

# Unix - Linux



Eine Einführung von Wolfgang Pulina

Universität Regensburg

Rechenzentrum

September 2007



# Vorwort zur vierten Auflage

Seit 17 Jahren halte ich am Rechenzentrum der Universität Regensburg den Kurs *Einführung in Unix/Linux*, und seit 17 Jahren wird dieser Kurs immer wieder von vielen Teilnehmern besucht. Dieses große Interesse ist umso erstaunlicher, als dass ein Betriebssystem wie Unix in der gesamten EDV-Landschaft eine eher periphere Rolle zu spielen scheint. Doch bei genauerer Betrachtung wird man feststellen, dass Unix nach wie vor einen zentralen Platz unter den Betriebssystemen einnimmt, vor allem dann, wenn es darum geht, hochverfügbare, skalierbare Leistungen zur Verfügung zu stellen. Es gibt aber noch einen weiteren Grund für das große Interesse an Unix-Betriebssystemen und dieser Grund heisst Linux. Aus der Spielwiese für Programmierer und Informatiker ist mittlerweile ein ernstzunehmendes System geworden, das fast schon routinemäßig in die IT-Landschaft der Privatwirtschaft integriert wird. Auch im häuslichen Bereich wird zunehmend Linux eingesetzt, wobei es hier hauptsächlich darum geht, die Alternativen zu einem anderen marktbeherrschenden Betriebssystem auszuloten.

Bei der Neugestaltung der kursbegleitenden Dokumentation wurde versucht, sowohl Linux-Neulingen als auch Linux-Fortgeschrittenen interessante Tipps und Anregungen zu geben. So wurde die Darstellung grafischer Werkzeuge ausgeweitet, wobei im Gegenzug die Behandlung von Themen wie vi, emacs und PC-X-Server-Emulation weichen musste. Neu hinzugekommen ist das Kapitel über die bash.

Ein einwöchiger Kurs kann natürlich nicht das gesamte Potential von Unix/Linux ergründen. Der Kurs soll vielmehr als Motivation betrachtet werden, sich intensiver mit dem Betriebssystem und seinen Anwendungen auseinanderzusetzen. Ich freue mich, falls mir dies mit vorliegender Publikation gelingen sollte, bin aber auch offen für alle Anregungen, die mir dabei helfen, etwas besser zu machen.

Zum Schluss möchte ich mich bei allen bedanken, die in Diskussionen und Gesprächen wertvolle Ideen zum Werden dieses Kurs-Skriptes beigetragen haben: - Vielen Dank.

Wolfgang Pulina - 28. September 2007



# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
1.1	Typographische Konventionen	1
1.2	Die Entwicklung von Unix	2
1.3	Charakterisierung von Unix	3
<b>2</b>	<b>Erste Schritte</b>	<b>5</b>
2.1	Zugang zu einem Unix-Rechner	5
2.1.1	Login über das Datennetz	6
2.1.2	Dateitransfer über das Datennetz	8
2.2	Passwörter	10
2.3	Benutzeroberflächen und Benutzerumgebung	11
2.4	Der grafische Texteditor gedit	16
2.5	Unix-Dokumentation	17
2.6	Spezialzeichen und reguläre Ausdrücke	20
<b>3</b>	<b>Das Dateisystem</b>	<b>23</b>
3.1	Dateiarten und Dateinamen	24
3.2	Kommandos mit Dateien	26
3.3	Kommandos mit Verzeichnissen	29
3.4	Dateischutz	31
3.5	Drucken von Dateien	32
3.6	Der Dateimanager konqueror	35
<b>4</b>	<b>Kommandos und Prozesse</b>	<b>39</b>
4.1	Der Login-Prozess	40
4.2	Kommandoeingabe	41
4.3	Prozesskontrolle	42
<b>5</b>	<b>Die csh/tcsh - Shell</b>	<b>47</b>
5.1	Kommandowiederholung	47
5.2	Vervollständigung von Kommandos, Variablen und Dateinamen	48
5.3	alias - Mechanismus	49
5.4	Ein- und Ausgabeumlenkung	50
5.5	Variable	51
5.5.1	Lokale Variable	51
5.5.2	Globale Variable	54
5.6	Shell-Programmierung	55
5.6.1	Die if - Anweisung	56
5.6.2	Die foreach - Schleife	56
5.6.3	Die while - Schleife	57
5.6.4	Die shift - Anweisung	58

5.6.5	Fallunterscheidung mit switch und case	58
5.6.6	Die break- und continue-Anweisung	59
5.6.7	Die goto-Anweisung	60
5.6.8	Allerlei Nützliches	60
<b>6</b>	<b>Die bash - Shell</b>	<b>63</b>
6.1	Kommandowiederholung	63
6.2	Vervollständigung von Kommandos, Variablen und Dateinamen	63
6.3	alias - Mechanismus	64
6.4	Ein- und Ausgabeumlenkung	64
6.5	Variable	65
6.6	Arithmetik	66
6.7	Shell-Programmierung	66
6.7.1	Die if - Anweisung	67
6.7.2	Die for - Schleife	67
6.7.3	Die while - Schleife	68
6.7.4	Die until - Schleife	68
6.7.5	Die select - Schleife	69
6.7.6	Fallunterscheidung mit case	70
6.7.7	shift-Statement	70
6.7.8	Signalhandling mit trap	70
6.7.9	Eingabe von Tastatur	71
6.7.10	break und continue	71
6.7.11	Funktionen	71
6.7.12	Sonstiges	72
<b>7</b>	<b>Textmusterverarbeitung mit awk</b>	<b>73</b>
7.1	Aufruf und Programmstruktur	73
7.2	Die Sprachelemente von awk	75
7.3	Bedingungen	77
7.4	Aktionen	78
7.4.1	Ausgabeeanweisungen	78
7.4.2	Kontrollanweisungen	79
7.5	Sonstiges	80
<b>A</b>	<b>Ausgewählte Web-Links</b>	<b>81</b>

# 1 Einleitung

## 1.1 Typographische Konventionen

In dieser Publikation werden häufig Kommandos, Dateiinhalte, Kommunikationsdialoge und Programm-Strukturen wiedergegeben, die für ein leichteres Verständnis der behandelten Themen dienen sollen. Die "Interaktion mit dem Unix-System" soll daher auch aus dem Schriftbild logisch erkennbar sein. Folgende typographische Konventionen sind in diesem Zusammenhang gültig.

<code>ls -al</code>	Dieser Text kann wörtlich übernommen werden. Meist handelt es sich dabei um Kommandos.
<code>cp <i>file_old</i> <i>file_new</i></code>	Bei diesem Text kann nur das <code>cp</code> wörtlich übernommen werden. Die beiden in Schrägschrift gesetzten Begriffe sind nur Platzhalter für konkrete Werte.
<code>head [-n] [<i>filename ...</i>]</code>	Alle in eckigen Klammern stehenden Angaben sind optional.
<code>/etc/hosts</code>	Dateinamen werden immer in Courier dargestellt.
<code>&lt;Ctrl&gt; f</code>	Control-Tastensequenz: die Control-Taste wird gedrückt gehalten und gleichzeitig die Taste <code>f</code> betätigt. Bei deutschen Tastaturen entspricht die <code>&lt;Strg&gt;</code> -Taste der <code>&lt;Ctrl&gt;</code> -Taste.
<code>host&gt;</code>	Rechner-Prompt; dahinter stehen die Kommandos, die am Bildschirm eingegeben werden können.

Dateiinhalte und Bildschirmausgaben werden in einfachem Courier wiedergegeben. Als Beispiel soll das Kommando `ls -l /etc/motd` dienen.

```
ls -l /etc/motd
```

```
-rw-r--r--  1 root      sys          54 Jan  9  2000 /etc/motd
```

Im laufenden Text werden Hervorhebungen entweder in *italic* oder in **bold** dargestellt. Nach Kommando-Eingaben ist immer die Betätigung der `<Return>`- bzw. der `<Eingabe>`-Taste erforderlich. In den Kommando-Beispielen wird dies nicht explizit dokumentiert.

## 1.2 Die Entwicklung von Unix

Das Betriebssystem Unix hat bereits eine über 30-jährige Entwicklungsgeschichte hinter sich. Die Entwicklung begann in einer Zeit, in der Computer noch sehr groß und teuer waren. Üblicherweise wurde auf diesen Großrechnern im sog. Stapelbetrieb gearbeitet. Das sah gewöhnlich so aus, dass ein Stapel Lochkarten in den Rechner eingelesen wurde, der Rechner dann die Bearbeitung vornahm und das Ergebnis auf einen Drucker ausgab. Falls das Ergebnis nicht den Wünschen entsprach, mussten die fehlerhaften Lochkarten ersetzt werden und der ganze Zyklus begann von neuem. Der Wunsch vieler Programmierer nach einer Arbeitsumgebung, die den interaktiven Dialog in einem Mehrbenutzerbetrieb und das Teamwork unterstützt, mündete in der Entwicklung eines neuen Betriebssystems: **Unix**.

Ken Thompson entwickelte 1969 bei den Bell Laboratories das erste Unix auf einer PDP-7, einem Rechner der Firma Digital Equipment. Dieses erste Unix war noch vollständig in Assembler geschrieben. Erst mit der Entwicklung der Programmiersprache C von Dennis Ritchie wurde das Betriebssystem 1971 in C umgeschrieben.

Die Verbreitung des Systems erfolgte zunächst im akademischen Umfeld, indem es kostenlos an Universitäten und wissenschaftliche Institute verteilt wurde. Vielfach wurde das System an verschiedenen Stellen weiterentwickelt, so dass im Laufe der Zeit unterschiedliche Unix-Derivate entstanden (PWB/UNIX, IS/1, XENIX). Seit 1982 werden Unix-Betriebssysteme von kommerziellen Systemanbietern wie z.B. IBM, HP und Sun vertrieben.

Da die Entwicklung verschiedener Unix-Systeme zu immer mehr Inkompatibilitäten führte, entschloss man sich, Anfang der 90er Jahre durch die Definition von Standards die konsequente Weiterentwicklung zu einem konvergenten Basissystem voranzutreiben. Eine wichtige Rolle spielten dabei Standardisierungsorganisationen wie **IEEE** und **X/Open**. Das *Institute of Electrical and Electronic Engineers*, kurz IEEE, ist eine der vielen industriellen Organisationen, die vom *American National Standards Institute* (ANSI) mit der Definition und Spezifizierung von Standards beauftragt worden sind. Das als **POSIX**-Ausschuss bekannte Gremium der IEEE besitzt die Aufgabe, den Schnittstellenstandard für alle Aspekte der Betriebsumgebung festzulegen.

Ziel von X/Open ist es, eine Grundlage für die Entwicklung einer gemeinsamen Anwendungsumgebung zu schaffen. So hat X/Open beispielsweise POSIX als Basisbetriebssystem-Schnittstelle unterstützt und das X Window System des MIT (Massachusetts Institute of Technology) als Benutzerschnittstelle übernommen. Die Arbeit der Organisation ist in Form von Referenzdokumenten mit der Bezeichnung *X/Open Portability Guide* oder *XPG* erhältlich, die als Leitfaden für Portierbarkeit dienen sollen.

1991 schlug schließlich die Geburtsstunde von **Linux**. Der Finne Linus Thorvalds entwickelte einen Betriebssystemkern, um vor allem die Leistung der damals aktuellen Intel 80386-Prozessoren ausnutzen zu können. Dabei bediente er sich freier Software, die im Rahmen des GNU-Projekts zur Verfügung gestellt wurde (GNU C-Compiler). Von Anfang an wurde darauf Wert gelegt, dass auch Linux den POSIX- und X11-Standards entspricht. Das besondere an Linux ist die Tatsache, dass die Entwicklung nicht in den Händen einer Firma liegt, sondern aufgeteilt ist auf Software-Entwickler, die verteilt über die ganze Welt arbeiten und über das Internet miteinander kommunizieren.

Am Beginn des 21. Jahrhunderts haben Unix- und Linux-Systeme eine Qualität erreicht, die ihnen einen festen Platz in der Betriebssystem-Landschaft sichern. Die Entwicklung dieser Systeme geht natürlich weiter. Wichtig ist dabei stets die konsequente Orientierung an offenen Standards.

## 1.3 Charakterisierung von Unix

Eine Charakterisierung von Betriebssystemen kann aus unterschiedlichen Perspektiven erfolgen. Da wäre zum einen die Unterscheidung von *Real-Time*- und *Time-Sharing*-Systemen, zum anderen die Aufteilung in *Client*- und *Server*-Systeme. Ferner können Kriterien wie Portierbarkeit und Skalierbarkeit zur Differenzierung herangezogen werden.

Abgesehen von den unterschiedlichen Funktionstypen besitzt ein Unix-System eine grundlegende Struktur, die durch nebenstehende Abbildung verdeutlicht wird. Die Basis des Betriebssystems bildet der *Kernel*, der Betriebssystemkern, der für die Ansteuerung der Hardware verantwortlich ist. Die Hauptaufgaben des Kernels sind die *Prozessverwaltung*, die *Hauptspeicherverwaltung*, die *Ein/Ausgabeverwaltung* und die *Dateiverwaltung*. Die Leistungsfähigkeit eines Unix-System hängt maßgeblich von der Qualität der Implementierung dieser Basisfunktionen ab. Als nächste Schicht im Strukturmodell folgen systemnahe Programme wie die *Shell* und sog. *Utilities*. Die *Shell* ist für die kommandoorientierte Interaktion zwischen Benutzer und System verantwortlich, während die *Utilities* eine Sammlung nützlicher Programme darstellen (das Kommando `ls` ist eine derartige Utility). Den Abschluss bilden die *Applications*, also die eigentlichen Anwendungsprogramme. Der Compiler `cc` gehört ebenso zu den Anwendungen wie z.B. das Office-Paket StarOffice.

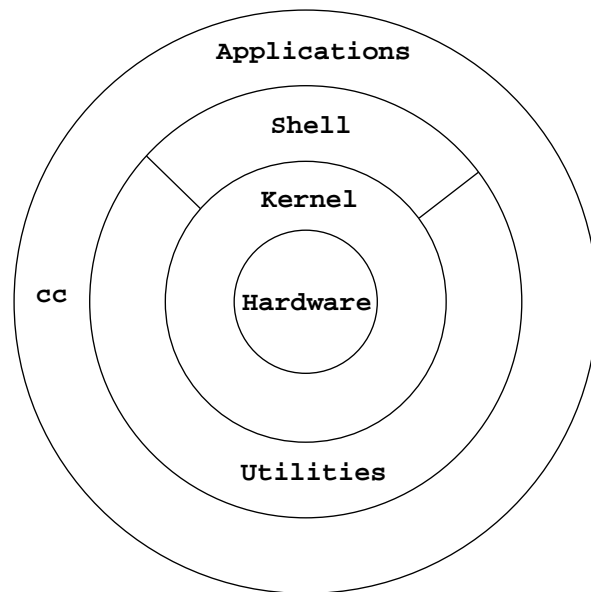


Abbildung 1.1: Struktur des Betriebssystems

Den Abschluss bilden die *Applications*, also die eigentlichen Anwendungsprogramme. Der Compiler `cc` gehört ebenso zu den Anwendungen wie z.B. das Office-Paket StarOffice.

Für eine kurze **Charakterisierung von Unix** sollen folgende Aspekte herangezogen werden.

- **Unix ist ein Dialogsystem**

Dies ist zwar in der heutigen Zeit nichts besonderes, bei der Entstehung von Unix waren aber immer noch die batch-orientierten Systeme in der Mehrzahl.

- **Unix ist primär ein Multi-User/Multi-Tasking-Betriebssystem**

Multi-User bedeutet, dass an einem Unix-System gleichzeitig mehrere Benutzer arbeiten können. Multi-Tasking heißt, dass jeder Benutzer auch mehrere Programme quasi gleichzeitig ablaufen lassen kann. Nebenbei sei bemerkt, dass es heute noch andere Betriebssysteme gibt, die diese Funktionalitäten entweder überhaupt nicht oder nur sehr eingeschränkt besitzen.

- **Unix unterstützt das virtuelle Speicherkonzept**

Das bedeutet, dass im Prinzip ein großes Programm auf relativ klein dimensionierter Hardware laufen kann. Erreicht wird dies durch die sog. virtuelle Speicheradressierung. Erst zur Laufzeit des Programms werden die virtuellen Adressen in physikalische Adressen umgesetzt.

- **Unix unterstützt das Pipe-Konzept**

Dabei können die Ausgabedaten eines Prozesses unmittelbar als Eingabedaten einem anderen Prozess übergeben werden.

- **Unix verfügt über eine hierarchische Dateiverwaltung**

Wie auch bei anderen Betriebssystemen ist die Dateistruktur baumartig aufgebaut.

- **Unix besitzt leistungsfähige Shells als Benutzerschnittstelle**

Die kommandoorientierte Interaktion mit dem System wird über Shells abgewickelt. Diese Shells bieten vielfältige Möglichkeiten für die Programmierung und Automatisierung von Vorgängen.

- **Unix ist fast vollständig in C programmiert**

Die Sprache C kombiniert die Effizienz einer Assemblersprache mit den Steuerstrukturen moderner Sprachkonzepte. Heute sind mehr als 95 % in C geschrieben.

- **Unix ist einfach in Datennetze zu integrieren**

Alle Funktionen für die Integration in Datennetze sind bereits vorhanden.

- **Unix ermöglicht eine hohe Skalierbarkeit**

Das Betriebssystem kann Rechner unterstützen, die einen, zwei, aber auch 128 Prozessoren besitzen können. Die Leistungsfähigkeit steigt dabei fast linear. Ferner können durch die Vernetzung von Rechner-Knoten in Kombination mit geeigneter Software noch weitere Leistungsreserven mobilisiert werden (Linux-Cluster-Computing).

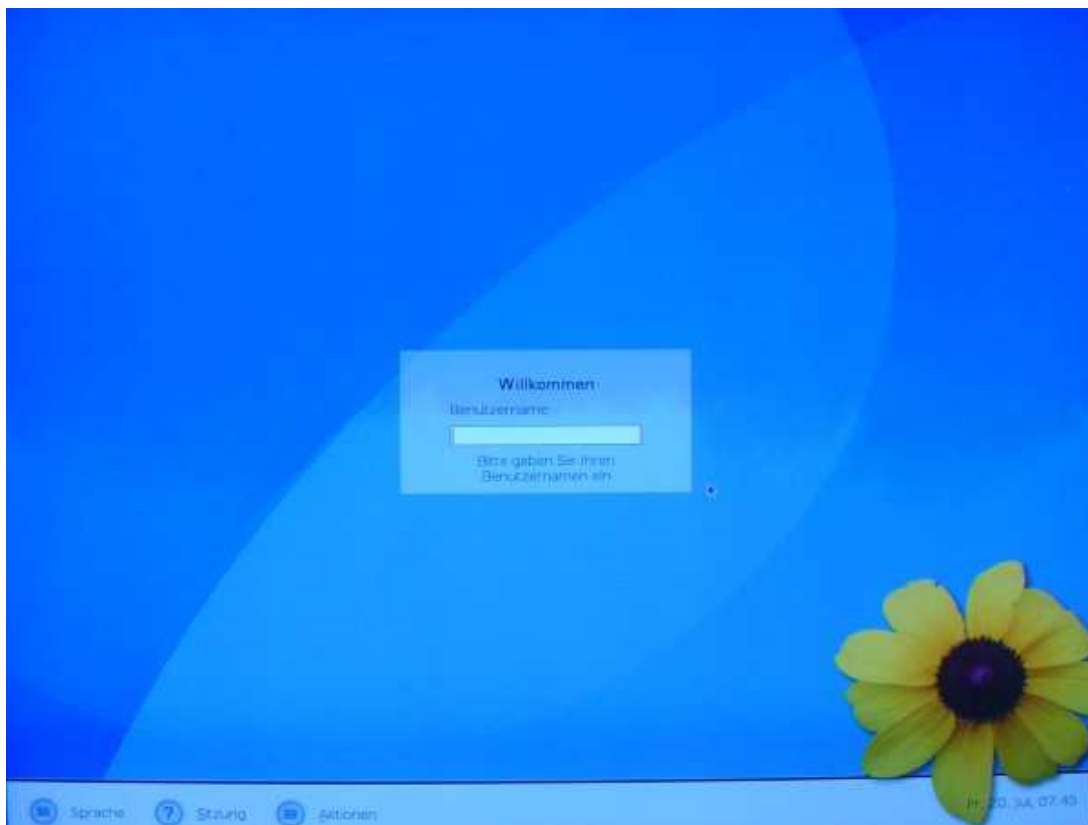
Unix/Linux-Rechner werden in den verschiedensten Bereichen eingesetzt: als Computeserver, Datenbankserver, Mailserver, Bootserver und natürlich als Web-Server, aber auch als Firewall-Systeme und für das Netzwerk-Management. Als Desktop-Systeme finden Unix-Rechner vor allem im technischen Computing als CAD-Arbeitsplätze Verwendung. Auch die Filmindustrie hat die Leistungsfähigkeit von Unix erkannt. So entstanden Animationsfilme wie "Das große Krabbeln", "Toy Story" oder "Spirit" vollständig auf Unix-Rechnern. Im Film "Titanic" wurden einige Sequenzen auf Linux-Rechnern gerechnet.

Unix erfüllt in höchstem Maße die Ansprüche an Zuverlässigkeit und Skalierbarkeit, so dass es aus dem Bereich der *Mission-Critical*-Anwendungen nicht mehr wegzudenken ist. Im Desktop-Bereich hat sich in den letzten Jahren vor allem Linux als ernsthafte Alternative zu einem anderen, marktbeherrschenden System etabliert.

## 2 Erste Schritte

### 2.1 Zugang zu einem Unix-Rechner

Der Zugang zu einem Unix-/Linux-System ist generell mit einer Authentifizierung des Benutzers verbunden. Der Benutzer meldet sich mit einem **Account-** bzw. **Benutzernamen** an und muss anschließend ein **Passwort** eingeben. Ein Einloggen auf einem Unix-/Linux-System kann direkt am Rechner selbst oder aber über das Datennetz erfolgen.



**Abbildung 2.1:** Login an einem Linux-Rechner

Das Login an der Console eines Unix-/Linux-Rechners erfolgt in der Regel über einen grafischen Login-Bildschirm. Obige Abbildung zeigt einen Login-Bildschirm auf einem Linux-Rechner. Über den Menüpunkt *Sitzung* können alternative Benutzeroberflächen eingestellt werden (Gnome, KDE, IceWM). Mit dem Unterpunkt *Terminal (abgesichert)* kann ein Einloggen ohne Benutzeroberfläche erfolgen.

Neben dem direkten Login an der Console eines Unix-/Linux-Rechners gibt es noch verschiedene Möglichkeiten, einen Unix-Rechner über das Datennetz zu nutzen. Dabei wird unterschieden zwischen reinen Terminal-Verbindungen und Verbindungen zum Austausch von Dateien.

## 2.1.1 Login über das Datennetz

Ausgangspunkt für das Login über das Datennetz ist ein lokaler Rechner, auf dem Unix/Linux oder MS-Windows installiert ist. Die Konfiguration des Rechners muss eine fehlerfreie Netz-Kommunikation erlauben (IP-Adresse, Netzmaske, usw.).

### Lokaler Rechner ist ein Unix/Linux-System

Die Programme **slogin** und **ssh** bieten die derzeit bevorzugte Methode, sich von einem lokalen in einen entfernten Unix-Rechner einzuloggen. Dabei läuft die gesamte Datenkommunikation dieser sog. **secure shell** zwischen den Rechnern **verschlüsselt** ab. Der entfernte Rechner (remote host) kann den Zugang erlauben, verbieten oder differenziert regeln. Beim erstmaligen Login wird im lokalen Home-Verzeichnis der sog. *public key* des remote host abgespeichert. Dies erfolgt im Rahmen eines interaktiven Dialogs.

```
host> slogin hostname      Remote Login mittels Rechnernamen
host> slogin ip-address    Remote Login mittels IP-Adresse

host> ssh hostname        Remote Login mittels Rechnernamen
host> ssh ip-address       Remote Login mittels IP-Adresse
```



Abbildung 2.2: Remote Login mittels secure shell

In obigem Beispiel loggt sich ein Benutzer namens `teul4338` mittels `ssh` von dem lokalen Rechner `rciplx2` in den Rechner `rex2` ein (der Rechnername wird hier in der kompletten Schreibweise als `rex2.rz.uni-regensburg.de` angegeben). Da das secure-shell-Login zu diesem Rechner zum ersten Mal erfolgt, muss der public key des Rechners `rex2` im lokalen Home-Verzeichnis des Benutzers abgespeichert werden (in der Datei `.ssh/known_hosts`). Diese Speicherung wird aber erst dann vorgenommen, wenn der Benutzer auf die Frage

Are you sure you want to continue connecting (yes/no)?

mit **yes** antwortet. Dabei ist wichtig, dass die Antwort **yes** ausgeschrieben wird! Die Eingabe von `y` genügt nicht. Das System speichert daraufhin den public key des remote host in der Datei `.ssh/known_hosts` und fordert den Benutzer auf, sein Passwort einzugeben. Beim wiederholten Login in den Rechner `rex2`

ist eine Abspeicherung des public keys nicht mehr notwendig. Nach dem **ssh**-Kommando wird sofort die Eingabe des Passworts verlangt.

Im allgemeinen ändern sich die public keys der entfernten Rechner nicht. Falls dies aufgrund einer Neuinstallation oder nach Datenverlust doch einmal passiert, ist es notwendig, den public key des entsprechenden Rechners aus der Datei `.ssh/known_hosts` zu entfernen. Bei erneuter Verbindungsaufnahme wird dann der neue public key abgespeichert.

Die Kommandos **slogin** und **ssh** lassen sich völlig identisch handhaben (**slogin** ist einfach ein symbolischer Link auf das Programm **ssh**). Beide Kommandos besitzen die gleichen Optionen. Die folgenden Beispiele werden daher nur mit dem Kommando **ssh** wiedergegeben. Statt des langen Rechnernamens wird der kurze Name `rex2` verwendet, was ev. eine nochmalige Abspeicherung des public host keys zur Folge hat. Kurze Rechnernamen können nur innerhalb des Uni-Datennetzes verwendet werden. Für ein secure-shell-Login vom heimischen PC ist die Angabe des langen Rechnernamens erforderlich.

```
host> ssh -l teu14338 rex2 Remote Login zum Rechner rex2 als Benutzer teu14338

host> ssh teu14338@rex2 Remote Login zum Rechner rex2 als Benutzer teu14338

host> ssh rex2 finger Remote Login mit anschließender Ausführung des finger-Kommandos

host> ssh rex2.rz.uni-regensburg.de Remote Login mit full-qualified Internet-Name
```

Letztes Beispiel zeigt, wie für die Adressierung eines Rechners der *full-qualified* Hostname verwendet wird. Dies ist immer dann erforderlich, wenn von außerhalb des Uni-Datennetzes auf den Rechner zugegriffen wird. Mit dem Kommando **exit** wird die secure-shell-Verbindung beendet.

Ein Login von Unix/Linux- zu Unix/Linux-System ist auch mit Programmen wie **telnet**, **rlogin** und **rsh** möglich. Sie stehen in der Regel auf allen Unix/Linux-Systemen zur Verfügung, sollten aber aus Sicherheitsgründen nicht mehr verwendet werden (Passwort wird im Klartext über das Datennetz übertragen).

## Lokaler Rechner ist ein MS-Windows-System

Voraussetzung für ein remote Login von MS-Windows zu einem Unix-/Linux-Rechner ist das Vorhandensein der secure-shell. An allen Rechnern der Universität kann die secure-shell für Windows über RZ-Setup installiert werden. Für heimische Windows-Rechner eignet sich das freie Programmpaket *putty* als secure-shell-Ersatz. Die Benutzung des secure-shell-Clients von Windows aus wird auf den Web-Seiten des Rechenzentrums unter *Webmaster* → *Nutzung und Zugang zentraler Webserver* → *Benutzung des Secure Shell-Client (SSH-Zugang) unter Windows* dokumentiert.

## 2.1.2 Dateitransfer über das Datennetz

Die Übertragung von Dateien zwischen Unix-/Linux-Rechnern sowie zwischen Unix-/Linux-Rechnern und MS-Windows-Rechnern setzt wie beim Login eine Authentifizierung des Benutzers voraus. Die Funktionalität ist aber ganz speziell auf den Transfer von Dateien ausgerichtet. Auf beiden Rechnern müssen die entsprechenden Programme installiert und aktiviert sein.

### Dateitransfer zwischen Unix/Linux-Rechnern

Die Übertragung von Dateien (*files*) zwischen Unix/Linux-Rechnern wird meist mit Programmen wie **scp** und **sftp** durchgeführt. Das **s** im Namen weist darauf hin, dass es sich hier um eine sichere Art der Datenübertragung handelt. Alle Daten werden **verschlüsselt** transferiert. Das Programm **ftp**, das auf jedem Unix-System vorhanden ist, kann zwar auch verwendet werden, allerdings mit dem gleichen Sicherheitsrisiko wie bei den Programmen **telnet** und **rlogin**. Passwörter und Daten werden bei **ftp** **unverschlüsselt** übertragen.

```
host> scp file rechner:
sicheres Kopieren einer Datei zu einem anderen Rechner

host> scp file rechner:subdir
sicheres Kopieren einer Datei zu einem anderen Rechner in ein untergeordnetes Verzeichnis

host> scp file username@rechner:
sicheres Kopieren einer Datei in das Home-Directory eines anderen Benutzers auf einem anderen Rechner

host> scp -r directory rechner:
sicheres, rekursives Kopieren eines kompletten Verzeichnisses zu einem anderen Rechner
```

Beim Kopieren mittels **scp** ist unbedingt darauf zu achten, dass hinter dem Rechnernamen ein **Doppel-****punkt** steht.

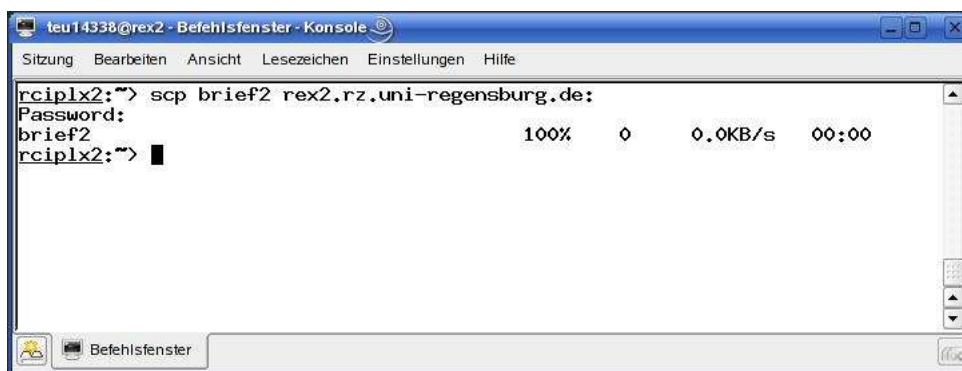


Abbildung 2.3: Sicheres Kopieren über das Datennetz mittels *scp*

In obigem Beispiel wird vom Rechner **rciplx2** eine Datei namens **brief2** zum Rechner **rex2** in das Home-Verzeichnis kopiert. Eine Statuszeile veranschaulicht den Fortschritt der Kopieraktion. Die Abspeicherung des **public-host-keys** ist in diesem Fall nicht mehr notwendig, da ja bereits früher eine **secure-shell**-Verbindung zu dem Rechner aufgebaut wurde.

Werden vom heimischen Linux-Rechner Daten mittels `scp` übertragen, so muss vor dem Rechnernamen der Benutzername stehen, unter dem man an der Uni Regensburg geführt wird.

```
mein-pc> scp protokoll.txt vip12345@rex2.rz.uni-regensburg.de:
```

In obigem Beispiel kopiert ein Benutzer, dessen RZ-Account an der Uni vip12345 heißt, eine Datei `protokoll.txt` von seinem heimischen Linux-PC in sein Home-Verzeichnis auf dem Uni-Rechner `rex2`.

Der Dateitransfer mittels `sftp` funktioniert genauso wie mit dem älteren `ftp`, nur mit dem Unterschied, dass die Daten verschlüsselt übertragen werden.

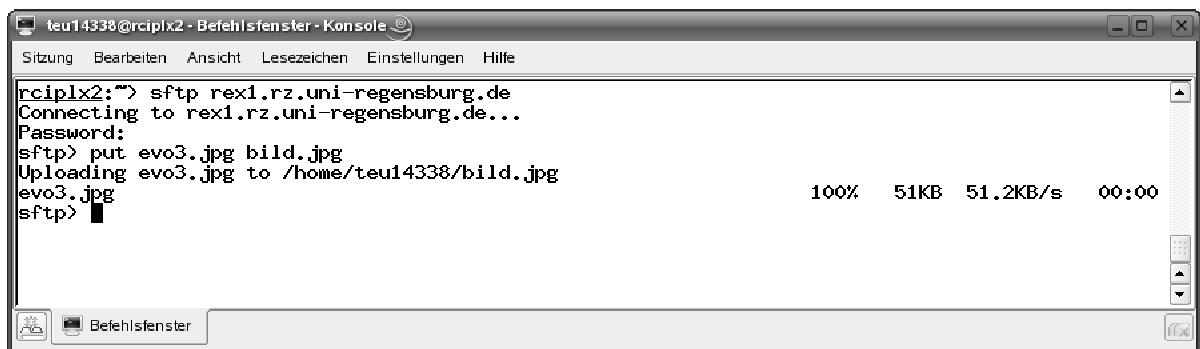


Abbildung 2.4: Sicherer Datentransfer mittels `sftp`

Im vorliegenden Beispiel wird eine Datei namens `evo3.jpg` vom Rechner `rciplx2` zum Rechner `rex1` übertragen. Nach dem `sftp`-Login wird der Transfer mit dem Kommando `put evo3.jpg bild.jpg` angestoßen. Dabei wird die neue Zieldatei mit dem Namen `bild.jpg` abgelegt. Das Kommando `exit` beendet das Programm.

```
sftp> help          gibt Informationen zu sftp aus
sftp> get filename Datei vom remote Rechner holen
sftp> put filename Datei zum remote Rechner schicken
sftp> mkdir directory neues Verzeichnis auf remote Rechner anlegen
```

Weitere Informationen zu `sftp` können mit dem Kommando `man sftp` aus den *online manual pages* abgerufen werden.

### Dateitransfer zwischen Unix/Linux- und MS-Windows-Rechnern

Mit der unter Windows vorhandenen `secure-shell` können auch Dateiübertragungen von und zu Unix-Rechnern vorgenommen werden. Die Benutzung des `secure-shell`-Clients zur Datenübertragung zwischen Linux- und Windows-Rechnern wird auf den Web-Seiten des Rechenzentrums unter *Webmaster* → *Nutzung und Zugang zentraler Webserver* → *Benutzung des Secure Shell-Client (SSH-Zugang) unter Windows* dokumentiert.

Beim Übertragen von Dateien zwischen einem Unix-Rechner und einem Windows-Rechner sollten die entsprechenden Dateinamen-Konventionen beachtet werden. So können in einem Eingabefenster von Windows nur Dateinamen beschränkter Länge abgebildet werden. Auch auf die Groß- und Kleinschreibung muss Rücksicht genommen werden.

## Unix-/Linux-Systeme unterscheiden grundsätzlich zwischen Groß- und Kleinschreibung!

Besonders bei Dateitransfers von Unix/Linux zu Windows ist dies relevant. Hierzu ein Beispiel: Auf Linux liegen zwei Dateien namens `brief.txt` und `Brief.txt` vor, die mittels `secure ftp` nach Windows übertragen werden sollen. Für das Unix-System sind dies zwei verschiedene Dateien. Werden die beiden Dateien hintereinander übertragen, so wird beim zweiten Transfer der Inhalt der ersten Datei einfach überschrieben. Windows kennt bzgl. der Dateinamen keine Unterscheidung zwischen Groß- und Kleinschrift.

Es gibt noch eine weitere Möglichkeit, von Windows auf Dateien von Linux-Systemen zuzugreifen. Ermöglicht wird dies durch einen sog. **samba**-mount zwischen Windows-Client und einem samba-Server, der üblicherweise auf einem Unix-Fileserver installiert ist. Über den Windows-Explorer kann damit ein Linux-Verzeichnis quasi als Windows-Laufwerk zur Verfügung gestellt werden. Die Vorgehensweise wird in der Dokumentation *Benutzung von Samba* auf der Web-Seite des Rechenzentrums unter *Webmaster -> Nutzung und Zugang zentraler Webservers* erläutert.

## 2.2 Passwörter

Benutzer-Accounts auf Unix/Linux-Rechner sind grundsätzlich passwortgeschützt. Passwörter dürfen nicht an andere Personen weitergegeben werden. Ausserdem ist darauf zu achten, dass das Passwort in bestimmten Zeitabständen geändert wird.

Die Passwortverwaltung an der Uni Regensburg unterscheidet sich von der auf heimischen Linux-Rechnern. Die Benutzerverwaltung an der Uni Regensburg wird zentral durch ein sog. Novell-eDirectory realisiert. Das bedeutet, dass jeder Benutzer immer das gleiche Passwort verwendet, gleich ob er sich auf einen Windows- oder Linux-Rechner einloggt.

Bei der Wahl eines Passworts sollten folgende Regeln beachtet werden.

- Das Passwort muss mindestens 8 Zeichen haben und sollte nicht länger als 15 Zeichen sein (um auf allen Systemen zu funktionieren).
- Es darf keine Umlaute, keine Leerstellen und nur kodierungssichere Sonderzeichen ( - \$ # [ ] ! ( ) . , \* : ; \_ ) enthalten.
- Es muß aus Buchstaben UND Zahlen bestehen (und evtl. den o.g. Sonderzeichen).
- Das Passwort darf nicht Ihr RZ-Account Name oder leer sein.
- Es muss sich genügend vom alten Passwort unterscheiden (d.h. keine 3 aufeinanderfolgenden Zeichen des alten Passworts dürfen im neuen Passwort vorkommen).
- Sie können kein Passwort setzen, das Sie in den letzten 2 Jahren (für mindestens 1 Tag) schon einmal verwendet haben.

Das Ändern des Passworts sollte grundsätzlich über das entsprechende Web-Formular erfolgen, zu finden über die RZ-Startseite -> *Benutzer* -> *Allgemeine Dienstleistungen* -> *Passwörter ändern* -> *RZ Passwort-änderung*.

Nach einer kurzen Zeit ist das neue Passwort auf allen Rechnern der Universität gültig. Diesen Hinweis **unbedingt beachten!** Nach einer Passwortänderung wird die Modifikation auf alle eDirectory-Server repliziert. Dieser Replikations-Vorgang benötigt natürlich eine gewisse Zeit, die erst verstreichen muss, bevor der Benutzer sich erneut in einen Linux-Rechner einloggen kann. Eine 5-minütige Wartezeit sollte daher eingehalten werden.

## 2.3 Benutzeroberflächen und Benutzerumgebung

Graphische Benutzeroberflächen dienen dazu, die Interaktion zwischen Benutzer und Rechner möglichst einfach zu gestalten. Alle Aktionen sollen leicht intuitiv erfasst werden können. Grundsätzlich kann jedes Unix/Linux-System auch ohne graphische Benutzeroberfläche betrieben werden. Auf Server-Systemen wird sogar bewusst darauf verzichtet, da dies nur unnütz Plattenplatz und Rechenzeit kosten würde.

Alle graphischen Benutzeroberflächen unter Unix/Linux basieren auf dem system- und netzwerk-unabhängigen **X Window System**.

Der Ursprung des X Window Systems liegt in zwei Projekten der frühen 80er Jahre: ein Window-Projekt der Stanford Universität, bekannt als *W* und ein Projekt *Athena* des Massachusetts Institute of Technology (MIT). 1983 gegründet und von MIT, Digital und IBM finanziert, hatte das Projekt *Athena* u.a. die Zielsetzung, ein netzwerkunabhängiges Graphikprotokoll innerhalb der bunt gemischten Rechnerumgebung der Universität zu entwickeln. Die MIT-Gruppe nahm *W* in ihr Entwicklungssystem auf und gab ihm den zufälligen Namen **X Window System**. 1986 bot MIT die erste kommerzielle Version des X Window Systems (X10) an und bildete 1987 eine breitere Industriepartnerschaft, das X Konsortium, um eine kontinuierliche Entwicklung zu gewährleisten. Konsortiumsgründungsmitglieder waren u.a. Tektronix, IBM, Sun, HP und AT&T.

Das X Window System ist ein Softwarestandard, der einen Rahmen für fensterbetriebene Anwendungen über das Netzwerk bietet. Es handelt als eine Art Mittelsmann zwischen einer Anwendung und einem Darstellungssystem (X Terminal, Workstation, PC). Bei einer Benutzeroberfläche, die auf dem X Window System basiert, spielen grundsätzlich drei Komponenten zusammen: X-Server, X-Client und Windowmanager.

Der **X-Server** läuft als eigener Prozess auf dem Darstellungssystem und verarbeitet den Input von Geräten wie Tastatur und Maus. Auch der Input vom X-Client wird für die Darstellung verwendet.

Der **X-Client** ist die eigentliche Anwendung (z.B. *xclock*, *xterm*, *xmapple*) und kommuniziert mit dem X-Server über das standardisierte X11-Protokoll.

Der **Windowmanager** ist verantwortlich für Fensterrahmen, Verschieben, Vergrößern und Verkleinern und Iconisieren von Fenstern. Je nach verwendetem Windowmanager sieht das Layout auf dem Desktop etwas unterschiedlich aus. Das sog. "Look and Feel" wird also vom Windowmanager bestimmt. Bekannte Windowmanager sind z.B. **twm**, **mwm** (Motif Window Manager), **fvwm** und **dtwm** (der Windowmanager unter CDE). Bei Linux-Systemen sind außerdem die Windowmanager **kwin** unter KDE und **icewm-gnome** unter Gnome weit verbreitet.

Durch die Aufteilung der beiden Funktionen Anwendung und Darstellung ist es möglich, X-Client und X-Server auf unterschiedlichen Rechnern ablaufen zu lassen. So kann der X-Client z.B. auf einer Sun Workstation laufen, als Darstellungssystem wird aber ein PC mit X-Server-Software verwendet. Dadurch kann die gleiche Benutzeroberfläche, die man unmittelbar an der Workstation zur Verfügung hat, auch auf dem PC verwendet werden.

Für die individuelle Gestaltung des Desktops kommen grundsätzlich zwei Möglichkeiten in Frage: Das interaktive Verändern des Layouts mit anschließendem Abspeichern und das Modifizieren von speziellen Dateien. Je nachdem, welcher Windowmanager eingestellt ist, muss entweder nach der ersten oder der zweiten Methode vorgegangen werden.

Erfolgte das Login unter CDE, KDE oder Gnome, so können mittels kleiner Hilfsprogramme (StyleManager, ControlCenter) vielfältige Modifikationen des Desktops interaktiv vorgenommen werden (Hintergrundfarbe, Fensterrahmen, Mausverhalten usw.). Beim Ausloggen wird dann das aktuell vorhandene Desktop-Layout automatisch abgespeichert.

Etwas schwieriger stellt sich die Modifikation des Layouts unter den Windowmanagern twm und fvwm dar. Für eine individuelle Gestaltung müssen hier diverse Dateien editiert werden. Im einzelnen handelt es sich um die Dateien `.xinitrc`, `.twmrc` bzw. `.fvwmrc` und `.Xdefaults` bzw. `.Xresources`. Derartige Modifikationen erfordern eine gewisse Erfahrung im Umgang mit dem X Window System. Gute Hilfen bieten hier die entsprechenden Dokumentationen und die online manual-pages.

Im folgenden wird eine kurze Übersicht über die drei am häufigsten verwendeten Desktop-Systeme KDE, Gnome und CDE gegeben.

## KDE-Desktop

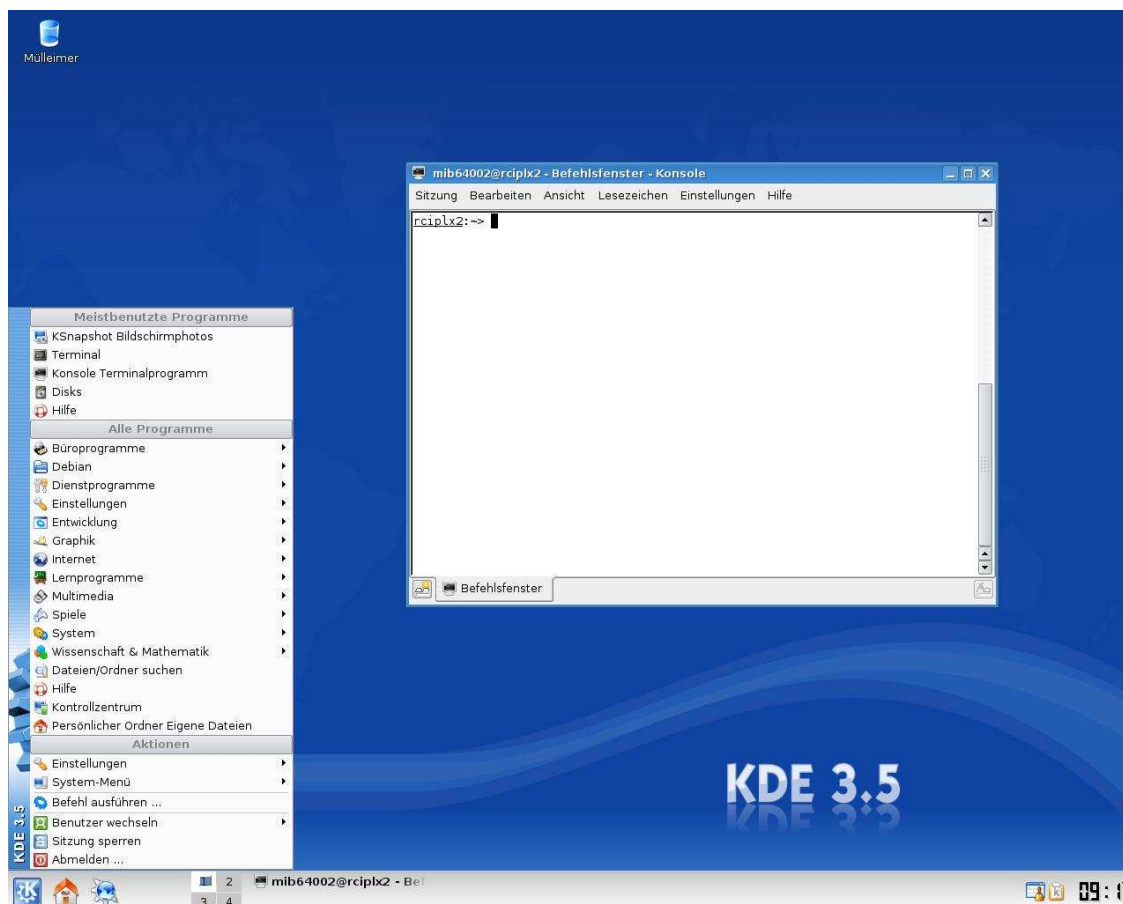
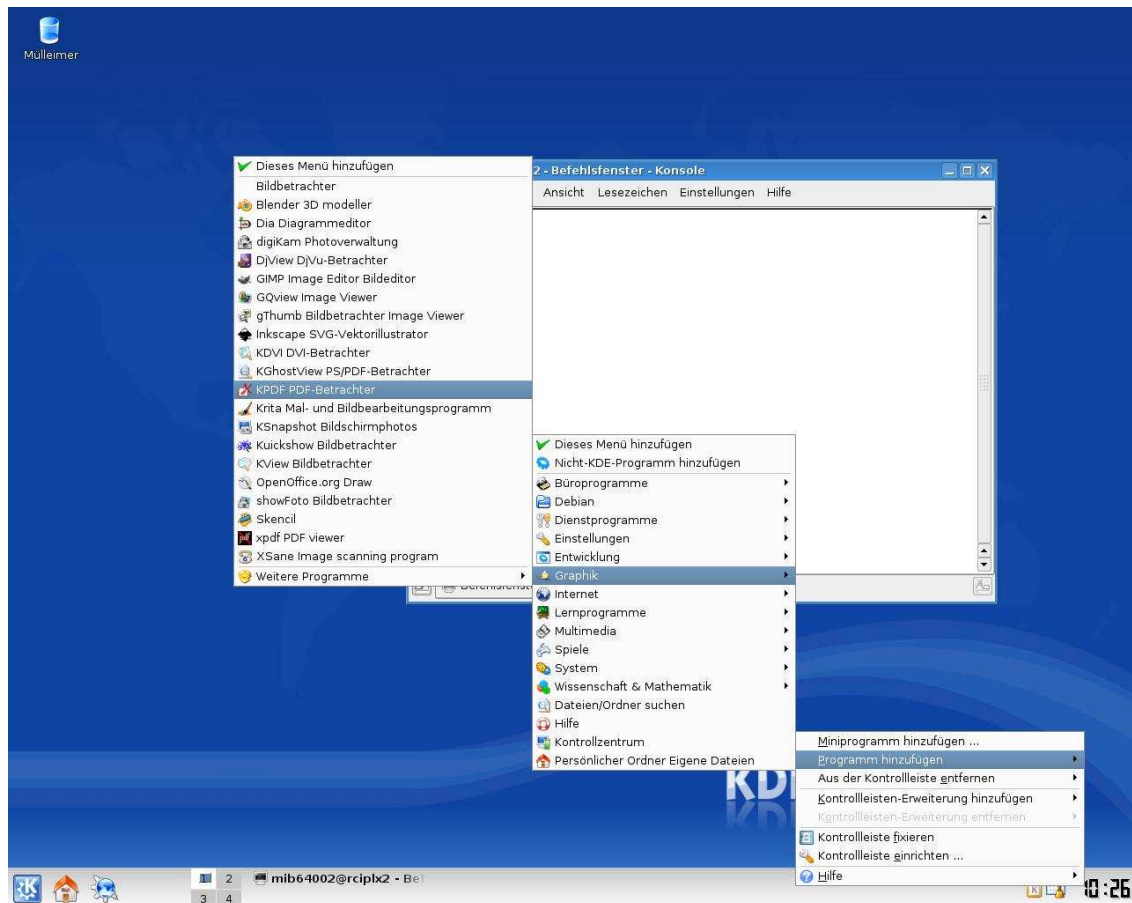


Abbildung 2.5: KDE-Desktop mit Anwendungsmenü und Eingabefenster

KDE ist in jeder Linux-Distribution enthalten und bietet eine Fülle von Anwenderprogrammen. Augenfälliges Merkmal des Desktops ist der *Session-Manager* an der Unterkante des Bildschirms. Dieser Session-Manager ist quasi die Schaltzentrale für den Desktop. Der Session-Manager besitzt standardmäßig vier Schaltknöpfe (1 - 4) für die Wahl des virtuellen Bildschirms. Links daneben sind drei Icons zu sehen. Mit einem einfachen Mausklick auf das Haus-Icon startet der Dateimanager *konqueror* und zeigt den Inhalt des eigenen Home-Verzeichnisses an. Ein Klick auf die Weltkugel startet den *konqueror* als allgemeinen File- und Web-Browser. Ganz links in der Ecke ist der Schaltknopf für das Aufklappen des Anwendungs-Menüs untergebracht. Darunter befindet sich auch die Funktion *Abmelden*, mit der ein Logout initiiert wird.

Der Kontrollleiste (Session-Manager) können zusätzliche Funktionen hinzugefügt werden. Dazu genügt ein rechter Mausklick auf die Kontrollleiste. Es öffnet sich ein kleines Menüfenster, aus dem man dann den Unterpunkt *Programm hinzufügen* auswählen kann. Daraufhin öffnet sich das bereits bekannte Anwendungs Menü, aus dessen Kategorien man sich die gewünschte Anwendung heraussucht.



**Abbildung 2.6:** Auswahl von neuen Start-Icons für die KDE-Kontrollleiste

In obigem Beispiel wird aus der Kategorie *Graphik* der *KPDF PDF-Betrachter* ausgewählt. Nach dieser Auswahl erscheint in der Kontrollleiste ein neues Icon, über das diese Anwendung nun mit einem einfachen Mausklick gestartet werden kann.



**Abbildung 2.7:** PDF-Betrachter als neues Start-Icons in der KDE-Kontrollleiste

Durch Rechtsklick auf das neue Icon kann es innerhalb der Kontrollleiste verschoben werden. Neben Anwendungsprogrammen sind in die Kontrollleiste auch noch sog. Miniprogramme integrierbar (z.B. Auslastungsanzeige). Enthält ein Element in der Kontrollleiste einen kleinen schwarzen Pfeil am oberen Rand, so bedeutet dies, dass sich hinter diesem Element mehrere Funktionen verbergen (z.B. das K-Startmenü in der linken unteren Ecke). Wird das Fenster einer Anwendung iconisiert, so wird das Icon in der Kontroll-

leiste abgelegt. Eine Reaktivierung erfolgt dann durch einen einfachen Mausklick auf das Icon. Über das K-Startmenü kann das *Kontrollzentrum* aufgerufen werden. Das Kontrollzentrum ermöglicht eine Modifizierung des KDE-Desktops. Damit können beispielsweise Schriften, Farben, Hintergründe und das Verhalten von Maus und Tastatur den individuellen Anforderungen angepaßt werden.

## Gnome-Desktop

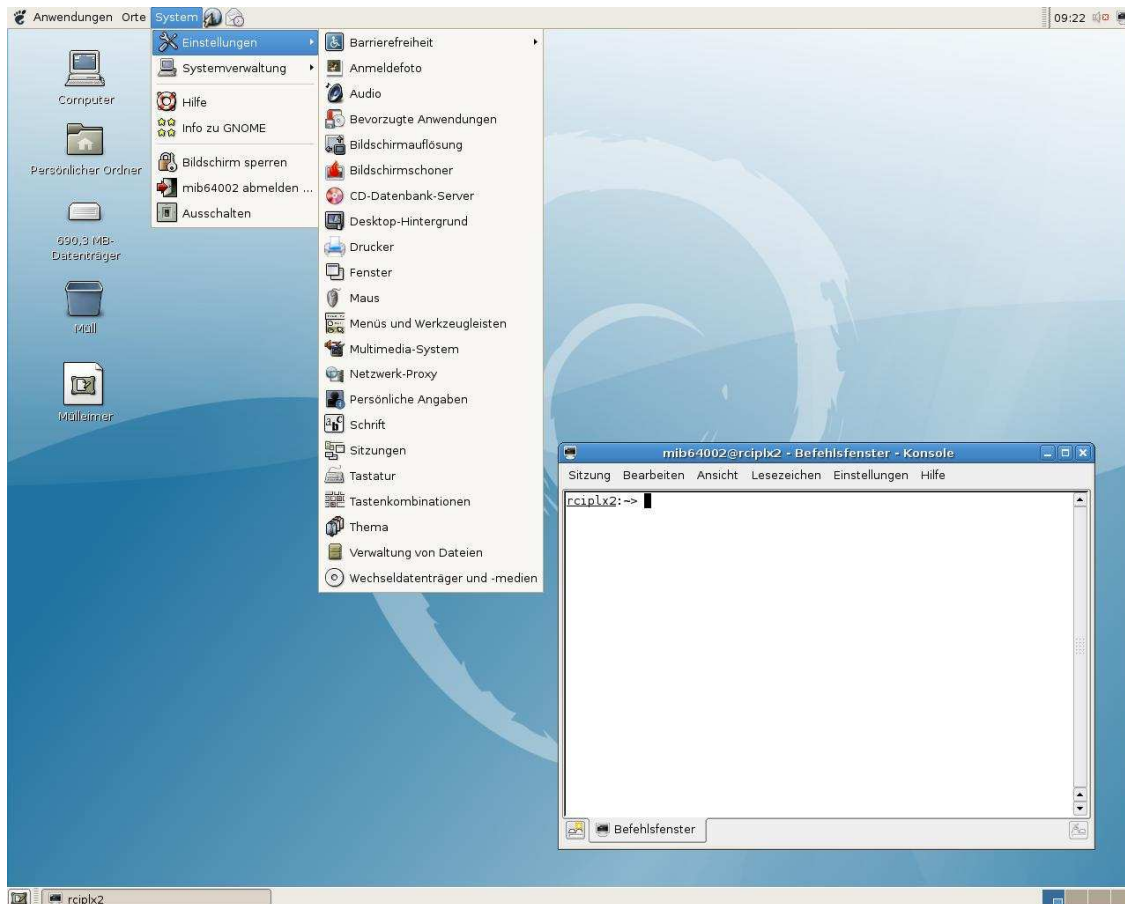
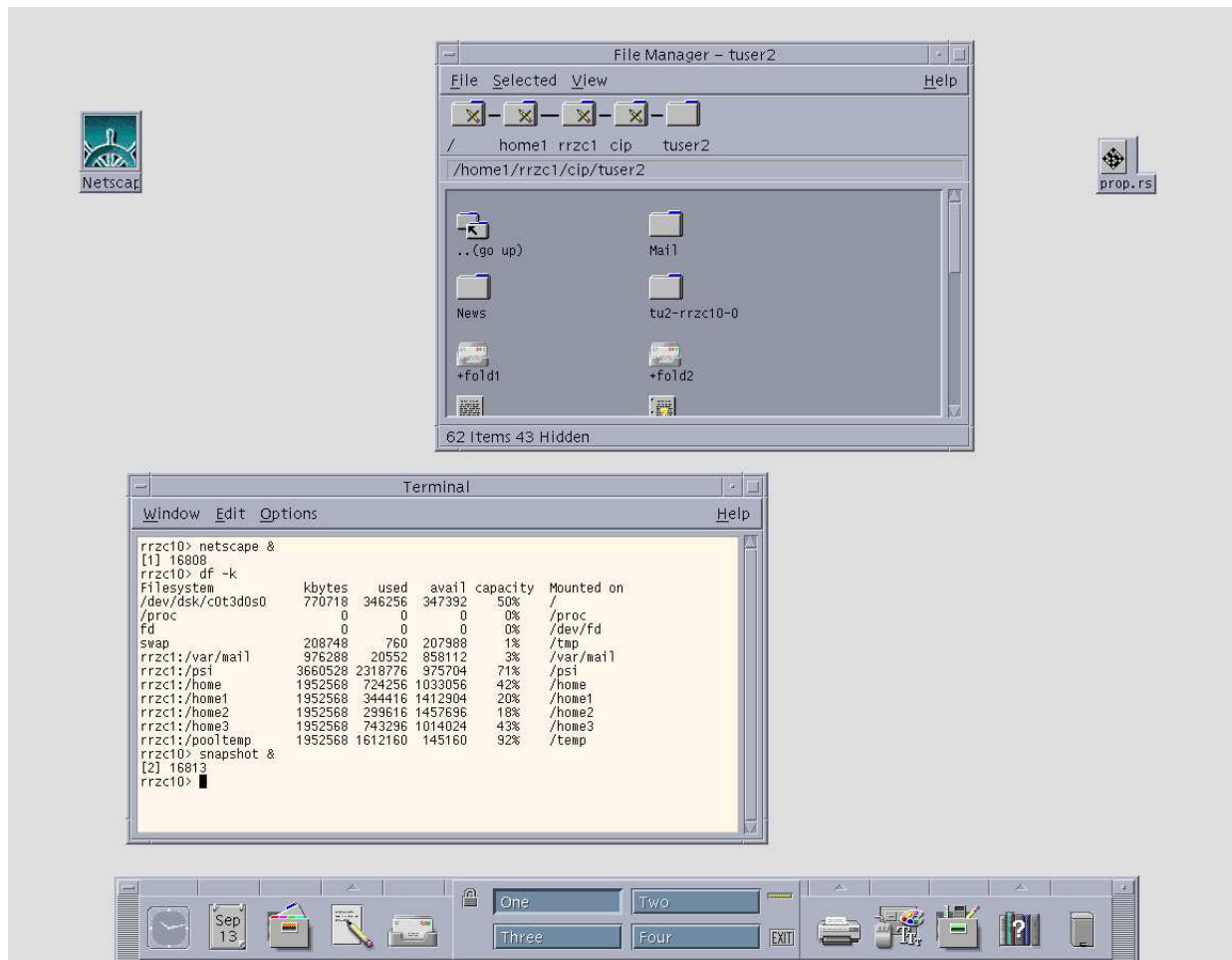


Abbildung 2.8: Gnome-Desktop mit Systemmenü und Eingabefenster

Auch unter dem Gnome-Desktop gibt es in der Standard-Konfiguration einen Session-Manager am unteren Bildschirmrand. Dieses "Panel" enthält vier virtuelle Bildschirme in der rechten unteren Ecke und nimmt die Icons von Anwendungsfenstern auf. Zusätzlich befindet sich am oberen Bildschirmrand eine weitere Leiste. In dieser Leiste lassen sich zum einen diverse Popup-Menüs aufrufen (*Anwendungen*, *Orte*, *System*), zum anderen befinden sich dort auch Schaltknöpfe für die Aktivierung von Anwendungen. Auf der restlichen freien Bildschirmfläche werden gewöhnlich die grafischen Anwendungen plazierte. In obigem Beispiel ist dies eine *Befehlsfenster-Konsole*. Wie in dem Bild zu sehen, können Aussehen und Funktion des Gnome-Desktops über das *System*-Menü Unterpunkt *Einstellungen* modifiziert werden (Maus- und Tastaturverhalten, Hintergrund usw.). Wie bei KDE kann auch unter Gnome der Funktionsumfang der Kontrollleisten erweitert werden. Dies erfolgt durch Rechtsklick auf eine Kontrollleiste und anschließender Auswahl der Funktion *Zum Panel hinzufügen*.

## CDE-Desktop



**Abbildung 2.9:** CDE-Desktop mit Dateimanager und Eingabefenster

Die CDE-Oberfläche (Common Desktop Environment) ist eine grafische Benutzeroberfläche, die architekturunabhängig auf Unix-Rechnern von Sun, HP, IBM und Digital zur Verfügung steht. Wie alle anderen Benutzeroberflächen, basiert auch CDE auf dem X11-Protokoll-Standard. Kennzeichen der CDE-Oberfläche ist wie auch bei KDE und Gnome ein Session-Manager am unteren Bildschirmrand. Dieser Session-Manager besitzt eine Reihe von Grundfunktionen, die individuell noch erweitert werden können. Befindet sich oberhalb eines Symbols im Session-Manager ein kleiner Pfeil, so kann hier ein Popup-Menü mit weiteren Funktionen aktiviert werden (in obigem Bild z.B. über dem Editor-Symbol). Eine gute Einführung in die Handhabung des Desktops erhält man über das Symbol Desktop-Help im Session-Manager-Window (das Symbol mit dem Fragezeichen).

Die drei gezeigten Desktop-Varianten dürften wohl den Hauptteil aller weltweit eingesetzten grafischen Benutzeroberflächen auf Unix/Linux-Systemen ausmachen. Als gemeinsames Element tritt hierbei die Kontrollleiste bzw. der Session-Manager hervor. Auch im gesamten läßt sich eine Ähnlichkeit in der Handhabung feststellen, was das abwechselnde Arbeiten auf verschiedenen Systemen vereinfacht. Die Wahl eines bestimmten Desktops bzw. Desktop-Layouts wird sich sicherlich aus dem persönlichen Geschmackempfinden des Benutzers ergeben. Die Funktionalität ist überall gleich.

## 2.4 Der grafische Texteditor gedit

Als Beispiel einer grafischen Anwendung soll hier der Texteditor **gedit** dargestellt werden. Dieser Editor ist kein Programm zur Textverarbeitung (wie z.B. OpenOffice oder Winword), sondern ein nützliches Werkzeug für die Erstellung von einfachen Textdateien.

Für das Starten des Editors genügt das Kommando **gedit** in einem Eingabefenster oder die Aktivierung über das KDE-Startmenü, Unterpunkte *Dienstprogramme* → *Editoren* → *Texteditor*. Je nach Desktop-Version kann sich der Editor auch in einem anderen Untermenü befinden.

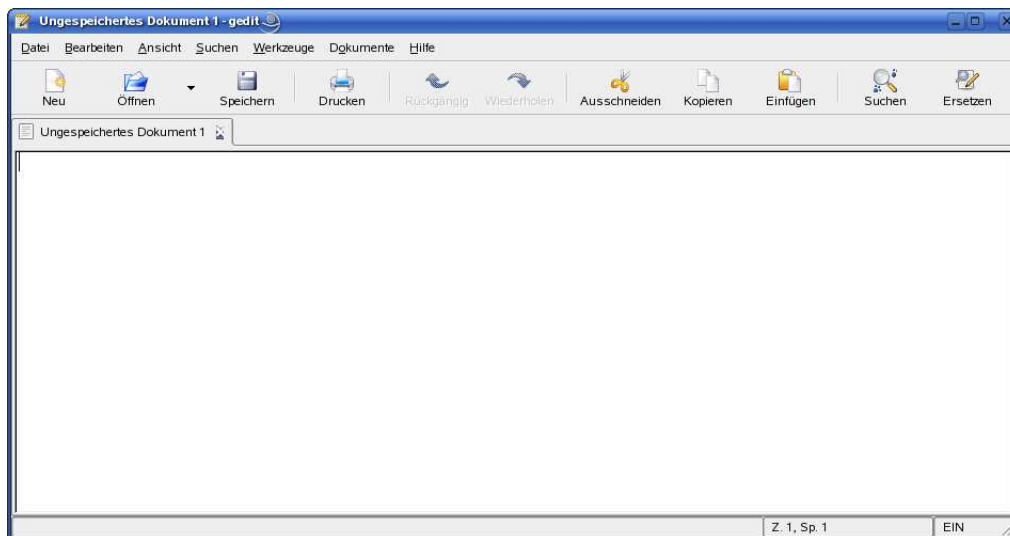


Abbildung 2.10: Der Texteditor gedit

Das grafische Editorfenster gliedert sich vertikal in fünf Bereiche. Ganz oben stehen die Begriffe für die Wahl der entsprechenden Auswahlmensüs (Datei, Bearbeiten, usw.), in der nächsten Leiste sind einige wichtige Funktionen zur Schnellwahl abgelegt, der dritte Bereich umfasst in Form von Reitern die geöffneten Dokumente. Den größten Bereich bildet die Texteingabefläche. Hier können nun beliebige Texteingaben erfolgen. Den Abschluß bildet eine Statuszeile, in der immer die aktuelle Cursorposition im Text angezeigt wird.

Beim Speichern des geschriebenen Textes über das Menü *Datei* → *Speichern unter* hat man die Möglichkeit, die Kodierung anzugeben. Je nachdem, welche Desktop-Umgebung vorhanden ist, wird entweder ISO- oder UTF-8-Kodierung voreingestellt. Bei vorliegendem Beispiel wird der Text in UTF-8 abgespeichert.

Mit dem Editor können auch mehrere Dateien gleichzeitig bearbeitet werden. Der Wechsel von einer zur anderen Datei wird durch die Aktivierung des entsprechenden Reiters durchgeführt.

Um den Editor individuell anzupassen, muss man aus dem Menü *Bearbeiten* den Unterpunkt *Einstellungen* auswählen. Hier können nun Zeilenumbruch, Schrift, Farbe und auch die Syntax-Hervorhebung für spezielle Texte definiert werden.



Abbildung 2.11: Wahl der Kodierung

## 2.5 Unix-Dokumentation

Der Nutzen jeder Software hängt entscheidend von der Qualität der verfügbaren Software-Dokumentation ab. Die Dokumentation von Unix/Linux-Betriebssystemen kann in unterschiedlichem Umfang und unterschiedlicher Form erfolgen. Neben der Unix-Dokumentation in Buchform (je nach System 10 - 20 Ordner) stehen dem Benutzer auch on-line-Hilfen zur Verfügung.

- **Manual-Pages** geben einen kompakten Überblick über die wichtigsten Kommandos und Themen.
- **Desktop-Hilfen** bieten eine Art Einführung zur Handhabung der graphischen Benutzeroberfläche.
- **Hypertext-Systeme** stellen die umfassendste Dokumentation des Systems dar. Die gesamte System-Dokumentation ist als Web-Dokument verfügbar.

Die klassische Form der Dokumentation bilden die **Manual-Pages**. Sie sind in jeder System-Distribution vorhanden und gliedern sich thematisch in acht Teile (sections):

- 1 User Commands
- 2 System Calls
- 3 Subroutines
- 4 Special files
- 5 File formats, conventions
- 6 Games
- 7 Macro packages and language conventions
- 8 Maintenance, commands and procedures

Innerhalb dieser acht Sektionen wird eine alphabetische Reihenfolge eingehalten. Die Dokumentation der einzelnen Themen ist maximal komprimiert.

Ein Dokumentationseintrag besteht meist aus folgenden Teilen:

NAME	Name und Kurzbeschreibung des Themas
SYNOPSIS	Syntax des Kommandos mit Optionen und Argumenten
DESCRIPTION	”Detaillierte Beschreibung”
OPTIONS	Beschreibung aller wählbaren Optionen
FILES	mit dem Thema zusammenhängende Dateien
SEE ALSO	Hinweise auf verwandte Themen
DIAGNOSTICS	Erklärung von stark verkürzten Fehlermeldungen
BUGS	bekannte Fehler
EXAMPLES	Verwendungsbeispiele (nicht immer vorhanden)

Die Anzahl und Art der Unterpunkte eines Dokumentationseintrages können von Fall zu Fall unterschiedlich sein. So kann durchaus einmal der Abschnitt `DIAGNOSTICS` fehlen, während dafür Punkte wie `USAGE` und `NOTES` erwähnt werden. Auch der Umfang der `DESCRIPTION` kann sehr verschieden sein. Oft ist die `DESCRIPTION` ausreichend, manchmal wünscht man sich aber eine ausführlichere Beschreibung.

Der wichtigste Teil der Doku-Einträge ist die Synopsis, wobei bei Kommandos folgendes gilt:

kommando	ist wörtlich zu übernehmen
[...]	alles in eckigen Klammern ist optional
file	Dateiname
...	eine Wiederholung des vorherigen Arguments ist möglich

Die Einträge der Manual-Pages können mit dem Kommando **man kommando** abgerufen werden. Eine Einführung in die Handhabung des man-Kommandos erhält man durch den Befehl **man man**.

```
host> man ls

User Commands                                                    ls(1)

NAME
    ls - list contents of directory

SYNOPSIS
    /usr/bin/ls [-aAbcCdFghilMnoprRstuxl@] [file...]

    /usr/xpg4/bin/ls [-aAbcCdFghilMnoprRstuxl@] [file...]

DESCRIPTION
    For each file that is a directory, ls lists the contents of
    the directory. For each file that is an ordinary file, ls
    repeats its name and any other information requested. The
    output is sorted alphabetically by default. When no argument
    is given, the current directory is listed. When several
    arguments are given, the arguments are first sorted
    appropriately, but file arguments appear before directories
    and their contents.

    ...
```

Obiges Listing zeigt den Bildschirm-Output des Kommandos **man ls** auf einem Unix-System (Sun).

Das gleiche Kommando auf einem Linux-System ausgeführt zeigt folgenden Output.

```
host> man ls

VDIR(1)                  FSF                  VDIR(1)

NAME
    ls - zeigt Verzeichnisinhalt an

ÜBERSICHT
    ls [OPTION]... [DATEI]...

BESCHREIBUNG
    Auflistung von Informationen der DATEIen (Standardvorgabe
    ist das momentane Verzeichnis). Alphabetisches Sortieren
    der Einträge, falls weder -cftuSUX noch --sort angegeben.

    -a, --all
        Einträge, die mit . beginnen, nicht verstecken.

    -A, --almost-all
        Keine Anzeige implizierter . und ..

    ...
```

Da auf Linux-Rechnern meist "Deutsch" als Sprachumgebung eingestellt ist, wird bei den manual-pages auch deutscher Text angezeigt. Aber noch andere Unterschiede zeigen sich. Der Beschreibungstext bei der Linux-Dokumentation ist sehr kurz, während unter Unix doch eine ausführlichere Beschreibung des Kommandos wiedergegeben wird. Oft wird man auch feststellen können, dass sich die Optionen für das gleiche Kommando auf den verschiedenen Unix- bzw. Linux-Varianten etwas unterscheiden.

Zwei weitere Anwendungsmöglichkeiten des **man**-Kommandos sind die Schlüsselwortsuche und der gezielte Zugriff auf Themen in speziellen Sektionen.

```
host> man -k key
Schlüsselwortsuche in den manual-pages; als key kann jeder beliebige Begriff
verwendet werden, der in irgendeiner Weise in der Dokumentation steht.

host> man -f key
Konkrete Suche nach einem bestimmten Begriff (key); es werden aus den
kompletten manual-pages nur die Einträge herausgesucht, die dem gesuchten
Begriff korrekt entsprechen.

host> man -k mount
Alle mit mount zusammenhängende Themen samt Kurzbeschreibung werden
aufgelistet.

host> man -f mount
Es werden nur Einträge, die den korrekten Begriff mount im NAME-Feld
besitzen, angezeigt.
```

Über eine Schlüsselwortsuche können zu einem bestimmten Begriff manchmal mehrere Einträge erscheinen. Hinter den Einträgen stehen in Klammern die Sektionen, in denen die Begriffe dokumentiert werden. Mit Hilfe dieser Sektionsnummern kann dann gezielt auf den gewünschten Dokumentationseintrag zugegriffen werden. Unter Linux erfolgt dies mit der Option **-s** und unter Sun Solaris mit der Option **-s**.

```
host> man -f autofs

autofs (8)          - Control Script for automounter
autofs (5)          - Format of the automounter maps

host> man -s 8 autofs
host> man -s 5 autofs
```

Die Manual-Pages stellen die grundlegende online-Dokumentation auf jedem Unix/Linux-System dar. Trotz der manchmal etwas knappen Beschreibung sind sie eine gute Hilfe bei der täglichen Arbeit. Eine Vertiefung in ein spezielles Thema erlauben die Manual-Pages allerdings nicht. Umfangreichere Dokumentationen bieten entweder gedruckte Medien oder sog. Dokumentations-Server. Aufgrund der besseren Aktualität sind Dokumentations-Server vorzuziehen. Sie enthalten die komplette gedruckte Dokumentation und unterstützen die themenbezogene Suche. Eine gute Dokumentation von Sun Solaris ist unter der Web-Adresse [docs.sun.com](http://docs.sun.com) zu finden. Zahlreiche Linux-Dokumentationen sind nicht nur als gedruckte Bücher im Buchhandel, sondern auch als Web-Dokumente im Internet vertreten. Als Beispiel soll hier die Adresse [www.tldp.org/guides.html](http://www.tldp.org/guides.html) genannt werden.

## 2.6 Spezialzeichen und reguläre Ausdrücke

Beim interaktiven Arbeiten auf Kommandozeilenebene gibt es eine Reihe von Zeichen, die für die Interpretation der Eingabe eine spezielle Bedeutung haben.

*	expandiert zu einem beliebigen String
?	expandiert zu genau einem Zeichen
[...]	expandiert zu genau einem Zeichen aus der angegebenen Zeichenliste
~	expandiert zum HOME-Directory
{...}	expandiert zu den angegebenen Strings
\$	Zeichen für die Referenz auf eine shell-Variable
\	maskiert das nachfolgende Sonderzeichen, Fortsetzung einer Zeile
'...'	maskiert den in Hochkommas eingeschlossenen Text
"..."	maskiert den in den doppelten Hochkommas eingeschlossenen Text
	ausser den Zeichen \$ \ `

Unter der *Maskierung* von Zeichen versteht man das Verbergen vor einer Interpretation seitens der login-shell. Dies ist vor allem dann erforderlich, wenn eine Zeichenkette (string) mit Sonderzeichen an ein externes Kommando weitergegeben werden soll.

Einige der oben erwähnten Spezialzeichen werden auch bei der Beschreibung *regulärer Ausdrücke* verwendet. Ein *regulärer Ausdruck* ist eine Folge von Textzeichen und/oder Spezialzeichen. Die Spezialzeichen stellen Operatoren dar, mit deren Hilfe komplexe Textmuster beschrieben werden können. Derartige Muster können in verschiedenen Kommandos als Textfilter verwendet werden.

Die Operatoren in regulären Ausdrücken haben folgende Bedeutung:

.	Platzhalter für ein beliebiges Zeichen
[...]	Platzhalter für genau ein Zeichen aus der angegebenen Menge
a*	a kann nullmal oder öfter hintereinander vorkommen
a?	a kann nullmal oder einmal vorkommen
a+	a kann einmal oder öfter hintereinander vorkommen
^	Vorkommen am Zeilenanfang
\$	Vorkommen am Zeilenende
	oder-Verknüpfung

Reguläre Ausdrücke werden bei der Verwendung mit den Kommandos **grep** und **egrep** in einfache Hochkommas eingeschlossen, bei der Verwendung mit **awk** stehen sie zwischen Schrägstrichen (/).

```
host> egrep '^...[al]' /etc/postfix/main.cf
```

```
relayhost = mailhub.uni-regensburg.de
alias_maps = hash:/etc/aliases
alias_database = hash:/etc/aliases
mailbox_command =
mailbox_size_limit = 0
```

In obigem Beispiel werden mit dem Kommando **egrep** aus der Datei `/etc/postfix/main.cf` alle Zeilen herausgefiltert, die dem angegebenen Textmuster entsprechen (die ersten drei Zeichen können beliebig aussehen, das vierte Zeichen darf aus einem `a` oder `l` bestehen und das Muster muss am Zeilenanfang erscheinen).

Für die Interaktion auf Kommandozeilenebene sind noch einige Control-Tastensequenzen relevant.

<Ctrl> u löscht den Text in der aktuellen Eingabezeile

<Ctrl> c bricht das laufende Programm ab

<Ctrl> z suspendiert ein laufendes Programm

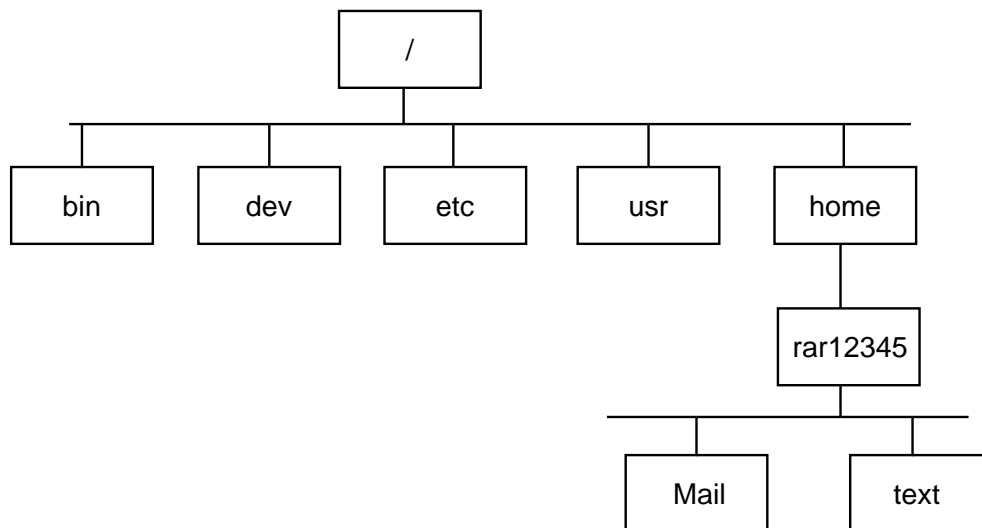
<Ctrl> d beendet eine Eingabe auf Standard-Input.

Die Eingabe von q beendet bei Auflistungen durch die Kommandos **man** und **more** die Ausgabe.



### 3 Das Dateisystem

Das Dateisystem unter Unix/Linux ist hierarchisch gegliedert. Ausgehend vom *root directory /* verzweigen sich Unterverzeichnisse mit System- und Benutzerdaten. Unter einem *Dateisystem* wird grundsätzlich eine komplette Hierarchie von Verzeichnissen und Dateien auf einer Festplatte oder einem Festplatten-Array verstanden. Meist kommen heute Festplatten-Arrays zum Einsatz, die aufgrund ihrer Konfiguration (RAID) eine hohe Fehlertoleranz aufweisen.



**Abbildung 3.1:** Hierarchisches Dateisystem

Im Verzeichnis `/etc` stehen u.a. Konfigurationsdateien für das System, das Directory `/dev` enthält spezielle Dateien für die Kommunikation mit Geräten und im Verzeichnis `/bin` stehen ausführbare Programme. Benutzerdaten werden in der Regel auf gesonderten Platten abgelegt (`/home`). Die Informationen über die Struktur des gesamten Dateisystems ist auf jeder Platte in speziellen Blöcken und in den sog. I-Node-Tabellen gespeichert.

Jedes Home-Verzeichnis ist ein Unterverzeichnis in einem großen Dateisystem (in obigem Beispiel wäre dies `/home/rar12345` auf dem Dateisystem `/home`). Alle Unix/Linux-Benutzer an der Uni Regensburg können aber noch weitere Verzeichnisse in anderen Dateisystemen besitzen, die sie zur Speicherung ihrer Daten verwenden können. So kann jeder Benutzer unter dem Dateisystem `/temp` ein Unterverzeichnis (Unterordner) erstellen für die Ablage temporärer Daten. Für die Speicherung wichtiger Daten ist aber das reguläre Home-Verzeichnis vorgesehen, das bei jedem Login voreingestellt ist. Jeder Benutzer kann in seinem Home-Verzeichnis bis zu einer Maximal-Grenze (*diskquota*) Daten ablegen. Die Daten im Homeverzeichnis werden täglich gesichert und können vom Benutzer selbst restauriert werden. Die Durchführung einer Daten-Wiederherstellung wird auf den Web-Seiten des Rechenzentrums erläutert.

### 3.1 Dateiartern und Dateinamen

Unter Unix existieren drei Dateiartern:

Directories	d	Inhaltsverzeichnisse
Normale Dateien	-	können Daten aller Art enthalten
Spezielle Dateien	c	charakter-orientierte Gerätedateien
	b	block-orientierte Gerätedateien
	p	named pipes für die Prozesskommunikation

Die mittlere Spalte zeigt für den entsprechenden Dateityp den Kennbuchstaben, der bei einem ausführlichen Directory-Listing mit dem Kommando `ls -l` ausgegeben wird.

```
host> ls -l

-rw-r----- 1 rar12345 stud      19 Sep  9 08:56 else
-rw-r----- 2 rar12345 stud      63 Sep  9 08:58 franz
-rw-r----- 1 rar12345 stud         0 Sep  9 08:55 hugo
-rw-r----- 2 rar12345 stud      63 Sep  9 08:58 ina
lrwxrwxrwx  1 rar12345 stud         4 Sep  9 08:59 sara -> else
drwxr-x--x  2 rar12345 stud     117 Sep  9 09:27 text
```

Die einzelnen Spalten der Ausgabe haben folgende Bedeutung:

```
-rw-r-----  das erste Bit dokumentiert den Typ der Datei, in den neun
                folgenden Bits ist der Zugriffsschutz gespeichert
                2  Anzahl der Hardlinks auf die Datei
rar12345      Datei-Eigentümer
stud         Zuordnung zu einer Gruppe (i.d.R. die Login-group)
63           Größe der Datei in Bytes
Sep  9 08:58 Datum der letzten Modifikation
franz        Dateiname
```

In obigem Directory-Listing sind zwei Besonderheiten zu sehen. Die Datei `ina` ist ein sog. *Hardlink* auf die Datei `franz` (erkennbar an der Zahl 2 in der Spalte der Hardlinks) und die Datei `sara` ist ein sog. *Symbolic Link* auf die Datei `else`.

Als *Hardlink* versteht man einen Verweis von einer Datei auf eine andere Datei. Die Dateinamen können unterschiedlich sein, der Inhalt dagegen ist identisch. Hardlinks können nur zwischen Dateien des gleichen Dateisystems geschlossen werden. Zeigt kein Verweis auf eine Datei, so ist die Anzahl der Hardlinks dieser Datei gleich 1. Hardlinks werden mit dem Kommando `ln vorhandene_Datei neue_Datei` angelegt.

Bei *Symbolic Links* erscheint als Typkennzeichnung ein `l` im Directory-Listing (siehe oben). Mit Symbolic Links können sowohl Verweise zu Dateien als auch zu Directories geschlossen werden. Im Gegensatz zu den Hardlinks können Symbolic Links auch über die Grenzen von Filesystemen hinausreichen. Symbolic Links werden erzeugt mit dem Kommando `ln -s vorhandene_Datei neue_Datei`.

**Dateinamen** dürfen unter Unix aus maximal 256 Zeichen bestehen. Es sind grundsätzlich alle Zeichen erlaubt auch Sonder- und Kontrollzeichen. Auf Dateinamen mit nicht darstellbaren Zeichen sollte allerdings verzichtet werden. Ebenso sollte man für Dateinamen keine Zeichen wählen, die nur für eine bestimmte

Landessprache gültig sind. Die Zeichen Ü, Ö, Ä und ß können je nach shell-Einstellung (ISO- oder UTF-8-Kodierung) entweder mit einem oder mit zwei Bytes dargestellt werden. Dateinamen, die in der einen Weise generiert werden, können in der jeweils anderen Umgebung nicht korrekt abgebildet werden. Das Dateisystem speichert die Dateinamen sowohl im ISO- als auch im UTF-8-Format.

Beispiele für gültige Dateinamen:

```
'1 2 345'
'Heute ist Dienstag'
'X'
'g*:8'
'ha.ha.ha.ha'
'bild.jpg'
'Süßholzraspel'
'programm.exe.exe.exe'
'programm'
```

In obiger Liste ist der Dateiname jeweils von Hochkommas eingeschlossen. Eine Aufteilung in Dateiname und Dateiextension, wie unter DOS, gibt es nicht. Die Verwendung von Sonderzeichen ist zwar möglich, sollte aber vermieden werden. Dateinamen, die nur ASCII-Zeichen enthalten sind sowohl in einer ISO- als auch UTF-8-Umgebung lesbar.

Achtung: Auch bei Dateinamen gilt: **UNIX unterscheidet Gross- und Kleinschreibung !!!**

Die freie Wahl von Dateinamen hat natürlich dort seine Grenzen, wo für bestimmte Zwecke Dateinamen-Konventionen bestehen. So werden z.B. C-Quelldateien mit der Endung `.c` gekennzeichnet, Fortran-Quellen enden mit `.f` und Objektdateien mit `.o`. Ferner gibt es unter Unix eine Reihe von Dateien, die mit einem Punkt beginnen (sog. Dot-files). Diese Dot-files werden auch "hidden files" genannt und fungieren primär als setup-files (`.login`, `.cshrc`). Sie sind mit dem Kommando `ls -a` aufzulisten.

Der Datei- bzw. Directory-Name dokumentiert nur den letzten Teil in der Bezeichnung der kompletten Hierarchie im Dateisystem. Strenggenommen setzt sich ein Datei- bzw. Directory-Name auch aus den davor befindlichen Verzeichnisnamen zusammen. Hat sich z.B. der Benutzer `rar12345` eingeloggt und danach mit dem Kommando `cd briefe` in das Unterverzeichnis `briefe` gesetzt, so lautet die komplette Bezeichnung des Unterverzeichnisses `/home/rar12345/briefe`. Auf Dateien, die sich in diesem Unterverzeichnis befinden, kann der Benutzer nun entweder mit der *absoluten* oder *relativen* Pfadangabe Bezug nehmen.

```
host> ls -l /home/rar12345/briefe/einladung.txt  absolute Referenz
host> ls -l ./einladung.txt                    relative Referenz
host> ls -l einladung.txt                      relative Referenz
```

An dieser Stelle soll die mit den bereits besprochenen Spezialzeichen zusammenhängende **Dateinamen-Expandierung** dokumentiert werden. Als Ausgangspunkt dient ein Unterverzeichnis namens `text` mit den darin befindlichen Dateien `alfred`, `berta`, `c1`, `c2`, `c3`, `christa`, `g*:8` und `zorro`. Zum Auflisten wird das Kommando `ls -l` verwendet.

```

host> ls -l c*           alle Dateien, die mit c beginnen
host> ls -l ??r*        alle Dateien, die als dritten Buchstaben ein r besitzen
host> ls -l ?????       alle Dateien, die genau aus 5 Zeichen bestehen
host> ls -l [ab]*       alle Dateien, die mit a oder b beginnen
host> ls -l c[13]*      die Dateien c1 und c3
host> ls -l [c-z]*     alle Dateien, deren erster Buchstabe zwischen c und z liegt
host> ls -l ?[e-h]*    alle Dateien, deren zweiter Buchstabe zwischen e und h
                        liegt
host> ls -l ?[*]*      alle Dateien, deren zweites Zeichen ein * ist
host> ls -l ?\**      alle Dateien, deren zweites Zeichen ein * ist
host> ls -l ~         alle Dateien im HOME-Directory
host> ls -l *{red}     alle Dateien, die mit dem string red enden
host> ls -l .         alle Dateien im aktuellen Directory
host> ls -l ..       alle Dateien im übergeordneten Directory

```

## 3.2 Kommandos mit Dateien

Das Kommando `ls -l` zum Auflisten von Inhaltsverzeichnissen haben wir bereits kennengelernt. Im folgenden sollen die wichtigsten Kommandos besprochen werden, die in Verbindung mit Dateien Verwendung finden.

### Inhalt einer Datei am Bildschirm ausgeben

```
cat [ -options ] [ filename ... ]
```

```

host> cat /etc/hosts      zeigt Inhalt der Datei /etc/hosts
host> cat -n /etc/hosts  zeigt Inhalt der Datei mit Zeilennummerierung

```

Die Ausgabe einer Datei auf dem Bildschirm ist natürlich nur dann sinnvoll, wenn die Datei auch lesbaren Text enthält (ASCII-, ISO- oder UTF-8-Kodierung).

### Inhalt einer Datei seitenweise am Bildschirm ausgeben

```
more [ -options ] [ filename ... ]
```

```

host> more /etc/mailcap   seitenweise Ausgabe der Datei
host> more +15 /etc/mailcap beginnt die Ausgabe ab Zeile 15

```

`more` ist ein mächtiges Kommando zum Auflisten von Textdateien. Während des Auflistens kann vorwärts und rückwärts (<Ctrl> b) geblättert werden, es können Textmuster gesucht und Leerzeilen komprimiert werden. Durch Betätigung der Leertaste wird immer um eine Seite bzw. Bildschirm weitergeblättert. Die Auflistung kann durch Eingabe von `q` vorzeitig abgebrochen werden.

### Ausgabe des Dateianfangs

```
head [ -n ] [ filename ... ]
```

```

host> head /etc/mailcap   Vom Inhalt der Datei /etc/mailcap werden
                        die ersten 10 Zeilen ausgegeben
host> head -20 /etc/mailcap Ausgabe der ersten 20 Zeilen der Datei

```

## Ausgabe des Dateiendes

```
tail [ -options ] [ filename ... ]
```

```
host> tail /etc/mailcap      gibt die letzten 10 Zeilen der Datei aus
host> tail -3 /etc/mailcap   gibt die letzten 3 Zeilen der Datei aus
host> tail +7 /etc/mailcap   zeigt ab Zeile 7 den gesamten Inhalt
```

## Zählen des Inhalts einer Datei

```
wc [ -clw ] [ filename ... ]
```

```
host> wc /etc/mailcap       gibt die Anzahl der Zeilen, Wörter und Zeichen der
                             Datei aus
host> wc -l /etc/mailcap    gibt die Anzahl der Zeilen der Datei aus
```

## Extrahieren von Zeilen mit bestimmten Textmustern

Für folgende Beispiele wird die Datei `mailcap` im Verzeichnis `/etc` verwendet (wird mit `cd /etc` als aktuelles Directory eingestellt).

```
grep [ -options ] [ pattern ... ] [ filename ... ]
```

```
egrep [ -options ] [ pattern ... ] [ filename ... ]
```

```
host> grep zip mailcap      alle Zeilen mit dem Muster zip
host> grep mpeg mailcap    alle Zeilen mit dem Muster mpeg
host> grep '^#' mailcap    alle Zeilen, die mit dem Zeichen # beginnen
host> grep '#$' mailcap    alle Zeilen, die mit dem Zeichen # enden
host> grep '^#$' mailcap   alle Zeilen, die genau aus dem Zeichen #
                             bestehen
```

```
host> grep '^.....[rh]' mailcap
    alle Zeilen, die mit fünf beliebigen Zeichen beginnen und deren sechstes
    Zeichen ein r oder h ist
```

```
host> grep '^.....[h-r]' mailcap
    alle Zeilen, die mit fünf beliebigen Zeichen beginnen und deren sechstes
    Zeichen im Bereich h - r liegt
```

```
host> egrep 'video|mpg' mailcap
    alle Zeilen die den string video oder mpg enthalten
```

```
host> egrep 'ol+' mailcap
    alle Zeilen, in denen der string ol vorkommt, wobei das l mehrfach hinter-
    einander vorkommen kann
```

**grep** und **egrep** sind wichtige Werkzeuge, um aus Text-Dateien Zeilen mit beliebigen Textmustern herauszufiltern. Reguläre Ausdrücke werden dabei in einfachen Hochkommas eingeschlossen. Der Unterschied zwischen beiden Kommandos liegt darin, dass `grep` standardmäßig nur eine eingeschränkte Auswahl an regulären Ausdrücken zulässt.

## Extrahieren von Spalten aus Dateien

```
cut [ -options ] [ filename ... ]
```

```
host> cut -b 1-2 mailcap      gibt die ersten beiden Bytes jeder Zeile der
                               Datei mailcap aus
host> cut -f 1 -d' ' mailcap  gibt das erste Feld jeder Zeile der Datei
                               mailcap aus
host> cut -f 1-2 -d' ' mailcap gibt das erste und zweite Feld jeder Zeile
                               der Datei mailcap aus
```

Je nach Option können mit `cut` Bytes, Zeichen oder Felder aus einer Datei "ausgeschnitten" werden. Bei Verwendung von `-f` sollte immer mit der Option `-d` das entsprechende Feldtrennzeichen definiert werden. In obigem Beispiel wird das Leerzeichen als Feldtrennzeichen angegeben.

## Dateien kopieren

```
cp [ -fip ] file_old file_new
cp [ -fip ] file directory
```

Sollen die Zugriffsrechte der ursprünglichen Datei beibehalten werden, so ist dies mit der Option `-p` anzugeben. `-f` bewirkt ein unbedingtes Kopieren und `-i` gibt vor dem Kopieren eine Warnung aus, falls die Zielfeile bereits existiert (Schutz vor Überschreiben).

## Dateien umbenennen

```
mv [ -fi ] file_old file_new
mv [ -fi ] file directory
```

Im zweiten Fall wird die Datei in das bereits vorhandene Directory verschoben.

## Dateien löschen

```
rm [ -fir ] file ...
```

Um unbeabsichtigte Löschoptionen zu vermeiden, ist es sinnvoll, das `rm`-Kommando mit der Option `-i` aufzurufen. Dies ist besonders dann zu empfehlen, wenn beim Löschen Metazeichen zur Dateinamen-Expandierung verwendet werden.

## Dateien suchen

```
find pathname expression
```

```

host> find . -name christa -print
Ausgehend vom aktuellen Verzeichnis wird die Datei christa gesucht.

host> find . -name c\* -print
Ausgehend vom aktuellen Verzeichnis werden alle Dateien und Directories
gesucht, die mit c beginnen.

host> find /usr -name \?\?r\* -print
Das gesamte Verzeichnis /usr wird nach Dateien und Directories durchsucht,
die im Namen als drittes Zeichen ein r besitzen.

host> find . -name christa -exec rm {} \;
Wenn die Datei christa gefunden wird, wird sie durch das rm-Kommando
gelöscht.

```

Das Kommando **find** durchsucht ausgehend vom angegebenen Directory rekursiv alle darunter liegenden Verzeichnisse nach Dateien, die den gewünschten Kriterien entsprechen. Sonderzeichen wie [ ] ? \* ( ) müssen mit dem Zeichen \ maskiert werden.

### Weitere nützliche Kommandos

```

host> diff file1 file2      vergleicht zwei Dateien
host> sort [ -options ] file  sortiert eine Datei
host> quota                gibt die aktuelle Plattenplatzbelegung aus
host> dquota              gibt die aktuelle Home-Quota aus
host> du -sk *             gibt die Größe von Dateien und Directories
                           aus
host> du -sk .??*         gibt die Größe von hidden Dateien und Direc-
                           tories aus
host> file filename       gibt den Typ einer Datei aus

```

Für die Verwaltung des eigenen Home-Verzeichnisses sind besonders die Kommandos **du** und **dquota** von großer Bedeutung, da damit die Plattenplatzbelegung im eigenen Verzeichnis differenziert betrachtet werden kann.

Da man ja unter Unix/Linux von den Dateinamen nicht auf den Inhalt der Dateien schließen kann, ist ein Programm wie **file**, das den Datei-Inhalt analysiert, sehr hilfreich. Je nachdem, von welcher Art der Inhalt ist, werden z.B. Meldungen wie "ASCII-file", "Postscript-file" usw. ausgegeben. Auch bei Textdateien kann mit **file** die Kodierung des Inhalts herausgefunden werden (ISO oder UTF).

## 3.3 Kommandos mit Verzeichnissen

Verzeichnisse (Ordner, Directories) sind im Grunde Dateien, die die Informationen über die in ihnen vorhandenen Dateien und Unterverzeichnisse enthalten.

### In ein Verzeichnis wechseln

```

cd dirname

host> cd text  Wechsel in das Verzeichnis text
host> cd /tmp  Wechsel in das Verzeichnis /tmp
host> cd ..    von der aktuellen Position eine Ordner-Stufe nach oben
host> cd      zurück ins HOME-Verzeichnis wechseln

```

Beim Verzeichniswechsel kann sowohl die absolute als auch die relative Directory-Notation verwendet werden.

### Aktuelles Verzeichnis anzeigen

```
pwd
```

### Verzeichnis erzeugen

```
mkdir [ -options ] dirname ...
```

```
host> mkdir text im aktuellen Verzeichnis wird das Directory text angelegt
```

### Verzeichnis kopieren

```
cp -r [ -ip ] quell_directory ziel_directory
```

```
host> cp -r text text2 rekursives Kopieren des Directories text nach
text2
```

Das in obigem Beispiel erwähnte Ziel-Verzeichnis `text2` sollte vor dem Kopieren noch nicht existieren. Falls dies doch der Fall ist, wird das komplette Quell-Verzeichnis `text` unter dem Directory `text2` angelegt. Beim Kopieren ganzer Verzeichnisse muss auf *symbolic links* geachtet werden. Symbolic links werden beim Kopieren aufgelöst und die entsprechenden Dateien sind dann mehrfach vorhanden. Soll daher die Link-Struktur beibehalten werden, ist folgende Vorgehensweise besser.

```
mkdir ziel_directory
cd quell_directory
tar -cvf - . | ( cd ziel_directory; tar -xpvf - )
```

Das Programm `tar` ist ein Tool zum Sichern von Dateien und Verzeichnissen in einer Archiv-Datei. Dabei wird die originale Struktur beibehalten.

### Verzeichnis umbenennen

```
mv [ -options ] old_directory new_directory
```

Das Umbenennen von Verzeichnissen erfolgt auf die gleiche Weise wie das Umbenennen von Dateien.

### Verzeichnis löschen

```
rmdir dirname ... löscht leeres Directory
rm -r dirname ... löscht rekursiv den ganzen Verzeichnis-Baum
```

Auch beim `rm`-Kommando können die Zeichen zur Dateinamens-Expandierung verwendet werden (aber Vorsicht bei der Verwendung von `*`).

## 3.4 Dateischutz

Der Zugriff auf Dateien und Directories wird über die sog. *accessrights* bzw. *Filemodi* geregelt. Über die aktuellen accessrights einer Datei oder eines Directories gibt das Kommando **ls -l** Auskunft (z.B. `-rw-r--r--`).

Es gibt bzgl. jeder Datei bzw. jedes Directories drei Benutzergruppen:

der *Eigentümer* ( `user=u` ), die *Gruppe* ( `group=g` ) und *alle anderen* ( `other=o` ).

Für jede Benutzergruppe kann bestimmt werden, ob sie die Datei

*lesen* ( `read=r` ), *schreiben* ( `write=w` ), *ausführen* ( `execute=x` ) darf.

Eine Protection-mask kann z.B. so aussehen:

`-rw-|r--|r--` entspricht binär `110|100|100` entspricht oktal `644`

Die ersten drei Bits symbolisieren den Zugriff für den Datei-Eigentümer, die nächsten drei Bits den Zugriff für die Gruppe und die letzten drei Bits den Zugriff für alle anderen. Bei obigem Beispiel darf der Datei-Eigentümer die Datei lesen und schreiben, die Gruppe und alle anderen besitzen nur lesenden Zugriff. Das erste Bit der Filemodi bezeichnet die Art der Datei ( `d` = Directory, `-` = normales file, `l` = symbolischer Link, `b`, `c` = spezielle Gerätedateien, `p` = named pipe).

Standardmäßig wird die Protection-mask neu erstellter Dateien komplementär zum sog. **umask**-Wert gebildet. Mit dem Kommando **umask** kann der aktuelle umask-Wert angezeigt oder auch verändert werden. Das `x`-Bit bleibt in der Regel vom eingestellten umask-Wert unbeeinflusst. Ein umask-Wert von `022` bewirkt beispielsweise eine Protection-mask von `644` bzw. `-rw-r--r--`. Das `x`-Bit wird nur dann automatisch gesetzt, wenn die Datei durch eine Kompilation gebildet wurde.

### Ändern der Zugriffsrechte

```
chmod [ -options ] mode filename
```

Die Modifikation der Zugriffsrechte erfolgt mit dem Kommando **chmod**. Dabei kann der einzustellende *mode* entweder **oktal** (z.B. `775`) oder **symbolisch** (`u=rwx, g=r`) angegeben werden.

```
host> chmod 777 berta           Protection-mask: -rwxrwxrwx
host> chmod 640 berta           Protection-mask: -rw-r-----
host> chmod g+x berta           Protection-mask: -rw-r-x---
host> chmod g+w,o+x berta       Protection-mask: -rw-rwx--x
host> chmod g-wx berta          Protection-mask: -rw-r----x
host> chmod u=rw,go= berta      Protection-mask: -rw-----
```

Bei der symbolischen Einstellung der filemodes bedeuten die Zeichen `+` Setzen von zusätzlichen Rechten, `-` Entzug von Rechten und `=` absolutes Setzen von Rechten.

Preisfrage: Was passiert, wenn ein Benutzer einer Datei, die ihm gehört, sämtliche Rechte entzieht?

Bei Directories besitzen die Protection-bits eine andere Bedeutung. Sie regeln hier den Zugang über das Kommando `cd` sowie die Möglichkeit, Dateien zu erstellen bzw. zu löschen. Folgende Übersicht soll dies verdeutlichen.

```

drwxrwxrwx  ls, cd und file create/delete für alle Benutzer möglich
drwxr-xr-x  nur ls und cd möglich für alle Benutzer
drwxr--r--  nur ls möglich für alle Benutzer
drwx--x--x  nur cd möglich für alle Benutzer
drwx----- nur noch Rechte für den Eigentümer des Directories

```

Durch die Zugriffsrechte, die für ein Directory gesetzt werden, können auch die darunterliegenden Verzeichnisse geschützt werden.

Neben den oben beschriebenen Zugriffsregelungen gibt es noch drei weitere Ausprägungen:

```

-rwsr-xr-x  set user id bit (user-s-bit, oktal 4000)
              wenn dieses Programm aufgerufen wird, wird nur für die Zeit der
              Ausführung die user-id des file-owners übernommen
-rwxr-sr-x  set group id bit (group-s-bit, oktal 2000)
              wenn dieses Programm aufgerufen wird, wird nur für die Zeit der
              Ausführung die group-id des file-owners übernommen
-rwxrwxrwt  sticky bit (wirkt global, oktal 1000)
              Bei Programmen bleibt das aufgerufene Programm nach Beendi-
              gung im swap-Bereich (höhere Performance).
              Bei gemeinsam genutzten Directories regelt es den Zugriff auf
              files von verschiedenen Benutzern.

```

### 3.5 Drucken von Dateien

Unter Linux haben sich in den letzten Jahren Drucksysteme durchgesetzt, die auf *cups* aufbauen. Der Ausdruck von Dateien kann entweder kommandoorientiert oder über Anwendungen erfolgen. Beim kommandoorientierten Verfahren werden hauptsächlich die System-V-Kommandos verwendet. Prinzipiell hat ein Ausdrucken an der Uni Regensburg nur bei einem positiven Drucker-Kontostand Erfolg.

```

lp -d RIO_CIP_PHY file  Ausdruck auf Linux-CIP-Pool-Drucker
lp -d RIO_LASER file    Ausdruck auf Drucker RIO_LASER im Rechen-
                        zentrum

lpstat -o printqueue    Anzeige aller Jobs in der genannten printqueue
lpstat -s                Anzeige aller installierten Drucker-Queues
lpstat -p -l            Ausführliche Anzeige aller Drucker-Queues

cancel printjob-nr      Eliminieren eines Printjobs

```

Es ist ratsam, bei jedem Druckauftrag auch explizit den Drucker anzugeben, auf den ausgegeben werden soll, da entweder kein default Drucker definiert ist oder der default Drucker in einem Raum steht, zu dem kein Zutritt möglich ist. Bei Ausgaben auf Drucker im Rechenzentrum müssen die Ausdrücke auch dort abgeholt werden. **Achtung:** Bei einem remote Login von einem Windows-PC wird ein unter Unix abgesetzter Druckauftrag nicht auf den Drucker im Windows-Pool umgeleitet!

**a2ps** ist ein nützliches Hilfsprogramm, um Text-Dateien für den Ausdruck vorzubereiten. Damit können auch Manual-Pages platzsparend ausgedruckt werden.

```
host> man chmod | a2ps -m -o - | lp -d RIO_CIP_PHY
```

Mit obigem Kommando werden die manual-pages von `chmod` auf den Drucker im Linux-Pool Physik ausgegeben, jeweils zwei Seiten auf einem A4-Blatt.

Auf allen Linux-Systemen gibt es auch grafische Programme für das Ausdrucken von Dateien. Stellvertretend für diese Hilfsprogramme soll hier das Tool **kprinter** besprochen werden, da dies einen integralen Bestandteil des KDE-Desktops darstellt.

Der Start des Programms erfolgt durch das Kommando **kprinter** oder durch **kprinter file**. Es erscheint folgendes Fenster.



Abbildung 3.2: Drucken mit kprinter

Falls die zu druckende Datei noch nicht als Kommando-Parameter eingegeben wurde, kann dies über den Schaltknopf *Optionen* nachgeholt werden.

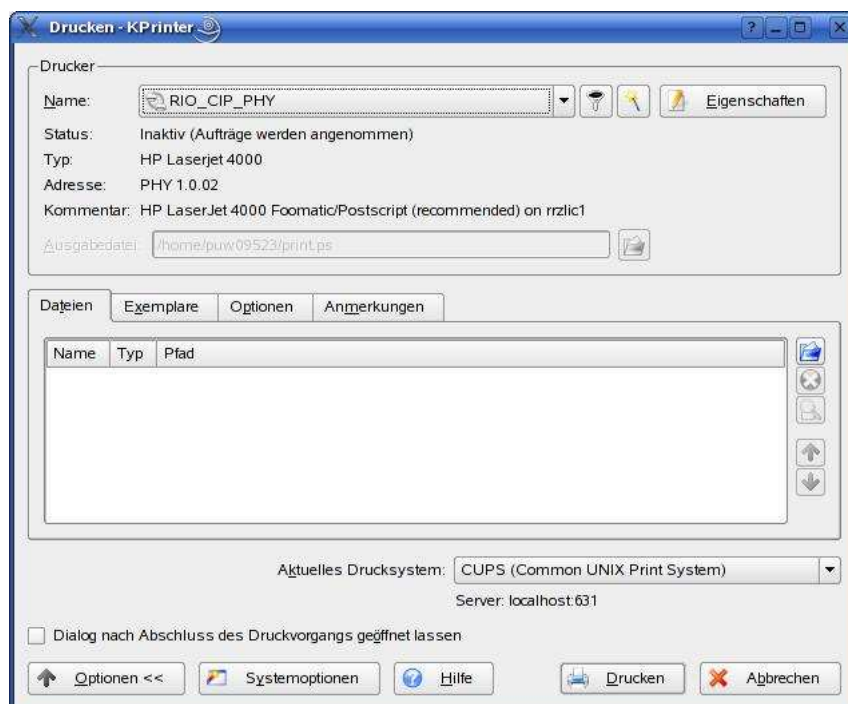


Abbildung 3.3: Auswahl von Dateien



Durch Klick auf das Ordnersymbol auf der rechten Seite des Fensters wird der Datei-Browser aktiviert, aus dem dann die zu druckenden Dateien ausgewählt werden können. Im Auswahlfeld des Drucker-Namens sollte der richtige Drucker eingestellt werden. In obigem Beispiel wird auf den Drucker RIO\_CIP\_PHY, den Drucker im Linux-Pool Physik, ausgegeben. Neben dem Namen enthält das Fenster noch eine Beschreibung der Hardware sowie den Standort des Druckers. Über den Schaltknopf *Drucken* wird schließlich der Ausdruck gestartet.

Verschiedene Druckoptionen lassen sich durch die Auswahl von *Eigenschaften* definieren. Die wichtigsten Angaben sind das Seitenformat und die Ausrichtung. Es ist unbedingt darauf zu achten, dass als Seitenformat A4 angegeben wird, da alle Drucker an der Uni Regensburg für den A4-Ausdruck eingestellt sind. Die Definition der Ausrichtung richtet sich nach dem Text, der ausgedruckt werden soll.

Nach dem Start des Druck-Jobs kann nachgeprüft werden, ob der Auftrag fehlerlos weitergeleitet wurde. In der Datei `/var/log/cups/error_log` werden alle Druckaufträge protokolliert. Sollte ein Fehler auftreten, so wird dies in dieser Datei festgehalten.

Alle Druckaufträge werden zunächst am lokalen Rechner in die entsprechende Warteschlange gestellt. Über das Drucksystem werden sie zu sog. Printservern weitergeleitet, von wo sie dann schließlich auf dem gewünschten Drucker ausgegeben werden. Ein Zugriff auf den Druckauftrag seitens des Benutzers ist nur auf dem lokalen System möglich. Ein bereits beim Drucker befindlicher Auftrag kann auch nur direkt am Drucker abgebrochen werden.

Manche Anwendungen ,wie z.B. OpenOffice, verwenden beim Ausdruck kein KDE-Hilfsprogramm, sondern nehmen die Ansteuerung des Druckers selbst vor. In diesem Fall wird das Druckbild von den Druck-Einstellungen der Anwendung bestimmt (Schriften, Seitenränder, Hintergrund). Oft ist es möglich, in der Anwendung eine Druckvorschau aufzurufen. Damit kann geprüft werden, ob das zu druckende Dokument den eigenen Vorstellungen entspricht, bevor es zum Drucker gesendet wird. Als Beispiel dafür soll die Druckvorschau des Editors `gedit` dienen (über das Menü *Datei* -> *Druckvorschau*).



Abbildung 3.4: Druckoptionen



Abbildung 3.5: Druckvorschau beim Editor `gedit`

Jede Seite wird im A4-Format dargestellt. Auch die meisten Web-Browser lassen eine Druckvorschau zu.

## 3.6 Der Dateimanager konqueror

Alle bisher dokumentierten Aktionen für die Verwaltung von Dateien und Ordnern lassen sich auch mit dem grafischen Dateimanager **konqueror** bequem erledigen. konqueror ist der Dateimanager unter KDE. Unter Gnome heißt das äquivalente Programm nautilus.



Der Start des Dateimanagers erfolgt durch einen einfachen Mausklick auf das Haussymbol in der unteren Kontrollleiste. Dabei deutet das Haus-Icon bereits an, dass der Dateimanager standardmäßig den Inhalt des persönlichen Home-Verzeichnisses wiedergibt. Wird konqueror dagegen mittels Kommandoingabe aufgerufen, so muss der persönliche Ordner erst explizit ausgewählt werden; konqueror ist nicht nur Dateimanager, sondern auch gleichzeitig Web-Browser.

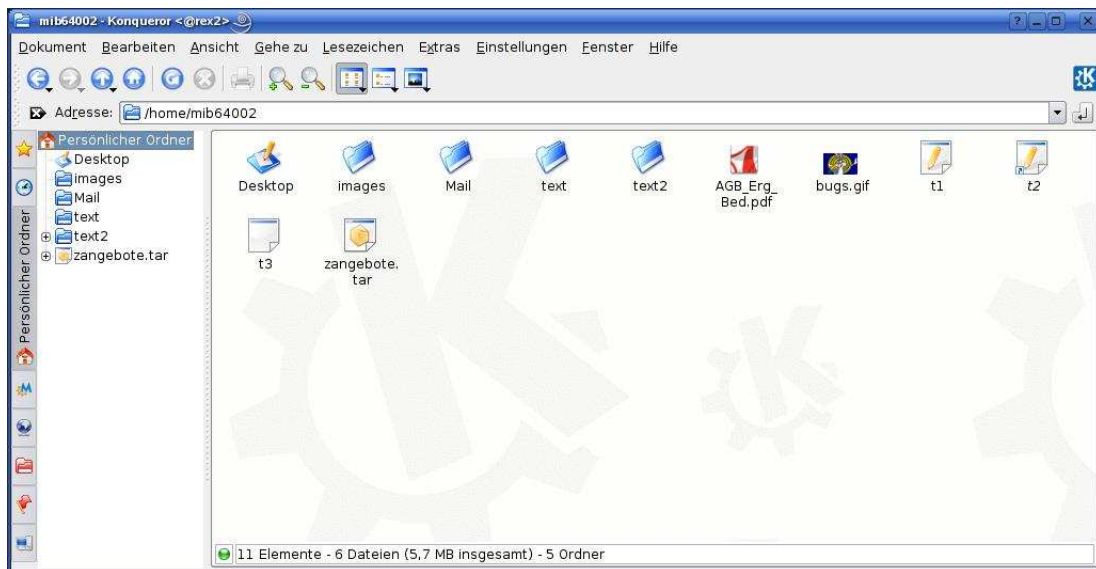


Abbildung 3.6: konqueror als Dateimanager

In obigem Beispiel wird mit konqueror das Home-Verzeichnis des Benutzers mib64002 abgebildet. Den oberen Teil des Fensters nehmen Menü-, Werkzeug- und Adressleiste ein. Dann folgt darunter der große Anzeigenbereich für Dateien und Ordner. Ganz unten ist noch eine Zeile mit Statusinformationen untergebracht. Der Anzeigenbereich für die Dateien und Ordner ist nochmals zweigeteilt: In der linken Spalte wird die baumartige Ordnerstruktur dargestellt, in der rechten Spalte ist jeweils der Inhalt des angewählten Ordners zu sehen. Am linken Rand sind noch einige Symbole zu sehen, die für weitere konqueror-Funktionen genutzt werden können.

Wenn der Dateimanager konqueror zum ersten Mal aufgerufen wird, zeigt er den Inhalt des Home-Verzeichnisses in einer Symbolansicht. Jeder Ordner und jede Datei wird mit einem Symbol dargestellt, das auf den Inhalt schließen läßt.

	Spezieller Desktop-Ordner		Ordner (Unterverzeichnis)
	Textdatei		leere Datei
	tar-Archiv		pdf-Dokument
	symbolic Link		Bild

Die Verzeichnis-Struktur lässt sich auch in anderen Ansichten darstellen. Über das Menü *Ansicht* und Unterpunkt *Anzeigemodus* kann die Darstellung geändert werden. Am informativsten ist die *Detaillierte Ordneransicht*.

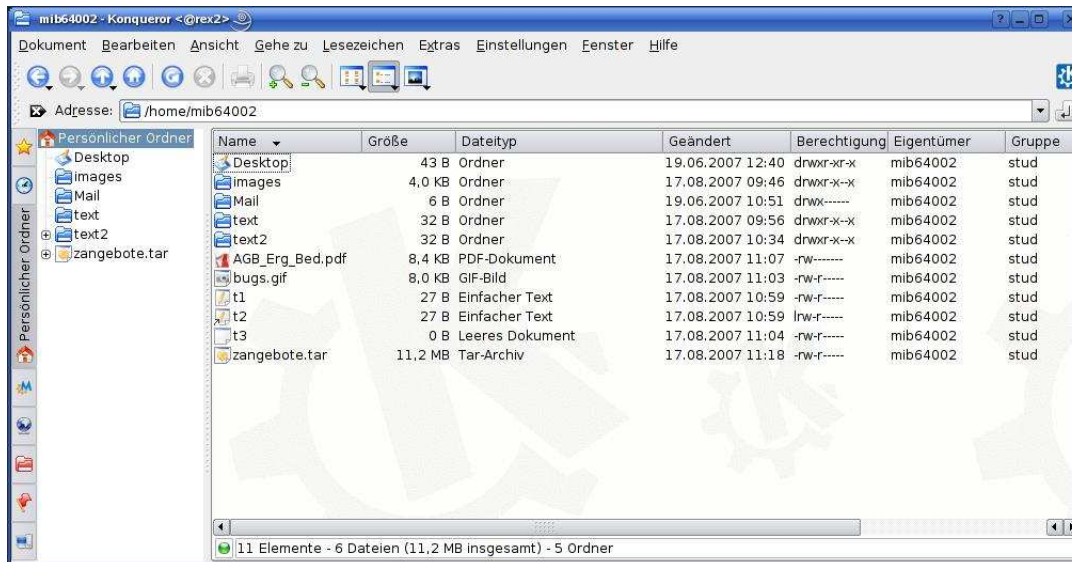


Abbildung 3.7: konqueror mit Detaillierter Ordneransicht

Wie beim Kommando `ls -l` sind in dieser Ansicht alle Dateiattribute zu sehen. Über das Menü *Ansicht* lassen sich auch *versteckte Dateien anzeigen*, also Dateien bzw. Ordner, deren Name mit einem Punkt beginnt. Bewegt man den Mauszeiger über ein Dateisymbol, so erscheinen ebenfalls die Informationen über diese Datei. Bei einem Links-Klick auf ein Dateisymbol versucht konqueror, die Datei entsprechend ihrem Inhalt zu öffnen.

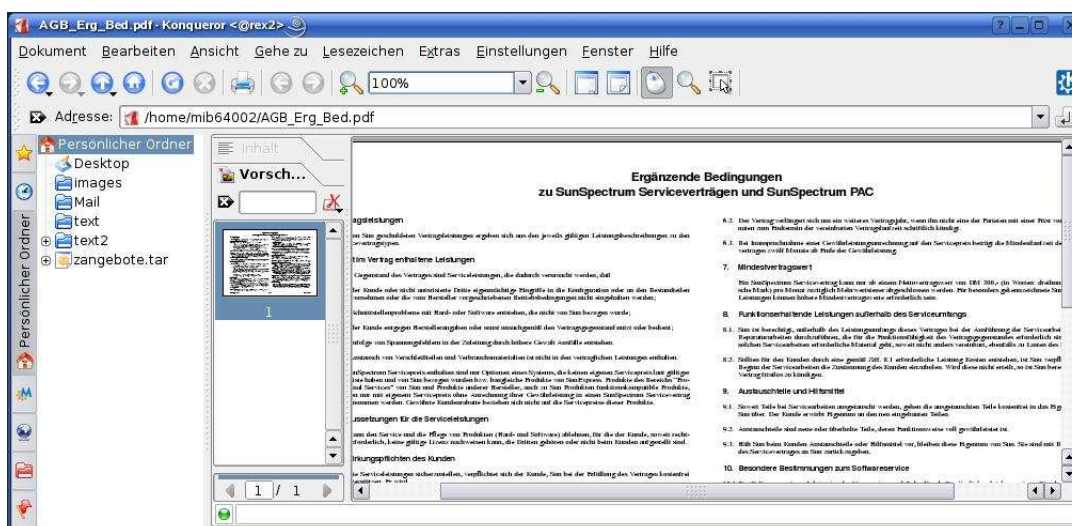


Abbildung 3.8: Vorschau eines pdf-Dokuments

In obigem Beispiel wurde die Datei `AGB_Erg_Bed.pdf` geöffnet. Da es sich bei dieser Datei um eine pdf-Datei handelt, wird sie vom integrierten pdf-reader als lesbares Dokument wiedergegeben. Die Adressleiste

zeigt dabei den kompletten Dateipfad an. Mit dem Links-Pfeil <- in der Werkzeugleiste kommt man zurück in die vorherige Ansicht. In dieser Weise lassen sich die wichtigsten Typen von Dateien in einer Vorschau darstellen.

Führt man einen Rechts-Klick auf ein Dateisymbol aus, so erscheint ein Funktions-Menü für die Dateiverwaltung. Mit diesen Funktionen lassen sich Dateien umbenennen, kopieren und löschen (*In den Mülleimer werfen*). Daneben gibt es noch Funktionen zum Verschlüsseln und zum Komprimieren. Über den Punkt *Eigenschaften* → *Berechtigungen* können die Zugriffsrechte der Datei oder des Ordners gesetzt werden. Über den Unterpunkt *Aktionen* → *Drucken* ist es möglich, die Datei auszudrucken, wobei hier das Hilfsprogramm *kprinter* verwendet wird.

Je nach Dateityp kann sich das Funktions-Menü etwas unterschiedlich zusammensetzen. So werden z.B. beim Rechts-Klick auf die tar-Datei die Funktionen *Vorschau in Archivprogramm* und *Entpacken* angeboten. Eine tar-Datei ist eine Zusammenfassung mehrerer Dateien. Mit der Funktion *Entpacken* werden die ursprünglichen Dateien wiederhergestellt. Auch bei zip-Dateien, einer anderen Form von Datei-Archiven, können diese Funktionen angewandt werden.

Um neue Ordner oder Dateien mit konqueror anzulegen, genügt es, aus dem Menü *Bearbeiten* den Punkt *Neu erstellen* auszuwählen. Aus dem dann folgenden Menü kann entweder ein Ordner, ein Dateityp oder eine Verknüpfung gewählt werden.

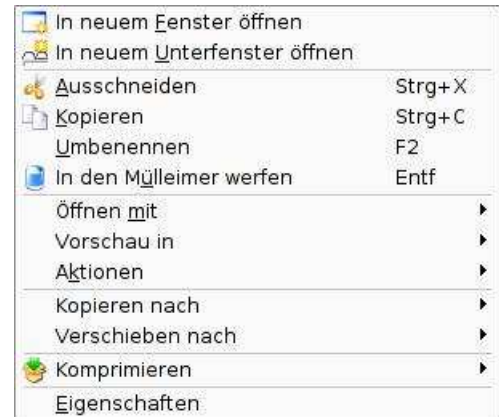


Abbildung 3.9: Funktionsmenü

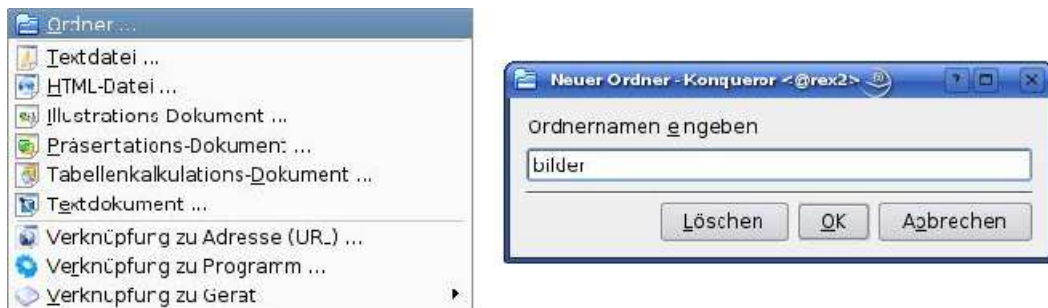


Abbildung 3.10: Neu erstellen eines Ordners

In vorliegendem Beispiel wird ein neuer Ordner namens `bilder` erstellt. Der neue Ordner erscheint dann sowohl in der Baumstruktur (linke Navigations-Spalte) als auch in der Ordneransicht. Soll nun eine vorhandene Datei in den neuen Ordner kopiert oder verschoben werden, so kann dies über oben erläutertes Funktions-Menü geschehen oder aber mittels "drag and drop" mit der linken Maustaste. Vor den Ordnersymbolen in der linken Spalte steht ein Pluszeichen, falls es darin weitere Unterordner gibt. Mit einem Mausklick auf das Pluszeichen wird diese Unterstruktur sichtbar.

Eine äquivalente Funktion zum `find`-Kommando wird über das Menü *Extras* → *Dateien suchen* zur Verfügung gestellt. `konqueror` stellt damit ein Werkzeug dar, mit dem in komfortabler Art und Weise die Verwaltung von Dateien und Verzeichnissen realisiert werden kann.



## 4 Kommandos und Prozesse

Beim Hochfahren (Booten) eines Unix/Linux-Systems wird eine Reihe von Prozessen gestartet. Ausgehend von einem Wurzel-Prozess wird so nach und nach eine baumartige, hierarchische Struktur von Prozessen erzeugt.

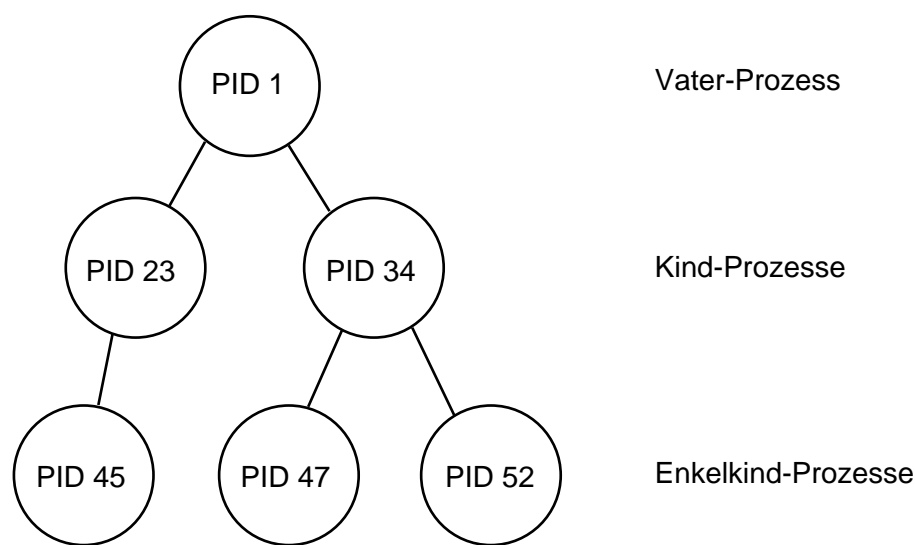


Abbildung 4.1: Hierarchische Prozess-Struktur

Ein Prozess ist ein in einem definierten Hard- und Software-Kontext laufendes Programm. Innerhalb eines Prozesses können mehrere Programme abgearbeitet werden, andererseits kann ein Prozess auch weitere untergeordnete sog. Kindprozesse erzeugen. Jeder Prozess besitzt eine eindeutige Prozessidentifikationsnummer (PID) und ist einem eindeutigen Benutzer zugeordnet (UID). Neue Prozesse können nur von einem bereits laufenden Prozess gestartet werden, so dass daraus eine gesamte Prozess-Hierarchie entsteht.

Der Urvater aller Prozesse ist der `init`-Prozess. Er besitzt die PID 1. Wie aus der Prozess-Hierarchie zu erkennen ist, sind die Prozesse voneinander abhängig. Da ein übergeordneter Prozess in der Regel einen untergeordneten Prozess kontrolliert, bedeutet eine Termination eines Prozesses auch ein Sterben aller von ihm abhängigen untergeordneten Prozesse.

Unter Unix/Linux wird zwischen Vordergrund- und Hintergrundprozessen unterschieden. *Vordergrundprozesse* sind interaktive Prozesse, die über ein Gerät (*Device*) kontrolliert werden (z.B. Login-shell). Die Klasse der *Hintergrundprozesse* wird eingeteilt in interaktive Hintergrundprozesse und in sog. Daemon-Prozesse. *Daemon*-Prozesse sind Endlosprozesse, die auf ein bestimmtes Ereignis warten und keinem Device zugeordnet sind. Der Prozess `nsd` ist ein solcher Daemon-Prozess. Er ist dafür zuständig, ein Filesystem remote über das Datennetz zur Verfügung zu stellen.

## 4.1 Der Login-Prozess

Beim Login wird ebenfalls ausgehend vom `init`-Prozess eine Hierarchie von Benutzer-Prozessen aufgebaut, an deren Ende der Start aller Anwendungen steht. Während des Login-Vorgangs werden, abhängig von der Login-shell und dem zu startenden Desktop, verschiedene Dateien ausgelesen und die dort befindlichen Definitionen aktiviert. Diese Dateien werden auch *setup*-Dateien genannt.

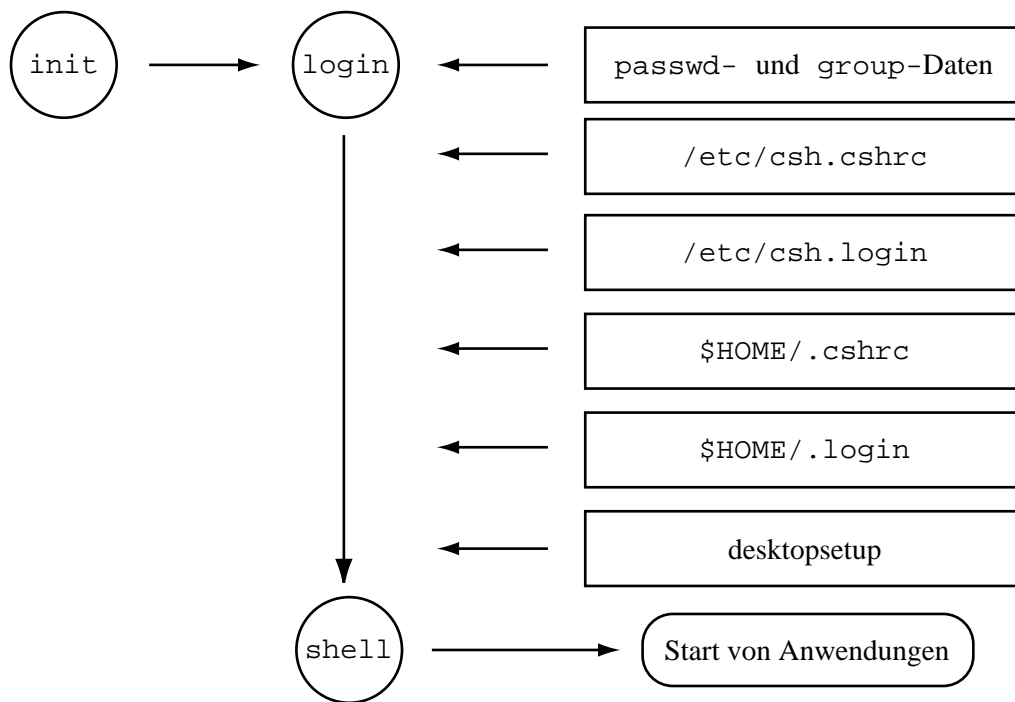


Abbildung 4.2: Login-Vorgang beim tcsh-Login

Jeder Login-Vorgang beginnt mit der Identitäts-Prüfung des Benutzers (`passwd`, `group`). Mit den Dateien `/etc/csh.cshrc` und `/etc/csh.login` werden zunächst globale Einstellungen vorgenommen, die für jeden Benutzer gelten, der sich auf dem System einloggt. Alle weiteren Dateien befinden sich im HOME-Verzeichnis des Benutzers und sind für die individuellen Einstellungen verantwortlich.

`/etc/csh.cshrc` In dieser Datei werden globale Pfade, Variable und aliases gesetzt.

```

set path = ( /usr/local/bin /usr/bin )
setenv MAIL ${HOME}/Mail/Inbox
setenv EDITOR vi
alias rm 'rm -i'

```

`/etc/csh.login` Hier befinden sich meist globale Einstellungen zum Terminaltyp.

```

/usr/bin/stty -istrip
setenv TERM vt100

```

`$HOME/.cshrc` Individuelle Erweiterungen zu den globalen Einstellungen.

```

set path = ( $path $HOME/bin )
source ~/.myalias

```

```
$HOME/.login  Individuelle Terminal-Settings. Definitionen für spezielle Anwen-
              dungen.
              setenv ENVIRONMENT 'NOBATCH'
```

All diese Dateien legen mit ihren Definitionen die gesamte Benutzerumgebung fest. Je nachdem, welche Login-shell für einen Benutzer eingetragen ist, werden die für diese Login-shell relevanten Dateien ausgelesen und die dort vorhandenen Einstellungen aktiviert. Die oben erwähnten Dateien sind für die `tcsh` als Login-shell gültig. Wird die `bash` als Login-shell benutzt, so werden für die Definitionen beim Login die Dateien `/etc/profile`, `$HOME/.bash_profile`, `$HOME/.bash_login` und `$HOME/.profile` sowie `$HOME/.bashrc` benutzt. Für das Desktop-Setup sind je nach gewählter Oberfläche die im HOME-Verzeichnis vorhandenen Unterverzeichnisse `.dt` (für CDE), `.kde` bzw. `.kde2` (für KDE) und `.gnome...` (für Gnome) sowie die setup-Dateien `.dtprofile` und `.kderc` verantwortlich.

## 4.2 Kommandoeingabe

Kommandos sind Bildschirm-Eingaben (meist in einem Terminal-Fenster), die von der aktuell laufenden shell (Kommandointerpreter) abgearbeitet werden. Unter Unix/Linux existieren drei verschiedene Typen von Kommandos:

- shell-interne Kommandos (z.B. `cd`, `echo`); diese Kommandos werden innerhalb der shell abgewickelt; es wird kein eigener Sub-Prozess gestartet
- externe Kommandos; die Abarbeitung erfolgt in einem untergeordneten Prozess
- Benutzerprogramme und -prozeduren

### Das einfache Kommando

```
command [ arguments ]
```

```
host> date           Ausgabe des aktuellen Datums
host> id             Ausgabe der eigenen Benutzeridentität
host> finger        Auflisten aller eingeloggten Benutzer
host> logname       Ausgabe des Login-Namens
host> mkdir text    Anlegen eines Subdirectories namens text
```

### Die Kommandofolge

```
command1; command2; command3; ...
```

```
host> date; finger; ls /tmp
      Ausgabe des aktuellen Datums, der eingeloggten Benutzer und Auflisten des
      Verzeichnisses /tmp
```

Die Kommandofolge besteht also aus mehreren einzelnen Kommandos, die jeweils durch ein Semikolon getrennt von links nach rechts abgearbeitet werden. Als Argumente können Dateinamen, Optionen und Ausdrücke verwendet werden. Argumente werden durch Leerzeichen voneinander getrennt.

## Die Kommandogruppe

```
( command1; command2; command3; ... )
```

```
host> ( date; finger; ls /tmp )
```

Ausgabe des aktuellen Datums, der eingeloggten Benutzer und Auflisten des Verzeichnisses /tmp

Die Gruppierung erfolgt durch Einschluss einer Kommandofolge in runde Klammern. Im Gegensatz zur Kommandofolge werden die Anweisungen in der Kommandogruppe in einem separaten untergeordneten Prozess abgearbeitet.

## Das Pipelining

```
command1 | command2 | command3 | ...
```

```
host> ps -ef | grep root | head -10
```

Das Kommando `ps -ef` gibt alle laufenden Prozesse aus, `grep root` filtert aus der gesamten Liste nur die Prozesse heraus, die dem Benutzer `root` gehören und `head -10` gibt von der verbleibenden Liste die ersten 10 Zeilen am Bildschirm aus.

```
host> ( ls -l /tmp; ls -l /var ) | wc -l
```

Übergabe des gesammelten Outputs beider `ls`-Kommandos an das Kommando `wc`

Der Output von `command1` wird unmittelbar als Input für `command2` verwendet, der Output von `command2` als Input für `command3` usw. Im zweiten Beispiel wird die Bedeutung der Kommandogruppe deutlich. Würden die runden Klammern wegfallen, so würde das `wc`-Kommando nur den Output vom unmittelbar davor stehenden `ls`-Kommando erhalten.

## 4.3 Prozesskontrolle

Ein laufendes Unix/Linux-System besteht aus einer Vielzahl von Prozessen, die entweder beim Booten des Systems oder durch Benutzer-Aktivitäten (Login, Programmaufruf) erzeugt werden. Jeder Prozess ist einem Benutzer zugeordnet und kann daher auch von diesem Benutzer kontrolliert werden. Viele Prozesse gehören z.B. dem Benutzer `root` und können nur vom Benutzer `root`, also dem Systemadministrator, beeinflußt werden. Auf alle Prozesse, die einem Benutzer gehören, hat dieser Benutzer selbst administrativen Einfluß.

Für das Auflisten der Prozess-Tabelle wird das Kommando `ps` verwendet. Immer wenn Informationen über laufende Prozesse benötigt werden, sollte dieses Kommando benutzt werden. Je nach gewählter Option liefert `ps` Informationen über alle Prozesse oder nur über Prozesse, die von der aktuellen Position in der Prozess-Hierarchie erzeugt wurden.

```
host> ps -f
```

UID	PID	PPID	C	STIME	TTY	TIME	CMD
puw09523	10005	9997	0	Oct 02	pts/4	0:01	tcsh
puw09523	13853	10005	0	08:24:44	pts/4	0:02	xdvi -bg white index
puw09523	13855	13853	0	08:25:01	pts/4	0:03	gs -sDEVICE=x11 -dNOPAUSE
puw09523	13848	10005	0	08:15:18	pts/4	0:01	/usr/openwin/bin/textedit

```
host> ps -l
```

```
F S  UID  PID  PPID  C  PRI  NI  ADDR  SZ  WCHAN  TTY  TIME  CMD
8 S  9523 10005 9997  0  41  20  f5bb2cc0  325  f5bb2e90 pts/4  0:01 tcsh
8 S  9523 13853 10005  0  57  20  f6000010  775  f5e64066 pts/4  0:02 xdvi
8 S  9523 13855 13853  0  67  20  f60a4980 1213  f606f8d4 pts/4  0:03 gs
8 S  9523 13848 10005  0  51  20  f60a3cc0 1070  f5e6430e pts/4  0:01 textedit
```

Mit den oben gezeigten Kommandos sieht man alle eigenen Prozesse, die von der aktuellen Position der Prozess-Hierarchie erzeugt wurden. Dabei kann auch an Hand der PID- und PPID-Nummern das Abhängigkeitsverhältnis in der Prozess-Hierarchie nachvollzogen werden. Sollen alle Prozesse angezeigt werden, die auf dem System laufen, so erfolgt dies mit dem Kommando **ps -ef | more**.

Die vom ps-Kommando gelieferten Daten haben im einzelnen folgende Bedeutung:

UID	Benutzer-Identifikation des Prozesseigentümers
PID	eindeutige Prozess-Nummer
PPID	Prozess-Nummer des Eltern-Prozesses (Parent PID)
STIME	Uhrzeit, wann der Prozess gestartet wurde
TIME	bisher vom Prozess verbrauchte CPU-Zeit
TTY	das dem Prozess zugeordnete Pseudo-Terminal
ADDR	Hauptspeicher- oder Plattenadresse des Prozesses
WCHAN	Adresse eines Events, auf den der Prozess wartet
CMD	Name des Kommandos, das den Prozess gestartet hat
PRI	aktuelle Priorität des Prozesses (je kleiner desto grösser die Priorität)
NI	nice-Parameter für die Prioritätsberechnung
SZ	Grösse des Prozesses im Hauptspeicher in kByte
C	systeminterner Scheduling-Parameter
F	systeminterne flags (hat nur noch historische Bedeutung)
S	Status (S=sleeping, O=running, R=runnable, Z=zombie, T=stopped)

Das **ps**-Kommando stellt ein wichtiges Hilfsmittel zur Prozess-Überwachung dar. Kombiniert mit den Kommandos **grep** und **sort** ermöglicht es verschiedene Selektionen aus der kompletten Prozess-Liste.

```
host> ps -ef | grep ssh  Selektion aller ssh-Prozesse
```

Es gibt noch weitere Kommandos, die Informationen über laufende Prozesse liefern. So geben die Befehle **ptree** unter Sun Solaris und **pstree** unter Linux alle Prozesse in hierarchischer Struktur aus.

```
host> ptree $LOGNAME | more  Prozess-Hierarchie unter Solaris
host> pstree -p $LOGNAME | more  Prozess-Hierarchie unter Linux
```

Beide Kommandos listen die Prozess-Struktur des eingeloggtten Benutzers auf.

Das Verhalten der eigenen Prozesse kann vom Prozesseigentümer gesteuert werden. So ist es möglich, Prozesse zu suspendieren, zu terminieren und mit verminderter Priorität zu starten. Auch das Absetzen von Hintergrundprozessen und die Beobachtung der Prozess-Aktivität wird in diesem Zusammenhang erwähnt.

## Prozess als Hintergrund-Prozess starten

```
command [ arguments ] &
nohup command [ arguments ] &
```

Ein Prozess wird durch Anfügen des Zeichens **&** in den Hintergrund versetzt. Als Information erhält man vom System eine relative [jobid] in eckigen Klammern und eine eindeutige Prozessidentifikationsnummer (PID). Standardausgabe- und Standardfehlerausgabe-Medium bleibt weiterhin das Pseudoterminal, von dem aus der Hintergrundprozess gestartet wurde. Das Starten eines Hintergrundprozesses mit dem Kommando **nohup** verhindert ein Terminieren des Prozesses beim Ausloggen aus dem System. Der Sinn von Hintergrundprozessen ist der, dass zum einen die interaktive Terminal-Eingabe nicht blockiert wird, und zum anderen ein Logout aus dem System möglich ist, ohne einen Abbruch des Hintergrundprozesses zu bewirken (beim Start mit **nohup**). Der Prozess wird mit **nohup** quasi aus der Hierarchie entkoppelt. Mit dem Kommando **jobs -l** werden alle gestarteten Hintergrundprozesse mit ihrem Status aufgelistet.

```
host> xclock &
[1] 22656

host> xterm &
[2] 22660

host> jobs -l

[1]  + 22656 Läuft           xclock
[2]  - 22660 Läuft           xterm
```

In vorliegendem Beispiel werden zwei graphische Anwendungen (**xclock**, **xterm**) als Hintergrundprozesse gestartet. In der Job-Liste werden die entsprechenden **jobids** und **PIDs** wiedergegeben. Ein Prozess kann auch noch nachträglich in den Hintergrund versetzt werden, indem nach der Kommandoingabe die Tastensequenz **<Ctrl> z** gedrückt wird und anschließend die Anweisung **bg** (für background) abgesetzt wird. Durch das **<Ctrl> z** wird der Vordergrundprozess suspendiert, also in einen Wartezustand versetzt.

### Prozess in den Vordergrund holen

```
%jobid
fg %jobid
```

Die **jobid** kann dazu benutzt werden, einen im Hintergrund laufenden Prozess in den Vordergrund zurückzuholen. Danach steht aber das Terminal für weitere Eingaben nicht mehr zur Verfügung. Allerdings kann mit **<Ctrl> z** eine Suspendierung erfolgen, mit der Alternative, den Prozess erneut in den Hintergrund schicken zu können.

```
%jobid &
bg %jobid
```

### Prozess suspendieren

```
stop %jobid
stop PID

host> stop 22656
host> jobs -l

[1]  + 22656 Angehalten (Signal)   xclock
[2]  - 22660 Läuft                 xterm
```

Für die Suspendierung von Prozessen kann neben der oben beschriebenen Tastensequenz **<Ctrl> z** auch das Kommando **stop** verwendet werden. Dabei kann man wahlweise die **jobid** oder die **PID** angeben.

## Prozess terminieren

```
kill PID
kill -9 PID

host> kill 22656
host> jobs -l

[1]      22656 Beendet                xclock
[2]  + 22660 Lauft                  xterm
```

Das vollstandige Beenden eines Prozesses wird mit dem Kommando **kill** erreicht. Mit der Option **-9** wird der Prozess bedingungslos abgebrochen, ohne diese Option versucht das System moglicherweise geoffnete Dateien noch ordnungsgema zu schlieen. In der Job-Liste wird der entsprechende Prozess als "Beendet" markiert.

## Prozess-Prioritat vermindern

```
nice [ +n|-n ] [ command ]
renice n PID

host> ps -l | egrep '(UID|22660)'

  F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
000 S  9523 22660  1244  0  69   0 -  1172 select pts/2    00:00:00 xterm

host> renice 5 22660

22660: Alte Prioritat: 0, neue Prioritat: 5

host> ps -l | egrep '(UID|22660)'

  F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
000 S  9523 22660  1244  0  72   5 -  1172 select pts/2    00:00:00 xterm
```

Das **nice**-Kommando startet einen Prozess mit verminderter Prioritat. Ohne die Angabe eines Wertes *n* wird standardmaig 4 gesetzt. Das **renice**-Kommando vermindert die Prioritat eines bereits laufenden Prozesses. Es ist nur eine Reduzierung der Prozess-Prioritat moglich. Je hoher der Wert *n*, desto niedriger ist die Prioritat. In der Prozess-Liste kann der gesetzte Prioritatswert in der Spalte NI abgelesen werden. Eine Verminderung der Prioritat ist immer dann sinnvoll, wenn verhindert werden soll, dass langlaufende Programme die gleichzeitige interaktive Arbeit zu sehr behindern. Unter Sun Solaris wird die Prioritat in der Spalte NI vom Ausgangswert 20 hochgezahlt.

## Prozess-Aktivitat beobachten

```
strace command [ arguments ]
strace -p PID

truss command [ arguments ]
truss -p PID
```

Das Kommando **strace** wird unter Linux, das Kommando **truss** unter Sun Solaris verwendet. Beide liefern als Output die vom System benutzten System-Calls.

Es existieren noch weitere Hilfsprogramme, mit denen die Prozess-Aktivität auf dem gesamten System beobachtet werden kann. Ein weit verbreitetes Tool ist das Programm **top**. **top** stellt alle Prozesse als dynamische Tabelle dar, die alle 5 bis 10 Sekunden aktualisiert wird.

```

10:33:27 up 4 days, 24 min, 2 users, load average: 0.26, 0.25, 0.16
75 processes: 74 sleeping, 1 running, 0 zombie, 0 stopped
CPU states: 0.6% user, 1.6% system, 0.0% nice, 97.8% idle
Mem: 254700K total, 211076K used, 43624K free, 968K buffers
Swap: 586364K total, 12300K used, 574064K free, 88064K cached

  PID USER      PRI  NI  SIZE  RSS SHARE STAT  %CPU  %MEM  TIME COMMAND
 27800 puw09523  16   0   952   952   744 R    0.9   0.3   0:02 top
   983 root       12  -10 83768 9.9M 2352 S <    0.5   3.9 1293m XFree86
 1177 puw09523  10   0  2264 2180 1580 S    0.1   0.8   0:14 xscreensaver
 1189 puw09523  10   0  3632 3260 2704 S    0.1   1.2   0:07 gnome-terminal
    1 root        8   0   476   436   416 S    0.0   0.1   0:11 init
    2 root        9   0     0     0     0 SW   0.0   0.0   0:00 keventd
    3 root       19  19     0     0     0 SWN  0.0   0.0   0:01 ksoftirqd_CPU0
    4 root        9   0     0     0     0 SW   0.0   0.0   0:16 kswapd
    5 root        9   0     0     0     0 SW   0.0   0.0   0:00 bdflood
    6 root        9   0     0     0     0 SW   0.0   0.0   1:25 kupdated
    7 root       -1 -20     0     0     0 SW<  0.0   0.0   0:00 mdrecoveryd
   23 root        9   0   700   668   532 S    0.0   0.2   0:00 devfsd
   99 root        9   0     0     0     0 SW   0.0   0.0   0:03 pagebuf_daemon
  117 root        9   0     0     0     0 SW   0.0   0.0   0:00 khubd
  200 root        9   0     0     0     0 SW   0.0   0.0   0:00 usb-storage-0
  201 root        9   0     0     0     0 SW   0.0   0.0   0:00 scsi_eh_0
  225 root        9   0   764   700   620 S    0.0   0.2   0:00 dhclient-2.2.x

```

Abbildung 4.3: Das Programm **top** zur Prozess-Beobachtung

Die Prozesse werden sortiert nach ihrem CPU-Verbrauch wiedergegeben, d.h. der Prozess, der am meisten CPU-Zeit verbraucht, steht ganz oben. Außerdem sind der Tabelle noch Angaben über Memory- und Swap-Ausnutzung zu entnehmen. Soll die Sortierung nach dem Memory-Verbrauch erfolgen, so kann dies mit der Tastenkombination **<Shift> M** initiiert werden. Eine Übersicht aller Funktionen ist durch Betätigung der Taste **h** abrufbar. Das Programm wird einfach durch Eingabe von **q** beendet.

Meist enthalten die graphischen Benutzeroberflächen auch noch Programme, die eine Beobachtung der Prozess-Aktivität zulassen. Unter Sun Solaris und CDE ist dies das Programm **sdtprocess**, unter Linux sind dies die Tools **gtop** und **kpm**.

## 5 Die csh/tcsh - Shell

Unter einer **shell** wird allgemein ein Kommando-Interpreter verstanden. Wie bei anderen Betriebssystemen übernimmt dieser Interpreter die Verarbeitung der eingegebenen Kommandos. Auf Unix/Linux-Systemen sind in der Regel folgende shells anzutreffen: **sh** (Bourne-shell), **bash** (Bourne-again-shell), **ksh** (Korn-shell), **jsh** (Job-Control-shell), **csh** (c-shell) und **tcsh** (eine Variante der csh, die vorzugsweise an der Uni Regensburg eingesetzt wird).

Was ist der Grund für diese Menge verschiedener shell-Varianten? So wie bei den Editoren besitzt auch jede shell gewisse Stärken, aber auch einige Schwächen. Es hängt vom geforderten Einsatz ab, welcher shell man den Vorzug gibt. Es gibt shells, die sich eher für die interaktive Arbeit eignen (**tcsh**, **bash**), andere sind dagegen für das Schreiben von shell-Prozeduren besser geeignet (**sh**, **ksh**).

Da an der Uni Regensburg die gesamte Benutzerumgebung auf die **tcsh** bzw. **csh** abgestimmt wurde, werden daher diese shells intensiver behandelt. Hinzu kommt, dass viele Mechanismen, die hier betrachtet werden, auch in anderen shell-Varianten in gleicher oder in ähnlicher Weise anzutreffen sind.

### 5.1 Kommandowiederholung

Alle Kommandos, die interaktiv eingegeben werden, werden in einer sog. *History-Liste* gesichert. Auf diese Liste kann zurückgegriffen werden, um bestimmte Kommandos wiederholt aufzurufen, ohne das gesamte Kommando nochmals eintippen zu müssen. Beim Login wird außerdem eine Datei namens `.history` angelegt, die die Speicherung der History-Liste übernimmt. So stehen die Kommandos sogar nach einem Logout beim nächsten Login wieder zur Verfügung. Die History-Liste kann man mit dem Kommando **history** ansehen.

```
host> history           Ausgabe der gesamten History-Liste
host> history | tail -5 Ausgabe der letzten 5 Kommandos
```

#### History-Kommandos

Die Referenz auf ein in der History-Liste stehendes Kommando erfolgt mit einem vorangestellten Rufezeichen **!**. Dies ist auch der Grund dafür, dass das Rufezeichen bei anderweitiger Verwendung maskiert werden muss.

```
host> !!    das zuletzt eingegebene Kommando wird ausgeführt
host> !17   das Kommando Nr. 17 in der History-Liste wird ausgeführt
host> !-4  das viertletzte Kommando wird ausgeführt
host> !vi  der letzte vi-Aufruf wird ausgeführt
```

Wie oben zu sehen, kann die Referenz auf ein Kommando der History-Liste entweder mit der Nummer oder mit einem Namen erfolgen. Beim Bezug auf einen Kommandonamen wird das letzte Kommando genommen.

## Kommando-Modifikation mittels History-Kommandos

```

0   1   2   3   4   5
ls -l berta hugo | more

```

Eine Kommando-Eingabe kann man sich als Aneinanderreihung einzelner Wörter vorstellen. Die Zeichen |, <, > und & gelten dabei als eigene Wörter. Auf die einzelnen Kommandobestandteile kann durch Angabe der entsprechenden Wortnummer in History-Kommandos Bezug genommen werden.

```

host> cp !!:2 ~/text2
      Die Datei berta wird in das Subdirectory text2 kopiert

host> chmod 666 !ls:2-3
      Der chmod-Befehl wird für die Dateien berta und hugo ausgeführt, dem
      zweiten und dritten Argument im letzten ls-Befehl

```

Für die Korrektur von Tippfehlern im zuletzt eingegebenen Kommando, gibt es eine einfache Möglichkeit.

```

host> more text2/briefe/ina.txt   für text2 soll text3 stehen
host> ^2^3                         Korrektur mit dem Zeichen ^

```

Neben den oben erwähnten History-Kommandos lässt die tcsh auch das Vorwärts- und Rückwärts-Blättern der History-Liste mit den Pfeiltasten zu. Dies ist für viele Benutzer oft einfacher, als sich die verschiedenen History-Modifikationen zu merken.

## 5.2 Vervollständigung von Kommandos, Variablen und Dateinamen

Mit der tcsh lässt sich eine Menge Tipp-Arbeit sparen. Bei der Eingabe von Kommandos, Variablen und Dateinamen genügt es, die ersten paar Zeichen zu schreiben und danach die <Tab>-Taste zu drücken, um den kompletten Namen zu vervollständigen.

```

host> fing<Tab>      wird zum Kommando finger komplettiert

host> fin<Ctrl> d    zeigt alle Kommandos an, die im Suchpfad mit fin
                   beginnen

find  find2perl  findaffix  finger

```

Gibt es mehrere Kommandos, Variable oder Dateinamen, die mit der gleichen Zeichenfolge beginnen, so kann mit der Tastensequenz <Ctrl> d eine Übersicht ausgegeben werden.

```

host> echo $ver<Tab>   Variablenname wird zu version komplettiert

host> echo $MAIL<Ctrl> d zeigt alle Variable an, die mit MAIL beginnen

MAIL  MAILCAPS  MAILSERVERNAME

host> more<Ctrl> d    Anzeige aller Dateien und Directories

briefe/  else  ina  rudi

host> more el<Tab>   Dateiname wird zu else komplettiert

```

Bei der Anwendung von <Ctrl> d auf Dateinamen werden Directories mit einem / hinter dem Namen gekennzeichnet. Durch mehrfaches Verwenden der <Tab>-Taste können auch längere Pfadnamen zusammengesetzt werden.

## 5.3 alias - Mechanismus

aliases sind *selbstdefinierte Kommandobezeichnungen*. Sie werden in erster Line deswegen definiert, um komplexere Kommandoangaben abzukürzen und den eigenen Bedürfnissen anzupassen.

<b>alias</b> <i>name definition</i>	Definition eines aliases
<b>unalias</b> <i>name</i>	Löschen einer alias-Definition
<b>alias</b>	Anzeigen aller gesetzten aliases

Durch die Eingabe von *name* werden die in der *definition* enthaltenen Kommandos ausgeführt.

```
host> alias ll ls -al
      Eingabe von ll führt das Kommando ls -al aus
host> alias sys 'ps -ef | more'
      Längere alias-Definitionen werden in Hochkommas eingeschlossen
```

### Verwendung von Argumenten in alias-Definitionen

!^ Platzhalter für das erste Argument des aktuellen Kommandos  
!\* Platzhalter für alle Argumente des aktuellen Kommandos

Für das find-Kommando soll ein alias definiert werden, bei dem genau das erste Argument (hier der Dateiname) übergeben werden soll.

```
host> alias ff 'find ~ -name \!^ -print'
host> ff filename
      Der filename wird in obigem alias als Argument eingesetzt
```

Es soll ein alias definiert werden, das ein Directory-Listing beliebig vieler Verzeichnisse seitenweise ausgibt (`1 dir1 dir2 dir3`).

```
host> alias l 'ls -l \!* | more'
```

Hier ist es erforderlich, dass alle Argumente in die Definition übernommen werden, da ja erst danach das seitenweise Auflisten mit dem more-Kommando folgt.

Weitere Beispiele:

```
host> alias cd 'cd \!*; set prompt="$cwd> "'
      Bei jedem cd wird das aktuelle Directory als prompt gesetzt
host> alias tm '(set d=`date`; echo $d[4])'
      Das Kommando tm gibt die aktuelle Uhrzeit aus
```

Bei der Interpretation von Kommandos wird folgende Abarbeitungs-Reihenfolge eingehalten: 1. Alias-Substitution, 2. Built-in Kommandos, 3. Kommandos im Suchpfad. Alias-Definitionen werden also immer als erste interpretiert. Dies ermöglicht eine Umdefinition von built-in und Suchpfad-Kommandos (Vorsicht: Umdefinition kann dazu führen, dass ursprüngliche Kommandos nicht mehr funktionieren). Sollen alias-Definitionen auch nach dem nächsten Einloggen und in untergeordneten shells zur Verfügung stehen, so sind sie in die Datei `.cshrc` bzw. `.myalias` einzutragen. Will man eingetragene Definitionen sofort aktivieren, so geschieht dies mit dem Kommando `source .cshrc` bzw. `source .myalias`.

## 5.4 Ein- und Ausgabeumlenkung

Die Ein- und Ausgabe von Kommandos erfolgt in der Regel innerhalb eines Terminals. Werden dagegen Dateien für die Eingabe bzw. Ausgabe verwendet, so spricht man von *Ein-* bzw. *Ausgabeumlenkung*.

### Ausgabeumlenkung

Bei der Ausgabe wird zwischen einer *Standardausgabe* und einer *Standardfehlerausgabe* unterschieden. Die Standardausgabe enthält alle Daten, die ein Kommando bei seinem regulären Ablauf ausgibt. Die Standardfehlerausgabe besteht dagegen aus Diagnose-Meldungen, die auf Probleme bei der Abarbeitung hinweisen sollen.

```

command > filename      Ausgabe wird in die Datei filename geschrieben
command >> filename     Ausgabe wird dem Inhalt der Datei filename angehängt
command >& filename      Diagnose-Meldungen werden in die Datei geschrieben
command >>& filename     Diagnose-Meldungen werden dem Inhalt der Datei angehängt

```

Wird nur die Ausgabeumlenkung für Diagnose-Meldungen mit `>&` definiert, so wird sowohl die Standardausgabe als auch die Standardfehlerausgabe in die gleiche Datei umgelenkt.

```

host> date > dirliste
      Die Ausgabe des aktuellen Datums wird in die Datei dirliste umgelenkt
host> ls -al >> dirliste
      Der Datei dirliste wird das Directory-Listing angefügt
host> (date; ls -al) > dirliste
      Die Ausgabe beider obigen Kommandos wird in einem Aufruf mittels Kommandogruppierung in die Datei dirliste umgelenkt.
host> find /var -name mail -print >& err_output
      Das find-Kommando sucht im Verzeichnis var nach dem Namen mail, wobei alle Diagnose-Meldungen in die Datei err_output umgelenkt werden

```

Das letzte Beispiel dokumentiert das Problem, Ausgabe und Fehlerausgabe zu trennen. Bei obigem `find`-Kommando werden daher nicht nur die Diagnose-Meldungen, sondern auch die Standardausgaben in die Datei `err_output` umgeleitet. Reguläre Ausgaben des `find`-Kommandos stehen somit vermischt zwischen den Fehlerausgaben. Sollen die Standardausgaben und die Fehlerausgaben in getrennte Dateien erfolgen, so muss das Kommando folgendermaßen abgesetzt werden.

```

host> (find /var -name mail -print > output) >& err_output

```

Preisfrage: Wie muss das `find`-Kommando aussehen, wenn die Standardausgabe ganz normal auf das Terminal, die Standardfehlerausgabe aber in eine Datei erfolgen soll?

Die Ausgabeumlenkung wird häufig in Zusammenhang mit Hintergrundprozessen verwendet, die mit dem `nohup`-Befehl gestartet wurden und nach einem Logout weiterlaufen. Da hierbei kein Terminal für die Ausgabe existiert, ist es wichtig, dass der Prozess ev. auftretende Ausgaben in eine Datei schreiben kann.

Mit der Umgebungs-Variablen `noclobber` können versehentliche Überschreibungen von Dateien verhindert werden. Ist die Datei, in die der Output geschrieben werden soll, bereits vorhanden, so erscheint die Meldung `datei existiert`. Bei Dateien, die für eine Anfügung von Output ausgewählt werden, wird `datei` oder `verzeichnis` nicht gefunden ausgegeben, falls die Datei noch nicht existiert. Will man diesen Sicherheits-Mechanismus wieder ausschalten, so muss die Variable `noclobber` deaktiviert werden (`unset noclobber`). Bei gesetztem `noclobber` können einzelne Überschreibungen durch Anfügen eines `!` an den Umlenkungs-Operator realisiert werden (`>!` `>>!` `>&!` `>>&!`).

## Eingabeumlenkung

Bei der Eingabe-Umlenkung wird der Inhalt einer Datei als Input für ein Kommando verwendet.

```
command < filename  Inhalt der Datei filename wird als input verwendet
command << wort    Unmittelbare Eingabe von Text, der mit wort bzw.
                  <Ctrl> d abgeschlossen wird (here document)
```

Die zweite Form der Eingabeumlenkung ist vor allem in Zusammenhang mit der Shellprogrammierung interessant.

```
host> mailx rar12345 < brief
host> tr "[A-Z]" "[a-z]" < report > report.low
```

Obiges `tr`-Kommando ist ein Beispiel für das gleichzeitige Verwenden von Ein- und Ausgabe-Umlenkung. Alle Grossbuchstaben in der Datei `report` werden in Kleinbuchstaben konvertiert und das Resultat in der Datei `report.low` abgelegt. Die Datei `report` wird dabei nicht verändert.

## 5.5 Variable

Variable werden zur Speicherung alphanumerischer Daten verwendet. Variablenamen beginnen immer mit einem Buchstaben. Man unterscheidet **lokale** und **globale** Variable. Lokale Variable sind nur innerhalb der aktuellen (current) shell bekannt. Globale Variable werden an untergeordnete Prozesse weitervererbt.

### 5.5.1 Lokale Variable

Für die Definition von lokalen Variablen existieren zwei Formen:

```
set varname = wert  wird für beliebige Werte verwendet
@ varname = wert    wird für numerische Werte verwendet
unset varname       Eliminiert eine lokale Variable
set                 zeigt alle definierten Variablen an
```

Die Referenz auf bestimmte Variablen wird durch ein vorangestelltes Dollarzeichen erreicht. Mit dem `echo`-Kommando wird der Inhalt einer Variablen ausgegeben.

```
echo $varname      gibt die Variablenwerte aus
echo $#varname     gibt die Anzahl der Argumente in der Variablen aus
```

Beispiele:

```
host> set m1 = doris else rudi
      Definition der Variablen m1

host> echo $m1
      es wird nur doris ausgegeben, da Leerzeichen als Trennzeichen wirken

host> set m1 = 'doris else rudi'
host> set m1 = "doris else rudi"
host> echo $m1
      nun werden in beiden Fällen alle drei Namen angezeigt
```

Man beachte in obigen Variablen-Definitionen die unterschiedliche Wirkungsweise der Maskierung. Einfache Hochkommata maskieren *alle* eingeschlossenen Zeichen, während dagegen Anführungszeichen die Interpretation der Zeichen \$, \ und ` zulassen.

Mit dem Kommando **echo \$#m1** kann die Anzahl der in der Variablen enthaltenen Argumente ausgegeben werden. Im vorliegenden Beispiel beträgt dieser Wert 1, obwohl die Variable `m1` aus 3 Namen besteht. Soll eine Variable aus einer indizierbaren Wortliste bestehen, so muss die Variable folgendermaßen definiert werden:

```
set varname = ( wert1 wert2 wert3 ... )
```

Die Variable besteht nun aus einzelnen Argumenten. Ein indizierter Zugriff auf ein einzelnes Argument erfolgt mit **set varname[n]** bzw. mit **\$varname[n]**.

Beispiele:

```
host> set m1 = ( doris else rudi )
      Definition der Variable m1 als word-list

host> echo $#m1          liefert jetzt den Wert 3
host> set m1[2] = peter  ersetzt das zweite Argument (else) durch
                        peter
host> echo ${m1}ratlos   Verkettung mit Textstring
```

Das letzte Beispiel demonstriert die Verkettung einer Variablen mit Text. Damit der Variablenname eindeutig interpretiert werden kann, muss er in geschweiften Klammern eingeschlossen werden. Als Ausgabe erhält man in obigem Fall `doris peter rudiratlos`.

### Zuweisungsoperationen mit numerischen Variablen

In folgender Tabelle wird von dem Wert 6 als Ausgangswert für die Variable `count` ausgegangen. Das Ergebnis kann mit dem Kommando **echo \$count** angezeigt werden.

Symbol	Beispiel	Ergebnis
=	@ count = 0	0
+=	@ count += 2	8
--	@ count -= 5	1
*=	@ count *= 8	48
/=	@ count /= 4	1
++	@ count ++	7
-	@ count -	5

## Arithmetische, bitweise und logische Operationen mit numerischen Variablen

Der Ausgangswert der Variable `num` soll hier 10 betragen, das Ergebnis wird mit `echo $r` ermittelt.

Symbol	Beispiel	Ergebnis
+	@ r = \$num + 4	14
-	@ r = \$num - 5	5
*	@ r = \$num * 10	100
/	@ r = \$num / 6	1
%	@ r = \$num % 3	1
»	@ r = (\$num » 2)	2
«	@ r = (\$num « 3)	80

Symbol	Beispiel	Ergebnis	
~	@ r = ~ \$num	-11	1 Komplement
!	@ r = ! \$num	0	logische Negation
	@ r = (\$num   7)	15	bitweise or
^	@ r = (\$num ^ 7)	13	bitweise excl. or
&	@ r = (\$num & 6)	2	bitweise and
	@ r = (\$num > 8    \$num < 4)	1	logisch or
&&	@ r = (\$num > 4 && \$num < 8)	0	logisch and

## Vergleichsoperationen

In dieser Tabelle beträgt der Anfangswert von `num` 5, die Variable `str` besitzt den Wert `xyz` und die Variable `pat` den Wert `abc`.

Symbol	Beispiel	Ergebnis
==	\$num == 5	true
!=	\$num != 5	false
>	\$num > 3	true
>=	\$num >= 3	true
<	\$num < 3	false
<=	\$num <= 5	true
==	\$str == xyz	true
!=	\$str != abc	true
=~	\$pat =~ [abc]*	true
!~	\$pat !~ *c	false

Vergleichsoperationen werden in Kontrollstrukturen benötigt (Beispiel: `if ($pat =~ [abc]*) echo richtig`).

## Besondere lokale Variable

Es existiert eine Reihe von Variablen, die allgemein definiert sind und vor allem in Zusammenhang mit shell-Prozeduren wichtige Funktionen übernehmen. Sie dienen der Übergabe von Argumenten und Systemwerten sowie dem interaktiven Input.

<code>\$argv</code>	enthält alle Argumente des Prozeduraufrufs
<code>\$*</code>	" " " " "
<code>\$argv[*]</code>	" " " " "
<code>\$n</code>	enthält das n-te Argument des Prozeduraufrufs
<code>\${n}</code>	" " " " "
<code>\$argv[n]</code>	" " " " "
<code>\$#</code>	enthält die Anzahl der Argumente
<code>\$#argv</code>	" " " "
<code>\$0</code>	enthält den Namen der aufgerufenen Prozedur
<code>\$argv[0]</code>	" " " " "
<code>\$_</code>	enthält den Namen des zuletzt ausgeführten Kommandos
<code>\$\$</code>	enthält die Process-Id der ausführenden shell
<code>\$status</code>	enthält den exit-status des zuletzt ausgeführten Kommandos
<code>\$?</code>	" " " " "
<code>\$&lt;</code>	liest eine Zeile von der Tastatur

Um beispielsweise den Tastaturinput einer Variablen zuzuweisen, definiert man `set input = $<`. Der exit-status eines Kommandos besitzt den Wert 0 nach erfolgreicher und ungleich 0 nach fehlerhafter Ausführung.

### 5.5.2 Globale Variable

Während lokale Variable nur in der aktuellen shell definiert sind, werden globale Variable auch an untergeordnete Prozesse weitervererbt. Die Definition globaler Variablen ist dann sinnvoll, wenn Variablenwerte im untergeordneten Prozess-Baum wirksam sein sollen. Globale Variable, die bereits nach dem Login zur Verfügung stehen sollen, werden in den setup-Dateien `.cshrc`, `.login`, `.profile` bzw. `.dtprofile` definiert.

<b>setenv</b> <i>varname wert</i>	Definition einer globalen Variablen
<b>unsetenv</b> <i>varname</i>	Eliminiert eine globale Variable
<b>echo</b> <i>\$varname</i>	Ausgabe des Variablenwertes
<b>setenv</b>	zeigt alle definierten globalen Variablen an
<b>printenv</b>	zeigt alle definierten globalen Variablen an
<b>env</b>	zeigt alle definierten globalen Variablen an

Viele globale Variable werden auch als *Environment-Variable* bezeichnet, da sie die Benutzer-Umgebung definieren. Auch die Namen globaler Variablen beginnen mit einem Buchstaben und können in Gross- oder Kleinschrift gesetzt werden. Es hat sich allerdings eingebürgert, globale Variable generell mit Grossbuchstaben zu schreiben um sie besser von lokalen Variablen unterscheiden zu können.

```
host> setenv STADT Regensburg  Definition der globalen Variablen STADT
host> echo $STADT             Ausgabe des Wertes von STADT
```

### Wichtige globale Variable

Standardmäßig werden beim Login bereits eine Reihe von globalen Variablen definiert, die die Benutzerumgebung festlegen. So wird in der Variablen `HOME` das Login-Verzeichnis gespeichert, in der Variablen `PATH` steht der Suchpfad für Kommandoeingaben. Die Variable `MAIL` enthält die Unix-Mailbox und in der Variablen `SHELL` ist die Login-shell abgelegt. Manche lokale Variable werden beim Login gleich in globale Variable exportiert (`path -> PATH`).

## 5.6 Shell-Programmierung

Unter Shell-Programmierung versteht man das Schreiben von Kommando-Prozeduren. Diese Kommando-Prozeduren sind nichts anderes als Dateien, die shell-Kommandos enthalten. In der ersten Zeile einer Kommando-Prozedur sollte dem System mitgeteilt werden, unter welcher shell die Prozedur abgearbeitet werden soll. Die erste Zeile bei `csch`-Prozeduren lautet daher `#!/bin/csh`, bei `tcsh`-Prozeduren `#!/usr/local/bin/tcsh` bzw. `#!/bin/tcsh`. Ansonsten gilt das Zeichen `#` als Kommentarzeichen.

```
#!/usr/local/bin/tcsh
#
# Datei: filecount
# script zum zaehlen von Dateien und Directories
# 02.05.97 - WP
#
        echo -n "Dateien und Directories"
        ls -al | wc -l
        exit
```

Nach dem Editieren der Datei müssen noch die `execute`-Rechte gesetzt werden, damit die Prozedur auch ausgeführt werden kann.

```
host> chmod u+x filecount   execute-rechte werden gesetzt
host> ./filecount           Aufruf der Prozedur
```

Werden `csch`- bzw. `tcsh`-Skripten mit Argumenten aufgerufen, so werden die Argumente auf die besonderen Variablen `$n`, `$argv[n]` abgebildet. In der Prozedur kann dann auf diese Variablen referenziert werden.

```
#!/usr/local/bin/tcsh
#
# Datei: arguments
# script zur Ausgabe von drei Input-Argumenten
#
# Usage: arguments arg1 arg2 arg3
# 02.05.97 - WP
#
        echo $1
        echo $argv[2]
        echo $3 $2 $1
        echo "$3 $2 $1"
        exit
```

Der Aufruf der Prozedur erfolgt mit Eingabe von Argumenten, die mit Leerzeichen voneinander getrennt werden. Die ersten drei Argumente werden den Variablen `$1`, `$2` und `$3` zugewiesen.

```
host> ./arguments ina otto doris

ina
otto
doris otto ina
doris otto ina
```

Um eine Kontrolle über den Ablauf einer shell-Prozedur zu erhalten, ist es erforderlich sog. Steuerstrukturen einzubauen. Wie eine Programmiersprache kennt auch die tcsh bzw. csh derartige Kontrollstrukturen.

### 5.6.1 Die if - Anweisung

```
if ( expression ) command

if ( expression ) then
  command(s)
else if ( expression ) then
  command(s)
else
  command(s)
endif
```

Die if-Anweisung existiert also als einzelne Anweisung oder als if-Block. Ein if-Block kann elseif- und else-Zweige besitzen. Es ist darauf zu achten, dass vor und hinter den runden Klammern mindestens ein Leerzeichen steht.

```
#!/bin/csh
#
# Datei: if.csh
# Beispiel zur if-Anweisung
#
# Usage: if.csh arg1 arg2 arg3 ...
# 02.05.97 - WP
#
    if ( $#argv <= 0 ) then
        echo {$0}: Aufruf ohne Argumente
        exit
    else
        echo {$0}: Es wurden $#argv Argumente uebergeben
    endif
exit
```

### 5.6.2 Die foreach - Schleife

```
foreach var ( wordlist )
  command(s)
end
```

Jedes Wort aus der *wordlist* wird der Variablen *var* zugewiesen und die *command(s)* ausgeführt. Die foreach-Schleife wird so oft durchlaufen, wie Elemente in der *wordlist* enthalten sind.

```
#!/bin/csh
#
# Datei: foreach.csh
#
    foreach dir ($path)
        ls $dir
    end
```

Obiges Skript verwendet in der foreach-Schleife die Variable `$path` als wordlist. Es werden alle Verzeichnisse gelistet, die im Suchpfad stehen.

```
#!/bin/csh
#
# Datei: foreach2.csh
#
    foreach datei (x1 x2 x3)
        echo $datei
        echo "$datei enthaelt diese Zeile" > $datei
    end
```

Diese Prozedur legt die drei Dateien namens `x1`, `x2` und `x3` an mit jeweils einer Zeile Inhalt.

### 5.6.3 Die while - Schleife

```
while ( expression )
    command(s)
end
```

Die while-Schleife kann mit numerischen oder string-Variablen benutzt werden. Die Schleife wird solange durchlaufen, bis *expression* false ist.

```
#!/bin/csh
#
    @ i = 1
    while ( $i <= 10 )
        echo "$i - te Schleife"
        @ i++
    end
```

Für das zeilenweise Einlesen einer Datei eignet sich die while-Schleife in folgender Form.

```
#!/bin/csh -f
#
# Zeilenweise Ausgabe einer Datei
#
if ($#argv <= 0) then
    echo " Usage: $0 filename < filename"
    exit
endif
#
    set a = `wc -l $1 | awk '{print $1}'`
    shift
#
    set i = 1
    while ( $i <= $a )
        set input = $<
        echo Zeile $i\: $input
        @ i++
    end
```

Für die Ausgabe der Usage wird in obiger Prozedur die Variable \$0 verwendet. In ihr wird der Name der Datei abgelegt, die die Prozedur enthält. Die Maskierung des Doppelpunktes beim zweiten echo-Statement ist notwendig, damit die shell dies nicht als Bestandteil eines History-Kommandos interpretiert. Das zeilenweise Einlesen einer Datei kann unter der bash einfacher gelöst werden.

#### 5.6.4 Die shift - Anweisung

**shift**                    wirkt auf die Argumentenliste argv  
**shift** *varname*        wirkt auf die Variable *varname*

shift verschiebt die Argumente einer wordlist nach links. Das ganz links stehende Argument fällt aus der wordlist (Argumentenliste) heraus.

```
#!/bin/csh
#
# Datei: shift.csh
# Beispiel fuer shift-Statement
#
    set line = $<
    set input = ( $line )
    foreach element ( $input )
        echo $input
        shift input
    end
```

#### 5.6.5 Fallunterscheidung mit switch und case

```
switch ( string )
    case pattern1:
        command(s)
    breaksw
    case pattern2:
        command(s)
    breaksw
    ...
    ...
    default:
        command(s)
    breaksw
endsw
```

Es wird nach einer Übereinstimmung von *string* und *pattern* gesucht und die dort befindlichen Kommandos ausgeführt. Wird keine Übereinstimmung gefunden, so werden die Kommandos unter *default* abgearbeitet. Im *pattern-string* können die *pattern-matching character* \*, ? und [ ] verwendet werden.

```
#!/bin/csh
#
# Datei: case.csh
# Beispiel fuer case
```

```

#
    set line = $<
    set input = ( $line )
    foreach element ( $input )
        switch ( $element )
            case [aA]*:
                echo "Name $element beginnt mit einem A"
                breaksw
            case [hH]*:
                echo "Name $element beginnt mit einem H"
                breaksw
            case [rR][uU][dD][iI]:
                echo "Der Name lautet $element"
                breaksw
            default:
                echo "Name interessiert mich nicht"
                breaksw
        endsw
    shift input
end

```

### 5.6.6 Die break- und continue-Anweisung

Ein **break** bewirkt den Abbruch der Abarbeitung innerhalb einer Schleife; die Abarbeitung wird beim ersten Kommando hinter der Schleife fortgesetzt.

```

#!/bin/csh
#
    set line = $<
    set input = ( $line )
    foreach element ( $input )
        if ( $element == "otto" ) break
        echo $element
    end
    echo "Jetzt bin ich hinter der Schleife"

```

**continue** bewirkt den Sprung zum Ende einer Schleife, wobei die Schleife wiederholt durchlaufen wird, solange die Bedingung dafür true ist.

```

#!/bin/csh
#
    set line = $<
    set input = ( $line )
    foreach element ( $input )
        if ( $element == "otto" ) continue
        echo $element
    end
    echo "Jetzt bin ich hinter der Schleife"

```

### 5.6.7 Die goto-Anweisung

Es wird ein unbedingter Sprung zum bezeichneten Label ausgeführt und mit der Abarbeitung der dahinter folgenden Kommandos begonnen. Das Label muss mit einem Doppelpunkt abgeschlossen am Zeilenanfang stehen.

```
#!/bin/csh
#
    set line = $<
    set input = ( $line )
    foreach element ( $input )
        if ( $element == "otto" ) goto ottolabel
        echo $element
    end
    exit
#
ottolabel:
    echo "otto geht ins schwimmbad"
    exit
```

### 5.6.8 Allerlei Nützliches

#### Eingabe von Tastatur

Die interaktive Eingabe über die Tastatur erfolgt mit der besonderen Variablen \$<. Der Cursor bleibt solange stehen, bis die Eingabe abgeschlossen ist. Der eingegebene Text wird der im set-Kommando definierten Variablen übergeben (**wichtig:** bei tcsh-Skripten wird mangels quoting nur das erste Wort übergeben).

```
set varname = $<
```

Dabei ist zu beachten, dass der eingegebene Text als einzelner kompletter string und nicht als Wortliste interpretiert wird. Soll aus dem eingegebenen Text eine wordlist entstehen, so muss der Inputstring in eine neue Variable übernommen werden. Dies gilt aber nur für csh-Skripten.

```
set line = $<           Eingabe wird als string der Variablen line zugewiesen
set input = ( $line )   Umwandlung des strings in eine wordlist
```

#### Prüfen von Dateieigenschaften

Die tcsh besitzt mit dem builtin-Kommando **filetest** eine Möglichkeit, die Eigenschaften von Dateien abzufragen. Als Ergebnis des Kommandos wird entweder false 0 oder true 1 zurückgeliefert.

```
filetest -r filename  Datei lesbar
filetest -w filename  Datei schreibbar
filetest -x filename  Datei ausführbar
filetest -e filename  Datei existiert
filetest -z filename  Datei ist leer
filetest -f filename  normale Datei
filetest -d filename  Directory
filetest -l filename  Datei ist ein symbolic link
```

In `tcsh`-Prozeduren wird diese Art des Tests meist in Zusammenhang mit der `if`-Anweisung verwendet.

```
if ( `filetest -e .cshrc` ) echo "Datei existiert"
```

Man beachte, dass das `filetest`-Kommando in umgekehrte einfache Hochkommas eingeschlossen werden muss.

Die `csh` kennt die gleichen Operatoren für das Prüfen von Dateieigenschaften. Dabei genügt es, den Operator zusammen mit dem Dateinamen in einem Ausdruck zu verwenden. Diese Schreibweise kann auch in `tcsh`-Prozeduren kompatibel benutzt werden.

```
if ( -e .cshrc ) echo "Datei existiert"
```

### Messung des Zeitverbrauchs mit `time`

Zur Messung des Zeitverbrauchs eines Shell-Skripts kann das builtin-Kommando `time` verwendet werden. `time` wird einfach dem Prozedur-Aufruf vorangestellt.

```
host> time ./arguments ina otto doris

ina
otto
doris otto ina
doris otto ina
0.07u 0.02s 0:00.12 75.0%
```

Obige Zeitangaben sagen aus, dass der Prozess 0.07 CPU-Sekunden im user-mode und 0.02 CPU-Sekunden im kernel-mode gelaufen ist. Die Gesamtlaufzeit beträgt 0.12 Sekunden, wobei nur 75% davon die summierte CPU-Zeit ausmacht.

### Fehlersuche in Shell-Prozeduren

Um Fehler in `csh`-Prozeduren aufzuspüren, muss die Prozedur explizit in einer eigenen `csh` aufgerufen werden. Für die Fehlerdiagnose sind spezielle Optionen der `csh` zu verwenden.

```
csh -vx skript arg1 arg2
```

Der obige `csh`-Aufruf einer Prozedur gibt jede Zeile vor und nach der Variablen-, File- oder Kommando-Substitution aus. So kann die Abarbeitung eines Shell-Skriptes gut nachvollzogen werden. Fehlerhafte Bereiche können schnell lokalisiert werden.



## 6 Die bash - Shell

Die **bash** wird als Kommandointerpreter unter Linux häufig verwendet. Sie ist der Nachfolger der klassischen Bourne-shell, daher auch ihr Name "bourne again shell". Analog zur **tcsh** werden hier die Besonderheiten dieser shell behandelt.

### 6.1 Kommandowiederholung

Wie unter der **tcsh** existiert die Möglichkeit, auf frühere Kommandos zurückzugreifen. Dies erfolgt ebenfalls mit dem Kommando **history**. Aus der erscheinenden Liste kann dann über die Nummer das gewünschte Kommando ausgeführt werden.

```
host> history

    9  ls /tmp
   10  echo $SHELL
   11  ps -ef
   12  finger
   13  history

host> !11
```

In obigem Beispiel wird das Kommando mit der Nummer 11 mit vorangestelltem Rufezeichen nochmals aufgerufen. Natürlich ist auch die Verwendung der Pfeiltasten (Pfeil nach oben/unten) für die Kommandorückholung möglich.

### 6.2 Vervollständigung von Kommandos, Variablen und Dateinamen

Die **bash** verfügt ebenfalls über einen Mechanismus zur Vervollständigung von Kommandos, Variablen und Dateinamen. Die Tipparbeit wird dadurch reduziert.

```
host> fin<Tab>      wird zum Kommando finger komplettiert
host> fin<Tab><Tab> zeigt alle Kommandos an, die im Suchpfad mit fin
                    beginnen

find  find2perl  findaffix  findsmb  finger
```

Gibt es mehrere Kommandos, Variable oder Dateinamen, die mit der gleichen Zeichenfolge beginnen, so kann mit der Tastensequenz <Tab><Tab> eine Übersicht ausgegeben werden.

```
host> echo $LOG<Tab>      Variablenname wird zu LOGNAME komplettiert
host> ls -l /etc/ali<Tab>  Vervollständigung zu /etc/aliases
```

Obige Beispiele zeigen die Namens-Vervollständigung für Variablen- und Dateinamen.

## 6.3 alias - Mechanismus

Mit dem builtin-Kommando **alias** werden Aliase definiert. Eine Übernahme von Eingabe-Parametern ist hier aber nicht möglich.

```
alias name=definition  Definition eines Aliases
unalias name          Löschen einer alias-Definition
alias                  Anzeigen aller gesetzten Aliases
```

```
host> alias sys='ps -ef | more'
```

Oben gesetztes alias zeigt alle auf dem System laufenden Prozesse seitenweise an. Sollen unter der bash alias-Definitionen nach jedem Login zur Verfügung stehen, so sind sie in die Datei `.bashrc` im Home-Verzeichnis einzutragen.

## 6.4 Ein- und Ausgabeumlenkung

Für die Ein- und Ausgabeumlenkung werden auch die Operatoren `>`, `>>`, `>&`, `>&&`, `<` und `<<` verwendet.

```
command > filename      Ausgabe wird in die Datei filename geschrieben
command >> filename     Ausgabe wird dem Inhalt der Datei filename angehängt
command >& filename     Ausgabe und Fehler-Ausgabe in die Datei filename
command >&& filename    Ausgabe und Fehler-Ausgabe werden dem Inhalt der Datei
                           angehängt
command < filename     Inhalt der Datei filename wird als input verwendet
command << word       Eingabe von Text, der mit word bzw. <Ctrl> d abge-
                           schlossen wird (here document)
```

```
host> tr ''[a-z]'' ''[A-Z]'' << ende
> hallo
> hallo
> ende
HALLO
HALLO
```

Obiges Beispiel eines *here documents* zeigt, dass die Eingabeumlenkung bis zum Erscheinen des Wortes ende erfolgt.

Zusätzlich zu den Mechanismen der c-shell-Varianten erlaubt die bash die Verwendung von Filedeskriptoren bei der Ein-/Ausgabeumlenkung. Dabei sind die Werte für die Standardeingabe (Tastatur) **0**, für die Standardausgabe (Bildschirm) **1** und die Standardfehlerausgabe (Bildschirm) **2** bereits vordefiniert. Weitere Filedeskriptoren können mit dem Kommando **exec** gesetzt werden (bis maximal 9).

```
fd<>datei  Öffnet datei zum Lesen und Schreiben mit filedescriptor fd
<&fd       Datei mit filedescriptor fd wird als Standardeingabe verwendet
>&fd       Datei mit filedescriptor fd wird als Standardausgabe verwendet
fd<&-      Schließen der Standardeingabedatei mit filedescriptor fd
fd>&-      Schließen der Standardausgabedatei mit filedescriptor fd
```

Ein kleines Beispiel soll die Verwendung von Filedescriptoren verdeutlichen.

```

host> exec 4<>datei_inout           Datei wird mit mit filedescriptor 4 zum
                                     Lesen und Schreiben geöffnet

host> echo "Heute regnet es" >&4     Ausgabeumlenkung in Datei mit filedes-
                                     criptor 4

host> 4>&-                           Schließen der Datei mit filedescriptor 4

host> cat datei_inout                 Ausgabe des Inhalts der Datei
Heute regnet es

```

Bei Ausgabeumlenkungen verhindert die shell-Variablen `noclobber` ein Überschreiben von bereits vorhandenen Dateien. Mit dem Operator `>|` kann dann ein Überschreiben erzwungen werden.

## 6.5 Variable

Wie auch die Bourne-shell kennt die `bash` Positionsparameter und Shellvariablen. Die Positionsparameter werden mit `$1`, `$2` usw. referenziert, Namen von Shellvariablen können frei definiert werden.

Für die Definition von Variablen existieren zwei Formen.

<code>varname=wert</code>	Definition einer lokalen Variablen
<code>declare varname=wert</code>	Definition einer lokalen Variablen
<code>unset varname</code>	Eliminiert eine lokale Variable
<code>declare</code>	zeigt alle definierten Variablen an
<code>set</code>	zeigt alle definierten Variablen an

Sowohl bei der direkten Zuweisung als auch bei der Definition mittels `declare` dürfen zwischen dem Variablennamen und dem Wert keine Leerzeichen stehen. Die Wertezuweisung mit `declare` hat den Vorteil, dass die Variable mit einem Attribut versehen werden kann. So wird mit der Option `-i` die Variable eindeutig als Integervariable identifiziert.

Variablen, die in oben dargestellter Schreibweise definiert werden, sind zunächst nur auf dem selben Shell-Level gültig (lokale Variable). Sollen Variablen als globale Variablen in untergeordnete Shell-Ebenen weitervererbt werden, so sind sie folgendermaßen zu definieren.

<code>export varname=wert</code>	Definition einer globalen Variablen
<code>declare -x varname=wert</code>	Definition einer globalen Variablen

Wie unter jeder anderen Linux-Shell gibt es auch eine Reihe von festgesetzten Standardvariablen.

<code>\$*</code>	enthält alle Positionsparameter als ein string
<code>\$@</code>	enthält alle Positionsparameter als einzelne strings
<code>\$n</code>	enthält den n-ten Positionsparameter
<code>\$#</code>	enthält die Anzahl der Positionsparameter
<code>\$0</code>	enthält den Namen der aufgerufenen Prozedur
<code>\$_</code>	enthält das letzte Argument des vorherigen Kommandos
<code>\$\$</code>	enthält die Process-Id der ausführenden shell
<code>\$?</code>	enthält den exit-status des zuletzt ausgeführten Kommandos

Neben diesen eingebauten lokalen Variablen gibt es auch eine Vielzahl von globalen bzw. environment-Variablen, die standardmäßig gesetzt werden. Hier sollen nur einige als Beispiel erwähnt werden.

```
$BASH    enthält den vollen Pfadnamen der bash
$HOME    enthält das Home-Verzeichnis des Benutzers
$OSTYPE  enthält den Namen des Betriebssystems
$RANDOM   liefert immer eine neue Zufallszahl zw. 0 und 32767
```

Werden außer den standardmäßig definierten Variablen beim Login noch weitere Variablen für diverse Zwecke benötigt, sollten sie in der Datei `.bashrc` eingetragen werden.

## 6.6 Arithmetik

Die ursprüngliche Bourne-shell kannte nur eine sehr eingeschränkte Form der Arithmetik. Für arithmetische Operationen stand das externe Kommando `expr` zur Verfügung. Bei der Entwicklung der `bash` wurde daher der Umfang für arithmetische Operationen erheblich erweitert. Im wesentlichen wurden die Funktionen der C-Shell-Varianten zusätzlich implementiert, wobei man die alte Form der Arithmetik aus Kompatibilitätsgründen beibehalten hat.

Arithmetische Ausdrücke werden in einfachen eckigen oder doppelten runden Klammern eingeschlossen. Die Operatoren haben die gleiche Bedeutung wie bei der `tcsh`.

```
host> echo ${7 + 5 * 6}   Ausdruck ergibt den Wert 37
host> echo $( (15 / 5) )  Ausdruck ergibt den Wert 3
host> Y=$((8 << 2))      Y wird der Wert 32 zugewiesen
```

Eine weitere Möglichkeit arithmetische Ausdrücke auszuwerten, bietet das Kommando `let`. Falls das Kommando erfolgreich ist, liefert es den exit-Status 0, andernfalls den exit-Status 1.

```
host> let X=(3 + 4)*10    Zuweisung in die Variable X
host> echo $X             Ausgabe des Variablenwertes von X (70)
host> let Z=(3 + 4)*10+Y  falls Y 32 beträgt, ist das Ergebnis 102
```

## 6.7 Shell-Programmierung

Nutzt man die Funktionalität der `bash` für das Schreiben von Kommando-Prozeduren, sollte in der ersten Zeile der Prozeduren `#!/bin/bash` stehen.

```
#!/bin/bash
#
# Datei: arguments.bash
# Usage: arguments arg1 arg2 arg3
# 24.09.07 - WP
#
    echo $1
    echo $3 $2 $1
    echo "$3 - $2 - $1"
    exit
```

Wie auch bei anderen Linux-Shells muss die Skript-Datei mit `chmod` ausführbar gesetzt werden. Der Aufruf erfolgt mit dem Dateinamen und den einzugebenden Parametern. Diese Positionsparameter werden in der Prozedur wie gewohnt mit `$1`, `$2`, `$3` usw. referenziert. Für die Ablauf-Kontrolle von `bash`-Prozeduren sind folgende Steuerstrukturen vorgesehen.

### 6.7.1 Die `if` - Anweisung

```

if [ expression ]
then
    command(s)
elif [ expression ]
then
    command(s)
else
    command(s)
fi

```

Die Schlüsselworte `then`, `elif`, `else` und `fi` sollten in separaten Zeilen stehen.

```

#!/bin/bash
#
# Usage: ./if.bash argument
#
    if [ $1 == "rudi" ]
    then
        echo "$1 muss heute Zeitungen austragen"
    elif [ $1 == "otto" ]
    then
        echo "$1 muss nach -----> Unterdinxbichl fahren"
    else
        echo "$1 hat heute frei"
    fi

```

### 6.7.2 Die `for` - Schleife

```

for laufvariable in wort1 wort2 ...
do
    command(s)
done

```

Die `for`-Schleife wird für jedes ihrer Argumente `wort1`, `wort2` usw. ausgeführt. Die *laufvariable* erhält dann jeweils das aktuelle Argument.

```

#!/bin/bash
#
# Datei: for.bash
#
    for VAR in `ls`
    do
        echo "Lese Datei $VAR"
    done

```

Die Argumentenliste in obiger for-Schleife wird durch ein ls-Kommando erzeugt (Ausführung durch umgekehrte Hochkommas).

### 6.7.3 Die while - Schleife

```
while [ expression ]
do
    command(s)
done
```

Die while-Schleife wird solange ausgeführt, wie *expression* true ist.

```
#!/bin/bash
#
# Datei: while.bash
#
    A=0
    while [ $A -le 10 ]
    do
        SUM=$((SUM += $A))
        echo "Wert von A    = $A"
        echo "Wert von SUM = $SUM"
        A=`expr $A + 1`
    done
    let Z=$SUM*2
    echo $Z
```

Die *expression* im while-Statement wurde kompatibel zur Bourne-shell-Syntax gewählt ( `-le` für `<=` ). In bash-Syntax kann für den Ausdruck auch ( `( A <= 10 )` ) geschrieben werden.

### 6.7.4 Die until - Schleife

```
until [ expression ]
do
    command(s)
done
```

Die until-Schleife wird solange ausgeführt, wie *expression* false ist.

```
#!/bin/bash
#
# Datei: until.bash
# Usage: ./until.bash arg1 arg2 arg3 ...
#
    i=1
    until [ $# = 0 ]
    do
        echo "Der $i-te Parameter: $1"
        shift
        i=`expr $i + 1`
    done
```

Die until-Schleife gibt alle beim Skriptaufruf eingegebenen Parameter zeilenweise auf dem Bildschirm aus. Dabei wird die besondere Variable \$# benutzt (enthält die Anzahl der eingegebenen Parameter). Mit `shift` wird dann bei jedem Schleifendurchlauf die Parameterliste um ein Element verkürzt, bis schließlich \$# = 0 ist. Der Ausdruck kann auch in der Form ( ( \$# == 0 ) ) geschrieben werden.

### 6.7.5 Die select - Schleife

Die select-Schleife bzw. das select-Kommando ist eine Konstruktion, die von der Korn-shell übernommen wurde. Sie wurde zur Ausgabe und Bearbeitung von Menüs eingeführt.

```
select variable in wort1 wort2 ...
do
    command(s)
done
```

**select** gibt zunächst ein durch die Wortliste gebildetes Menü mit vorangestellter Ziffer aus. Falls die Wortliste fehlt, werden die eingegebenen Parameter verwendet. Die Auswahl aus dem Menü erfolgt dann durch Eingabe der entsprechenden Ziffer. Das ausgewählte Wort wird schließlich in die Variable geschrieben. Die select-Schleife wird solange wiederholt, bis ein `break`-, `return`- oder `exit`-Kommando erreicht wird. Auch `<Strg>d` bricht die Schleife ab.

```
#!/bin/bash
#
# Datei: select.bash
# Usage: ./select.bash
#
PS3="Wohin willst du fahren? "
select wahl in Argentinien Belgien England Frankreich BEENDEN
do
    if [ "$wahl" = "BEENDEN" ]
    then
        break
    else
        echo "Du hast -- $wahl ($REPLY) -- gewaehlt"
    fi
done
```

Nach dem Aufruf bleibt das Skript mit dem Prompt PS3 stehen und wartet auf Eingabe.

```
1) Argentinien
2) Belgien
3) England
4) Frankreich
5) BEENDEN
Wohin willst du fahren?
```

Nach Eingabe einer Ziffer von 1 bis 4 erscheint immer wieder die entsprechende Ausgabe auf dem Bildschirm. Nur die Eingabe von 5 bewirkt den Abbruch der Prozedur.

### 6.7.6 Fallunterscheidung mit case

```

case wort in
    muster1) command(s) ;;
    muster2) command(s) ;;
esac

```

Bei Übereinstimmung von *wort* mit einem *muster* wird die dahinter stehende Kommandosequenz ausgeführt. Die Verwendung von Metazeichen ist zulässig.

```

#!/bin/bash
#
# Datei: case.bash
#
    for i in `ls`
    do
        case $i in
            [a-g]*) echo "Datei faengt mit a-g an $i" ;;
            [h-p]*) echo "Datei faengt mit h-p an $i" ;;
            [q-z]*) echo "Datei faengt mit q-z an $i"
                    zeilenzahl=`wc -l $i | awk '{print $1}'`
                    echo " Sie besitzt $zeilenzahl Zeilen" ;;
        esac
    done

```

### 6.7.7 shift-Statement

Im Gegensatz zu den C-Shell-Varianten wirkt ein `shift` nur auf die Liste der Positionsparameter (siehe until-Schleife).

### 6.7.8 Signalhandling mit trap

Die Reaktion der `bash` auf eintreffende Signale kann mit `trap` festgelegt werden. Die Definition der Signale kann mittels Nummern oder symbolischer Namen erfolgen. Eine Übersicht über die Bedeutung dieser Signale kann mit dem man-Kommando `man -s 7 signal` abgerufen werden.

```

trap 'command(s)' signumber
trap 'command(s)' sigsymbol

```

Folgendes Beispiel soll die Verwendung von `trap` verdeutlichen.

```

#!/bin/bash
#
# Datei: trap.bash
#
    trap 'echo hallo; echo hallo' 2
    trap 'ls' 15
    trap 'finger' HUP
#
    echo -n "Eingabe: "; read input
    echo $input

```

Je nachdem, welches Signal eintrifft, wird die davor definierte Kommando-Liste ausgeführt. Das `read`-Statement dient in diesem Fall nur dazu, das Skript an dieser Stelle anzuhalten.

### 6.7.9 Eingabe von Tastatur

Mit dem Kommando **read** kann eine Eingabe vom Standardeingabegerät einer oder mehreren Variablen zugewiesen werden.

```
read variable1 variable2 variable3 ...
```

Die Eingabe erfolgt in der Regel von der Tastatur. Mit dem Operator für die Eingabeumlenkung `<` kann aber auch eine Datei als Standardeingabe verwendet werden. Sind beim Einlesen mehr Worte als Variablen vorhanden, so werden alle überzähligen Worte der letzten Variablen im `read`-Statement zugewiesen.

```
#!/bin/bash
#
# Datei: read.bash
#
#           line=1
#
#           while [ ! "$line" = "" ]
#           do
#               read line
#               echo "Ausgabe $line"
#           done
```

Obiges Skript liest eine Zeile von der Standardeingabe in die Variable `line` ein und gibt den Inhalt der Variablen anschließend aus.

### 6.7.10 break und continue

Die beiden Kommandos **break** und **continue** können für das Abbrechen von **for**-, **while**-, **until**- und **select**-Schleifen verwendet werden (wie bei C-Shell).

### 6.7.11 Funktionen

Bei häufig benötigten Operationen ist es oft sinnvoll, diese Operationen innerhalb eines Shell-Skripts in Form einer Funktion zu definieren.

```
function name
{
    command(s)
}
```

Der Name *name* der Funktion ist beliebig wählbar. Bei den Funktionsaufrufen können Argumente übergeben werden, die innerhalb der Funktion als Positionsparameter erscheinen. Variable, die nur für die Funktion gelten sollen, werden mit dem Kommando **local** definiert (funktionslokale Variablen). Mit dem Kommando **return** wird eine Funktion verlassen.

```
#!/bin/bash
#
# Datei: function.bash
#
function number
{
    declare eingabe="$1" rueckgabe=1
    while true
    do
        case $eingabe in
            [0-9]*) eingabe=${eingabe#?}
                    rueckgabe=0;;
            "") return $rueckgabe;;
            *) return 1;;
        esac
    done
}
#
if number $1
then
    echo "$1 ist eine Zahl"
else
    echo "$1 ist keine Zahl"
fi
```

Obiges Shell-Skript prüft, ob das eingegebene Argument eine rein numerische Zahl ist. Für die Prüfung wird die Funktion `number` verwendet. Die Funktion liefert den Wert 0, wenn eine Zahl vorliegt, andererseits den Wert 1, falls keine reine Zahl eingegeben wurde.

### 6.7.12 Sonstiges

Wie auch bei der C-Shell kann bei der `bash` das Zeitverhalten eines Shell-Skripts mit dem vorangestellten `time`-Kommando ermittelt werden. Auch das Kommando `times` ermittelt diese Werte.

Die Fehlersuche in Shell-Skripten erfolgt durch Aufruf der `bash` mit den Optionen `-vx`.

```
bash -vx skript arg1 arg2
```

Bei der Abarbeitung wird jede Zeile vor und nach der Variablen-, File- oder Kommando-Substitution aufgelistet. Fehler können damit schnell lokalisiert und bereinigt werden.

# 7 Textmusterverarbeitung mit awk

**awk** ist eine Programmiersprache, die für das Durchsuchen und die Manipulation von Textdaten entworfen wurde. Der Name **awk** leitet sich von den drei Autoren **A**ho, **W**einberger und **K**ernighan ab.

Die Basisfunktion der Sprache besteht im *zeilenweisen* Durchsuchen der Eingabe nach vom Benutzer spezifizierten Auswahlbedingungen, denen entsprechende Aktionen zugeordnet werden können. Die Anwendungsbereiche von **awk** liegen in der Datenauswertung und -aufbereitung, der Datenvalidation und der Datentransformation.

Für einige der nachfolgenden Beispiele wird eine Datei namens `daten` mit folgendem Inhalt verwendet:

```
23      otto      huber
34      hugo      habicht
43      OTTO     Albers  Hamburg
12078677345444
11      ina      deter
0       rudi      ratlos
26      peter   lustig
44      elly     heuss
88      mani     held
14      egon     erpel
76      bodo     ballermann
90      andrea   doria
```

## 7.1 Aufruf und Programmstruktur

Es gibt die Möglichkeit, den **awk**-Programmtext über eine Programmdatei oder direkt einzugeben. Der **awk**-Aufruf kann auch innerhalb eines shell-scripts erfolgen.

```
awk 'programmtext' [inputfile ... ]
```

```
awk -f programmdatei [inputfile ... ]
```

Nach dem Aufruf wird der Programmtext auf syntaktische Korrektheit geprüft. Fehlermeldungen kennzeichnen den ungefähren Ort des Fehlers (`awk: syntax error near line 10`). Laufzeitfehler dagegen werden genau lokalisiert (`awk: division by zero, record number 10`).

Der übergeordnete Kontrollfluss im **awk**-Programm besteht aus einer Hauptschleife, die nacheinander alle Eingabezeilen liest. Jede Eingabezeile wird sequentiell mit allen aufgeführten Bedingungen verglichen.

wenn Bedingung erfüllt -> dann Aktion ausführen

Nicht erfüllte Bedingungen verursachen einen Sprung zur nächsten Bedingung. Aktionen stehen immer in geschweiften Klammern. Eine Aktion muss immer auf derselben Programmzeile beginnen wie die zugehörige Bedingung. Ansonsten können **awk**-Programme formatfrei geschrieben werden. Aus Übersichtlichkeit ist aber eine tabellarische Struktur ratsam.

Beispiele zu den Anwendungsbereichen von **awk**:

## Datenauswertung und -aufbereitung

```
host> awk '{sum+=$1; print $1, sum}' daten
```

Summiert jeweils das erste Feld jeder Zeile in der Variablen sum auf und gibt den Feldwert und die Summe aus

## Datenvalidation

```
host> awk '/^[0-9]+$/' {print "Zeile ist numerisch:" $0}' daten
```

Alle Zeilen, die rein numerisch sind, werden ausgegeben

## Datentransformation

```
host> awk '$2=="OTTO" {$2="HANS"; print}' daten
```

Untersucht jede Zeile, ob das zweite Feld den Inhalt OTTO besitzt und ersetzt OTTO durch HANS

Bedingung oder Aktion können auch weggelassen werden. Wenn nur der Aktionsteil vorhanden ist, wird dieser für jede Eingabezeile ausgeführt.

```
{sum+=$1; print $1, sum}
```

Wenn nur die Bedingung gegeben ist, wird die Eingabezeile vollständig in die Standardausgabe kopiert.

```
$2=="OTTO" entspricht $2=="OTTO"{print}
```

Ein- und Ausgabedatei werden als Folge von Zeilen gesehen, die durch ein Zeilentrennzeichen voneinander getrennt werden. Jede Zeile besteht für awk aus einer Folge von Feldern, die durch Feldtrennzeichen getrennt sind. Die einzelnen Felder einer Zeile werden mit \$1, \$2, ... angesprochen. Die gesamte Eingabezeile wird mit \$0 referenziert. Achtung: Die awk-Variablen \$1, \$2 usw. haben nichts mit den gleichnamigen Variablen in shell-Skripten zu tun!

host> awk '{print \$3, \$1}' daten	Ausgabe des dritten und ersten Feldes jeder Zeile
host> awk '{print \$0}' daten	Ausgabe jeder Zeile
host> awk '{print \$2, "lacht"}' daten	Ausgabe jedes zweiten Feldes mit zusätzlichem Text
host> awk '{print \$2, \$2, \$2}' daten	Feld 2 wird dreimal hintereinander ausgegeben

## 7.2 Die Sprachelemente von awk

Die Syntax von awk stimmt in vielen Punkten mit der C-Syntax überein.

### Datentypen und Variable

Es existieren numerische Daten und Zeichenkettendaten. Zeichenketten werden in Anführungsstriche gesetzt. Variable brauchen nicht deklariert werden, sie sind bei ihrer ersten Erwähnung im Programm implizit deklariert. Variablennamen beginnen grundsätzlich mit einem Buchstaben ( `abc`, `e534`, `a[10]` ).

Neben den frei definierbaren Variablen gibt es einige eingebaute Variable.

Statische eingebaute Variable:

RS	Trennzeichen für Eingabezeilen (LF)
ORS	Trennzeichen für Ausgabezeilen (LF)
FS	Feldtrennzeichen für die Eingabe (SPACE, TAB)
OFS	Feldtrennzeichen für die Ausgabe (SPACE, TAB)
OFMT	Steuerstring für die Ausgabe numerischer Werte

Dynamische eingebaute Variable:

NR	Number of Record, aktuelle Zeilennummer
NF	Number of Fields, aktuelle Feldanzahl der Eingabezeile
FILENAME	Name der aktuellen Eingabedatei

### Operatoren und Ausdrücke

Es gibt numerische und nichtnumerische Operationen. Das Ergebnis einer Operation ist abhängig vom Operator und den Werten der verwendeten Operanden. Der **Typ der Operation** dient als primäres Entscheidungskriterium für die Wertermittlung.

Als **numerische** Operatoren sind die Zeichen `+` `-` `*` `/` `%` `++` `--` zugelassen. Klammerausdrücke sind erlaubt. Als **nicht-numerische** Operation ist die Verkettung definiert.

host> <code>awk '{print 3 + 4}'</code>	ergibt 7, da numerische Operation
host> <code>awk '{print "3" + "4"}'</code>	ergibt ebenfalls 7, da die Operation numerisch ist
host> <code>awk '{print 3 + 4"5"}'</code>	Mischung von numerischer und nicht-numerischer Operation; die Verkettung wird zum Schluss ausgeführt; Ergebnis ist 75
host> <code>awk '{print 3 + (4"5")}'</code>	Ergebnis ist 48, da die Klammern die Auswertungsreihenfolge beeinflussen

Als **Vergleichsoperatoren** sind dieselben Operatoren wie in der Programmiersprache C gebräuchlich: `==` `!=` `<` `>` `<=` `>=` . Wenn die Vergleichsoperanden unterschiedlichen Typs sind, wird der numerische Operand als Zeichenkettentyp interpretiert. Vorsicht bei Zeichenkettenvergleichen: Die Werte werden linksbündig verglichen (d.h. `"12" < "2"` ist true). Verknüpfungen von Vergleichsoperationen erfolgen mit den Operatoren `!` (NOT), `&&` (AND) und `||` (OR).

```
host> awk '$2 >= "e" && $2 < "f" && $2 != "egon"' daten
Die Verknüpfung ist für alle Namen zutreffend, die mit einem e beginnen, aber
nicht egon lauten
```

## Textfunktionen

**substr**( *s*, *p* [, *l* ] )

**substr** schneidet einen Teilstring aus String *s*, beginnend bei Position *p* mit der Länge *l*.

```
host> awk '{print substr($3, 2, 2)}' daten
schneidet aus Feld 3 einen 2-stelligen String
```

```
host> awk '{print substr($3, 2)}' daten
extrahiert aus Feld 3 alles ab der Zeichenpostion 2
```

**sprintf**( *a1*, ..., *an* )

**sprintf** formatiert den Wert der Ausdrücke *a1* ... *an* gemäß dem in *f* angegebenen Format (siehe `printf`).

## Numerische Funktionen

**length**( *s* )

**length** gibt die aktuelle Länge der Zeichenkette *s* zurück; wenn Argument *s* fehlt, wird die gesamte input-Zeile genommen.

```
host> awk '{print length($3)}' daten
gibt die Länge des dritten Feldes jeder Zeile aus
```

**index**( *s1*, *s2* )

**index** gibt die Anfangsposition der Zeichenkette *s2* innerhalb der Zeichenkette *s1* an; falls *s2* nicht gefunden wird, ergibt dies 0

```
host> awk '{print index($3,"mann")}' daten
sucht in der dritten Spalte nach der Zeichenkette mann
```

**split**( *s*, *a* [, *t* ] )

**split** spaltet die Zeichenkette *s* in Teilketten auf, die durch das Trennzeichen *t* bestimmt sind; die Teilketten werden den Elementen des Feldes *a* zugewiesen; Ergebnis der Funktion ist die Zahl der gefundenen Teilketten.

```
host> awk '{print split($0,feld), feld[1], feld[2],
feld[3]}' daten
hier wird als Feldtrennzeichen <space> bzw. <tab> benutzt
```

```
host> awk '{print split($0,feld,"e"), feld[1], feld[2],
feld[3]}' daten
hier werden die Teilketten durch den Buchstaben e getrennt
```

Als weitere numerische Funktionen sind die mathematischen Funktionen **sqrt**, **log**, **exp** und **int** definiert (Beispiel: `awk '{print sqrt(split($0,feld))}' daten`).

## 7.3 Bedingungen

Bedingungen dienen in erster Linie zur Selektion von Zeilen des Eingabetextes. Als Bedingungen sind reguläre Ausdrücke und Vergleichsausdrücke zugelassen. Reguläre Ausdrücke müssen bei `awk` in Schrägstriche / eingeschlossen werden.

```
host> awk '/OTTO|egon/ {print NR, $0}' daten
      gibt die Zeilen, in denen OTTO oder egon vorkommen, mit Zeilennummer
      aus
host> awk '/[oO][tT][tT][oO]/ {print NR, $0}' daten
      gibt alle Zeilen aus, in denen Otto vorkommt
host> awk '$2 ~/i|p|m/ {print}' daten
      gibt alle Zeilen aus, in deren zweites Feld i, p oder m vorkommt
host> awk 'length > 25' daten
      gibt alle Zeilen aus, deren Länge 25 Zeichen überschreitet
host> awk 'NR >= 2 && NR <= 7' daten
      es werden nur die Zeilen 2 bis 7 ausgegeben
```

Es ist auch möglich, Bedingungen zu spezifizieren, die einer Start- und einer Ende-Situation entsprechen. Es werden dann alle Eingabezeilen selektiert, die zwischen der Start- und der Ende-Bedingung liegen.

```
host> awk '/elly/,/bodo/ {print}' daten
      selektiert alle Zeilen von elly bis bodo
host> awk 'NR == 2,NR == 7 {print}' daten
      startet Ausgabe bei Zeile 2 und endet bei Zeile 7
```

### Spezielle Bedingungen

Spezielle Bedingungen übernehmen die Programmsteuerung vor bzw. nach der Hauptschleife.

**BEGIN** { *Aktion* } wird vor dem Lesen der ersten Eingabezeile ausgeführt; kann benutzt werden, um Variablen vorzubersetzen, Überschriften auszugeben usw.

**END** { *Aktion* } wird nach dem Lesen der letzten Zeile ausgeführt, für die Ausgabe von Statistiken, Meldungen, usw.

```
host> awk 'BEGIN {print "Hier fang ich an"} \
NR==2,NR==7 {print NR, $0} \
END {print "Hier ist das Ende"}' daten
```

Obiges Beispiel zeigt auch die Verwendung von `awk`-Statements über mehrere Zeilen. Die Fortsetzung des Programmtextes wird am Zeilenende mit Backslash markiert. Ist der Programmtext dagegen in einer eigenen `awk`-Programmdatei abgespeichert, so werden sowohl die den Programmtext kennzeichnenden einfachen Hochkommas als auch die Backslashes weggelassen.

## 7.4 Aktionen

Aktionen stehen immer in geschweiften Klammern. Innerhalb von Aktionen sind folgende Zuweisungen gültig: = += -= \*= /= %=

### 7.4.1 Ausgabeanweisungen

In awk existieren die beiden Ausgabeanweisungen **print** und **printf**. **print** benutzt die Einstellungen der eingebauten Variablen `OFS`, `ORS` und `OFMT`; numerische Werte mit Kommastellen werden in einem Standardformat ausgegeben, das durch `OFMT` bestimmt ist (`%.6f` bedeutet 6 Nachkommastellen). Ganze Zahlen und Zeichenketten werden in ihrer aktuellen Länge ausgegeben. Die Ausgabe kann in eine Datei umgelenkt oder einem Kommando weitergegeben werden.

```
awk '{print $1*1.14167}' daten
```

alle Werte der ersten Spalte werden mit 1.14167 multipliziert

```
awk '{print "Ergebnis: " $1*1.14167 > "/tmp/erg"}' daten
```

Ausgabe wird in die Datei `/tmp/erg` umgelenkt

Die Ausgabeanweisung **printf** entspricht der gleichnamigen C-Anweisung. Im `printf`-Kommando beschreibt ein Steuerstring das Format der Ausgabe. Die Formatbeschreibung für ein Argument wird mit einem Prozentzeichen `%` eingeleitet, danach folgt ein Kennbuchstabe, der den Argumententyp beschreibt.

Kennbuchstaben:	d	Ganzzahl
	ld	Lange Ganzzahl
	f	Kommazahl ohne Exponent
	e	Kommazahl in Exponentenschreibweise
	g	e oder f
	s	Zeichenkette

awk versucht eine automatische Typanpassung der Argumente an das spezifizierte Format. Als Steuerzeichen für die Ausgabe sind die Zeichen `\n` für Zeilenvorschub und `\t` für Tabulator zugelassen.

```
awk '{printf "%f", $1}' daten
```

Ausgabe aller Werte als Kommazahlen mit 6 Nachkommastellen

```
awk '{printf "%6.4f\n", $1}' daten
```

Ausgabe mit 4 Nachkommastellen und Zeilenvorschub

```
awk '{printf "%6.4e\t%s\t%-10s\n", $1, $2, $3}' daten
```

Ausgabe einer numerischen und zweier alphanumerischer Spalten, jeweils linksbündig und durch Tabulatoren getrennt

```
awk '{printf "%6.4e\t%s\t%+10s\n", $1, $2, $3}' daten
```

Ausgabe wie oben, nur dritte Spalte wird rechtsbündig ausgerichtet

Wenn `printf` ohne Argumente verwendet wird, entspricht die Ausgabe der von `print` (ganze Zeile wird ohne Zeilenvorschub ausgegeben).

## 7.4.2 Kontrollanweisungen

Wie in shell-Prozeduren können auch in awk-Programmen Kontrollstrukturen für die Ablaufsteuerung verwendet werden.

### Die Verbundanweisung

Mehrere Anweisungen können mit geschweiften Klammern zu einer Gruppe von Anweisungen zusammengefasst werden. Dies nennt man Verbundanweisung.

```
awk '{ {wert=$1*3.4} {print $2,$3, "Ergebnis:" wert} }' daten
```

### Die if-Verzweigung

```
if ( expression ) Anweisung [ ; else Anweisung ]
```

```
host> awk '{if($1 > 100) \
    $1="****" \
    else \
    sum+=$1 \
    print $1} \
    END {print "Summe:" sum}' daten
```

Obige Verzweigung kann auch in einer Zeile geschrieben werden:

```
awk '{if($1 > 100) $1="****"; else sum+=$1;print $1} \
    END {print "Summe:" sum}' daten
```

Wichtig: Bei der Verzweigung unter awk ist zu beachten, dass im if- bzw. else-Zweig immer nur genau **eine** Anweisung stehen darf. Hierbei kann es sich auch um eine Verbundanweisung handeln.

### Die while-Schleife

```
while ( expression ) Anweisung
awk '{i=1; while(i<=NF) {print $i; i++} }' daten
```

### Die for-Schleife

```
for ( expression1; compare_expression; expression2 ) Anweisung
awk '{for (i=1; i<=NF; i++) print $i}' daten
```

In beiden obigen Beispielen werden alle Felder jeder Zeile in einer eigenen Zeile ausgegeben.

### Schleifensteuerung

Mit den beiden Anweisungen **break** und **continue** kann der Ablauf von while- und for-Schleifen beeinflusst werden. Durch ein **break** wird die Schleife verlassen und mit der nächsten Anweisung hinter der Schleife fortgefahren. Bei einem **continue** wird sofort mit dem nächsten Schleifen-Durchlauf begonnen, ohne die restlichen Anweisungen in der Schleife auszuführen.

```
awk '{for (i=1; i<=NF; i++) {print $i; if($i=="bodo") break} }' daten
```

Im Beispiel werden alle Felder jeder Zeile in einer eigenen Zeile ausgegeben. Wird das Feld namens bodo erreicht, werden die restlichen Felder dieser Zeile nicht mehr angezeigt.

## 7.5 Sonstiges

Kommentare werden im Aktionsteil von awk mit dem Zeichen # eingeleitet.

Bei Verwendung von awk-Aufrufen in shell-Skripten muss der Unterschied zwischen den shell-Variablen und den Feldvariablen von awk beachtet werden. Die Positionsparameter \$1, \$2 usw., die mit dem Prozeduraufruf in das shell-Skript übernommen werden, haben mit den Feldvariablen \$1, \$2 ... in awk nichts gemein. Es gibt drei Möglichkeiten, shell-Variable in den awk-Aufruf zu übernehmen.

```
echo $shell_var | awk '$1 ...'
echo $1 | awk '{print $1}'
```

In diesem Fall wird der Inhalt der shell-Variablen \$1 mit dem echo-Befehl an den awk-Aufruf weitergeleitet. awk interpretiert den über die Pipe ankommenden Input als Feld \$1, \$2 usw..

```
awk ...'{ ... awk_var ...}' awk_var=$shell_var filename
awk '{print user}' user=$USER daten
```

Bei dieser Variante wird die Variablenübergabe durch die Definition von awk-Variablen mit gleichzeitiger Zuweisung des Inhalts der shell-Variablen beim awk-Aufruf realisiert. Dabei können auch mehrere awk-Variable definiert werden.

```
awk ...'{ ... "$shell_var" ...}' filename
awk '{var="$USER"; print var}' daten
```

Die dritte Alternative besteht darin, die shell-Variable direkt im awk-Aufruf zu referenzieren. Dies erfolgt dadurch, dass die shell-Variable erst in einfache und dann nochmals in doppelte Hochkommas (Anführungszeichen) eingeschlossen wird.

Sollen dagegen awk-Variable an shell-Variable übergeben werden, so erfolgt dies durch die bereits bekannten Zuweisungsmechanismen.

```
set shell_var = `awk ...'{print awk_var ...}' filename`
@ shell_var = `awk ...'{print awk_var ...}' filename`
shell_var=`awk ...'{print awk_var ...}' filename`
```

Bei der Zuweisung auf eine Bourne-Shell-Variable darf zwischen Variable, Gleichheitszeichen und awk-Statement kein Leerzeichen stehen.

```
set vname = `awk '{print $2}' daten`
```

Die Zuweisung kommt durch das sofortige Ausführen des awk-Kommandos mittels den umgekehrten einfachen Hochkommas zustande.

# A Ausgewählte Web-Links

<http://www.comptechdoc.org/os/linux/usersguide/>

<http://docs.sun.com>

<http://www.tldp.org>

<http://www.linuxfibel.de>