennenlernen und kleine Shellskripte schreiben können.

Das Skriptum dient im Kurs Wirtschaftsinformatik II (Bachelorstudium Wirtschaftsinformatik) als Basis für die begleitenden Übungen zur Bash-Programmierung.

Wir bedanken uns bei allen Studierenden und KollegInnen für Ihre Anregungen sowie bei allen, die an diesem Skriptum mitgewirkt haben, insbesondere bei Tobias Roithmeier und Felix Kraus.

München, im März 2019

Peter Mandl, Björn Rottmüller

# Inhaltsverzeichnis

|   |    |
|---|----|
| 1 Einführung .....  | 8  |
| 1.1 Überblick über Linux-Derivate .....                               | 8  |
| 1.2 Bourne, Korn, C-Shell, Bash, ... ..                               | 10 |
| 1.3 Terminals und Shells .....  | 11 |
| 1.4 Starten und Beenden der Bash .....                                | 12 |
| 2 Grundlegende Konzepte .....   | 13 |
| 2.1 Linux-Verzeichnisse und Linux-Dateien .....                       | 13 |
| 2.2 Linux-Prozesse .....  | 14 |
| 2.3 Kommandosyntax .....  | 15 |
| 2.4 Heimatverzeichnis und aktuelles Verzeichnis .....                 | 17 |
| 2.5 Prozessumgebung und Umgebungsvariablen .....                      | 18 |
| 2.6 Reguläre Ausdrücke .....  | 21 |
| 2.7 Handbuchseiten und Hilfefunktion .....                            | 22 |
| 2.8 Übungen .....   | 24 |
| 2.8.1 Hinweise für Studierende der Hochschule München .....           | 24 |
| 2.8.2 Übung zu den Handbuchseiten .....                               | 24 |
| 2.8.3 Übung zum Kommando help .....                                   | 25 |
| 2.8.4 Übung zum Anlegen von Umgebungsvariablen .....                  | 25 |
| 2.8.5 Übung zum Kommando typeset oder declare .....                   | 25 |
| 2.8.6 Übung zum Erzeugen und Exportieren von Umgebungsvariablen ..... | 26 |
| 2.8.7 Übung zu alias .....  | 26 |
| 3 Bash-Kommandos für die Dateibearbeitung .....                       | 27 |
| 3.1 Dateien und Verzeichnisse .....                                   | 27 |
| 3.2 Kommandoübersicht .....   | 30 |
| 3.2.1 Kommando echo .....   | 30 |
| 3.2.2 Kommando pwd .....  | 31 |
| 3.2.3 Kommando cd .....   | 31 |
| 3.2.4 Kommando ls .....   | 31 |
| 3.2.5 Kommando touch .....  | 33 |
| 3.2.6 Kommando mkdir .....  | 33 |
| 3.2.7 Kommando rmdir .....  | 33 |

## 1 Einführung

---

|  |    |
|--|----|
| 3.2.8 Kommando cp .....                                | 34 |
| 3.2.9 Kommando cat .....                               | 34 |
| 3.2.10 Kommando less oder more.....                    | 34 |
| 3.2.11 Kommando mv.....                                | 35 |
| 3.2.12 Kommando rm .....                               | 35 |
| 3.2.13 Kommando ln.....                                | 36 |
| 3.2.14 Kommando wc .....                               | 36 |
| 3.3 Ein- und Ausgabekanäle von Prozessen .....         | 37 |
| 3.4 Pipes .....  | 38 |
| 3.5 Quoting und Expandierungen.....                    | 39 |
| 3.6 Kommando-Historie .....                            | 41 |
| 3.7 Übungen .....                                      | 43 |
| 3.7.1 Übung zur Kommando-Historie.....                 | 43 |
| 3.7.2 Übung zur Datei- und Verzeichnisbearbeitung..... | 43 |
| 4 Linux-Berechtigungen .....                           | 48 |
| 4.1 Berechtigungskonzept .....                         | 48 |
| 4.2 Spezielle Kommandos.....                           | 50 |
| 4.2.1 Kommando chmod .....                             | 51 |
| 4.2.2 Kommando chown .....                             | 51 |
| 4.2.3 Kommando chgrp .....                             | 52 |
| 4.2.4 Kommando umask .....                             | 52 |
| 4.3 Login- und Bash-Profiling.....                     | 54 |
| 4.4 Übungen .....                                      | 57 |
| 4.4.1 Übung zum Berechtigungskonzept.....              | 57 |
| 4.4.2 Übung mit umask .....                            | 58 |
| 4.4.3 Übung .bashrc verändern .....                    | 58 |
| 5 Linux-Internas.....                                  | 59 |
| 5.1 Linux-Bootvorgang .....                            | 59 |
| 5.2 Init-System.....                                   | 60 |
| 5.2.1 SysVinit .....                                   | 61 |
| 5.2.2 Systemd .....                                    | 62 |
| 5.2.3 Exkurs: Startup mit launchd unter OS X .....     | 62 |
| 5.3 Einige Kommandos für Prozesse .....                | 63 |
| 5.3.1 Kommando ps .....                                | 63 |
| 5.3.2 Kommando pstree.....                             | 64 |

---

|       |   |    |
|-------|---|----|
| 5.3.3 | Kommando top .....  | 65 |
| 5.3.4 | Kommando kill .....   | 66 |
| 5.4   | Hintergrundprozesse und Dämonen .....                         | 67 |
| 5.5   | Übungen .....   | 69 |
| 5.5.1 | Übung mit ps.....   | 69 |
| 5.5.2 | Übung mit top und pstree .....                                | 69 |
| 5.5.3 | Übung mit Hintergrundprozess .....                            | 70 |
| 5.5.4 | Übung zum Beenden eines Prozesses.....                        | 70 |
| 6     | Grundlagen der Bash-Programmierung .....                      | 71 |
| 6.1   | Aufbau und Aufruf eines Bash-Skripts .....                    | 71 |
| 6.2   | Übergabeparameter und Rückgabewerte .....                     | 72 |
| 6.3   | Bedingungen .....   | 73 |
| 6.4   | if-Anweisung .....  | 75 |
| 6.5   | for-Schleife.....   | 76 |
| 6.6   | while-Schleife.....   | 76 |
| 6.7   | Bash-Funktionen.....  | 77 |
| 6.8   | Globale und lokale Variablen .....                            | 79 |
| 6.9   | Übungen .....   | 80 |
| 6.9.1 | Übung: Erstes Bash-Skript.....                                | 80 |
| 6.9.2 | Übung zur Parameterübergabe .....                             | 80 |
| 6.9.3 | Übung zum Debugging von Bash-Skripts .....                    | 81 |
| 6.9.4 | Übung mit if-Anweisung.....                                   | 81 |
| 6.9.5 | Übung mit for-Schleife.....                                   | 82 |
| 6.9.6 | Übung mit while-Schleife .....                                | 83 |
| 6.9.7 | Übung: Bash-Skript zum Bewegen von Dateien .....              | 83 |
| 6.9.8 | Übung: Bash-Skript zum Bewegen von Dateien (Erweiterung)..... | 86 |
| 6.9.9 | Übung zum Übersetzen und Ausführen von Java-Programmen .....  | 87 |
| 7     | Weitere interessante Kommados .....                           | 91 |
| 7.1   | Kommandos zur Mustersuche .....                               | 91 |
| 7.1.1 | Kommando grep .....   | 91 |
| 7.1.2 | Kommando egrep .....  | 92 |
| 7.2   | Kommandos zur Datei- und Datenbearbeitung .....               | 93 |
| 7.2.1 | Kommando tr .....   | 93 |
| 7.2.2 | Kommando head .....   | 93 |
| 7.2.3 | Kommando tail .....   | 93 |

|   |     |
|---|-----|
| 1 Einführung  |     |
| 7.2.4 Kommando printf .....                             | 94  |
| 7.2.5 Kommando find .....                               | 94  |
| 7.3 Sonstige, nützliche Kommandos .....                 | 95  |
| 7.3.1 Kommando whois.....                               | 95  |
| 7.3.2 Kommando which .....                              | 95  |
| 7.3.3 Kommando who .....                                | 95  |
| 7.4 Übersicht über weitere interessante Kommandos ..... | 96  |
| 8 Zusammenfassung .....                                 | 97  |
| Anhang: Gebräuchliche Linux-Editoren .....              | 98  |
| Editor vi .....   | 98  |
| Editor nano .....                                       | 100 |
| Editor emacs .....                                      | 101 |
| Literaturhinweise und Web-Links .....                   | 103 |
| Sachwortverzeichnis .....                               | 104 |



## 1 Einführung

Dieses Kapitel gibt einen ersten Einblick in das Betriebssystem Linux und in die Shell-Programmierung, soweit dies für das Verständnis der Bash erforderlich ist. Die wichtigsten Linux-Derivate und einige heute verfügbare Shell-Dialekte werden kurz vorgestellt.

### **Zielsetzung**

Der Studierende soll einen Überblick über die verschiedenen Linux-Derivate und Shell-Dialekte erhalten. Er soll am Ende des Kapitels in der Lage sein, sich an einem Linux System an- und abzumelden, sowie eine entsprechende Shell zu starten und beenden.

### **Wichtige Begriffe**

Linux, Linux-Derivat, Linux-Distribution, Login, Logout, Terminal, Shell, Bash.

## 1.1 Überblick über Linux-Derivate

Der 25. August 1991 gilt als die von unterschiedlichen Quellen verifizierte Geburtsstunde von Linux. An diesem Tag kündigte Linus Benedict Torvalds, ein finnischer Student der Universität Helsinki, den ersten Kernel seines Betriebssystems via Usenet-Posting an. Aus einem selbstgeschriebenen Terminalemulator entwickelte sich ein vollständiges Betriebssystem, das schließlich als „Linux“ bezeichnet wurde. Daraus zweigen bis heute unterschiedliche Distributionen ab, welche sich in einigen Aspekten unterscheiden. Gemeinsam ist ihnen aber der Kernel, also das Herzstück des Betriebssystems. Obwohl Andrew Tanenbaum, der Entwickler des Betriebssystems „MINIX“, damals behauptete, Linux sei obsolet, ist Linux heute das meistgenutzte Betriebssystem im professionellen und wissenschaftlichen Umfeld. Etliche Supercomputer, darunter ein Großteil der NASA-Rechner, werden unter Linux betrieben. Anfang 1992 kündigte Linus Torvalds eine Lizenzänderung an. Ende 1992 wurde die erste Version unter einer neuen Lizenz, der GNU General Public License veröffentlicht. Diese Lizenz erlaubt das Verwenden, Untersuchen, Verbreiten und Verändern durch Endbenutzer.



Es gibt heute viele Distributionen, von denen einige auch im Enterprise-Bereich mit entsprechenden Services verkauft werden. Eine Übersicht hierzu gibt Abbildung 1-1.

Obwohl Linux üblicherweise mit graphischer Benutzeroberfläche installiert werden kann, ist es möglich darauf zu verzichten. Insbesondere im Serverbetrieb, in dem Linux- und Netzwerkspezialisten auf Sicherheit achten müssen, ist meist die Nutzung der Kommandozeile schneller und effizienter. Die meisten Linux-Distributionen erlauben durch Live-Images das direkte Booten von CDs bzw. DVDs, USB-Sticks und anderen Speichermedien für erste Versuche und auch für Diagnose-Einsätze.

Die Distribution Fedora verdankt ihren Namen seinem Vorgänger, dem Red Hat Linux. Dessen Logo zeigt einen roten Filzhut, auch Fedora genannt. Fedora ist der direkte Nachfolger Red Hats und dient als Testumgebung. Dieser Status sorgt für seine Beliebtheit innerhalb der Community. Red Hat entwickelte sich zu Red Hat Enterprise Linux, einer Distribution mit Subscription, weiter. Diese Distribution ist für Unternehmen (Enterprises) abgestimmt, was alleine schon am signifikant längeren Lebenszyklus einer Version (ca. zehn Jahre) erkennbar wird. Der Lebenszyklus beträgt bei Fedora im Vergleich dazu etwa ein Jahr.

Eine ebenfalls sehr häufig verwendete Distribution ist CentOS (Community Enterprise Operating System). CentOS ist binärkompatibel zu Red Hat Enterprise Linux und stellt eine kostengünstige Alternative dar, da ebenso auf die Bedürfnisse großer Unternehmen ausgerichtet ist. Binärkompatibilität bedeutet, dass Programme auch ohne erneutes Kompilieren auf einem anderen System ausführbar sind.

Das Linux-Derivat OpenSUSE, welches frei verfügbar ist, hat seinen Fokus eher auf Benutzerfreundlichkeit gelegt. Das aus SUSE Linux entstandene Betriebssystem, welches früher ebenfalls in einer Enterprise Version erhältlich war, hat ein eigenes Installations- und Konfigurationswerkzeug namens „YaST“, Abkürzung für („Yet another Setup Tool“).

Debian Linux ist nach dem von Debian abstammenden Ubuntu das meist verbreiteste, freie Desktop-Betriebssystem. Es wird neben Red Hat Enterprise Linux und Windows auch auf der internationalen Raumstation ISS eingesetzt.

Das bekannteste Desktop-Linux-Derivat ist wohl Ubuntu. Es wird kostenlos bereitgestellt. Dies soll auch mit dem Namen ausgedrückt werden, denn „Ubuntu“ kommt aus den Sprachen der afrikanischen Völker Zulu und Xhosa und steht für „Menschlichkeit“ und „Gemeinsinn“. Der Schwerpunkt liegt auf der Be-

## 1 Einführung

nutzerfreundlichkeit, die Einstiegshürde für Linux-Neulinge ist vergleichsweise gering.

Android ist wohl das unbewusst am meisten genutzte Betriebssystem, das auf einem Linux-Kernel aufsetzt. Auch von Android werden Derivate gelistet, z. B. das Fire OS, welches von Amazon für eigene Produkte (z. B. Kindle, Fire Phone) verwendet wird. Schließlich ist noch das von Google entwickelte Chrome OS zu nennen.

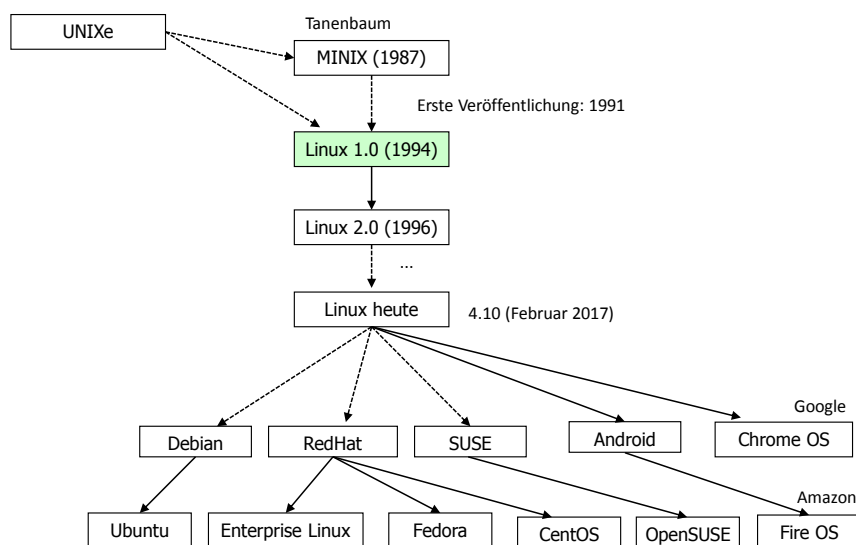


Abbildung 1-1: Entwicklung von Linux und den wichtigsten Distributionen

## 1.2 Bourne, Korn, C-Shell, Bash, ...

Eine Shell ist in Linux ein Kommandointerpreter, der in einem Terminal ausgeführt wird. Die Shell ist auf den ersten Blick vergleichbar mit der Kommandozeile (Programm cmd.exe) aus der Microsoft-Welt. Neben dem Ausführen einfacher, einzelner Befehle ermöglicht eine Shell unter Linux die Entwicklung und Ausführung äußerst leistungsfähiger Programme (sog. Shell-Skripte), die in der Praxis zum Beispiel zur Administration von Serverrechnern eingesetzt werden. So werden beispielsweise auf Serverrechnern automatische Backups von Dateien mit Shell-Skripten realisiert.

Unter einer Shell versteht man die traditionelle Benutzerschnittstelle unter Unix- und Unix-ähnlichen Betriebssystemen. Der Begriff Shell ist allerdings als Oberbegriff für verschiedene Kommandozeileninterpreten zu verstehen, sodass viele unterschiedliche „Shells“ existieren. Diese unterscheiden sich meist in den verfügbaren Befehlen bzw. in der Syntax der Befehle. Die gängigsten Shells sollen kurz erwähnt werden:

- Bourne-Shell (sh): Die Bourne-Shell ist Vorfahre vieler heutiger Shells. Noch heute finden sich in vielen Unix-Distributionen Nachfolger der Bourne-Shell.
- Weitere Shells sind die Korn-Shell (ksh), die C-Shell (csh), die Thompson-Shell (osh) und die TENEX-C-Shell (tcsh). Die Korn-Shell wurde für Unix System V entwickelt, die C-Shell entstand für Berkeley-Unix, die Thompson-Shell war in den 70er-Jahren die Unix-Standard-Shell und wurde später durch die Bourne-Shell ersetzt. Die TENEX-C-Shell ist eine Weiterentwicklung der C-Shell und wird heute noch häufig unter Linux und auch unter macOS verwendet.
- Bash (Bourne-again-shell): Die Bash ist eine Weiterentwicklung der Bourne-Shell und heute die Standard-Shell für viele Linux-Distributionen. Sie enthält viele Merkmale der anderen Shells. Bash ist auch unter macOS und vielen anderen Unix-Derivaten verfügbar.

Die meisten Shells bieten ähnliche Möglichkeiten, teilweise mit identischer, aber auch mit unterschiedlicher Syntax an. Beispielhafte Funktionen sind das Starten von Programmen in einem eigenen Prozess, sowie die Umlenkung von Ein- und Ausgaben, welche die Programme benötigen bzw. produzieren (z. B. Umlenken der Ausgabe eines Programms in eine Datei oder in die Eingabe für ein weiteres Programm). Auch Kontrollstrukturen wie Schleifen (for, while, ...) und Bedingungen (if – then – else) werden unterstützt, so dass man ganze Shell-Programme als Folge von Shellkommandos, die in Kontrollstrukturen eingebettet werden, entwickeln kann. Dies nennt man auch Shell-Programmierung. Diese Shell-Programme werden in Editoren geschrieben, können in Dateien, als sog. Shell-Skripte abgelegt und in der Kommandozeile durch Angabe des Dateinamens zum Ablauf gebracht werden.

### 1.3 Terminals und Shells

Ein Benutzer meldet sich in Linux wie in Unix zunächst über ein sog. Terminal an. Im Gegensatz zu den ersten Unix-Varianten, in denen Terminals physikalische Einheiten mit Bildschirm und Tastatur für den Zugang zu den Rechnern waren, handelt es sich heute nicht mehr um klassische Terminals, sondern vielmehr um

eine grafische Benutzerumgebung. Informationen zu den im Betriebssystem konfigurierten Terminals fand man früher in Konfigurationsdateien des Systems wie z. B. `/etc/ttys`.

Der Anmeldevorgang wird also üblicherweise durch eine grafische Benutzeroberfläche unterstützt, aber auch ein Zugang über eine einfache Textkonsole ist möglich. Vor allem bei Serversystemen ist die Textkonsole die übliche Einstiegsvariante.

Typische grafische Arbeitsumgebungen bzw. Desktops unter Linux sind KDE<sup>1</sup>, GNOME<sup>2</sup>, und XFCE<sup>3</sup>. Terminals können also heute aus der grafischen Linux-Oberfläche gestartet werden, was sich je nach Linux-Distribution immer ein wenig anders darstellt. Man muss also in der jeweiligen GUI etwas suchen.

### 1.4 Starten und Beenden der Bash

Zum Starten einer Bash gibt man einfach den Namen der Shell als Kommando in der Kommandozeile ein. Will man die Bash verlassen, verwendet man das Kommando `exit`:

```
$ bash
... beliebige Kommandos eingeben
$ exit
```

In Linux wird beim Starten der Bash ein neuer Prozess erzeugt, in dem der Kommandointerpreter gestartet wird. Dieser Prozess erhält einen eigenen Adressraum und erbt die Eigenschaften des Elternprozesses (Mandl 2014).

Hinweis: Das Zeichen „\$“ wird auch als Kommando-Prompt (engl. “auffordern”) bezeichnet und kann – je nach Einstellung – variieren. Es ist sogar über die Shell beliebig um zusätzliche Informationen (z. B. Rechnername, Benutzername, Datum, ...) erweiterbar.

---

<sup>1</sup> <http://www.kde.de>, letzter Zugriff am 21.02.2019.

<sup>2</sup> <https://www.gnome.org/>, letzter Zugriff am 21.02.2019.

<sup>3</sup> <http://www.xfce.org/>, letzter Zugriff am 21.02.2019.

## 2 Grundlegende Konzepte

### Zielsetzung des Kapitels

Der Studierende soll am Ende des Kapitels die grundlegenden Konzepte von Linux (Datei, Dateisystem, Verzeichnis, Prozess, Umgebungsvariablen, reguläre Ausdrücke) erläutern können. Ebenso soll der Studierende die Kommandosyntax und grundlegende reguläre Ausdrücke kennen und anwenden können.

### Wichtige Begriffe

Prozess, Datei, Verzeichnis, Verzeichnisbaum, Umgebungsvariable, regulärer Ausdruck.

### 2.1 Linux-Verzeichnisse und Linux-Dateien

Das Linux-Dateisystem ist logisch in einer hierarchischen Baumstruktur organisiert, die heute einer Standardisierung unterliegt. Der Standard wird als Filesystem Hierarchy Standard (FHS) bezeichnet. FHS ist eine Richtlinie für die logische Verzeichnisstruktur Unix-ähnlicher Betriebssysteme und wird von der Linux Foundation (Linux Foundation), einem gemeinnützigen Konsortium, gepflegt.

Das Dateisystem beginnt mit einer „Wurzel“ (auch Rootverzeichnis genannt), die mit dem Symbol „/“ beginnt und ist in Subverzeichnisse strukturiert. Die folgende Tabelle beschreibt die wichtigsten Subverzeichnisse unter dem Rootverzeichnis, die in einem FHS-konformen Dateisystem vorhanden sein sollten. Nicht jede Distribution hält sich genau an die Vorgaben. Unter jedem Subverzeichnis kann natürlich wieder ein komplexer Unterbaum liegen.

| Bezeichnung | Inhalt   |
|-------------|--|
| /bin        | Binärdateien, grundlegende Befehle   |
| /boot       | Bootloader zum Starten des Systems   |
| /dev        | Gerätedateien, externe Geräte (z. B. Festplatte) werden in sog. Gerätedateien beschrieben, die aus Sicht des Anwenders wie eine einfache Datei aussehen. |

## 2 Grundlegende Konzepte

---

|        |  |
|--------|--|
| /etc   | Systemkonfiguration für den Rechner  |
| /home  | Verzeichnis, in dem für die Benutzer Subverzeichnisse angelegt werden            |
| /lib   | Bibliotheken und Kernel-Module   |
| /media | Einhängeposition für Unterdateisysteme wechselnder Datenträger wie z. B. CD-ROMs |
| /mnt   | Temporär eingebundene (gemountete) Dateisysteme                                  |
| /opt   | Anwendungsprogramme  |
| /proc  | Informationen zu laufenden Prozessen   |
| /root  | Verzeichnis der Rootberechtigung (Admin-Kennung)                                 |
| /sbin  | Binärdateien für Systemprogramme   |
| /tmp   | Ablage für temporäre Dateien   |
| /srv   | Datenablage für Systemdienste  |
| /usr   | Programmdateien  |
| /var   | Veränderliche Dateien wie Logging-Dateien  |
| ...    |  |

### 2.2 Linux-Prozesse

Jedes Programm wird in Linux in einer Ablaufumgebung gestartet, die als Prozess bezeichnet wird. Ein Prozess erhält einen eigenen Speicher (virtueller Adressraum genannt) zugeordnet und konkurriert mit anderen Prozessen um die vorhandenen CPUs.

Welche Prozesse im System gerade aktiv sind, kann man sich über Kommandos genauer anschauen. Beispielsweise dienen die Kommandos `top` und `ps` dazu, sich eine Übersicht über alle laufenden Prozesse auf dem Bildschirm auszugeben.

Ruft man ein Programm auf, wird ein Prozess erzeugt. Unter Linux kann man auch angeben, dass nicht unbedingt ein eigener Prozess mit eigenem Adressraum erzeugt wird (das ist nämlich aufwändig). In diesem Fall wird ein Thread erzeugt. Für die weitere Betrachtung ist das aber nicht von Belang (siehe hierzu (Mandl 2014)).

Beim Systemstart werden auch schon spezielle Systemprozesse erzeugt. Prozesse bekommen die Namen der in ihnen laufenden Programme. Einige wichtige Systemprozesse sind `initd` bzw. `systemd`, `inetd`, `cron` und `syslogd`.

Während ihrer Laufzeit wird ihnen ein eindeutiger Identifier, die sog. Prozess-Identifikation oder PID zugeordnet. Ein Prozess, der einen anderen erzeugt, wird als Elternprozess bezeichnet. Der neu erzeugte Prozess heißt Kindprozess.

Auch die Prozesse sind damit in einer Baumstruktur angeordnet. Der erste Prozess, der im System gestartet wird (je nach Startprozedere ist das z. B. der `initd`-Prozess oder der `systemd`-Prozess oder unter macOS der `launchd`-Prozess) erzeugt weitere Prozesse, diese wiederum neue und so entsteht der Prozessbaum. Wenn man innerhalb einer Bash-Session ein Kommando aufruft, wird dies in einem eigenen Prozess ausgeführt. Es wird also von der Bash ein Kindprozess für die Ausführung des Kommandos erzeugt.

### 2.3 Kommandosyntax

Die Bash ist ein Kommandointerpreter, der Eingabezeichen untersucht und ein Kommando erkennen muss. Erst wenn ein Kommando syntaktisch fehlerfrei identifiziert wurde, kann es ausgeführt werden. Dazu braucht es einer speziellen Sprachdefinition. Um Kommandos zu beschreiben ist eine Metasyntax erforderlich. Hierzu soll ein erster Einstieg gegeben werden. Ein Bash-Kommando ist allgemein wie folgt aufgebaut:

|   |
|---|
| Kommandoname [ <code>&lt;optionen&gt;</code> ] [ <code>&lt;argumente&gt;</code> ] |
|---|

`Kommandoname` gibt die eindeutige Bezeichnung eines Kommandos an. Der `Kommandoname` kann – muss nicht – um eine Liste von `Optionen` ergänzt werden. `Optionen` beginnen normalerweise, aber nicht in jedem Fall, mit den Zeichen „-“ und sind durch einen Buchstaben gekennzeichnet. Anschließend können `Parameter` (`Argumente`) angegeben werden. Die eckigen Klammern deuten an, dass die Eingaben optional, also nicht zwingend sind. Jedes Kommando hat natürlich seine eigenen `Optionen` und `Argumente`.

Bevor wir auf Bash-Kommandos im Einzelnen eingehen, sollen zunächst allgemeine Aspekte zur Syntax erläutert werden. Man spricht hier auch von einer Metasyntax, die zur Beschreibung der eigentlichen Kommandosyntax dient.

Man unterscheidet grundsätzlich interne und externe Kommandos:

- Interne Kommandos startet die Bash selbst, sie sind also Bestandteil der Bash.

## 2 Grundlegende Konzepte

---

- Externe Kommandos führt die Bash nicht selbst aus, sondern startet dafür eine ausführbare Datei (z. B. ein eigenes Bash-Skript oder ein Programm. Die Kommandos `ls` und `pwd` sind beispielsweise externe Kommandos, während es sich bei `cd`, `echo`, `exit` usw. um interne Kommandos handelt.

Externe Kommandos müssen über den eingestellten Suchpfad (Umgebungsvariable `PATH`, siehe weiter unten) im Zugriff sein. Es können beliebige Programme, auch selbstentwickelte Bash-Skripts sein.

Für die Nutzung in der Bash spielt es aber keine wesentliche Rolle, ob es sich um ein internes oder externes Kommando handelt.

Eine Zeichenfolge, die von der Bash als zusammengehörig betrachtet wird, bezeichnen wir als Token. Ein Bezeichner oder Name ist dann ein Token, dass nur aus alphanumerischen Zeichen und Unterstrichen besteht und mit einem Buchstaben oder Unterstrich beginnt. Grundsätzlich wird zwischen Groß- und Kleinschreibung unterschieden.

Die folgende Tabelle gibt die in der Bash verwendeten Sonderzeichen und wichtige reservierte Wörter an:

| Sonderzeichen  | Bedeutung  |
|--|--|
| & ; ( ) < > Leerzeichen Tabulator  | Zeichen dienen als Trenner für Tokens und haben unterschiedliche Funktionen. „ “ ist z. B. das Pipe-Symbol (siehe unten). Leerzeichen, Tabulator, Zeilenumbruch sind Trennzeichen, die in einer speziellen Umgebungsvariablen mit dem Namen <code>IFS</code> (Internal Field Separator) definiert sind. Umgebungsvariablen werden weiter unten eingeführt. |
| && ( ) ! Newline   | Operatoren, „&&“ ist z. B. das logische Und-Symbol, „!“ die logische Negation.   |
| case do done elif else esac fi for function if in select then until while time | Reservierte Wörter mit Sonderbedeutungen für Anweisungen in der Bash. Die Schlüsselwörter <code>case</code> , <code>if</code> , <code>do</code> , ... werden in Sprachkonstrukten genutzt.   |
| ` ' "  | Die Apostrophe haben in der Bash eine besondere Bedeutung und werden unter dem Begriff „Quoting“ noch erläutert.   |



Weiterhin verwenden wir folgende Notation:

| Notation                  | Bedeutung  |
|---------------------------|--|
| <code>a b</code>          | Auswahl: entweder a oder b.                        |
| <code>{a,b,c}</code>      | Auswahl aus einer Liste von Tokens.                |
| <code>[a,b]</code>        | Optionale Angabe, entweder a oder b                |
| <code>&lt;name&gt;</code> | Platzhalter mit vorgegebenen Namen für einen Wert. |

**Hinweis.** Ganz eindeutig ist unsere Metasyntax leider nicht. Die eckigen Klammern `[` und `]` werden in manchen Kommandos auch als Bestandteil der Anweisungsfolgen verwendet und sind damit nicht der Metasyntax zur Beschreibung von Befehlen, sondern der tatsächlichen Befehlssyntax zuzuordnen. Dies ist beispielsweise bei geklammerten Bedingungen in `if`- oder `while`-Statements der Fall. Auch das sogenannte Pipesymbol `|` wird in der Bash anderweitig genutzt. Die Nutzung ergibt sich jeweils aus dem Kontext.

**Noch ein Hinweis.** Man kann übrigens auch jedes Kommando beliebig umbenennen oder neue Kommandos zusammenstellen, indem man das Kommando `alias` nutzt. Im folgenden Beispiel wird ein neues Kommando mit der Bezeichnung `list` eingeführt, das bei seinem Aufruf das Kommando `ls -al` ausführt:

```
alias list="ls -al"
```

## 2.4 Heimatverzeichnis und aktuelles Verzeichnis

Jedem Benutzer wird ein spezielles Verzeichnis im Dateisystem von Linux zugeordnet. Der Verzeichnisname stimmt in der Regel mit dem Benutzernamen überein. Dieses Verzeichnis wird als *Heimatverzeichnis* oder *Home-Verzeichnis* bezeichnet. In diesem Verzeichnis wird man nach dem Login-Vorgang positioniert. Im Dateisystem wird es üblicherweise unter dem Subverzeichnis `/home` angelegt. In der Umgebungsvariable `$HOME` wird der Name des aktuellen Home-Verzeichnisses gespeichert.

Im Home-Verzeichnis kann man seine eigenen Dateien und Verzeichnisse verwalten und hat dazu alle Rechte. Möchte man Dateien in anderen Verzeichnissen (z. B. in Home-Verzeichnissen anderer Benutzer) bearbeiten, so benötigt man meist spezielle Rechte. Das Rechtekonzept von Linux werden wir weiter unten noch genau-

## 2 Grundlegende Konzepte

---

er betrachten. Ein spezieller Benutzer, der sog. Superuser, darf alle Dateien im gesamten Dateisystem bearbeiten.

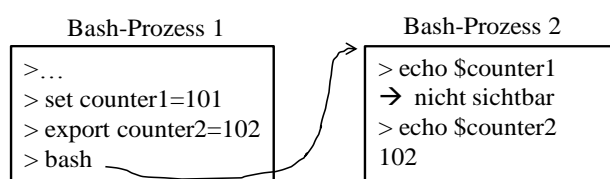
Mit dem Kommando `cd` (change Directory), das wir später noch kennenlernen werden, können wir z. B. von jeder Position im Dateisystem auf das Home-Verzeichnis zurückkommen, in dem wir einfach folgendes Kommando in der Bash eingeben:

```
cd
Alternativen:
cd $HOME
cd ~
```

### 2.5 Prozessumgebung und Umgebungsvariablen

Linux-Prozesse verwalten Variablen in einer prozessinternen Umgebung. Man nennt sie daher auch Umgebungsvariablen. Variablen bekommen Namen und können mit Werten belegt werden. In Linux-Prozessen können verschiedene Variablen hinterlegt werden, die auch in Bash-Skripten und anderen Programmen benutzt werden können. Man unterscheidet zwischen vordefinierten, benutzerdefinierten und exportierten Umgebungsvariablen.

Vordefinierte Umgebungsvariablen sind unter Linux schon systemweit vorhanden. Beispiele hierfür sind `HOME`, `LOGNAME` und `PATH`. `HOME` enthält den Pfad zum privaten Verzeichnis eines Benutzers (Home-Verzeichnis), `LOGNAME` enthält die Benutzerkennung des aktuell eingeloggten Benutzers und `PATH` enthält die Standardsuchpfade, also alle Dateipfade, in denen bei Aufruf eines Programms die ausführbare Datei gesucht wird.



**Abbildung 2-1: Vererben von Umgebungsvariablen**

Wenn ein vorhandener Prozess einen weiteren Prozess startet, vererbt er seine Umgebung mit allen exportierten Umgebungsvariablen an den neu erzeugten Prozess. Der neu erzeugte Prozess kann dann aber seinerseits die Variablen verändern,

ohne dass dies der erzeugende Prozess mitbekommt. Vordefinierte Umgebungsvariablen sind meist auch exportiert (siehe HOME und LOGNAME).

Benutzerdefinierte Variablen können mit beliebigen Namen angelegt werden. Exportierte Variablen sind auch in Prozessen sichtbar, die von einem Prozess erzeugt wurden.

In der Abbildung 2-1 ist skizziert, wie sich eine nicht exportierte, benutzerdefinierte Umgebungsvariable im Vergleich zu einer exportierten beim Aufruf einer neuen Bash verhält. Die Bash wird hier in einem neuen Prozess gestartet. Die exportierte Variable `counter2` wird auch in die Umgebung des neuen Prozesses vererbt, die nicht exportierte Variable `counter1` dagegen nicht.

Umgebungsvariablen werden üblicherweise mit einem vorangestellten „\$“-Zeichen angesprochen, bei der Zuweisung eines Werts auf eine Variable wird aber das „\$“-Zeichen nicht angegeben.

Für die Bearbeitung von Umgebungsvariablen gibt es verschiedene, sehr ähnlich funktionierende Befehle. Hierzu gehören `typeset`, `declare`, `set`, `env` und `printenv`.

Umgebungsvariablen können auch mit dem Befehl `typeset` oder mit dem Befehl `declare` angelegt werden. Das Anzeigen aller vordefinierten und benutzerdefinierten Umgebungsvariablen ist ebenfalls mit `typeset` oder `declare` möglich. In diesem Fall kann dann auch ein spezieller Datentyp (beispielsweise Integer) vereinbart werden.

| typeset      |  |
|--------------|--|
| Beschreibung | Dient zum Anlegen und Anzeigen von Umgebungsvariablen.   |
| Syntax       | <pre>typeset &lt;optionen&gt;</pre> <p>Optionen können u.a. sein:</p> <ul style="list-style-type: none"> <li>i = Anlegen einer Variable vom Typ Integer</li> <li>a = Anlegen einer Variable vom Typ String</li> <li>...</li> <li>x = Anlegen einer exportierten Variable</li> </ul> <p>Achtung: Je nach Betriebssystem und Version können die Optionen abweichen. Schauen Sie also mit dem Befehl <code>man</code> in die Handbuchseiten</p> |

## 2 Grundlegende Konzepte

|           |  |
|-----------|--|
| Beispiele | <pre># Alle Variablen auflisten typeset  # Erzeugen einer Integervariable mit Wertangabe typeset -i var1=123</pre> |
|-----------|--|

Ebenso können ganze Arrays von Variablen des gleichen Typs erzeugt werden. Dies wird in unserer Einführung aber nicht weiter betrachtet. Der Befehl `declare` verfügt über die gleichen Parameter.

Im Folgenden sind noch weitere Beispiele zur Erzeugung von Umgebungsvariablen mit dem Befehl `typeset` bzw. `declare` dargestellt.

| Beispiel                                | Beschreibung  |
|---|---|
| <code>typeset -i mycounter=11234</code> | Anlegen der Integervariable <code>mycounter</code> mit dem Wert 11234                 |
| <code>declare -i mycounter=99999</code> | Die Variable wird einfach überschrieben   |
| <code>typeset -a mystring=abc</code>    | Anlegen der Stringvariable <code>mystring</code> mit dem Wert „abc“                   |
| <code>typeset -ix exportvar=113</code>  | Anlegen und Exportieren einer Integervariable <code>exportvar</code> mit dem Wert 113 |
| <code>declare -x</code>                 | Anzeigen aller exportierten Umgebungsvariablen  |
| <code>typedef -ix</code>                | Anzeigen aller exportierten Umgebungsvariablen vom Typ Integer                        |

Eine Variable kann auch einfach durch eine Zuweisung oder durch Nutzung des Befehls `set` erzeugt werden. Der zugeordnete Typ ist dann „String“, das heißt, der Variable können beliebige, alphanumerische Zeichenketten (ohne Sonderzeichen) zugeordnet werden.

Eine einfache Verwendung von Umgebungsvariablen ist im Folgenden dargestellt. Die Variablen `MEINEVARIABLE1` und `MEINEVARIABLE2` werden durch einfache Zuweisung eines Wertes (hier „abc“) bzw. mit Hilfe des `set`-Befehls erzeugt. Mit dem Befehl `unset` wird eine Variable aus der Umgebung gelöscht.

```
> MEINEVARIABLE1=abc
> set MEINEVARIABLE2=abc
> PATH=/bin
> PATH=$PATH:$HOME
> Hallo="Hallo Linux"
# Entfernen einer Variable aus der Umgebung
> unset MEINEVARIABLE2
```

Der Befehl `set` ohne weitere Parameter zeigen ebenfalls alle vordefinierten und benutzerdefinierten Umgebungsvariablen an, die Befehle `env` und `printenv` jedoch nur die exportierten Umgebungsvariablen.

## 2.6 Reguläre Ausdrücke

Reguläre Ausdrücke (Regular Expressions) erleichtern das Arbeiten in der Bash. Bei der Verwendung von bestimmten Linux- oder Bash-Befehlen bietet es sich an, auf reguläre Ausdrücke zurückzugreifen. Damit muss nicht mehr jede Datei einzeln angesprochen werden, sondern es können z. B. Ausdrücke erstellt werden, die eine zusammengehörige Menge an Dateien filtern. Einen kurzen Auszug aus der Liste der Möglichkeiten bietet folgende Tabelle:

| Syntax                     | Bedeutung   |
|----------------------------|---|
| *                          | Beliebig viele Zeichen.                                     |
| ?                          | Genau für ein Zeichen.                                      |
| [a-z]                      | Genau ein Zeichen zwischen a und z.                         |
| [!Bb]                      | Weder ein ‚B‘ noch ein ‚b‘ erlaubt.                         |
| {info, hinweis, hilfe}.txt | Eine der drei Dateien info.txt, hinweis.txt oder hilfe.txt. |
| b* info*                   | Alle Dateien, die mit ‚b‘ oder ‚info‘ beginnen.             |

Weiterführende Informationen zu regulären Ausdrücken finden sich beispielsweise unter (Expressions). Im Folgenden finden Sie erste Anwendungen von regulären Ausdrücken. Die weitere Nutzung folgt in Übungen.

## 2 Grundlegende Konzepte

| Beispiel                       | Beschreibung   |
|--------------------------------|--|
| <code>cp *.txt test</code>     | Kopiert sämtliche Dateien mit der Endung <code>txt</code> in den Ordner (Verzeichnis) mit der Bezeichnung <code>test</code> .<br>Voraussetzung: Der Ordner <code>test</code> ist vorhanden und Dateien mit dieser Endung existieren.   |
| <code>mv ??april* april</code> | Verschiebt alle Dateien, in deren Dateinamen das Wort <code>april</code> vorkommt in ein Verzeichnis namens <code>april</code> .<br>Voraussetzung: Der Ordner <code>april</code> ist vorhanden und Beispieldateien mit Namen wie beispielsweise <code>01april2009.txt</code> oder <code>09april2007.txt</code> sind ebenfalls vorhanden. |
| <code>rm bin/*.class</code>    | Löscht alle Dateien, die die Zeichenfolge <code>*.class</code> enthalten, aus dem Verzeichnis <code>bin</code> .   |

### 2.7 Handbuchseiten und Hilfefunktion

Um Informationen zu Bash-Kommandos zu erhalten, kann man die Handbuchseiten (engl. manual) des Systems nutzen. Hierfür gibt es das Kommando `man`.

Die Handbuchseiten sind in sog. Sektionen geordnet, die durchnummeriert sind. Beispielsweise enthält die Sektion 1 Informationen zur Überschrift „General commands (tools and utilities)“, die Sektion 2 zu „System calls and error numbers“ und die Sektion 3 zu „Libraries“ usw.

| man          |   |
|--------------|---|
| Beschreibung | Zeigt die Handbuchseiten (manual page) zum spezifizierten Befehl (Kommando) an.   |
| Syntax       | <code>man [&lt;gruppe&gt;] [&lt;optionen&gt;] &lt;kommando&gt;</code><br>Die Gruppe, besser auch Abschnitt genannt, wählt einen bestimmten Abschnitt aus den Handbuchseiten aus.                        |
| Beispiele    | # Die Handbuchseite mit allen Sektionen für das<br># Programm <code>echo</code> anzeigen<br><code>man echo</code><br># Die Handbuchseite mit der Sektion<br># für das Programm <code>ps</code> anzeigen |

|  |  |
|--|--|
|  | <code>man 1 ps</code><br># Hilfe für das Kommando <code>man</code> ausgeben<br><code>man --help</code> |
|--|--|

Das Programm `man` wird mit der Taste „q“ beendet. Die Navigation erfolgt mit den Pfeiltasten nach oben bzw. unten.

Weiterhin kann man mit dem Kommando `help` gezielt die Syntax für ein konkretes internes Bash-Kommando anzeigen lassen.

| help         |  |
|--------------|--|
| Beschreibung | Zeigt die Syntax für ein internes Bash-Kommando an.                                      |
| Syntax       | <code>help &lt;kommando&gt;</code>   |
| Beispiele    | # Zeigt die Kommandosyntax des Kommandos <code>echo</code> an.<br><code>help echo</code> |

## 2.8 Übungen

### 2.8.1 Hinweise für Studierende der Hochschule München

Auf unseren Laborrechnern sind virtuelle Arbeitsplatzrechner aus der LRZ Compute Cloud mit Windows 8 als Betriebssystem verfügbar. Auf den virtuellen Windows-Maschinen ist auch der Hypervisor mit der Bezeichnung Virtualbox mit einer vorkonfigurierten Ubuntu-Linux-Instanz eingerichtet. Dort finden Sie nach dem Login eine grafische Oberfläche; in der Sie auch eine Terminalemulation starten können, in der auch die Bash verfügbar ist.

Der Zugang zum Windows-System erfolgt über die Kennung „HM-...“ (Hochschul-Passwort), die für jeden Studierenden eingerichtet ist.

#### **Tipps für das Arbeiten am eigenen Rechner:**

Im eigenen Notebook oder PC, auf dem Windows 10 können Sie das Linux-Derivat Ubuntu oder ein anderes Linux-Derivat aus dem Windows Store laden. Damit können Sie auch die Bash ausführen. Mehr Informationen finden Sie unter <https://docs.microsoft.com/en-us/windows/wsl/about> (zugegriffen am 21.02.2019).

Sie können sich auch selbst auf Ihrem privaten Rechner einen Hypervisor wie VirtualBox einrichten und ein Image für Ubuntu oder CentOS erzeugen.

Unter macOS können Sie ebenfalls ein Terminal nutzen, in dem die meisten Kommandos wie beschrieben getestet werden können. Hierfür ist nichts weiter zu installieren.

Selbstverständlich können Sie auf Ihrem privaten Rechner auch ein Linux-Derivat wie Ubuntu oder CentOS als Betriebssystem direkt installieren und Ihre Übungen darauf ausführen.

### 2.8.2 Übung zu den Handbuchseiten

Lesen Sie die Handbuchseiten zur `bash` mit dem Kommando `man`. Geben Sie auch die Kommandoübersicht mit `man -help` aus. Üben Sie den Umgang mit den Kommandos. Sie müssen noch nicht alles verstehen, was in den Handbuchseiten steht. Sie sollten vor allem versuchen, die Syntax der Kommandobeschreibung zu verstehen.



### 2.8.3 Übung zum Kommando help

Betrachten Sie sich mit den Kommandos `help echo` und `help if` die internen Kommandos der Bash und notieren Sie wichtige Aspekte, Sie müssen noch nicht alles verstehen.

### 2.8.4 Übung zum Anlegen Anlegen von Umgebungsvariablen

Betrachten Sie sich mit dem Kommando `env` bzw. `printenv` oder auch `set` die in Ihrer Bash-Session vorhandenen Umgebungsvariablen und machen Sie sich einige Notizen dazu.

Erzeugen Sie eine eigene Umgebungsvariable, die Ihren Namen hat und weisen Sie einen beliebigen Wert zu. Führen Sie das Kommando `env`, `printenv` oder `set` erneut aus und suchen Sie nun in der Ausgabe nach Ihrer Umgebungsvariablen.

---

---

---

---

---

---

### 2.8.5 Übung zum Kommando typeset oder declare

Erzeugen Sie mit dem Kommando `typeset` oder `declare` eine Umgebungsvariable mit dem Namen `i1` vom Typ Integer und weisen ihr den Wert 1350 zu. Erzeugen Sie eine zweite Umgebungsvariable `i2` vom gleichen Typ und weisen Sie ihr den Wert von `i1` zu. Zeigen Sie die neuen Umgebungsvariablen auf der Konsole an und löschen Sie diese anschließend wieder.

---

---

---

---

---

---

### 2.8.6 Übung zum Erzeugen und Exportieren von Umgebungsvariablen

Erzeugen Sie mit dem Kommando `typeset` zwei Umgebungsvariablen `int1` und `int2` vom Typ Integer und belegen Sie diese mit beliebigen korrekten Werten.

Exportieren Sie diese Variablen anschließend mit dem Kommando `export` oder `typeset` und überzeugen Sie sich, dass die Variablen auch wirklich exportiert wurden. Starten Sie dazu von der gleichen Bash aus wieder eine Bash und prüfen Sie, ob die neuen Umgebungsvariablen auch in der neuen Bash sichtbar ist.

Führen Sie die Aufgabe anschließend mit einer nicht exportierten Umgebungsvariable aus. Was ist der Unterschied?

---

---

---

---

---

---

### 2.8.7 Übung zu alias

Definieren Sie sich ein eigenes Kommando `ml`, das die Manual-Page des Kommandos `man` ausgibt.

---

---

---

---

## 3 Bash-Kommandos für die Dateibearbeitung

### Zielsetzung des Kapitels

Der Studierende soll am Ende dieses Kapitels den Umgang mit Dateien und Verzeichnissen in der Bash beherrschen. Hierzu gehören das Anlegen, Ändern, Verschieben und Löschen von Dateien und Verzeichnissen im Dateisystem von Linux. Weiterhin soll der Studierende Ein- und Ausgabekanäle umlenken und Pipes in Bash-Kommandos nutzen können.

### Wichtige Begriffe

Dateisystem, Datei, Verzeichnis, Zugriffsrechte, relative und absolute Pfadnamen, Pipe.

### 3.1 Dateien und Verzeichnisse

Wie bereits erläutert verwaltet Linux Verzeichnisse und Dateien in einer hierarchischen Baumstruktur. Verzeichnisse (Syn: Directory oder Ordner) bieten dem Nutzer die Möglichkeit, seinen Datenbestand sinnvoll zu strukturieren. Um Dateien und Verzeichnisse wiederzufinden, erhalten sie Namen, die aus Groß- und Kleinbuchstaben sowie aus Sonderzeichen bestehen können. Alle Zeichen außer dem Schrägstrich "/" und dem Nullzeichen (hexadezimal X`00`) sind erlaubt. Auch Umlaute und Leerzeichen sind möglich. Problematisch sind Sonderzeichen, die auch eine andere Bedeutung haben, wie z.B. |, >, <, ?, \*, wenn man Dateinamen in Kommandos als Argumente verwenden möchte, was eigentlich meist notwendig ist.

Es ist daher zu empfehlen, dass für Datei- und Verzeichnisnamen keine Umlaute, Leerzeichen und Sonderzeichen genutzt werden. Der Unterstrich „\_“, der Punkt „.“ und das Minuszeichen „-“ können aber gut verwendet werden, um die Lesbarkeit von Dateinamen zu verbessern.

Oft werden Dateinamen auch durch Einfügen von Punkten strukturiert bzw. in zwei Teilen dargestellt. Der Teil des Dateinamens nach dem letzten Punkt wird als *Suffix*, Dateinamenserweiterung (filename extension), Dateierweiterung oder Dateiteilendung bezeichnet. Ein Suffix macht meist eine Aussage über den Dateityp. Beispielsweise verwendet man als Suffix üblicherweise .txt, .ppt, .c, ja-

### 3 Bash-Kommandos für die Dateibearbeitung

---

va, .class, .html und viele mehr. Dies ist aber nur eine Konvention bzw. übliche Nutzung. Viele Programme nutzen aber Suffixe, um Dateien automatisch zu erkennen.

Dateinamen dürfen maximal 255 Zeichen lang sein. Auf die genaue Schreibweise ist unbedingt zu achten. So ist beispielsweise die Datei mit dem Namen „Abc“ eine andere als mit dem Namen „abc“.

Im Dateisystem wird der vollständige Name einer Datei oder eines Verzeichnisses als *absoluter Pfadname* bezeichnet. Er beginnt mit der Wurzel "/" des Dateisystems. Der Schrägstrich "/" wird benutzt, um die Namen der aufeinanderfolgenden Verzeichnisse (Subverzeichnisse) abzugrenzen.

Beispiel: /home/mandl/testprogramm.c

Man kann aber Dateien oder Verzeichnisse von jeder Stelle des Dateibaums, an der man sich gerade befindet, adressieren. Hier wendet man die relative Adressierung bzw. den *relativen Pfadnamen* an. Befindet man sich beispielsweise unter /home im Dateiverzeichnis, lässt sich die obige Datei mit mandl/testprogramm.c adressieren.

Befindet man sich in dem Verzeichnis, in dem auch die Datei liegt, z. B. unter /home/mandl, kann man die Datei auch nur mit dem einfachen Namen ansprechen. Im Beispiel wäre das testprogramm.c.

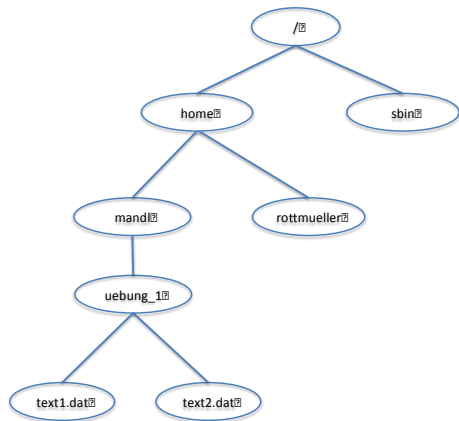
Das Sonderzeichen „.“ in einem relativen Pfadnamen verweist auf das aktuelle Arbeitsverzeichnis, die Sonderzeichenkombination „..“ (zwei Punkte) verweist auf das im Dateibaum übergeordnete Verzeichnis in Richtung der Wurzel (in der Informatik ist die Wurzel meistens oben während sie im richtigen Leben oft unten ist).

Anhand des Dateibaumauszugs aus Abbildung 3-1 lässt sich demonstrieren, was mit absoluter und relativer Pfadangabe gemeint ist. Der Baumausschnitt zeigt mehrere Verzeichnisse und zwei Dateien (text1.dat und text2.dat). Der oberste Knoten hat keine Vorgänger und wird daher als Wurzel bezeichnet. Einige Knoten sind Subverzeichnisse (home, rottmueller, ...). Unterhalb einer Datei kann kein weiterer Knoten mehr angelegt werden, unterhalb von Verzeichnissen schon. Wichtig dabei ist, wo man sich im Baum gerade befindet. Dies findet man mit dem Kommando pwd heraus.

Bei Angabe der absoluten Pfade kann man eine Datei oder ein Verzeichnis von jeder Position aus adressieren, da man immer den vollständigen Namen angibt. Der vollständige bzw. absolute Pfadname der Datei text1.dat ist in unserem Teilbaum /home/mandl/uebung\_1/text1.dat.

Befindet man sich im Verzeichnis `/home/rottmueller`, kann man die Datei `text1.dat` durch die Angabe des folgenden relativen Pfadnamens erfolgen:

`../mandl/uebung1/text1.dat`



**Abbildung 3-1: Exemplarischer Auszug eines Dateibaums**

Weitere Beispiele für gültige Dateinamen sind in der folgenden Tabelle aufgeführt:

|  |  |
|--|--|
| <code>PeterMandl.txt</code>            | Datei im aktuellen Verzeichnis   |
| <code>BjoernRottmueller.doc</code>     | Datei im aktuellen Verzeichnis   |
| <code>../bjoern.rottmueller.doc</code> | Relativer Pfadname der die Datei <code>bjoern.rottmueller.doc</code> im aktuellen Verzeichnis adressiert.  |
| <code>/home/br/text_1.doc</code>       | Absoluter Pfadname der Datei mit dem Namen <code>text_1.doc</code> .   |
| <code>../users/Uebung1.zip</code>      | Datei <code>Uebung1.zip</code> mit relativem Pfadnamen. Von der aktuellen Position im Verzeichnisbaum aus wird ein Verzeichnis („..“) zurückgegangen und dann ins Verzeichnis <code>users</code> positioniert. |
| <code>../../peter_mandl.de</code>      | Sonderbedeutung „..“ wird hier im relativen Pfadnamen zweimal genutzt.   |

Im Dateisystem wird eine Datei in einer internen Datenstruktur beschrieben. Der Inhalt ist nur für die nutzenden Anwendungen bzw. Kommandos interessant. Für

das Betriebssystem ist der Inhalt nur eine Folge von Oktets und kann semantisch alles möglich sein (eine Bilddatei, eine Textdatei, eine Programmcode-Datei, ...).

Im Dateisystem kann man neben diesen, als regulär bezeichneten Dateien, auch solche Dateien anlegen, die eigentlich nur Verknüpfungen (Links) auf andere Dateien oder Verzeichnisse repräsentieren. Man unterscheidet *symbolische Links* und harte Links (hard links). Zum Erzeugen von symbolischen und harten Links verwendet man das Kommando `ln`. Technisch wird eigentlich im Dateisystem nur Verwaltungsinformation abgelegt, mit der man auf die Originaldatei zugreifen kann. Mit Links werden wir uns später noch beschäftigen.

Im Folgenden werden wir die wichtigsten Kommandos für die Bearbeitung von Dateien und Verzeichnissen betrachten.

## 3.2 Kommandoübersicht

### 3.2.1 Kommando echo

| echo         |  |
|--------------|--|
| Beschreibung | Textzeile ausgeben.  |
| Syntax       | <code>echo [&lt;shortOption&gt;] [&lt;string&gt;]</code>   |
| Beispiele    | <pre>echo "Hello World!"<br/># -n unterdrückt den Zeilenumbruch nach Ausgabe<br/># von "Hello"<br/>echo -n "Hello"<br/># zeigt einen Hilfetext für echo an<br/>help echo</pre> |

Mit dem Kommando `echo` können Sie auch Steuerzeichen ausgeben. Steuerzeichen beginnen mit einem Backslash („\“). Damit das Steuerzeichen auch also solches erkannt wird, müssen Sie bei Aufruf von `echo` die Option `-e` angeben, sollen Steuerzeichen explizit nicht erkannt werden, geben Sie die Option `-E` an.

Typische Steuerzeichen sind `\n` für „Neue Zeile“, `\a` für die Ausgabe einer Terminalglocke (Klingel) und `\t` für die Ausgabe eines horizontalen Tabulatorsprungs.

Beispiele:

|   |   |
|---|---|
| <code>echo -e „Dies ist ein \n Umbruch“</code>  | Nach „Dies ist ein“ wird auf eine neue Zeile gewechselt.  |
| <code>echo -E „Dies ist kein \n Umbruch“</code> | Das Steuerzeichen für „Neue Zeile“ wird ignoriert. Es wird folgende auf dem Terminal ausgegeben:<br><code>&gt;Dies ist kein \n Umbruch</code> |

### 3.2.2 Kommando pwd

| pwd          |  |
|--------------|--|
| Beschreibung | Gibt das aktuelle Arbeitsverzeichnis aus.                    |
| Syntax       | <code>pwd [-LP]</code>                                       |
| Beispiele    | # zeigt Pfad an in dem man sich befindet<br><code>pwd</code> |

### 3.2.3 Kommando cd

| cd (change directory) |  |
|-----------------------|--|
| Beschreibung          | Führt einen Verzeichniswechsel aus.  |
| Syntax                | <code>cd &lt;verzeichnis&gt;</code>  |
| Beispiele             | # wechselt in das angegebene Verzeichnis<br><code>cd temp</code><br># wechselt in das eigene Heimatverzeichnis<br><code>cd</code> oder <code>cd ~</code> oder <code>cd \$HOME</code><br># wechselt in das übergeordnete Verzeichnis.<br><code>cd ..</code> |

### 3.2.4 Kommando ls

| ls (list)    |  |
|--------------|--|
| Beschreibung | Zeigt den Inhalt des aktuellen oder des angegebenen Verzeichnisses an. |

### 3 Bash-Kommandos für die Dateibearbeitung

---

|           |  |
|-----------|--|
| Syntax    | <code>ls &lt;optionen&gt; &lt;verzeichnis&gt;</code>   |
| Beispiele | <pre># zeigt den Inhalt des Verzeichnis temp an ls temp  # zeigt ausführlichere Informationen an ls -l  # auch versteckte Dateien und Verzeichnisse werden an- gezeigt. Das sind Dateien, die mit einem „.“ beginnen. ls -a  # Kombination von Optionen ls -al</pre> |



### 3.2.5 Kommando touch

| touch        |  |
|--------------|--|
| Beschreibung | Mit dem Kommando kann man die Zeitstempel für den letzten Zugriff und die letzte Änderung von Dateien verändern. Ohne weitere Angabe werden alle Zeiten (letzter Zugriff, letzte Änderung) auf die aktuelle Systemzeit gesetzt. Existiert eine Datei mit angegebenen Namen noch nicht, wird sie erzeugt. |
| Syntax       | <code>touch &lt;optionen&gt; &lt;dateiname_1&gt; ... &lt;dateiname_n&gt;</code>  |
| Beispiele    | <pre># aktualisiert das Datum von zwei Dateien touch myfile1.txt myfile2.dat # aktualisiert das Änderungsdatum der Datei # datei1.dat auf 24. Dez. 2018, 10:03 Uhr touch -t 201812241003 datei1.dat</pre>  |

### 3.2.6 Kommando mkdir

| mkdir        |  |
|--------------|--|
| Beschreibung | Verzeichnis anlegen.   |
| Syntax       | <code>mkdir &lt;verzeichnisname&gt;</code>   |
| Beispiele    | <pre># Legt das Verzeichnis temp im aktuellen Verzeichnis, # in dem man sich gerade befinden (siehe pwd) an mkdir temp</pre> |

### 3.2.7 Kommando rmdir

| rmdir (remove directory) |  |
|--------------------------|--|
| Beschreibung             | Löscht das/die angegebene(n) Verzeichnis(se) endgültig. Das angegebene Verzeichnis muss bereits leer sein (siehe rm-Kommando). |
| Syntax                   | <code>rmdir &lt;verzeichnisname&gt;</code>   |
| Beispiele                | <pre># Löscht das Verzeichnis temp rmdir temp</pre>  |

#### 3.2.8 Kommando cp

| cp (copy)    |   |
|--------------|---|
| Beschreibung | Kopieren von Dateien / Verzeichnissen.  |
| Syntax       | <pre>cp &lt;alteDatei&gt; &lt;neueDatei&gt; cp &lt;dateiname&gt; &lt;zielverzeichnis&gt; cp &lt;dateiname_1&gt; &lt;dateiname_2&gt; ... &lt;dateiname_n&gt; &lt;zielverzeichnis&gt;</pre> |
| Beispiele    | <pre># kopiert die Datei "bsp.txt" mit gleichem Namen # in das Verzeichnis /home/user1 cp /temp/bsp.txt /home/user1/</pre>  |

#### 3.2.9 Kommando cat

| cat          |   |
|--------------|---|
| Beschreibung | Liest eine Datei ein und schreibt sie auf Standardausgabe (stdout).           |
| Syntax       | <pre>cat &lt;dateiname&gt;</pre>  |
| Beispiele    | <pre># zeigt Inhalt der bsp.txt Datei auf dem Bildschirm an cat bsp.txt</pre> |

#### 3.2.10 Kommando less oder more

| less         |  |
|--------------|--|
| Beschreibung | Gibt jeweils eine Bildschirmseite einer Datei aus.   |
| Syntax       | <pre>less &lt;dateiname&gt; more &lt;dateiname&gt;</pre>   |
| Beispiele    | <pre># blättert in der Datei err.log less err.log  # blättert eine Seite weiter f, &lt;space&gt;  # blättert eine Seite zurück b  # blättert eine Zeile weiter</pre> |

|  |  |
|--|--|
|  | <pre>&lt;enter&gt;, &lt;Pfeil nach unten&gt; # blättert eine Zeile zurück &lt;Pfeil nach oben&gt; # Blättern beenden q</pre> |
|--|--|

### 3.2.11 Kommando mv

| mv (move)    |  |
|--------------|--|
| Beschreibung | Verschieben oder Umbenennen von Dateien oder Verzeichnissen.   |
| Syntax       | <p>Umbenennen einer Datei mit dem Namen &lt;Datei_1&gt; in den Namen &lt;Datei_2&gt;</p> <pre>mv &lt;Datei_1&gt; &lt;Datei_2&gt;</pre> <p>Verschieben einer Datei in das angegebene Verzeichnis</p> <pre>mv &lt;Datei&gt; &lt;Verzeichnis&gt;</pre>  |
| Beispiele    | <pre># benennt dat1.txt in dat2.txt um mv dat1.txt dat2.txt  # verschiebt alle Dateien, in deren Dateinamen das Wort # april vorkommt in ein Verzeichnis namens april. # Voraussetzung: Der Ordner april ist vorhanden und # Beispieldateien mit Namen wie 01april2009.txt oder # 09april2007.txt sind ebenfalls vorhanden mv ??april* april</pre> |

### 3.2.12 Kommando rm

| rm (remove)  |  |
|--------------|--|
| Beschreibung | Löscht die angegebene(n) Datei(en) endgültig.  |
| Syntax       | <pre>rm &lt;optionen&gt; &lt;dateiname_1&gt; ... &lt;dateiname_n&gt;</pre> <p>Wenn man den Inhalt der Unterverzeichnisse einschließlich des angegebenen Verzeichnisses auch gleich löschen möchte (recurse), nutzt man</p> |

### 3 Bash-Kommandos für die Dateibearbeitung

|           |   |
|-----------|---|
|           | <code>rm -r &lt;Ordner&gt;</code><br>Löschen schreibgeschützter Dateien erzwingen (force)<br><code>rm -f &lt;Datei&gt;</code>                                     |
| Beispiele | <code># löscht die beiden Dateien</code><br><code>rm dat1.txt dat2.txt</code><br><code># löscht das Verzeichnis temp mit Inhalt</code><br><code>rm -r temp</code> |

#### 3.2.13 Kommando ln

| ln           |   |
|--------------|---|
| Beschreibung | Softlink oder Hardlink anlegen.<br><br>Unter einem Hardlink versteht man einen Verweis auf eine andere Datei innerhalb des Dateisystems. Die Rechte der Datei (werden weiter unten erläutert) bleiben in diesem Fall unverändert.<br><br>Ein Softlink hingegen hat ähnliche Funktionalität, allerdings können damit auch Rechte angepasst und Verzeichnisse adressiert werden.<br><br>Beide Link-Typen sind durch ein führendes „l“ in der Berechtigungskette (sichtbar über <code>ls -la</code> ) zu erkennen. |
| Syntax       | <code>ln &lt;optionen&gt;</code>  |
| Beispiele    | <code># Symbolischen Link t2.dat erzeugen, der auf</code><br><code># t1.dat zeigt</code><br><code>ls -s t1.dat t2.dat</code>  |

#### 3.2.14 Kommando wc

| wc (word count) |  |
|-----------------|--|
| Beschreibung    | Mit dem Kommando kann man die Anzahl der verwendeten Zeilen, Wörter und Zeichen einer oder mehrerer Dateien ausgeben lassen. |
| Syntax          | <code>wc &lt;optionen&gt; &lt;dateiname_1&gt;...&lt;dateiname_n&gt;</code>   |
| Beispiele       | <code># zählt nur die Zeilen, der angegebenen Datei</code>   |

|  |   |
|--|---|
|  | <pre>wc -l test1.dat # zaehlt nur die Wörter, der angegebenen Datei wc -w test1.dat # zählt nur die Zeichen, der angegebenen Datei wc -c test1.dat # Zählt alle Zeilen, Wörter und Zeichen der beiden # angegebenen Dateien und gibt sie aus wc test1.dat test2.txt</pre> |
|--|---|

### 3.3 Ein- und Ausgabekanäle von Prozessen

Jeder Linux-Prozess hat drei Standardkanäle für die Ein-/Ausgabe:

- Standardeingabekanal, auch als stdin bzw. systemintern mit dem Filedescriptor 0 bezeichnet. Üblicherweise ist die Standardeingabe auf das Terminal gelegt.
- Standardausgabekanal, auch als stdout, bzw. systemintern mit dem Filedescriptor 1 bezeichnet. Die Standardausgabe ist standarmäßig mit dem Bildschirm verbunden.
- Standardfehlerkanal, auch als stderr, bzw. systemintern mit dem Filedescriptor 2 bezeichnet. Auch stderr ist üblicherweise mit dem Bildschirm verbunden.

Ein Filedescriptor (auch Handle genannt) kennzeichnet prozessintern eine geöffnete Datei. Er wird angegeben, wenn man auf eine Datei mit Ein- bzw. Ausgabeoperationen zugreifen möchte. In der Bash kann man die Filedescriptoren auch direkt nutzen, wenn man die Kanäle umlenken möchte (mehr dazu in (Mandl 2014)).

Eine Umlenkung (Redirection) der Kanäle ist durch Nutzung der Zeichen „>“ und „<“ möglich. Mit „>“ lenkt man die Ausgabe um, mit „<“ die Eingabe. Auch die Zeichenfolge „>>“ hat eine besondere Bedeutung. Sie wird verwendet, wenn man bei der Ausgabe an eine Datei hinten anfügen möchte. Bei Nutzung von „>“ wird die Datei immer neu beschrieben.

Beispiele:

```
# Umlenken der Standardausgabe in die Datei ls.dat
ls -al > ls.dat

# Umlenken der Standardfehlerausgabe in die Datei error.log.
# in diesem Fall steht in der Datei nach Ausführung des Befehles ein Text, der
# angibt, dass die verwendete Option falsch ist
ps -XXX 2> error.log

# Umlenken der Standardausgabe in die Datei ls.dat. Die Ausgabe wird an das
# Ende der Datei angehängt
ps -ax >> ls.dat

# Umlenken der Standardeingabe. Das Kommando liest von der Datei in.data
wc < in.data

# Umlenken der Standardausgabe auf das Null-Device und damit Verwerfen der
# Ausgabe
echo "Mist" > /dev/null
```

## 3.4 Pipes

Unter einer Pipeline oder Pipe versteht man unter Linux die Möglichkeit, die Standardausgabe eines Programms in die Standardeingabe eines anderen Programms zu schreiben. Die Ausgaben, die ein Programm (ein Kommando) liefert, können also als Standardeingabe an ein anderes Programm übergeben werden. Hierfür verwendet man das Zeichen „|“ (Pipe-Symbol).

Beispiel:

```
ls | wc -w
```

In dieser Kommandofolge wird die Ausgabe von `ls` (zur Erinnerung: `ls` listet den Inhalt eines Verzeichnisses auf) durch Nutzung einer Pipe nach `wc` (Word-Count) umgeleitet. Statt einer Liste von Dateien und Unterverzeichnissen erhält man dadurch die Anzahl der Wörter, die der `ls`-Befehl auf dem Bildschirm anzeigt.

Auch eine Pipeline bestehend aus Anweisungen, die jeweils mit einer Pipe verbunden sind, ist möglich.

Beispiel:

```
ls | wc | wc -w
```

In diesem Beispiel wird die Ausgabe des Kommandos `ls` über `wc` verarbeitet. Die Ausgabe des Kommandos `wc` über eine weitere Pipe an den nächsten `wc`-Befehl zur Verarbeitung weitergegeben.

Pipes kann man also nutzen, um mehrere Kommandos in ganzen Verarbeitungsketten zu verbinden.

## 3.5 Quoting und Expandierungen

Mit Quoting (Graiger 2009) bezeichnet man in der Bash den Schutz spezieller Zeichen mit Sonderbedeutung vor einer Interpretation. Die geschützten Zeichen werden von der Bash nicht als Sonderzeichen interpretiert. Hierfür werden Apostrophe, auch als Quotes bezeichnet, und das Zeichen Backslash verwendet:

- Ein Backslash schützt das unmittelbar folgende Zeichen.
- Wird eine Zeichenkette in einfache Apostrophe (einfaches Hochkomma `'`) geschrieben, wird die ganze Zeichenkette geschützt.
- Eine Zeichenkette, die in doppelten Hochkommas steht, wird ebenfalls geschützt. Nur ein paar wenige Sonderzeichen sind vom Schutz ausgenommen. Hierzu gehören `$`, ```, Backslash und `!`.
- Eine besondere Bedeutung hat schließlich noch das Zeichen ``` (umgekehrter Apostroph). Ein Kommando, das in umgekehrten Apostrophen steht, wird von der Bash ausgeführt. Die Bash liefert an die Stelle der Ausführung das Ergebnis des Kommandos aus.

### 3 Bash-Kommandos für die Dateibearbeitung

---

Beispiele:

```
# Inhalt von $HOME wird ausgegeben
echo "$HOME"

# oder
echo $HOME

# Inhalt von $HOME wird nicht ausgegeben
echo '$HOME'
$HOME

# In der Umgebungsvariable RESULT wird das Ergebnis des ls-Befehls gespeichert
RESULT=`ls`

# Ausgabe des Ergebnisses von ls auf Bildschirm
echo $RESULT
```

Wenn der Bash-Interpreter ein eingegebenes Kommando analysiert und zerlegt hat, werden noch vor der Ausführung sog. Expandierungen durchgeführt. Nach (Grainger 2009) sind dies sieben an der Zahl, die in einer vorgegebenen Reihenfolge durchgeführt werden. Die folgende Tabelle gibt einige wichtige Ersetzungsregeln mit einfachen Beispielen an. Auch die Nutzung von Umgebungsvariablen, von regulären Ausdrücken und Quotings wird in den Beispielen gezeigt.

| Typ                                  | Beschreibung   | Beispiel   |
|--------------------------------------|--|--|
| Kommandoexpandierung                 | Ersetzen von geschweiften Klammern {...} in Kommandos  | echo M{aus,umm}<br>Maus Mumm<br>echo {a, b}{y,z}<br>ay az by bz          |
| Tildenersetzung                      | Das Tildezeichen ~ wird durch das Homeverzeichnis expandiert.  | vi ~/.bashrc<br>vi<br>/home/mandl/.bashrc                                |
| Parameter- und Variablenexpandierung | \$-Zeichen zeigen zu expandierende Parameter bzw. Variablen an. Die Namen der Parameter/Variablen werden in geschweifte Klammern | verzeichnis=heimat<br>heimat=/home/mandl<br>echo \${verzeichnis}<br>oder |



### 3 Bash-Kommandos für die Dateibearbeitung

|                            |   |  |
|----------------------------|---|--|
|                            | gesetzt oder können direkt angegeben werden.                | echo \$verzeichnis<br>heimat<br>echo \${!verzeichnis}<br>/home/mandl                     |
| Befehls-<br>ersetzungen    | Ersetzen der Ausgabe eines Befehls in der Befehlszeile.     | lsergebnis=\${ls}<br>echo \$lsergebnis oder<br>echo `ls`<br><...Ausgabe von ls-Kommando> |
| Dateinamen-<br>ersetzungen | Musterauswertung (siehe reguläre Ausdrücke für Dateinamen). | ls *.txt<br>text1.txt text2.txt ...  |

### 3.6 Kommando-Historie

Die Bash führt die Historie der eingegebenen Kommandos in einem Kommandospeicher mit. Über den Befehl `history`, ein internes Kommando der Bash, kann die Historie genutzt werden. Komplexe Befehle müssen so nicht nochmals eingetippt werden. Die Historie kann aufgelistet werden, es können gezielt bestimmte Kommandos der Historie erneut aufgerufen oder gelöscht werden. Auch der gesamte Kommandospeicher kann gelöscht werden. Die Speicherkapazität der Kommando-Historie wird in der Regel begrenzt, so dass nur die *n* letzten Kommandos aufgezeichnet werden (Standard: 500 Zeilen). Die Größe kann man auch über die Umgebungsvariable `HISTFILESIZE` verändern

```
# Löschen des Kommandospeichers
history -c

# Ändern der maximalen Zeilenanzahl in der Historie auf 100 Zeilen
HISTFILESIZE=100

# Ausgabe des Kommandospeichers
history
1 history
ls
...
pwd
```

### 3 Bash-Kommandos für die Dateibearbeitung

---

```
history
1 history
2 ls
3 pwd
4 history
$ history -d 3
$ history
1 history
2 ls
3 history
4 history -d 3
5 history
# Aufruf des 2. Kommandos aus dem Kommandospeicher
!2
ls
```

In der Historie kann auch mit den Pfeiltasten geblättert werden. Dies wird am häufigsten verwendet.

Der Kommandospeicher wird standardmäßig im Home-Verzeichnis eines Benutzers standardmäßig in der Datei `~/.bash_history` persistent abgelegt und kann daher auch Session-übergreifend beim nächsten Login wiederverwendet werden. Man kann auch eine andere Datei verwenden, wenn man den Dateinamen dafür in die Umgebungsvariable `HISTFILE` einträgt.

Will man die ganze Historie löschen geht dies auch über folgende Kommandofolge:

```
history -c
history -w
```

Alternativ kann man dasselbe auch über das Kommando `cat` erreichen:

```
# Löschen des ganzen Kommandospeichers über die Session hinweg:
cd
cat /dev/null > ~/.bash_history
```

## 3.7 Übungen

### 3.7.1 Übung zur Kommando-Historie

Lassen Sie sich die Kommando-Historie in Ihrer Bash-Session ausgeben und versuchen Sie einige Befehle aus der Historie aufzurufen. Löschen Sie danach die Kommandos-Historie und geben Sie nochmals das Kommando `history` ein. Notieren Sie sich wichtige Aspekte.

---

---

---

---

---

---

---

---

### 3.7.2 Übung zur Datei- und Verzeichnisbearbeitung

Öffnen Sie ein Terminal unter Linux und gehen Sie in die Bash.

1. In welchem Verzeichnis befinden Sie sich gerade?

---

---

---

---

2. Lassen Sie sich den Inhalt des aktuellen Verzeichnisses anzeigen.

---

---

---

3. Erzeugen Sie im Home-Verzeichnis die drei Dateien `f1.txt`, `f2.txt` und `f3.txt`.

---

---

---

### 3 Bash-Kommandos für die Dateibearbeitung

---

4. Legen Sie im Home-Verzeichnis ein Verzeichnis mit dem Namen `uebung_winf` an. Nachdem Sie den Befehl `mkdir` ausgeführt haben, lassen Sie sich mit `ls` den Inhalt des aktuellen Verzeichnisses anzeigen. Der Ordner `uebung_winf` müsste nun angezeigt werden.

5. Wechseln Sie in das von Ihnen erstellte Verzeichnis `uebung_winf`. Lassen Sie sich den Inhalt des Verzeichnisses anzeigen.

6. Löschen Sie das Übungsverzeichnis.

7. Sie sollten sich jetzt im HOME-Verzeichnis befinden. Wenn nicht, dann wechseln Sie dorthin. Erzeugen Sie mit Hilfe des Befehls `echo > test.txt` oder `touch test.txt` eine neue Datei. Erstellen Sie mit dem Kopierbefehl ein paar Kopien der erstellten Datei. Geben Sie diesen Dateien die Namen `test_1.txt`, `test_2.txt`, `test_3.txt`, etc.

### 3 Bash-Kommandos für die Dateibearbeitung

8. Versuchen Sie nun alle Dateien, die mit `test` beginnen und als Dateiendung `.txt` besitzen, in ein neues Verzeichnis zu kopieren. Evtl. muss vorher ein anderer Befehl abgesetzt werden.

9. Lassen Sie sich eine beliebige binäre Datei (z. B. eine Zip-Datei) mit Hilfe des `cat`-Befehls anzeigen. Wie sieht die Ausgabe aus?

10. Verschieben Sie mit dem Befehl `mv` alle Dateien mit der Endung `.txt` in das Verzeichnis mit dem Namen `uebung_winf2`.

### 3 Bash-Kommandos für die Dateibearbeitung

---

11. Wechseln Sie nun in das Verzeichnis `uebung_winf` und benennen Sie die Datei `test.txt` in `test0.txt` um. Das Verzeichnis muss evtl. erneut angelegt werden.

---

---

---

---

12. Schreiben Sie mehrere Textzeilen in die Datei `text.txt`. Hierzu helfen der Befehl `echo` und die Ausgabeumleitungsoperatoren.

---

---

---

---

---

13. Zählen Sie die Anzahl der Zeilen in der gerade erzeugten Datei und lassen Sie sich diese auf der Kommandozeile anzeigen. Nutzen Sie dabei das Kommando `cat` eine Pipe und das Kommando `wc`.

---

---

---

---

14. Schauen Sie sich die letzten verwendeten Befehle in Ihrem Kommandospeicher an (`history`) und nutzen sie den letzten Befehl erneut. Löschen Sie anschließend den benutzten Befehl und geben Sie den Kommandospeicher erneut aus.

---

---

---

---

15. Erzeugen Sie in der aktuellen Bash eine weitere Bash und zeigen Sie nun den Kommandospeicher erneut an. Was sehen Sie? Verlassen Sie die neue Bash (mit Kommando `exit`) und zeigen Sie den Kommandospeicher nochmals an. Was sehen Sie nun?

---

---

---

---

## 4 Linux-Berechtigungen

### Zielsetzung des Kapitels

Der Studierende soll am Ende dieses Kapitels das Linux-Berechtigungskonzept und den Login-Vorgang mit Bash-Profiling beschreiben und anwenden können.

### Wichtige Begriffe

Eigentümer, Gruppe, Login-Shell, `bashrc`, `.bash_login`, `.bash_logout`, `etc/profile`, `/etc/bash.bashrc`, `umask`, `chmod`, `chown`.

### 4.1 Berechtigungskonzept

Im Heimatverzeichnis und in allen Verzeichnissen, die unterhalb des Heimatverzeichnisses liegen, kann der Benutzer<sup>4</sup> beliebig über seine Rechte verfügen. Wenn ein Benutzer aber in anderen Verzeichnissen arbeiten möchte, ist er bestimmten Einschränkungen unterworfen. Diese sind durch das Linux-Berechtigungskonzept geregelt.

Es gibt einen Benutzer, der alles machen darf, auch alle systemrelevanten Dateien bearbeiten. Er wird als Superuser oder Systemadministrator bezeichnet und hat die Benutzerkennung `root`.

Im Berechtigungskonzept unterscheidet sich das Linux- vom Windows-System erheblich. So wird zu jeder Datei explizit ein Eigentümer gespeichert. Auch die Gruppe, zu der diese Datei gehört wird im Dateisystem vermerkt. Außerdem wird spezifiziert, wer die Datei lesen, ändern oder ausführen darf. Die Informationen lassen sich über den `ls`-Befehl mit der Option `-l` anzeigen.

Jede Datei und jedes Verzeichnis ist durch die Zugriffsrechts-Tripel Lesen (r), Schreiben (w) und Ausführen (x) gekennzeichnet. Dieses Tripel gibt es für die Rechte des Eigentümers/Besitzers (owners), für die Rechte der Gruppe (group), zu den die Datei gehört und für alle anderen (other).

Eine Gruppe hat in Linux mehrere Benutzer. Die Zuordnung trifft der Superuser.

---

<sup>4</sup> Dies gilt natürlich genauso für Benutzerinnen, wie alles, was in diesem Skriptum erläutert wird.



Diejenige Benutzerkennung, die eine Datei erzeugt, hat zunächst alle Rechte. Die neu angelegte Datei erhält eine Standardeinstellung, die üblicherweise das Lese- und Schreibrecht für den Eigentümer, das Leserecht für die Gruppe und andere beinhaltet. Mit dem Kommando `umask` kann man die Voreinstellung verändern.

Die neu erzeugte Datei erhält als Eigentümer den Benutzer, der sie anlegt und die Gruppe zugeordnet, in der er sich befindet. Die Dateien des Superusers erhalten die Gruppe `root`.

Die Zugriffsrechte werden mit den Buchstaben `r`, `w`, `x` (`r` = read; `w` = write; `x` = execute) bezeichnet, aber auch die Angabe von Zahlen ist möglich: 4 steht für `r`, 2 für `w` und 1 für `x`. Der Eigentümer wird mit „`u`“ abgekürzt, die Gruppe mit „`g`“ und andere mit „`o`“. Alle Benutzer (Eigentümer, Gruppe und andere) werden mit „`a`“ abgekürzt. In Abbildung 3-1 wird dies nochmals verdeutlicht. Das linke Zeichen „`d`“ deutet auf ein Verzeichnis hin. Hier kann auch ein „`l`“ bei einem Link oder das Zeichen „`-`“ bei einer normalen Datei stehen.

| d | r | w | x | r | w | x | r | w | x |
|---|---|---|---|---|---|---|---|---|---|
|   | 4 | 2 | 1 | 4 | 2 | 1 | 4 | 2 | 1 |
|   | u |   |   | g |   |   | o |   |   |
|   | a |   |   |   |   |   |   |   |   |


Abbildung 3-1: Linux-Zugriffsrechte


Um die Zugriffsrechte von Dateien und Verzeichnissen anzuzeigen, nutzt man den Befehl `ls -al`. Eine typische Ausgabe könnte in etwa so aussehen:

```

-rwxr-xr-x 2 ifwXX users 43 Nov 17 15:13 bash1.sh
drwxr-xr-x 4 root root 4096 Nov 17 15:13 copy

```





Besonders interessant sind hier die ersten 10 Zeichen - die Berechtigungen. Darum werden diese hier nochmals erläutert (`r` = read; `w` = write; `x` = execute):

|                     |
|---------------------|
| d   rwx   r-x   r-x |
|---------------------|

Ganz links steht entweder das Zeichen „`-`“, wenn es sich um eine Datei handelt, oder ein „`d`“ im Falle eines Verzeichnisses. Danach kommen die Dreiergruppen mit den Zugriffsrechten für den Eigentümer, die Gruppe und für andere. Im Rah-

## 4 Linux-Berechtigungen

---

men des Berechtigungskonzepts sind noch die Spalten „Eigentümer“ und „Gruppe“ interessant.

Zur Veränderung der Zugriffsrechte dient das Kommando `chmod`. Die Berechtigungen können entweder durch die Angabe der genannten Buchstaben (r, w, x) oder auch über die Angabe einer adäquaten Dezimalzahl modifiziert werden. Hierfür werden die Rechte in drei Dreier-Bitgruppen aufgeteilt, wobei jede Bitgruppe die Wertigkeiten  $2^0$  (niederwertigstes, rechtes Bit),  $2^1$  (mittleres Bit)  $2^2$  (höherwertiges, linkes Bit) enthält. Damit lässt sich dann für jede Bitgruppe eine Umrechnung ins Oktale durchführen. Für jede Bitgruppe sind die Oktalwerte zwischen 0 und 7 möglich. Die folgende Tabelle zeigt einige Beispiele:

| Buchstaben  | Binär       | Dezimal |
|-------------|-------------|---------|
| rwX r-X r-X | 111 101 101 | 7 5 5   |
| rwX rwX rwX | 111 111 111 | 7 7 7   |
| rw- r-- --- | 110 100 000 | 6 4 0   |

In der Praxis hat sich bewährt, Rechte immer dediziert anzugeben und feingranular zu steuern. Verzeichnisse bzw. Dateien sollten mit den minimal benötigten Rechten ausgestattet sein.

Für die Dateiverwaltung werden Befehle benötigt, um die Zugriffsrechte von Dateien bzw. Verzeichnissen verändern zu können. Diese werden im Folgenden kurz erläutert.

Besonderheiten wie etwa das sog. Sticky-Bit werden in dieser Einführung nicht betrachtet (siehe hierzu z. B. (Wikipedia Sticky)).

### 4.2 Spezielle Kommandos

Im Folgenden werden einige Kommandos für die Rechtevergabe erläutert. Die Kommandos werden nicht umfassend, sondern nur einführend dargestellt. Die vollständigen Beschreibungen mit allen Optionen können über die Hilfefunktion `man` angezeigt werden.

### 4.2.1 Kommando chmod

| chmod (change mode) |  |
|---------------------|--|
| Beschreibung        | Ändert die Dateizugriffsrechte für Dateien und Verzeichnisse   |
| Syntax              | <pre>chmod [&lt;optionen&gt;] &lt;modus&gt; &lt;dateiname_1&gt; ... &lt;dateiname_n&gt;</pre> <p>&lt;modus&gt; gibt an, welche Rechte gesetzt werden. Die Angabe ist in Buchstaben- (r, w, x) oder in Dezimalschreibweise (z. B. 755) zulässig. Die Rechte werden entweder ergänzt (mit „+“) oder entfernt („-“). Rechte lassen sich auch dediziert nur auf Eigentümer, die Gruppe oder andere vergeben.</p>   |
| Beispiele           | <pre># Datei myscript.sh ausführbar machen chmod +x myscript.sh  # setzt für die angegebene Datei das Lese-, Schreib- und # Ausführungsrecht für alle Benutzer chmod 777 test1.dat  # für alle Dateien unterhalb des Verzeichnisses uebung1 wird # das Ausführungsrecht vergeben (Hinweis: -R ist nicht immer # implementiert, z. B. nicht in macOS). chmod -R +x uebung1  # für die angegebene Datei werden alle Rechte des Eigentümers # (Lesen, Schreiben und Ausführen) ergänzt. „u“ steht für # den Eigentümer chmod u+rw test1.dat</pre> |

### 4.2.2 Kommando chown

| chown (change owner) |   |
|----------------------|---|
| Beschreibung         | Setzt oder verändert den Eigentümer bzw. die Gruppenzugehörigkeit.  |
| Syntax               | <pre>chown [&lt;optionen&gt;] &lt;owner&gt;[:&lt;group&gt;] &lt;dateiname_1&gt; ... &lt;dateiname_n&gt;</pre> <pre>chown [&lt;Optionen&gt;] :&lt;group&gt; &lt;dateina- me_1&gt; .. &lt;dateiname_n&gt;</pre> |

## 4 Linux-Berechtigungen

|           |  |
|-----------|--|
|           | Der Aufruf mit der Option <code>-R</code> wirkt sich auf den kompletten Unterordner aus.   |
| Beispiele | <pre># Der Eigentümer der Datei file1 wird auf mandl und # die Gruppe auf dako gesetzt chown mandl:dako file1  # der Eigentümer der Datei file1 wird auf dakouser # gesetzt chown dakouser file1</pre> |

### 4.2.3 Kommando chgrp

| chgrp (change group) |   |
|----------------------|---|
| Beschreibung         | Setzt oder verändert die Gruppenzugehörigkeit.  |
| Syntax               | <pre>chgrp [&lt;optionen&gt;] &lt;gruppe&gt; &lt;Dateiname&gt;</pre> <p>Der Aufruf mit der Option <code>-R</code> wirkt sich auf den kompletten Unterordner aus.</p>                          |
| Beispiele            | <pre># Der aktuelle User ist in der Gruppe musik, er ändert # die Gruppenzuordnung des Verzeichnisses # /opt/musik/ und seinem kompletten Inhalt zu musik chgrp -c -R musik /opt/musik/</pre> |

### 4.2.4 Kommando umask

Durch den Befehl `umask` kann die Dateierzeugungsmaske verändert werden, nach der beim Anlegen einer Datei bzw. eines Ordners die Zugriffsrechte gesetzt werden.

Es ist zu beachten, dass mittels des `umask`-Befehls nur bei neuen Verzeichnissen Ausführungsrechte vergeben werden. Bei Dateien werden diese aus Sicherheitsgründen nicht vergeben, auch wenn die voreingestellte Maske diese beinhaltet.

| umask        |  |
|--------------|--|
| Beschreibung | Der Befehl wird verwendet, um Zugriffsrechte standardmäßig für die laufende Sitzung festzulegen. |

|          |   |
|----------|---|
| Syntax   | <pre>umask &lt;mode&gt;</pre> <p>Das Setzen der Maske (mode) bewirkt das Gegenteil des <code>chmod</code>-Kommandos. Wird ein Recht in der Dateierzeugungs-<br/>maske markiert, wird es „ausmaskiert“, d.h. das Zugriffsrecht wird beim anschließenden Erzeugen einer Datei nicht gesetzt.</p> <p>Das Kommando bezieht sich immer nur auf die zukünftige Erzeugung von Dateien.</p> <p>Das Ausführungsrecht kann mit <code>umask</code> nicht automatisch zugelassen werden.</p>  |
| Beispiel | <pre># Setzen der Dateierzeugungsmaske so, dass in Zukunft # für den Eigentümer und für die Gruppe das Lese- und # das Schreibrecht gilt, für andere nichts # (= Rechteeinstellung: 660) umask 007 umask 006 oder umask 00006 oder umask 0007 oder umask u=rw,g=rw,o= # Ausgabe der aktuellen umask-Einstellung umask # Setzen der Dateierzeugungsmaske so, dass in Zukunft # für die Gruppe und andere das Schreibrecht entfernt # aber das Leserecht vorhanden ist und für den # Eigentümer, Lesen und Schreiben erlaubt sind umask 022</pre> |

Die folgende Tabelle soll Sie bei der Ermittlung der richtigen `umask`-Einstellungen unterstützen. Es zeigt die gewünschten Standardeinstellungen für den Eigentümer der Datei, für die Gruppe und für andere. Der Eigentümer soll in Zukunft standardmäßig das Lese- und auch das Schreibrecht erhalten, die Gruppe nur das Leserecht und andere sollen gar keine Rechte bekommen. Im Beispiel werden drei

## 4 Linux-Berechtigungen

---

mögliche Varianten für den `umask`-Befehl vorgestellt, die alle zum gleichen Ergebnis führen. Nach Eingabe des Kommandos werden alle im Folgenden erzeugten Dateien mit diesen Rechten ausgestattet.

Das Ausführungsrecht kann übrigens aus Sicherheitsgründen nicht mit `umask` voreingestellt werden. Daher führen die Alternativen 1 und 2 zum gleichen Ergebnis. Die Einstellung für das Ausführungsrecht in Alternative 2 wird ignoriert.

| Owner                  | Group            | Other            | Erläuterung   |
|------------------------|------------------|------------------|---|
| Lese- und Schreibrecht | Leserecht        | Keine Rechte     | Gewünschte Standardrechte für neu erzeugte Dateien  |
| <code>rw-</code>       | <code>r--</code> | <code>---</code> | Rechte mit <code>ls</code> -Befehl dargestellt  |
| 6                      | 4                | 0                | Parameter für <code>chmod</code> -Befehl zum Einstellen dieser Rechte   |
| 0                      | 2                | 6                | Alternative 1: Parameter für <code>umask</code> -Befehl   |
| 1                      | 3                | 7                | Alternative 2: Parameter für <code>umask</code> -Befehl   |
| <code>u=rw</code>      | <code>g=r</code> | <code>o=</code>  | Alternative 3: Parameter für <code>umask</code> -Befehl (Parameter mit Kommas trennen: <code>umask u=rw,g=r,o=</code> ) |

### 4.3 Login- und Bash-Profiling

Das Verhalten der Bash lässt sich anpassen und nach individuellen Bedürfnissen gestalten. Im Folgenden werden zwei Varianten aufgezeigt, wie dies realisiert werden kann.

Es gibt in der Bash mehrere Möglichkeiten um den Startup-Vorgang anzupassen. Kommandos können damit bereits beim Start einer Shell automatisch ausgeführt werden. Kommandos werden hierfür in bestimmten Dateien hinterlegt, wobei man systemweite und benutzerspezifische Startup-Dateien unterscheidet.

Systemweite Startup-Dateien liegen gewöhnlich im Verzeichnis `/etc`, benutzerspezifische Startup-Dateien im Home-Verzeichnis. Werden Startup-Dateien gefunden, so werden diese ausgeführt, d.h. alle Kommandos, die sich in den Dateien

befinden, werden aufgerufen. Vorhandene systemweite Startup-Dateien werden vor den benutzerspezifischen ausgeführt.

Die erste Shell, die ein Benutzer startet, wird auch als Login-Shell bezeichnet. Login-Shells werden direkt nach dem Anmelden eines Benutzers an einem Terminal gestartet. Man kann die Bash auch explizit als Login-Shell aufrufen.

Aufruf: `bash --login` oder `bash -`

Login-Shells werden durch das Kommando `logout` beendet. Nicht-Login-Shells, die interaktiv ausgeführt werden (Aufruf einfach durch das Kommando `bash`) können nicht mit dem Kommando `logout` beendet werden. Man verwendet hier das Kommando `exit`. In den Startup-Dateien kann man auch ganze Shell-Skripte, also Folgen von Kommandos eintragen. Mit Shell-Skripten werden wir uns später noch befassen.

Startup-Dateien müssen Leserechte besitzen. Die erste gefundene Datei mit Leserecht wird jeweils ausgeführt, die folgenden nicht mehr.

Bei Aufruf einer Login-Shell werden die sog. profile-Dateien in folgender Reihenfolge ausgeführt:

- `/etc/profile` (systemweit)
- `~/.bash_profile` (benutzerspezifisch)
- `~/.bash_login` (benutzerspezifisch)
- `~/.profile` (benutzerspezifisch)

Die `.profile`-Dateien werden nur ausgeführt wenn eine Login-Shell geöffnet wird. Die Änderung der Datei `/etc/.profile` erfordert entsprechende Administratorrechte (man sagt auch root-Berechtigung).

Beim Starten einer Shell, die nicht Login-Shell ist, werden folgende Dateien ausgeführt:

- `/etc/bashrc` (systemweit)
- `~/.bashrc` (benutzerspezifisch)

Das Verändern der Datei `/etc/bash.bashrc` erfordert entsprechende Rechte.

Beispiel:

Will man beispielsweise einen Begrüßungstext bei jedem Start der Bash ausführen, so kann man in die Datei `~/.bashrc` z. B. die Zeile

```
echo „Hallo Linux-User“
```

## 4 Linux-Berechtigungen

---

einfügen. Startet man dann eine Bash erscheint in der ersten Zeile der definierte Text.

Die Ausführung der `bashrc`-Dateien erfolgt bei jedem Start eines Terminals oder einer Konsole, außer bei Login-Shell.

Lassen sie sich dazu weiterführende Informationen anzeigen, indem Sie über die Kommandozeile die Manual-Page der Bash aufrufen: `man bash` (unter dem Punkt „Invocation“).

### Exkurs:

Andere Shells nutzen auch Startup-Dateien bei Login oder beim Starten einer neuen Shell. Sollte die Bash nicht die Standard-Shell in Ihrem System sein und Sie wollen gleich nach dem Login automatisch in die Bash wechseln, können Sie das durch Eintragen des Kommandos `bash` in die entsprechende Startup-Datei erreichen. Wenn z. B. `tcsh` Ihre standardmäßig eingestellte Shell ist, wird zunächst `$HOME/.tcshrc` aufgerufen und danach `$HOME/.login`. Sie können dann den Befehl `bash` in die Datei `$HOME/.login` eintragen und es erfolgt beim Loginvorgang gleich ein Wechsel in die Bash.



## 4.4 Übungen

### 4.4.1 Übung zum Berechtigungskonzept

Erzeugen Sie mit dem `touch`-Befehl eine neue Datei in Ihrem Home-Verzeichnis und lassen Sie sich anschließend mit Hilfe des `ls`-Befehl die Berechtigungsinformationen dazu anzeigen.

---

---

---

---

Ändern sie nun die Berechtigung für diese Datei auf `r--r--r--`. Versuchen Sie anschließend den Inhalt der Datei zu ändern (am einfachsten über einen Editor). Was passiert und warum?

---

---

---

---

---

---

---

---

## 4 Linux-Berechtigungen

---

Füllen Sie in der folgenden Tabelle die fehlenden Lücken aus!

| Befehl                     | binär                    | Anzeige bei <code>ls -l</code> |
|----------------------------|--------------------------|--------------------------------|
| <code>chmod 440 dat</code> |                          |                                |
|                            |                          | <code>r-xrwx--x</code>         |
| <code>chmod 027 dat</code> | <code>000 010 111</code> |                                |
|                            | <code>010 010 000</code> |                                |

### 4.4.2 Übung mit `umask`

Setzen Sie die Berechtigungsmaske mit dem `umask`-Kommando so, dass der Eigentümer lesen und schreiben darf, die Gruppe nur lesen und anderen gar nichts erlaubt ist.

Erzeugen Sie nun eine beliebige neue Datei und betrachten Sie deren Zugriffsrechte mit dem `ls`-Kommando.

---

---

---

---

---

---

### 4.4.3 Übung `.bashrc` verändern

Ändern Sie nun die Datei `.bashrc` in Ihrem Home-Verzeichnis so ab, dass bei jedem Start eines Terminals ein Begrüßungstext und der Inhalt Ihres Home-Verzeichnisses ausgegeben wird.

---

---

---

---

---

---

## 5 Linux-Internas

### Zielsetzung des Kapitels

Der Studierende soll am Ende des Kapitels die wichtigsten Linux Prozesse kennen und beschreiben können. Der Studierende soll einen klassischen Bootvorgang in gängigen Linux-Distributionen nachvollziehen können.

### Wichtige Begriffe

systemd, initd, Dämon, Booten, Bootmanager.

### 5.1 Linux-Bootvorgang

Vor dem eigentlichen Startvorgang wird beim Einschalten eines Rechners die Hardware (CPU, BIOS-ROM, DMA-Controller, Tastatur, ...) initialisiert. Als nächstes erfolgen der Test und die Initialisierung von System-Erweiterungen (RAM, Schnittstellen, Festplatten-Controller). Wenn diese Tests erfolgreich abgelaufen sind, dann sucht das BIOS bzw. die neuere BIOS-Version UEFI auf den Bootgeräten nach einer gültigen Bootsequenz.

Etwas vereinfacht dargestellt lädt das BIOS/UEFI den MBR (*Master Boot Record*) des ersten eingetragenen Bootmediums. Der MBR enthält neben der Partitionstabelle ein Programm, das die Partitionstabelle auswertet. Ist eine dieser Partitionen mit einem „bootable Flag“ markiert, wird deren Bootsektor angesprungen und der dort enthaltene Code ausgeführt.

Ein Bootmanager wie etwa GRUB<sup>5</sup> wird benötigt, wenn mehrere Betriebssysteme auf einem Rechner installiert sind. Der Benutzer kann damit beim Booten wählen, welcher Bootvorgang ausgeführt wird (z. B. Windows-Partition oder Linux). Die Initialisierung aller Kernelteile (virtuelle Speicherverwaltung, Interruptroutinen, Zeitgeber, Scheduler, Dateisystem, Ressourcen der Interprozesskommunikation) erfolgt im Anschluss.

---

<sup>5</sup> Siehe [https://de.wikipedia.org/wiki/Master\\_Boot\\_Record#GRUB](https://de.wikipedia.org/wiki/Master_Boot_Record#GRUB). Zuletzt gelesen am 09.03.2018.

Abbildung 5-1 zeigt den Bootvorgang im Gesamten, beginnend mit dem Bootloader bis zum Start eines Terminals bzw. je nach Konfiguration des Systems einer grafischen Oberfläche.

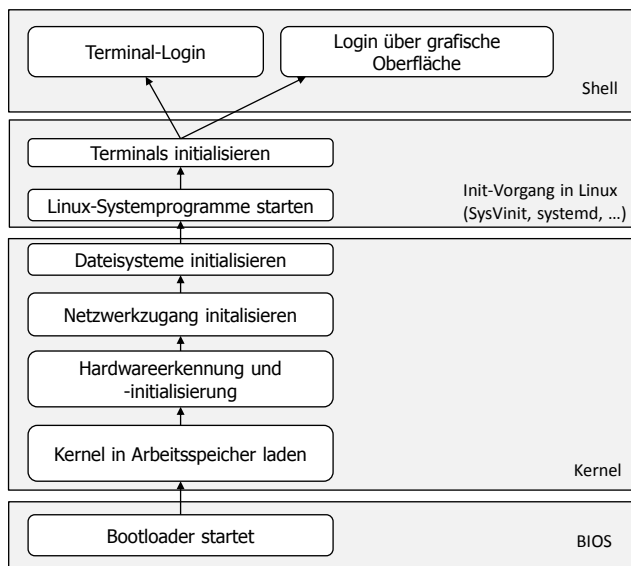


Abbildung 5-1: Linux-Bootvorgang<sup>6</sup>

### 5.2 Init-System

Bei Linux wird das System stufenweise hochgefahren. Nach dem Booten des Systems wird im Betriebssystem Zug um Zug der Start konfigurierter Programme (Dienste) vorgenommen. Unter Linux bzw. Unix gibt es hierfür verschiedene Mechanismen:

- SysVinit (Init-V): Ältere Unix-Variante, Ursprung: System V, auch unter Linux häufig eingesetzt, z. B. in Cent OS bis V6 genutzt
- Service Management Facility (SMF) von Solaris
- systemd: Neue Lösung, auch ab CentOS V7 und Debian 8
- upstart: Eingesetzt bei der Linux-Distribution Ubuntu, wird aber abgelöst durch systemd
- launchd bei OS X und iOS

---

<sup>6</sup> Quelle: <http://de.wikipedia.org/wiki/Booten#/media/File:Linux-bootvorgang.svg>.

Die verschiedenen Linux-/Unix-Distributionen unterscheiden sich zum Teil deutlich. Pfade und Dateien in dieser Übung beziehen sich auf die CentOS-Distribution, die auf den Hochschul-Rechnern installiert ist. Für andere Unixe/Linuxer wie beispielsweise OS X, Ubuntu oder Knoppix sind die Pfade nur bedingt anwendbar.

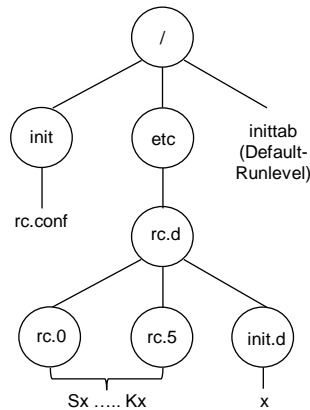
### 5.2.1 SysVinit

Bei SysVinit ruft der init-Prozess mit der PID 1 abhängig von der Kernelkonfiguration erste Dämon-Prozesse ins Leben. Außerdem ist der init-Prozess der Ursprungsprozess aller weiteren Prozesse. Für den Startup können verschiedene Runlevel konfiguriert werden. Runlevels legen grob gesagt fest, welche Programme automatisch gestartet werden.

Für den SysVinit-Startup-Vorgang (skizziert in Abbildung 5-1) sind folgende Verzeichnisse und Dateien relevant:

- Die Runlevels sind in der Datei `/etc/inittab` definiert.
- Ein Runlevel wird durch die Ausführung der entsprechenden `rc`-Datei initialisiert. Es handelt sich hierbei um Skript-Dateien, in der Start-Anweisungen für eine Anwendung oder für das Starten von Systembestandteilen hinterlegt sind.
- In der Datei `rc.conf` ist festgelegt für welches Runlevel welche Dienste gestartet werden sollen.
- Die Start-Skripte für die zu startenden Dienste liegen unter dem Verzeichnis `/etc/rc.d`. Für jedes Runlevel gibt es hier ein entsprechendes Verzeichnis (`rc.0` bis `rc.5`), in dem wiederum die einzelnen Start- und auch die Stop-Skripte für die Dienste enthalten sind, genauer gesagt sind es nur Links. Die eigentlichen Start-/Stop-Skripte sind im Verzeichnis `/etc/init.d` abgelegt.
- Die Links zum Starten beginnen mit „S“, die Links zum Stoppen mit „K“ (Kill). Start- als auch Stop-Skript nutzen oft dasselbe Skript. Ob ein Starten oder Stoppen initiiert werden soll, wird über einen Aufrufparameter des Skripts gesteuert.

Ist beispielsweise in der Datei `/etc/inittab` das Runlevel auf 3 eingestellt, werden beim Systemstart alle Skripte gestartet, die in den Verzeichnissen `/etc/rc.0`, `/etc/rc.1`, `etc/rc.2` und `etc/rc.3` enthalten sind. Der Superuser ist für die Verwaltung dieser Skripte verantwortlich. Standardmäßig wird bei einem Desktop-Linux die graphische Oberfläche gestartet, was im Runlevel 5 erfolgt. Das Runlevel 0 dient zum Beenden (Shutdown) des Systems, Runlevel 1 ist meist mit dem Einbenutzerbetrieb verbunden, bei Runlevel 2 ist man im Mehrbenutzerbetrieb und mit Runlevel 3 wird der Netzwerkzugang des Systems initialisiert.



**Abbildung 5-2: Für SysVinit relevante Dateien**

Es sollte beachtet werden, dass die Runlevel-Vergabe bei den verschiedenen Linux-Distributionen variieren kann. Am besten verschafft man sich selbst im System einen Überblick, indem man die oben skizzierten Verzeichnisse und Dateien inspiziert.

Mit dem Kommando `telinit` kann man den Runlevel einstellen, mit dem Befehl `who -r` den aktuellen Runlevel ausgeben.

```
# Runlevel anzeigen
who -r
# Runlevel verändern
telinit 5
```

### 5.2.2 Systemd

Ein neues Startup-System unter Linux verbreitet sich immer mehr. Es ist wesentlich flexibler und mächtiger, aber auch komplexer. Es wird als *systemd* bezeichnet. Ein Einstieg für die Einarbeitung in die Arbeitsweise und Architektur von *systemd* ist in (wikipedia 2016c) zu finden.

### 5.2.3 Exkurs: Startup mit launchd unter OS X

Unter OS X wird anstelle von `initd` oder *systemd* der Prozess *launchd* als Elternprozess für alle weiteren Prozesse gestartet. Weitere Ausführungen hierzu finden Sie unter (Levin 2013).

## 5.3 Einige Kommandos für Prozesse

Um einen Überblick über aktuell laufende Prozesse zu erhalten, gibt es einige Kommandos, die man unter Linux nutzen kann (Kommandos `ps`, `ps tree`, `top`).

Ein Prozess kann einem anderen Prozess sog. Signale zukommen lassen, womit sich beispielsweise Prozesse auch beenden lassen. Hierzu dient das Kommando `kill`, das wir ebenfalls erläutert.

### 5.3.1 Kommando `ps`

| ps (process status) |   |
|---------------------|---|
| Beschreibung        | Zeigt die Prozesse mit ihrem Status an.   |
| Syntax              | <code>ps &lt;optionen&gt;</code><br>Optionen können u.a. sein:<br>l = langes Format<br>a = alle Prozesse, auch die anderer User<br>x = Auch Dämon-Prozesse anzeigen<br>f = Darstellung des Prozessbaums |
| Beispiele           | # Nur Prozesse, denen gerade eine CPU zugeordnet ist,<br># anzeigen<br><code>ps r</code><br># Oft verwendete Nutzung<br><code>ps aux</code>   |

Sollte die Ausgabe des `ps`-Befehls oder auch eines anderen Befehls länger als eine Seite sein (lange unübersichtliche Ausgaben), kann ein Blättermodus über das Kommando `more` genutzt werden. Dazu muss z. B. der `ps`-Befehl mit dem `more`-Befehl über eine Pipe verknüpft werden:

```
ps aux | more
```

Kurze Erläuterung der wichtigsten Ausgaben des `ps`-Kommandos:

| PID        | TTY                                 | STAT   | TIME                      | CMD          | ... |
|------------|-------------------------------------|--------|---------------------------|--------------|-----|
| Prozess-ID | Bezeichnung des aktuellen Terminals | Status | Bisher benötigte CPU-Zeit | Programmname |     |

Im Feld Status, das den aktuellen Prozesszustand wiedergibt, werden einige Abkürzungen verwendet, u.a:

- S = sleeping (Prozess wartet auf eine CPU)
- R = running (Prozess ist aktiv, CPU zugeordnet)
- SW = sleeping und ausgelagert, d. h. der Prozess ist nicht im Hauptspeicher (siehe Hauptspeicherverwaltung (Mandl 2015))
- T = Traced / Stopped
- Z = Zombie (Prozess, der während des Beendens noch auf das Abholen seines Exit-Status wartet)

### 5.3.2 Kommando `pstree`

| pstree       |   |
|--------------|---|
| Beschreibung | Ausgabe der aktuell laufenden Prozesse als Baumdarstellung.   |
| Syntax       | <p><code>pstree &lt;optionen&gt; [&lt;pid&gt; &lt;user&gt;]</code></p> <p>Optionen sollen nicht weiter betrachtet werden.</p> <p>Mit <code>&lt;pid&gt;</code> kann ein Prozess angegeben werden, ab dem der Baum angezeigt wird. Wird nichts angegeben, beginnt die Ausgabe beim Prozess mit der PID 1 (z. B. init-Prozess, je nach Startup-Mechanismus).</p> <p>Mit <code>&lt;user&gt;</code> wird ein Benutzer angegeben. In diesem Fall werden nur die Prozesse des angegebenen Benutzers angezeigt.</p> |
| Beispiele    | <pre>pstree pstree 4120 pstree mandl</pre>  |



### 5.3.3 Kommando top

| top          |   |
|--------------|---|
| Beschreibung | <p>Das Kommando gibt eine Liste aller Prozesse, die gerade im System sind, aus und aktualisiert diese periodisch. Die Ausgabe ist sehr umfassend und beinhaltet neben der Prozess-Identifikation (PID), der genutzten CPU-Zeit durch die Prozesse auch Speicherverwaltungsinformationen und einiges mehr.</p> <p>Das Kommando kann auch interaktiv genutzt werden, es nimmt verschiedene Kommandos während der Laufzeit entgegen und wird mit dem Kommando „q“ (quit) beendet. Für unsere Zwecke soll die einfache Ausgabe einer Prozessliste ausreichen.</p> |
| Syntax       | <code>top &lt;optionen&gt;</code>   |
| Beispiele    | <pre>top # Anzeige der Prozesse mit den PIDs 2030, # 1, 2 und 3 top -p 2030 -p 1 - p 2 -p 3</pre>   |

### 5.3.4 Kommando kill

| kill         |  |
|--------------|--|
| Beschreibung | <p>Das Kommando dient dazu, einem Prozess ein Signal zu senden, auf das er in einer Signalaroutine (sofern im Programmcode vorgesehen) reagieren kann. Gibt es keine entsprechende Signalaroutine im Programm, wird der Prozess terminiert. Die meisten Signale können von Prozessen auch explizit über den Aufruf einer entsprechenden Systemfunktion ignoriert werden.</p> <p>Signale werden unter Linux verwendet, um zwischen Prozessen bestimmte Ereignisse anzuzeigen (zu signalisieren).</p> <p>Eine Liste aller Signale kann mit dem Kommando <code>kill -l</code> angezeigt werden. Deren Bedeutung kann über die durch das Kommando ausgegebenen Bezeichnungen im Internet ermittelt werden.</p> |
| Syntax       | <code>kill [-&lt;signal&gt;] &lt;PID&gt;</code>  |
| Beispiele    | <pre># Beenden des Prozesses mit der PID 1234, standardmäßig # wird das Signal mit der Nummer 15 gesendet. kill 1234  # Beenden des Prozesses mit der PID 1234, indem das # Signal 9 (auch symbolisch mit „KILL“ bezeichnet) # gesendet wird. kill -9 1234  oder kill -KILL 1234  # Liste aller Signale ausgeben. kill -l</pre>  |

## 5.4 Hintergrundprozesse und Dämonen

Normalerweise wird ein Kommando (Programm) beim Start immer dem Terminal zugeordnet, in dem es aufgerufen wird. Unter einem Dämon bzw. Dämon-Prozess versteht man einen Prozess, der ein Programm im Hintergrund ohne eine Terminalzuordnung ausführt. Meist stellen Dämon-Prozesse definierte Dienste zur Verfügung. Beispielsweise startet der Dämon-Prozess `cron` andere Programme bzw. Skripte zu festgelegten Zeiten. Andere Dämonen unter Linux sind z. B. `httpd` (Webserver) und `syslogd` (zentraler Logserver).

Ein Prozess kann als Hintergrundprozess ablaufen, indem beim Start das Zeichen „&“ angehängt wird:

```
WC &
```

Das Kommando `WC` wird bei diesem Beispiel als Hintergrund-Prozess gestartet. In diesem Fall ist der Prozess aber noch mit dem Terminal verbunden. Beendet man die Login-Shell, wird auch dieser Hintergrundprozess beendet. Möchte man das nicht, muss man den Prozess zum Dämon-Prozess machen. Um einen Prozess zum Dämon-Prozess zu machen, d. h. ihn vom Terminal abzuhängen, benötigt man das Kommando `nohup`. In der Spalte TTY (Terminalzuordnung) bei der Ausgabe des `ps`-Befehls steht bei Dämon-Prozessen „?” oder „??“, jedoch nicht in jedem Linux-Derivat und auch nicht bei OS X.

**Exkurs.** Genauer gesagt wird mit Hilfe des Kommandos `nohup` für einen Prozess das Hangup-Signal (auch mit `SIGHUP` bezeichnet) ignoriert. Das Signal wird normalerweise beim Beenden einer Login-Shell vom System an alle Prozesse gesendet, die in der gleichen Session (in einer Login-Shell) erzeugt wurden. Damit werden sie über das Logout-Ereignis informiert und üblicherweise beendet. Ignoriert ein Prozess das Signal mit `nohup`, wird der entsprechende Prozess nicht terminiert.

Wenn ein Prozess, der als Dämon laufen soll, noch mit der Standardeingabe, Standardausgabe oder dem Standardfehlerkanal verbunden ist, beendet er sich beim Logout trotzdem. In diesem Fall müsste man beim Starten des Prozesses alle benutzten Standardkanäle auf Dateien umlegen. Beispielsweise kann ein Kommando `WC`, das eine Eingabe vom Standardeingabekanal erwartet, nicht ohne Terminal arbeiten. Wenn die Standardeingabe beim Start nicht aus einer Datei kommt, terminiert der `WC`-Prozess spätestens beim Logout.

| nohup (no hang up) |  |
|--------------------|--|
| Beschreibung       | Startet ein Kommando als Dämon-Prozess oder macht einen laufenden Prozess zum Dämon-Prozess. Die Standardausgabe des Kommandos wird auf eine Datei <code>nohup.out</code> im Verzeichnis, in dem der Prozess gestartet wurde, geschrieben (hinten angehängt). Kann die Datei nicht im aktuellen Verzeichnis erzeugt werden, so wird sie im Heimatverzeichnis angelegt (es wird dabei die Umgebungsvariable <code>HOME</code> abgefragt).   |
| Syntax             | <code>nohup &lt;kommando&gt; &lt;argumente&gt;</code>  |
| Beispiele          | <pre># Start des Kommandos ps in einem Dämon-Prozess, # die Ausgabe wird auf die Datei nohup.out umgelenkt. nohup ps  # wie vorher, nur im Hintergrund. nohup ps&amp;  # Starten des Kommandos sleep im Hintergrund, # anschließend wird der Prozess zum Dämon gemacht. sleep &amp; 1234 (Prozess-Identifikation = PID) nohup 1234  # Mit sleep wird eine bestimmte Zeit in Sekunden # gewartet. Dies wird hier im Hintergrund in einem # Dämonprozess durchgeführt. nohup sleep 10&amp;  Done          nohup sleep 10</pre> |

## 5.5 Übungen

### 5.5.1 Übung mit ps

Lassen Sie sich die aktuell laufenden Prozesse über den Befehl `ps` anzeigen. Versuchen Sie die Ausgaben zu erklären. Variieren Sie dabei die Ausgabe über Kommando-Optionen. Notieren Sie sich interessante Aspekte.

---

---

---

---

---

---

---

---

### 5.5.2 Übung mit top und pstree

Zeigen Sie die aktuell laufenden Prozesse mit dem `top`-Kommando und den Prozessbaum mit dem Kommando `pstree` an. Versuchen Sie die Ausgaben mit Hilfe von Web-Recherchen zu verstehen und notieren Sie sich interessante Aspekte.

Hinweis: `pstree` gibt es nicht unter OS X. (`pstree` kann über HomeBrew, Macports oder Fink nachinstalliert werden.)

---

---

---

---

---

---

---

---

---

---

### 5.5.3 Übung mit Hintergrundprozess

Starten Sie das Kommando `sleep 1000` (wartet 1000 Sekunden lang und beendet sich anschließend) als Dämonprozess (Prozess ohne zugeordnetes Terminal).

Versuchen Sie danach herauszufinden, ob der gestartete Dämonprozess tatsächlich noch läuft, auch wenn Sie das Terminal, mit dem Sie den Prozess erzeugt haben, beenden und anschließend in einem neuen Terminal innerhalb von ca. 1000 Sekunden mit dem Kommando `ps aux | grep sleep` prüfen, ob der Prozess noch da ist. Woran erkennen Sie an der Ausgabe, dass es ein Dämonprozess ist?

---

---

---

---

---

### 5.5.4 Übung zum Beenden eines Prozesses

Starten Sie eine neue Bash und beenden Sie anschließend den Bash-Prozess aus einem anderen Terminal heraus.

---

---

---

---

---

Öffnen Sie ein Terminal und geben Sie den Befehl `wc` ein. Öffnen Sie ein zweites Terminal und versuchen Sie herauszubekommen, welche PID der `wc`-Prozess besitzt. Beenden Sie nun das `wc`-Programm des anderen Terminals mit Hilfe des `kill`-Kommandos.

---

---

---

---

---

## 6 Grundlagen der Bash-Programmierung

### Zielsetzung des Kapitels

Der Studierende soll am Ende dieses Kapitels in der Lage sein, den Aufbau und die Funktionsweise eines Bash-Skripts und die wichtigsten Kontrollstrukturen zu erklären. Weiterhin sollte er ein Bash-Skript parametrisieren können.

### Wichtige Begriffe

Schleifen, while, for, Bedingungen, test, if, then, else, Parameter, Übergabeparameter, Funktion, globale Variable, lokale Variable.

### 6.1 Aufbau und Aufruf eines Bash-Skripts

Um den Quelltext zu erstellen, ist ein einfacher Texteditor wie `vi`, `nano` oder `emacs` ausreichend. Bedienungsanleitungen für diese Editoren finden Sie im Anhang oder auf verschiedenen Webseiten.<sup>7</sup>

Nachfolgend ist ein sehr einfaches Skript zu sehen:

```
#!/bin/bash
echo "Servus Linux!"
```

Ganz am Anfang eines Skripts wird die Shell spezifiziert, mit der die weiteren Anweisungen interpretiert werden soll. Die Zeichenkombination `#!` wird auch als Shebang oder Magic Line bezeichnet. Die Bezeichnung stammt wohl aus der Abkürzung für die beiden Anfangszeichen „sharp bang“ oder „hash bang“, denn unter Unix bzw. Linux bezeichnet man das Ausrufezeichen auch mit „bang“ und das Doppelkreuz als hash oder sharp. Linux übergibt die Kommandofolge in der Datei dann dem genannten Kommandointerpreter `bash`. Man könnte auch einen anderen Interpreter, wie z. B. `ksh` angeben.

In der zweiten Zeile wird dann ein Text ausgegeben. Hat man das Skript erstellt und in einer Datei abgespeichert, so muss man diese Datei mit dem Befehl `chmod` als ausführbar (Modus: `+x`) markieren.

---

<sup>7</sup> Ein möglicher Suchbegriff bei Google wäre „vi editor Übersicht“.

Anschließend wechseln Sie in das Verzeichnis, in der Sie die Datei etwa unter dem Namen `servus.sh` abgespeichert haben und führen diese mit dem Befehl

```
./servus.sh
```

aus. In diesem Fall wird das Bash-Skript in einer eigenen Shell, die als Subshell bezeichnet wird, gestartet. Die Subshell läuft in einem eigenen Linux-Prozess und erbt die Umgebung der aufrufenden Shell. Alle exportierten Umgebungsvariablen werden als Kopien an die neue Shell übergeben. Ein Verändern von Umgebungsvariablen der rufenden Shell ist nicht möglich.

Möchte man ein Bash-Skript in der Shell laufen lassen, in welcher der Aufruf getätigt wird, benutzt man eine andere Notation, die als Punkt-Kommando bezeichnet wird. Das Punkt-Kommando ist ein Build-In-Kommando der Bash. Nutzt man es, werden keine Subshell und auch kein eigener Linux-Prozess erzeugt. Das aufgerufene Bash-Skript hat Zugriff auf alle Umgebungsvariablen der aufrufenden Shell und kann diese auch verändern. Der Aufruf sieht dann allgemein wie folgt aus:

```
. ./<dateiname> <parameter1> <parameter2>
```

Beispielaufruf: `. ./compare Linux Linux`

Das Punkt-Kommando kann man zum Beispiel sehr gut nutzen, um Umgebungsvariablen initial durch ein spezielles Bash-Skript zu setzen. Bei Nutzung des Punkt-Kommandos muss das aufgerufene Bash-Skript auch nicht das Ausführungsrecht besitzen.

**Vorsicht.** Achten Sie bei Verwendung von Copy&Paste aus dem Skriptum in die Linux-Kommandozeile darauf, dass damit auch nicht sichtbare Sonderzeichen übertragen werden können, die der Bash-Interpreter nicht interpretieren kann. Also lieber abtippen!

## 6.2 Übergabeparameter und Rückgabewerte

Für gewöhnlich möchte man ein Bash-Skript durch Benutzereingaben parametrisieren. Dies erfolgt über Argumente (Parameter), die beim Aufruf an das Bash-Skript übergeben werden. Im Bash-Skript werden die Parameter durch Variablen repräsentiert. Die Parameter werden als Stellungsparameter einfach durchgezählt. Bei Ihrer Verwendung muss das „\$“ vorangestellt werden.

Manchmal möchte man wissen, ob ein Bash-Skript oder ein Programm oder ein Kommando erfolgreich abgeschlossen oder ob es vorher unerwartet beendet wurde. Hierzu liefert jedes Bash-Skript automatisch einen sog. *Exit-Code*. Ebenso ist es



manchmal in einem zu erstellenden Skript hilfreich ist, die Anzahl der übergebenen Parameter zu ermitteln. Auch dies lässt sich abfragen.

Die folgende Tabelle zeigt einige Standard-Variablenamen, die hierfür verwendet werden können:

| Variablen-name       | Bedeutung  |
|----------------------|--|
| <code>\$#</code>     | Anzahl der übergebenen Argumente. Kann z. B. in einem Skript verwendet werden, um zu prüfen, ob alle Parameter vorhanden sind.   |
| <code>\$0</code>     | Name des ausgeführten Skripts  |
| <code>\$1</code>     | Wert des 1. übergebenen Parameters   |
| <code>\$2 ...</code> | Wert des 2. übergebenen Parameters ...   |
| <code>\$?</code>     | Gibt den Exit-Code des zuletzt ausgeführten Skripts bzw. Programms aus. Ein Exit-Code <code>!= 0</code> deutet meist auf einen Fehler hin, allerdings legt dies der Programmierer des Skripts bzw. des Programms fest.<br><br>Der Exit-Code wird mit dem Befehl <code>exit</code> vom gerufenen Programm/Skript zurückgegeben. |

Beim Aufruf eines Bash-Skripts werden die Parameter einfach durch Leerzeichen getrennt übergeben. Bei mehr als neun Parameter müssen die Variablen im Bash-Skript mit den Bezeichnern „`${10}`“, „`${11}`“ usw. angesprochen werden.

### 6.3 Bedingungen

| test bzw. [ ... ] |   |
|-------------------|---|
| Beschreibung      | Überprüft Dateien/Verzeichnisse und vergleicht Werte.   |
| Syntax            | <pre>test &lt;expression&gt; oder [ _&lt;expression&gt;_ ]</pre> <p><code>&lt;expression&gt;</code> ist hier ein logischer Ausdruck, der sich wieder aus mehreren Ausdrücken zusammensetzen kann.<br/>Beispiele:<br/>String-Vergleich</p> |

|  |   |
|--|---|
|  | <p><code>&lt;string1&gt; = &lt;string2&gt;</code> oder<br/><code>&lt;string1&gt; == &lt;string2&gt;</code></p> <p>Überprüfung, ob eine Datei existiert<br/><code>-e &lt;file&gt;</code></p> <p>Überprüfung, ob eine Datei existiert<br/><code>-a &lt;file&gt;</code></p> <p>Überprüfung, ob ein Verzeichnis existiert<br/><code>-d &lt;file&gt;</code></p> <p>Wahr, wenn String leer ist<br/><code>-z &lt;string&gt;</code></p> <p>Wahr, wenn String nicht leer ist<br/><code>-n &lt;string&gt;</code></p> <p>Wahr, wenn beide Ausdrücke wahr sind (log. Und)<br/><code>test &lt;expression1&gt; -a &lt;expression1&gt;</code></p> <p>Alternative:<br/><code>[_&lt;expression1&gt;_] &amp;&amp; [_&lt;expression2&gt;_]</code></p> <p>Wahr, wenn einer der beiden Ausdrücke wahr ist (log. Oder)<br/><code>test &lt;expression1&gt; -o &lt;expression2&gt;</code></p> <p>Alternative:<br/><code>[_&lt;expression1&gt;_]    [_&lt;expression2&gt;_]</code></p> <p>Weitere Vergleichsoperatoren:</p> <ul style="list-style-type: none"><li><code>-eq</code> (Prüfung auf numerische Gleichheit)</li><li><code>-ne</code> (Prüfung auf numerische Ungleichheit)</li><li><code>-le</code> (Prüfung auf kleiner oder gleich, numerisch)</li><li><code>-ge</code> (Prüfung auf größer oder gleich, numerisch)</li><li><code>-gt</code> (Prüfung auf größer, numerisch)</li><li><code>-lt</code> (Prüfung auf kleiner, numerisch)</li></ul> <p>Weitere Möglichkeiten siehe Handbuchseiten (man-Kommando)</p> |
|--|---|

|           |   |
|-----------|---|
| Beispiele | <pre> # Prüfung, ob die Datei test.sh vorhanden ist und # Ausgabe einer entsprechenden Meldung test -e test.sh &amp;&amp; echo "Datei vorhanden"    echo "Datei nicht vorhanden"  # Alternative Schreibweise [_-e test.sh_] &amp;&amp; echo „Datei vorhanden“    echo „Datei nicht vorhanden“  # Prüfung der Variablen i auf kleiner als 10 [_\$i -lt 10_] oder test \$i -lt 10  # Prüfung zweier mit log. Und verknüpfter Ausdrücke [_\$A == \$B_] &amp;&amp; [_\$C == \$B_] oder test \$A -eq \$B -a \$C -eq \$B </pre> |
|-----------|---|

Wie im letzten Beispiel ersichtlich wird, kann man statt des `test`-Befehls auch eckige Klammern (`[` und `]`) verwenden. Bei dieser Verwendung muss jedoch auf die Leerzeichen nach und vor der eckigen Klammer geachtet werden.

## 6.4 if-Anweisung

| if / then / else |  |
|------------------|--|
| Beschreibung     | Verzweigungsmöglichkeit anhand einer Bedingung.  |
| Syntax           | <pre> if [_&lt;bedingung&gt; _]; then     &lt;code&gt; else     &lt;code&gt; fi </pre> |
| Beispiele        | <pre> # Prüft, ob Verzeichnis existiert if !_[_-d '/tmp' _]; </pre>                    |

|  |  |
|--|--|
|  | <pre> then     echo "/tmp existiert nicht"; else     echo "/tmp exists"; fi </pre> |
|--|--|

## 6.5 for-Schleife

| for-Schleife |  |
|--------------|--|
| Beschreibung | Kopfgesteuerte Schleife.   |
| Syntax       | for <bedingung> do <code> done   |
| Beispiel     | <pre> # Zählt in einer Schleife einen Zähler hoch, solange der # Zähler kleiner 10 ist. for ((i = 0; i &lt; 10; i++)); do echo \$i; done  # In einem Bash-Skript würde man besser so schreiben: for ((i = 0; i &lt; 10; i++)); do     echo \$i; done  # Alternative for i in {0..10}; do echo \$i; done </pre> |

## 6.6 while-Schleife

| while-Schleife |   |
|----------------|---|
| Beschreibung   | Kopfgesteuerte Schleife.  |
| Syntax         | while <bedingung>; do <code>; done  |
| Beispiel       | <pre> # Zählt in einer Schleife einen Zähler hoch, solange der # Zähler kleiner 10 ist. i=0 while ((i &lt; 10)); </pre> |

```
do
    echo $i;
    ((i += 1));
done
```

## 6.7 Bash-Funktionen

Bash-Funktionen dienen der Strukturierung des Programms. Mehrfach nutzbarer Code kann damit nur einmal geschrieben und wiederverwendet werden, wie dies bei Funktionen und Methoden anderer Programmiersprachen der Fall ist.

Eine Bash-Funktion muss im Skript vor ihrer ersten Nutzung deklariert werden. Die Deklaration sieht allgemein wie folgt aus:

```
function <funktionsname>()
{
    <kommando1>;
    <kommando2>;
    ...
}
```

Bash-Funktionen erhalten keinen Typ und auch die Parameter sind nicht genauer festgelegt. Die Parameter werden nicht explizit deklariert, sondern sind Stellungsparameter, die beim Aufruf übergeben werden.

Beispiel:

```
function f1() {
    echo "Die Funktion f1 wurde aufgerufen"
    echo "Uebergabener Parameter: $1"
    if [ "$1" == "1" ] # auch "=" als Vergleichsoperator zulässig
    then
        # Funktion mit Exit-Status = 1 verlassen
        return 1
    else
        # Funktion mit Exit-Status = 0 verlassen
        return 0
    fi
}
```

## 6 Grundlagen der Bash-Programmierung

```
fi  
}
```

Hier wird eine Funktion mit dem Namen `f1` deklariert. Aufgerufen wird sie im Skript zum Beispiel wie folgt:

```
f1 "1"
```

Ein anderer Aufruf könnte beispielsweise in einer `if`-Anweisung wie folgt durchgeführt werden:

```
if f1 "1"  
then  
...  
fi
```

Es können beliebig viele Stellungsparameter übergeben werden. Diese werden in der Funktion mit den Bezeichnern `$1`, `$2`, usw. verwendet. Die Parameter beziehen sich also innerhalb der Funktion nicht mehr auf das übergeordnete Bash-Skript, sondern gelten lokal nur für die Funktion. Das gilt auch für `$*`, `$#` usw. Nur der Parameter `$0` behält den Namen des Skripts.

Die Rückgabe eines Wertes erfolgt mit der Anweisung `return`.

Mit dem Kommando `typeset` kann man alle deklarierten Funktionen auflisten lassen.

| typeset      |  |
|--------------|--|
| Beschreibung | Zeigt die im Prozess bekannten Funktionen an   |
| Syntax       | <pre>typeset &lt;optionen&gt;</pre> <p>Optionen können u.a. sein:</p> <p><code>f</code> = alle bekannten funktionen einschließlich des gesamten Programmcodes</p> <p><code>F</code> = nur die Definitionen alle bekannten Funktionen</p> |
| Beispiele    | <pre># Alle Funktionen auflisten<br/>typeset -f</pre> <pre># Definitionen aller Funktionen auflisten<br/>typeset -F</pre>  |

## 6.8 Globale und lokale Variablen

In der Bash kann man innerhalb von Funktionen lokale Variablen definieren. Hierzu muss man der Variablen bei der Definition lediglich das Schlüsselwort `local` voranstellen.

```
local var=wert
```

Eine so definierte Variable hat dann eine Lebensdauer, die nach dem Funktionsaufruf wieder endet. Eine globale Variable mit demselben Namen kann trotzdem verwendet werden.

Beispiel:

```
# Globale Variable
var="global"
Local_function() {
# Lokale Variable mit gleichem Namen als globale Variable
    local var="lokal"
    echo $var
}
# Hauptfunktion
echo $var
Local_function
echo $var
```

Bei der Ausführung des Skripts werden folgende Ausgaben getätigt:

```
you@host > ./myscript
global
lokal
global
```

## 6.9 Übungen

### 6.9.1 Übung: Erstes Bash-Skript

Legen Sie eine Datei mit dem einfachen Beispiel-Skript von oben ("Servus Linux!") als Inhalt an und bringen Sie es zum Ablauf. Notieren Sie die erforderlichen Aktionen, um das Programm zum Ablauf zu bringen.

---

---

---

---

---

---

---

---

### 6.9.2 Übung zur Parameterübergabe

Schreiben Sie ein einfaches Bash-Skript, in dem Sie zwei Parameter übergeben und im Skript auf dem Bildschirm ausgeben. Bringen Sie das Skript zum Laufen.

---

---

---

---

---

---

---

---

---

---

---



### 6.9.3 Übung zum Debugging von Bash-Skripts

Ein weiterer hilfreicher Befehl lautet `bash -x <Programmname>`. Informieren Sie sich, was dieser macht und testen Sie ihn anhand Ihres zuvor erstellten Skripts.

---

---

---

---

---

---

---

### 6.9.4 Übung mit if-Anweisung

Der `test`-Befehl wird häufig in Verzweigungen bei `if`-Anweisungen benötigt. Schreiben Sie ein kleines Bash-Skript, das überprüft, ob die ersten beiden übergebenen Parameter gleich sind und dazu eine entsprechende Meldung ausgibt. Sie können wieder das Skript von vorher erweitern.

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---



### 6.9.6 Übung mit while-Schleife

Schreiben Sie ein kleines Bash-Script, das in einer while-Schleife einen Zähler von 1 bis 20 zählt und jeweils den aktuellen Zählerwert auf den Bildschirm ausgibt.

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

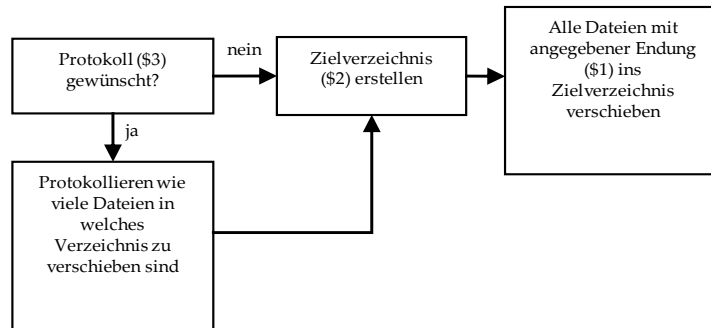
---

### 6.9.7 Übung: Bash-Skript zum Bewegen von Dateien

Entwickeln Sie ein Bash-Skript, mit dem Dateinamen `mvprot.sh`. Es soll im aktuell gewählten Verzeichnis alle Dateien mit der angegebenen Endung in ein neu erstelltes Unterverzeichnis verschieben und die Anzahl der verschobenen Dateien sowie deren Namen – falls gewünscht – in eine Protokolldatei im Quellverzeichnis schreiben. Das Skript besitzt also die drei Parameter *Dateiendung*, *Name des Unterverzeichnisses* und ein *Flag* für die Auswahl, ob ein Protokoll über den Vorgang erstellt werden soll. Zum Entwickeln und Testen Ihrer Lösung legen Sie sich bitte ein neues Verzeichnis mit einigen Testdateien an.

Der Ablauf ist im folgenden Diagramm skizziert:

## 6 Grundlagen der Bash-Programmierung



Hinweis: Mit der Kommandofolge

```
AnzahlDateien=`ls *.txt | wc -w`
```

weisen Sie der Variable `AnzahlDateien` die Anzahl der Dateien mit der Endung „.txt“, die sich im aktuellen Verzeichnis befinden, zu. Durch die Anwendung der sog. Backquotes (Vorsicht: nicht die einfachen Hochkommata) wird eine Kommandosubstitution durchgeführt (siehe Quoting). Das bedeutet, dass die in Backquotes eingeschlossenen Kommandos zusammenhängend ausgeführt werden und die Ausgabe der gesamten Befehlsfolge als Ergebnis in die Variable geschrieben wird. Alternativ dazu kann man das Kommando in der Bash wie folgt notieren:

```
AnzahlDateien=$(ls *.txt | wc -w)
```

Mit diesem Mechanismus könnten Sie z. B. die Anzahl der verschobenen Dateien zählen. Aber es gibt auch andere Möglichkeiten.

---

---

---

---

---

---

---

---

---

---

## This image shows a single sheet of white paper with horizontal ruling lines. The lines are evenly spaced and run across the width of the page. There are no margins, text, or other markings on the paper.

### 6.9.8 Übung: Bash-Skript zum Bewegen von Dateien (Erweiterung)

Versuchen Sie nun, Ihr Shell-Skript so zu erweitern, dass eine Prüfung der übergebenen Parameter stattfindet. Zudem soll überprüft werden, ob das zu erstellende Verzeichnis bereits existiert. In diesem Fall soll dem Anwender eine entsprechende Fehlermeldung ausgegeben werden

Ferner ist gefordert, eine Hilfsfunktion zu implementieren, die eine kurze Beschreibung des Programms und der möglichen Parameter liefert. Als dritte Erweiterung soll überprüft werden, ob Dateien mit der übergebenen Änderung existieren und verschoben werden müssen. Falls nicht soll ebenfalls eine entsprechende Fehlermeldung ausgegeben werden.

This image shows a single sheet of white paper with horizontal ruling lines. The lines are evenly spaced and run across the width of the page. There are no margins, text, or other markings on the paper.

**Hinweis.** Diese Übung ist für Fortgeschrittene gedacht und optional!

Erweitern Sie das vorgegebene Bash-Skript, welches das zu den Übungen beigelegte Java-Programm `Fakultaet.java` zur Berechnung von Fakultäten übersetzt in eine Jar-File mit der Klasse `Fakultaet.class` erzeugt und anschließend das Programm aus der Jar-File heraus startet. Alle Fehlersituationen sind abzufangen und entsprechend mit Meldungen zu kommentieren. Bitte beachten Sie, dass Sie sowohl bei der Übersetzung als auch beim späteren Starten des Programms noch Fremdbibliotheken benötigen.

Das Java-Programm erwartet eine Zahl  $x$ , mit  $0 \leq x \leq 20$ , als Eingabeparameter. Der benötigte Eingabeparameter für das Programm ist im Skript einzulesen und auf Plausibilität zu prüfen. Nur wenn der eingegebene Wert korrekt ist, darf das Java-Programm aufgerufen werden.

Nutzen Sie in Ihrem Skript die bereits vorhandenen Bash-Funktionen. Sie können auch Bash-Befehle nutzen, die im Skriptum nicht besprochen wurden.

Im Bash-Skript sind die evtl. benötigten Umgebungsvariablen `JAVA_HOME`, `CLASSPATH` und `PATH` geeignet zu belegen.

## 6 Grundlagen der Bash-Programmierung

---

Prüfen Sie die Returnwerte der aufgerufenen Programme `javac` und `jar` und geben Sie je nach Erfolgs- oder Fehlerfall folgende Exit-Codes zurück:

- 0: Erfolgreiche Verarbeitung
- 1: Übersetzungsfehler
- 2: Fehler beim Erzeugen der Jar-File
- 3: Fehler bei der Parametereingabe

Überlegen Sie sich vor Beginn der Programmierung den Ablauf und skizzieren Sie ihn in einem Ablaufdiagramm oder in Pseudocode-Notation.

Verwenden Sie zum Programmieren einen Linux-Editor wie z. B. `vi` oder `emacs` (siehe Anhang). Versetzen Sie das Bash-Skript direkt im Sourcecode mit entsprechender Dokumentation. Testen Sie das Skript unter Nutzung geeigneter Testfälle (richtige und auch fehlerhafte Eingaben prüfen).

Programmvorgabe (auch als Datei verfügbar):

```
#!/bin/bash
SRC_DIR="src"
JAVA_SRC_DIR="${SRC_DIR}/main/java"
SRC_FILES="sources.txt"
BIN_DIR="bin"
TMP_DIR="tmp"
RES_SRC_DIR="${SRC_DIR}/main/resources"
MANIFEST_FILE="${RES_SRC_DIR}/META-INF/MANIFEST.MF"
LIB_DIR="lib"
LOGS_DIR="logs"
JAR_FILE_NAME="WInfoJavaExample"
CLASSPATH=""
NEEDED_PROGS="java javac"

# function checks, if given variable is numeric
function is_numeric() {
    CHECK_VALUE=$(echo ${1} | tr -d 0-9)
    if [ -z "${CHECK_VALUE}" ]
    then
        return 0
    else
```



```
        return 1
    fi
}

# function checks, if defined programs are available on the
# system
function check_needed_progs() {
    for PROG in ${NEEDED_PROGS}
    do
        echo "check if Program: ${PROG} is present"
        which ${PROG}
        if [ ${?} -ne 0 ]; then
            echo "Program ${PROG} is missing"
        else
            echo "Program ${PROG} is available"
        fi
    done
    echo ""
}

# function for finding all JAVA source files for compile
# process
function find_source_files() {
    find ${SRC_DIR} -name "*.java" > ${SRC_FILES}
}

# function for setting JAVA CLASSPATH, needed for compile
# process
function set_classpath() {
    export CLASSPATH="lib/log4j-api-2.1.jar:lib/log4j-core2.1.jar"
}

# function, which does the compile process
function compile_code() {
    if [ ! -d ${BIN_DIR} ]; then
        mkdir ${BIN_DIR}
    fi
}
```

## 6 Grundlagen der Bash-Programmierung

---

```
fi
javac -d ${BIN_DIR}/ -sourcepath ${JAVA_SRC_DIR}/ @${SRC_FILES}
}

# command for creating runnable JAR file
function make_jar_file() {
    jar cvfm ../${JAR_FILE_NAME}.jar ../${MANIFEST_FILE} *
    RET_VAL=${?}
    if [ ${RET_VAL} -ne 0 ]; then
        echo "Error during JAR Create Process. "
        Echo "Check Code or Dependencies and run Skript again!"
        exit 2
    fi
}

# function, for the preparations before JAR creates process
function prepare_and_make_jar() {
    if [ ! -d ${TMP_DIR} ]; then
        mkdir ${TMP_DIR}
    fi
    cp -r ${BIN_DIR}/* ${TMP_DIR}
    cp -r ${RES_SRC_DIR}/* ${TMP_DIR}
    cd ${TMP_DIR}
    make_jar_file
    cd ..
}

# cleans up the directories
function cleanup() {
    rm -rf ${BIN_DIR}
    rm -rf ${TMP_DIR}
    rm -rf ${LOGS_DIR}
    rm ${JAR_FILE_NAME}.jar
    rm ${SRC_FILES}
}

# Please write your script here
```

## 7 Weitere interessante Kommandos

### Zielsetzung des Kapitels

Den Studierenden sollen in diesem Kapitel weitere interessante Kommandos vorgestellt werden. Diese können optional bei den Übungen zur Bash-Programmierung genutzt werden.

### Wichtige Begriffe

grep, fgrep, egrep, find, whois, who, which, printf, head, tail, tr.

## 7.1 Kommandos zur Mustersuche

### 7.1.1 Kommando grep

| grep         |   |
|--------------|---|
| Beschreibung | Dient der Suche nach Textzeilen aus Dateien die dem übergebenen Muster entsprechen.<br><br>Das zu suchende Muster darf auch als „regular Expression“ angegeben werden.  |
| Syntax       | grep [ <optionen>] &lt;muster&gt; [&lt;dateiname_1&gt; ... [&lt;dateiname_n&gt;]</optionen>   |
| Beispiele    | # gibt alle Zeilen aus der Datei „test.txt“ auf dem<br># Bildschirm in denen „Hello“ enthalten ist.<br>grep Hello test.txt<br><br># gibt alle Zeilen aus der Bash-History auf dem Bildschirm<br># in denen cp enthalten ist.<br>history   grep cp |

### 7.1.2 Kommando egrep

| egrep        |  |
|--------------|--|
| Beschreibung | <p>Alias-Bezeichnung für <code>grep -E</code>; kann mit erweiterten regulären Ausdrücken (ERE) umgehen.</p> <p>Im Muster werden die Metazeichen unterstützt.</p>   |
| Syntax       | <pre>egrep  [&lt;optionen&gt;]  &lt;muster&gt;  [&lt;dateiname_1&gt; ... [&lt;dateiname_n&gt;]</pre> <p>Einige interessante Optionen:</p> <ul style="list-style-type: none"> <li>-l Ausgabe der Dateinamen aller Dateien, die as Muster enthalten.</li> <li>-c Ausgabe der Anzahl von Matches.</li> <li>-i Suchen unter Vernachlässigung von Groß- und Kleinschreibung</li> <li>-n Ausgabe der Zeilennummern der Zeilen, in denen das Muster gefunden wurde</li> <li>-v Komplement des regulären Ausdrucks ist das Suchkriterium</li> </ul> <p>Interessante Metadaten:</p> <ul style="list-style-type: none"> <li>^ Muster muss am Anfang der Zeile stehen, z. B. '^AB' sucht alle Zeilen, die mit „AB“ beginnen.</li> <li>\$ Muster muss am Ende der Zeile stehen, z. B. '01\$' sucht alle Zeilen, die mit „01“ enden.</li> </ul> |
| Beispiele    | <p># Alle Zeilen aus der Datei /etc/group ermitteln, die mit nofork oder nogroup beginnen</p> <pre>egrep '^no(fork group)' /etc/group</pre>  |

## 7.2 Kommandos zur Datei- und Datenbearbeitung

### 7.2.1 Kommando tr

| tr           |  |
|--------------|--|
| Beschreibung | Zeichen ersetzen oder löschen  |
| Syntax       | tr [<optionen>] <set1> [<set2>]  |
| Beispiele    | # alle Zeichen in Großbuchstaben transformieren<br>echo „hElLo“   tr [:lower:] [:upper:] |

### 7.2.2 Kommando head

| head         |  |
|--------------|--|
| Beschreibung | Den ersten Teil von Dateien ausgeben   |
| Syntax       | head [<optionen>] [<dateiname_1> ... [<dateiname_n>]   |
| Beispiele    | # zeigt die ersten fünf Zeilen an<br>head -n5 test.txt<br><br># zeigt alle Zeilen von der ersten bis zur fünftletzten an<br>head -n-5 test.txt |

### 7.2.3 Kommando tail

| tail         |  |
|--------------|--|
| Beschreibung | Den letzten Teil von Dateien ausgeben  |
| Syntax       | tail [<optionen>] [<dateiname_1> ... [<dateiname_n>]   |
| Beispiele    | # Inhalt von Logdatei anzeigen<br>tail -f /var/log/messages<br><br># zeigt die letzten fünf Zeilen an<br>tail -n5 test.txt<br><br># zeigt alle Zeilen von der fünften bis zur letzten an<br>tail -n+5 test.txt |

### 7.2.4 Kommando printf

| printf       |  |
|--------------|--|
| Beschreibung | Dient der formatierten Ausgabe von Daten   |
| Syntax       | printf <format> [<argumente>]<br>printf <optionen>   |
| Beispiele    | SURNAME="Mustermann"<br>FIRSTNAME="Max"<br>printf "Surname: %s\nName: %s\n" "\$SURNAME"<br>"\$FIRSTNAME" |

### 7.2.5 Kommando find

| find         |  |
|--------------|--|
| Beschreibung | Dient dem Aufsuchen von Dateien und Ordnern im Dateisystem des Rechners.<br><br>Sinnvoll ist es einen „Startpunkt“ für die Suche zu setzen. Wird dieser nicht gesetzt gilt standardmäßig das aktuelle Verzeichnis.   |
| Syntax       | find [-H] [-L] [-P] [<startpunkt>] [<expression>]  |
| Beispiele    | # sucht alle Dateien, rekursiv, also auch in den<br># Subverzeichnissen vom aktuellen Verzeichnis aus,<br># auf welche in der letzten Minute zugegriffen wurde.<br>find -amin 1<br><br># sucht alle Dateien mit der Endung .sh und dem Typ<br># „Datei“ im Verzeichnis „/<br>find / -type f -name „*.sh“ |

## 7.3 Sonstige, nützliche Kommandos

### 7.3.1 Kommando whois

| whois        |  |
|--------------|--|
| Beschreibung | Abruf der Besitzinformationen einer Domain                         |
| Syntax       | whois [<optionen>] [<query>]                                       |
| Beispiele    | # Ruft Informationen über die Domain hm.edu ab.<br>whois -v hm.edu |

### 7.3.2 Kommando which

| which        |   |
|--------------|---|
| Beschreibung | Kompletten Pfad von (Shell) Kommandos anzeigen  |
| Syntax       | which [<optionen>] <programmname>   |
| Beispiele    | # Pfad zum Editor vi anzeigen<br>which vi<br># alle Pfade und Aliase zum Editor vi anzeigen<br>which vi -a<br># Pfade ohne Aliase zum Editor vi anzeigen<br>which vi --skip-alias |

### 7.3.3 Kommando who

| who          |  |
|--------------|--|
| Beschreibung | Angemeldete Benutzer auflisten   |
| Syntax       | who [<optionen>]   |
| Beispiele    | # letzte Startzeit des Systems anzeigen<br>who -b<br># alle angemeldeten Benutzer auflisten<br>who -u<br># alle Informationen zu angemeldeten Benutzern anzeigen<br>who -a |

### 7.4 Übersicht über weitere interessante Kommandos

In der untenstehenden Tabelle finden Sie noch weitere interessante Kommandos. Diese werden nicht detailliert beschrieben. Informationen hierzu finden Sie in den Manpages auf dem Linux System oder in (Kofler2013).

|       |         |        |        |
|-------|---------|--------|--------|
| alias | expr    | passwd | tee    |
| awk   | fg      | sed    | time   |
| bg    | file    | set    | uniq   |
| clear | htop    | sleep  | unset  |
| cut   | info    | sort   | unzip  |
| date  | jobs    | stat   | watch  |
| df    | killall | sudo   | whoami |
| du    | locate  | tar    | zip    |



## 8 Zusammenfassung

Dieses Skriptum sollte einen ersten Einblick in die Bash und die Bash-Programmierung verschaffen. Die grundlegenden Kommandos und Programmstrukturen wurden vermittelt.

Die Bash bietet noch eine Fülle weiterer Möglichkeiten, die dem fleißigen Studierenden zur Fortbildung überlassen werden. Die Literaturhinweise dienen als Einstieg hierfür.

## Anhang: Gebräuchliche Linux-Editoren

Editoren benötigen Sie z. B. zur Programmierung von Bash-Skripts. Im diesem Anhang finden Sie die wichtigsten Hinweise zur Nutzung der Editoren vi, nano und emacs. Alle drei Editoren sind in Linux-Systemen standardmäßig vorhanden.

### Editor vi

Kommando zum Starten:

```
$ vi [<Dateiname bzw. Pfad zur Datei>]
```

Existiert der angegebene Dateiname nicht, wird eine neue, leere Datei mit dem angegebenen Namen angelegt, achten Sie also auf eine korrekte Pfadangabe.

Man unterscheidet den Kommando- und den Editiermodus. Um der angelegten oder geöffneten Datei einen Text hinzuzufügen, muss in den Editiermodus gewechselt werden, dieser wird über die Tastensequenz `ESC, i` oder `ESC, a` betreten. Durch `ESC` wechselt man wieder vom Editier- in den Kommandomodus. Unterschiede äußern sich dahingehend, dass bei ersterer Methode die Eingaben links bei der letzteren die Eingaben rechts vom Cursor eingefügt werden. Es gibt noch mehrere unterschiedliche Methoden für das Einfügen.

Im bereits erwähnten Kommandomodus können folgende Kommandos wie folgt ausgeführt werden:

|  |  |
|--|--|
| <code>x</code> – einfaches Betätigen der Taste <code>x</code>  | Löscht das Zeichen an der Cursorposition   |
| <code>dd</code> – doppeltes Betätigen der Taste <code>d</code> | Löscht Zeile des aktuellen Aufenthaltsortes des Cursors (z. B. Cursor befindet sich in Zeile 13 -> Zeile 13 wird gelöscht) |
| <code>yy</code> – doppeltes Betätigen der Taste <code>y</code> | Kopiert die aktuelle Zeile in einen Puffer   |
| <code>p</code> – einfaches Betätigen der Taste <code>p</code>  | Fügt Pufferinhalt an aktuellem Cursorstandort ein  |
| <code>u</code> – einfaches Betätigen der Taste <code>u</code>  | Macht die gesamte letzte Eingabe rückgängig  |

|  |   |
|--|---|
| <code>/&lt;Suchmuster&gt;</code> - Slash und anschließende Eingabe des Suchbegriffs        | Sucht den Text ab Cursorposition vorwärts nach eingegebenem Suchmuster ab und hebt dieses bei erfolgreicher Suche hervor.   |
| <code>?&lt;Suchmuster&gt;</code> - Fragezeichen und anschließende Eingabe des Suchbegriffs | Sucht den Text ab Cursorposition rückwärts nach eingegebenem Suchmuster ab und hebt dieses bei erfolgreicher Suche hervor.  |
| <code>:set number</code> – Doppelpunkt und anschließende Eingabe von <i>set number</i>     | Zeigt Zeilenangabe innerhalb der geöffneten Datei an.   |
| <code>:w[q] [Dateiname] [!]</code> - Doppelpunkt und anschließende Eingabe von <i>w</i>    | <p>Speichert die aktuelle Datei im aktuellen Verzeichnis, falls kein Pfad angegeben wurde.</p> <p>Die zusätzliche Eingabe schließt die Datei im Anschluss – Speichern &amp; Schließen.</p> <p>Die zusätzliche Eingabe von <i>!</i> erlaubt das Überschreiben einer bereits vorhandenen Datei gleichen Namens.</p> |
| <code>:q[!]</code> - Doppelpunkt und anschließende Eingabe von <i>q</i>                    | <p>Schließt die aktuelle Datei.</p> <p>Bei getätigter Änderung kommt ein Hinweis, dass noch nicht gespeichert wurde. Die Datei wird nicht geschlossen. Dies kann aber durch Anfügen von <i>!</i> umgangen werden – hier wird die Datei dann ohne Übernahme der Änderungen geschlossen.</p>                        |
| <code>:s/&lt;Suchmuster&gt;/&lt;Ersetzungsmuster&gt;/g</code>                              | Sucht nach Suchmuster und ersetzt dieses in der aktuellen Zeile.  |

## Editor nano

Kommando zum Starten:

```
$ nano [<Dateiname bzw. Pfad zur Datei>]
```

Der Editor `nano` verfügt über eine gute Hilfeseite. Außerdem werden am unteren Rand des Fensters wichtige Kommandos und die dafür notwendigen Tastenkombinationen angezeigt. Bei `nano` handelt es sich um einen eher auf Komfort ausgelegten Editor, es werden häufig Fragen bezüglich der Verifikation einzelner Aktionen gestellt, welche per Tastendruck dann ausgeführt oder noch abgebrochen werden können. Innerhalb der integrierten Hilfeseite steht das Symbol `^` für die STRG-Taste und `M` für die ALT-Taste.

Folgende Kommandos können die Arbeit mit `nano` erleichtern:

|                            |                                      |
|----------------------------|--------------------------------------|
| STRG + G                   | Hilfeseite                           |
| STRG + X                   | Schließen                            |
| STRG + W                   | Suche                                |
| STRG + O                   | Zwischenspeichern                    |
| STRG + \ (ALT GR + ß)      | Suchen & Ersetzen                    |
| STRG + K                   | Zeile Ausschneiden                   |
| STRG + U                   | Einfügen                             |
| STRG + C                   | Position des Cursors anzeigen lassen |
| ALT + \ (ALT GR + ß); Pos1 | Zum Anfang springen                  |
| ALT + / (SHIFT + 7); Ende  | Zum Ende springen                    |
| ALT + } (ALT GR + 0)       | ausgewählte Zeile einrücken          |
| ALT + { (ALT GR + 7)       | ausgewählte Zeile ausrücken          |
| ALT + U                    | Letzte Operation rückgängig machen   |

## Editor emacs

Kommando zum Starten:

Über graphische Oberfläche:

```
$ emacs [<Dateiname bzw. Pfad einer Datei>]
```

Über Konsole/Terminal:

```
$ emacs -nw [<Dateiname bzw. Pfad einer Datei>]
```

Der Editor `emacs` verfügt über eine eigene, individuell anpassbare Benutzeroberfläche, kann aber auch als reines Kommandozeilentool aufgerufen werden. Der Editor verfügt über eine farbenreiche, übersichtliche Bildwiedergabe. Die Installation von Plugins ist möglich, der Editor `emacs` kann also nicht nur graphisch individuell angepasst, sondern auch praktisch nach persönlichen Vorlieben erweitert werden. In `emacs` steht der Buchstabe `C` für `Control`, womit die `STRG`-Taste gemeint ist, das `M` (`Meta`) steht für die `ALT`-Taste.

Folgende Kommandos können im Umgang mit dem Editor `emacs` sehr hilfreich sein:

|                   |                               |
|-------------------|-------------------------------|
| STRG + X STRG + F | Datei öffnen                  |
| STRG + X STRG + S | Datei speichern               |
| STRG + X STRG + W | Datei speichern unter         |
| STRG + X STRG + C | Emacs beenden                 |
| STRG + S          | Vorwärtssuche                 |
| STRG + R          | Rückwärtssuche                |
| STRG + A          | Zum Anfang der Zeile springen |
| STRG + E          | Zum Ende der Zeile springen   |
| STRG + D          | Ein Zeichen löschen           |
| ALT + D           | Ein Wort löschen              |
| STRG + X U        | Aktion rückgängig machen      |
| STRG + H T        | Tutorial                      |

## Anhang: Gebräuchliche Linux-Editoren

---

|   |   |
|---|---|
| STRG + S <Suchmuster> ALT + %<br>(SHIFT + 5) <Ersetzungsmuster> | Suchen und Ersetzen eines gewünschten Suchmusters innerhalb einer Datei. Dieser Vorgang muss dann für jedes gefundene Exemplar durch Drücken von „Y“ bestätigt werden |
| STRG + G  | Abbrechen   |
| STRG + Einfügen   | Kopieren des aktuell markierten Strings   |
| SHIFT + Einfügen  | Einfügen des kopierten Strings  |

## Literaturhinweise und Web-Links

Graiger Ch. (2009) Bash-Programmierung Einstieg und professioneller Einsatz. entwickler-press. 2009.

Heisse (2011) <http://www.heise.de/ix/artikel/Kerzenflut-1322441.html>. letzter Zugriff am 29.01.2018.

Kofler M. (2013) Linux-Kommandoreferenz: Shell-Befehle von A bis Z. Galileo Computing. 2013

Levin J. (2013) Mac OS X an iOS Internals. John Wiley & Sons, Inc. 2013.

Linux Foundation (2018): The Linux Kernel Development, <http://linuxfoundation.org>, letzter Zugriff am 29.01.2018.

Mandl P. (2014) Grundkurs Betriebssysteme Architekturen, Betriebsmittelverwaltung, Synchronisation, Prozesskommunikation, Virtualisierung, Springer Vieweg (Verlag). 4. Auflage, 2014.

Regular-Expressions (2018) info <http://www.regular-expressions.info/>. letzter Zugriff am 29.01.2018.

SelfLinux (2018) [http://www.selflinux.org/selflinux/html/linux\\_geschichte.html](http://www.selflinux.org/selflinux/html/linux_geschichte.html). letzter Zugriff am 29.01.2018.

Wikipedia Linux (2018) [http://de.wikipedia.org/wiki/Geschichte\\_von\\_Linux](http://de.wikipedia.org/wiki/Geschichte_von_Linux). letzter Zugriff am 29.01.2018.

Wikipedia Sticky (2018) [https://de.wikipedia.org/wiki/Sticky\\_Bit](https://de.wikipedia.org/wiki/Sticky_Bit), Zugriff am 29.01.2018.

Wikipedia Systemd (2018) <https://de.wikipedia.org/wiki/Systemd>, Zugriff am 29.01.2018.

## Sachwortverzeichnis

- Absoluter Pfadname 28
- Benutzerdefinierte Umgebungsvariable 18
- Berechtigungskonzept 48
- Bootloader 59
- Bootvorgang 59
- cat 34
- cd 31
- chgrp 52
- chmod 51
- chown 51
- cp 34
- echo 30
- egrep 92
- exit 55
- Expandierungen 39
- Exportierte Variable 18
- FHS 13
- find 94
- for-Schleife 76
- grep 91
- GRUB 59
- Harter Link 30
- head 93
- history 41
- if-Anweisung 75
- Init-System 60
- kill 66
- Kommandosyntax 15
- launchd 60
- less 34
- ln 36, 95
- Login-Shell 55
- logout 55
- ls 31
- mkdir 33
- more 34
- mv 35
- Pipe 38
- printf 94
- Prozess 14
- ps 63
- pstree 64
- Punkt-Kommando 72
- pwd 31
- Quoting 39
- rc 61
- Redirection 37
- Reguläre Ausdrücke 21
- relativer Pfadname 28
- rm 35
- rmdir 33
- Runlevel 61
- stderr 37
- stdin 37
- stdout 37
- Subshell 72
- Suffix 27
- Symbolischer Link 30
- systemd 62
- SysVinit 61
- tail 93
- top 65
- touch 33
- tr 93
- umask 52
- Umgebungsvariable 18
- Vordefinierte Umgebungsvariable 18
- wc 36
- while-Schleife 76
- who 95
- whois 95