Einführung in LINUX

Shells

erstellt durch:

Name:	Karl Wohlrab
Telefon:	09281 / 409-279
Fax:	09281 / 409-55279
EMail:	mailto: Karl.Wohlrab@fhvr-aiv.de

Anmerkungen

Bei dem vorliegenden Scriptum handelt es sich um ein Rohmanuskript im Entwurfsstadium. Das Script wird begleitend zur Lehrveranstaltung fortgeschrieben und überarbeitet.

Es erhebt keinen Anspruch auf Vollständigkeit und Korrektheit. Inhaltliche Fehler oder Ungenauigkeiten können ebenso wenig ausgeschlossen werden wie Rechtschreibfehler.

Inhaltsverzeichnis

1. Shells

Die Shell ist die Schnittstelle zwischen dem Anwender und dem Betriebssystem LINUX. Mit ihrer Hilfe werden Kommandos ausgeführt und Programme gestartet. Der Begriff Kommandointerpreter rührt daher, dass die Shell zunächst die Eingaben des Benutzers deutet und dann an die jeweilige Anwendung weitergibt. Außerdem stellt sie eine leistungsfähige Programmiersprache zur Verfügung, mit der Arbeitsabläufe automatisiert werden können. Dabei ermöglichen es einige besondere Shell-Kommandos innerhalb dieser Programme Variablen zu verwenden, Abfragen und Schleifen zu bilden usw. Letztendlich handelt es sich hierbei um einfache Textdateien, die aufgrund Ihrer besonderen Verwendung als Shell-Skripte bezeichnet werden.

Für jeden Anwender, der im System einen Benutzeraccount hat, ist eine default shell vorgesehen. Diese wird bei der Einrichtung des Benutzers in der Datei /etc/passwd hinterlegt. Mit dem Befehl chsh kann der Anwender diese Einstellung nachträglich anpassen.

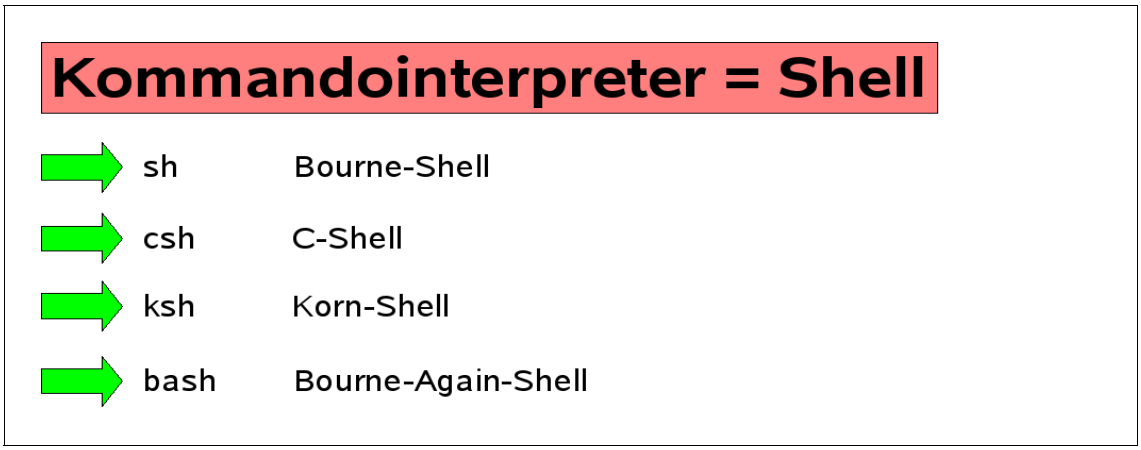

Abbildung 1: Wichtige Shells unter LINUX

Eine Liste der verwendbaren Shells findet sich in /etc/shells.

Beispiele:
Abfrage der installierten bash-Version:

```
echo $BASH_VERSION
```

Abfrage der gerade aktiven Shell:

```
echo $0
```

1.1 Überblick über wichtige LINUX-Shells

1.1.1 Die sh-Shell

Bourne Shell (die erste Shell, angelehnt am ALGOL68). Wird auf heutigen UNIX-Systemen insbesondere im Rahmen von Shell-Scripten eingesetzt. Im Dialogbetrieb kommt meist die Weiterentwicklung die **bourne again shell** (bash) zum Einsatz.

1.1.2 Die bash-Shell

Die **bourne again shell** (bash) ist der Standardkommandointerpreter unter LINUX. Sie ermöglicht die Eingabe und Ausführung von Kommandos. Zudem stellt Sie eine eigene Programmiersprache zur Verfügung, die zur Erstellung von ***Shell-Scripts*** verwendet werden kann. In den folgenden Abschnitten wird zunächst auf die Verwendung der bash eingegangen. Themen sind hier beispielsweise die Ein- und Ausgabeumleitung, die Kommunikation zwischen mehreren Prozessen und die Verwaltung von Shell-Variablen.

1.1.3 Die csh-Shell

C-Shell (angelehnt an C),

1.1.4 Die ksh-Shell

Korn Shell (das "Beste" aus Bourne- und C-Shell). Die ksh wird ebenfalls häufig im Dialogbetrieb eingesetzt.

1.2 Aufgaben der Shell

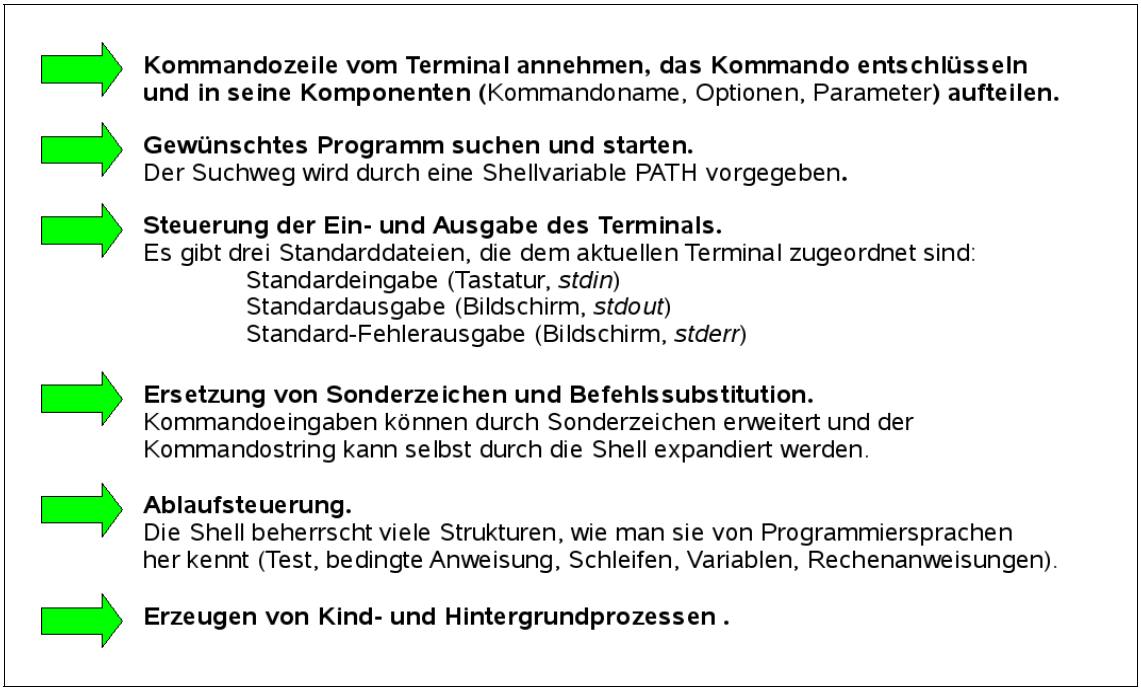

Abbildung 2: Aufgaben der Shell unter LINUX

Es gibt weiterhin die Möglichkeit, Programme "im Hintergrund" zu starten und am Bildschirm weiterzuarbeiten, während das Programm läuft. Dies wird durch ein & hinter dem Kommando realisiert.

Sobald die Shell bereit ist, Kommandoeingaben anzunehmen, meldet sie sich mit einem Bereitschaftszeichen (Prompt). Folgende Prompt-Zeichen sind üblich:

- $ als normales Prompt,
- # als Prompt für Superuser und
- > Prompt, wenn noch weitere Eingaben erwartet werden.

Wie vieles in der Shell, kann auch der/das Prompt beliebig modifiziert werden

1.3 Grundbegriffe der Shell-Programmierung

1.3.1 Kommandobearbeitung der Shell

Ein Kommando wird erst ausgeführt, wenn der Benutzer am Ende der Kommandozeile die Return-Taste drückt. Eine genauere Kenntnis des dann vonstatten gehenden Ablaufs erlaubt es, zu verstehen, warum etwas nicht so klappt, wie man es sich vorgestellt hat. Daher sollen diese Schritte hier kurz beschrieben werden. (Anmerkung: Einige der Kommandotrenner, `...`, Jokerzeichen und Variablen werden erst später behandelt.):

1. Die Shell liest bis zum ersten Kommandotrenner (& && || ; > <) und stellt fest, ob Variablenzuweisungen erfolgen sollen oder die Ein-Ausgabe umgelenkt werden muss.
2. Die Shell zerlegt die Kommandozeile in einzelne Argumente. Sie trennt die einzelnen Argumente durch eines der Zeichen, die in der Shell-Variablen IFS (Internal Field Separator) stehen, normalerweise Leerzeichen, Tabs und Newline-Zeichen.
3. Variablenreferenzen, die in der Kommandozeile stehen, werden durch ihre Werte ersetzt.
4. Kommandos, die in `...` oder bei der bash in $(...) stehen, werden ausgeführt und durch ihre Ausgabe ersetzt.
5. *stdin*, *stdout* und *stderr* werden auf ihre "Zieldateien" umgelenkt.
6. Falls in der Kommandozeile noch Zuweisungen an Variablen stehen, werden diese ausgeführt.
7. Die Shell sucht nach Jokerzeichen und ersetzt diese durch passende Dateinamen.
8. Die Shell führt das Kommando aus.

1.3.2 Shellvariablen

Variable sind frei wählbare Bezeichner (Namen), die beliebige Zeichenketten aufnehmen können.

- Bestehen die Zeichenketten nur aus Ziffern, werden sie von bestimmten Kommandos als Integer-Zahlen interpretiert (z. B. expr).
- Bei Variablen der Shell sind einige Besonderheiten gegenüber anderen Programmiersprachen zu beachten.

Im Umgang mit Variablen lassen sich grundlegend drei Formen unterscheiden:

- Variablendeklaration
- Wertzuweisung
- Wertreferenzierung

Beispiele zur Verdeutlichung des Sachverhaltes:

Deklaration:	**`USRBIN=`**
Wertzuweisung	**`USRBIN=/usr/local/bin`**
Wertreferenzierung	**`echo $USRBIN`**

Im Allgemeinen werden Variablen in der Shell nicht explizit deklariert. Vielmehr ist der Wertzuweisung die Variablendeklaration implizit enthalten. Wird eine Variable dennoch ohne Wertzuweisung deklariert, so wird bei der Wertreferenzierung ein leerer String ("") zurückgegeben.

1.3.2.1 Grundsätze zur Verwendung von Shell-Variablen

- Variable sind frei wählbare Bezeichner (Namen), die mit einem Buchstaben beginnen und bis zu 200 Zeichen lang sein dürfen. Leerzeichen innerhalb von Variablen sind (normalerweise) nicht erlaubt (und schlechter Stil).
- Ist der Wert, d. h. der Inhalt einer Variablen gemeint, wird ein $-Zeichen vor den Namen gestellt (siehe oben).
- Mittels spezieller Funktionen kann eine Variable auch numerisch oder logisch interpretiert werden.
- Shell-Skripten dürfen ihrerseits wieder Shell-Skripten oder Programme aufrufen. Dabei muss die Vererbung einer Variablen ausdrücklich festgelegt werden (--> Exportieren durch den export-Befehl), sonst kann eine andere Shell - oder ein beliebiges anderes Programm - nicht darauf zugreifen.
- Jede Subshell läuft in einer eigenen Umgebung, d. h. Variablendefinitionen (und die Wirkung verschiedener Kommandos) in der Subshell sind nach deren Beendigung wieder "vergessen". Es ist nicht möglich, den Wert einer Variablen aus einem Unterprogramm in die aufrufende Ebene zu übergeben.
- Die Zuweisung eines Wertes an die Variable erfolgt mittels des Gleichheitszeichens:

VAR=Wert	Wert ist ein String (ohne Leerzeichen und Sonderzeichen)
VAR="Wert"	Wert ist ein String (Ersetzung eingeschränkt, Leerzeichen und Sonderzeichen dürfen enthalten sein)
VAR=`kommando` VAR=$(kommando)	Wert der Variablen ist die Ausgabe des Kommandos (Newline --> Leerzeichen)

- Es hat sich die Konvention eingebürgert, Variablen zur Unterscheidung von Kommandos groß zu schreiben.
- Soll der Wert der Variablen mit einem String konkateniert werden, ist der Name in geschweifte Klammern einzuschließen - damit die Shell erkennen kann, wo der Variablenname endet. Bei der Zuweisung von Zahlen an Shell-Variable werden führende Leerzeichen und führende Nullen ignoriert.

1.3.2.2 Beispiele zur Verwendung von Shellvariablen:

Kommando-Eingaben beginnen mit "$ ".

```
$ VAR="Hello World"
$ echo $VAR
Hello World!
$ echo '$VAR'
$VAR
```

```
$ VAR=`pwd`
$ echo "Aktuelles Verzeichnis: $VAR"
Aktuelles Verzeichnis: /home/wohlrab
```

```
$ VAR=`pwd`
$ echo ${VAR}/bin
/home/wohlrab/bin
```

```
$ VAR=/usr/tmp/mytmp
$ ls > $VAR
```

Das letzte Beispiel schreibt die Ausgabe von **`ls`** in die Datei **`/usr/tmp/mytmp`**

Enthält eine Variable ein Kommando, so kann dies Kommando durch Angabe der Variablen ausgeführt werden, z. B.:

```
$ VAR="ls -la"
$ $VAR
```

1.3.3 Vordefinierte Variablen

Beim Systemstart und beim Aufruf der Dateien /etc/profile (System-Voreinstellungen) und .profile (benutzereigene Voreinstellungen), die ja auch Shellskripts sind, werden bereits einige Variablen definiert. Alle aktuell definierten Variablen können durch das Kommando `set` aufgelistet werden.

Einige vordefinierten Variablen sind neben anderen:

Variable	Bedeutung
HOME	Home-Directory (absoluter Pfad)
PATH	Suchpfad für Kommandos und Skripts
MANPATH	Suchpfad für die Manual-Seiten
MAIL	Mail-Verzeichnis
SHELL	Name der Shell
LOGNAME USER	Login-Name des Benutzers
PS1	System-Prompt ($ oder #)
PS2	Prompt für Anforderung weiterer Eingaben (>)
IFS	(internal field separator) Trennzeichen, meist CR, Leerzeichen und Tab)
TZ	Zeitzone (z. B. MEZ)

Die komplette Liste der Variablen (sowohl vordefinierte Variablen als auch Benutzerdefinierte Variablen kann mit dem Kommando

```
set
```

angezeigt werden.

1.3.4 Spezielle Shell-Variablen

Variable	Bedeutung	Kommandobeispiel
$-	gesetzte Shell-Optionen	set -xv
$$	PID (Prozessnummer) der Shell	kill -9 $$ (Selbstmord)
$!	PID des letzten Hintergrundprozesses	kill -9 $! (Kindermord)
$?	Exit-Status des letzten Kommandos	cat /etc/passwd ; echo $?

1.3.5 Umgebungsvariablen

Die bash verwendet verschiedene Dateien zur Initialisierung der Umgebung. Wenn sie als loginshell aufgerufen wird, werden diese Dateien in der folgenden Reihenfolge ausgewertet.

1. `/etc/profile`
2. /etc/profile.local (sofern vorhanden)
3. `./.profile`
4 `./etc/bash.bashrc`
5. `./.bashrc`

Individuelle Änderungen an ihrer Umgebung können die Benutzer in ~./.profile bzw. ~./.bashrc vornehmen.

1.3.6 Basiskonfiguration - Tastenbelegung und Prompt

Die Konfiguration der Tastatur kann global in der Datei `/etc/inputrc` oder benutzerspezifisch in `~/.inputrc` eingestellt werden.

Dies ist unter Umständen notwendig, da deutsche Sonderzeichen und die Funktionstasten Pos1, Ende und Entf nicht immer wie gewohnt sind arbeiten.

Die erwähnten Dateien steuern die Funktion readline, die bash-intern zur Verarbeitung von Tastatureingaben verwendet wird. In einem Konsolenfenster müssen deshalb aber noch lange nicht alle Funktionstasten richtig interpretiert werden. Der Prompt zeigt üblicherweise

den angemeldeten Benutzer, den Computernamen, das aktuelle Verzeichnis und mit welchen Rechten der eingeloggte Anwender ausgestattet ist.

Weitere Einstellmöglichkeiten bieten sich durch die Umgebungsvariable PS1.

1.3.7 Wichtige Tasten und Tastenkombinationen

Bash-Tastenkombination	Funktion
Pfeiltasten nach oben bzw. unten	Scrollt durch die zuletzt eingegebenen Kommandos
Pfeiltasten nach rechts bzw. links	Bewegt den Cursor vor bzw. zurück
Pos1, Ende	Bewegt den Cursor an den Beginn, an das Ende der Zeile
Strg + A, Strg + E	Bewegt den Cursor an den Beginn, an das Ende der Zeile (Wie Pos1 und Ende)
Alt + B, Alt + F	Bewegt den Cursor wortweise rückwärts, vorwärts
Backspace, Entf	Zeichen rückwärts, vorwärts löschen
Alt + D	Wort rechts vom Cursor löschen
Strg + K	Bis zum Ende der Zeile löschen
Alt + T	Die beiden vorangegangenen Wörter tauschen
Tab	Expansion des Kommando- oder Dateinamens
Strg + L	Löscht den Bildschirm
Strg + R	Sucht nach früher eingegebenen Kommandos, erneutes Betätigen geht zum nächsten Suchergebnis

1.3.8 Kommandoeingabe

Die bash unterstützt Sie bei der Kommandoeingabe mit vielen praktischen Tastenkürzeln und Sondertasten.

Insbesondere können Sie mit den Pfeiltasten die zuletzt eingegebenen Kommandos aufrufen, ausführen oder nachträglich bearbeiten. Diese Kommandos werden beim Ausloggen in einer Datei gespeichert und stehen somit nach dem nächsten Einloggen wieder zur Verfügung.

Außerdem können Sie, sollten die Ausgaben eines Befehls aufgrund ihres Umfangs nicht mehr vollständig sichtbar sein, mit shift + Bild auf bzw. shift + Bild ab die vorangegangenen Seiten betrachten. Dies ist solange möglich, bis Sie die Konsole wechseln. Auf diese Art und Weise können Sie beispielsweise nachvollziehen, was beim Systemstart abgelaufen ist.

Eine weitere nützliche Funktion ist die automatische Expansion von Kommando- und Dateinamen. Haben Sie die ersten Buchstaben beispielsweise eines Befehls eingegeben, wird Ihnen nach Betätigung von Tab der Befehlsname ergänzt, wenn dieser bereits eindeutig ist, andernfalls wird nach einem kurzen Signalton und einem nochmaligen Bestätigen mit Tab eine Liste mit allen zur Verfügung stehen Kommandos oder Dateien angezeigt. Diese Funktion verschleiert jedoch, wo sich ein Programm eigentlich befindet.

Abfrage des Verzeichnisses für ein gesuchtes Kommando:

```
whereis <kommandoname>
which <kommandoname>
```

Ähnliche Expansionsmechanismen treffen Sie auch bei Verzeichnis- und Pfadnamen an.

1.3.9 Alias-Abkürzungen

Mit dem Kommando alias werden Abkürzungen für Kommandos festgelegt. Bei der Verarbeitung der Kommandozeile wird zunächst geprüft, ob das erste Wort ein Alias ist und dann gegebenenfalls durch den vollständigen Text ersetzt. Die bash durchsucht darüber hinaus bei Pipes, Kommandosubstitutionen oder bei der sequentiellen Ausführung von Kommandos alle vorkommenden Kommandonamen auf Abkürzungen.

Beispiel: Festlegung eines Aliases:

```
alias more=less
```

Die Alias Abkürzungen werden vorrangig gegenüber gleichnamigen Kommandos behandelt. Das kann dazu genutzt werden, um den unerwünschten Aufruf eines Kommandos zu vermeiden.

Einmal festgelegte Abkürzungen sind bis zum Verlassen der shell gültig, können jedoch mit dem Befehl unalias wieder gelöscht werden. Alias Anweisungen, die in die Dateien .profile oder .bashrc aufgenommen sind, werden beim Einloggen geladen.

1.3.10 Ein- und Ausgabeumleitung

1.3.10.1 Dateideskriptoren

Bei der Ausführung von Kommandos in der bash stehen drei Dateideskriptoren zur Verfügung, die auf Betriebssystemebene wie Dateien behandelt werden.

0	**stdin**	**Standardeingabe** Gerade ausgeführte Programme lesen aus dieser die Eingaben des Anwenders. Normalerweise ist dies die Tastatur.
1	**stdout**	**Standardausgabe** An diese werden alle Ausgaben des Programms geschickt. Im Regelfall ist dies der Bildschirm bzw. das Terminal.
2	**stderr**	**Standardfehlerausgabe** Fehlermeldungen werden üblicherweise ebenfalls im aktuellen Terminal ausgegeben.

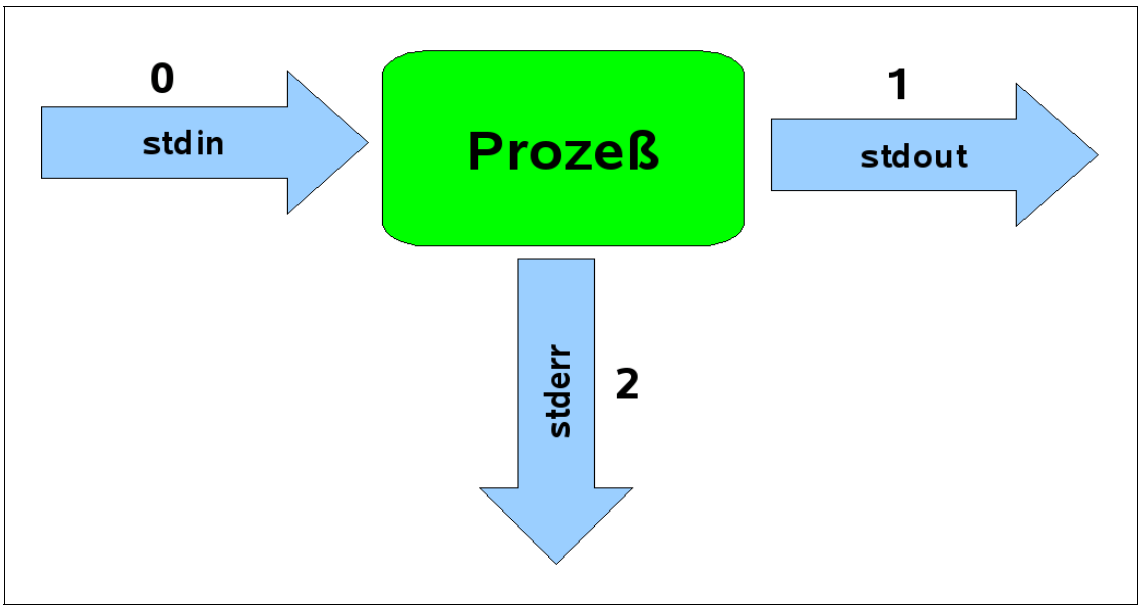

Abbildung 3: Standard-Datenkanäle unter LINUX

Diese drei Dateideskriptoren können jedoch angepasst werden. So ist es beispielsweise möglich, die Standardausgabe des einen Kommandos zur Standardeingabe eines anderen Kommandos zu machen.

Zudem kann man Ausgaben eines Programms in eine Datei umleiten.

Syntax	Funktion
`ls *.tex > inhalt`	Gibt das Ergebnis von ls in der Datei inhalt aus
`ls *.gz >> inhalt`	Hängt die Ausgabe von ls an die Datei inhalt an
`2> datei`	Leitet alle Fehlermeldungen in die angegebene Datei
`>& datei`	Leitet sowohl die Ausgabe als auch die Fehler in die Datei "datei"
`< datei`	Leitet die Eingaben an das Kommando weiter

Beispiele:

```
ls   l > inhalt
ls   l >> inhalt
make_prog < sourcefile > exefile 2> errorfile
```

Hinweise und Tipps:

Die Umleitung von *stdout* und *stderr* in dieselbe Datei würde prinzipiell eine zweimalige Angabe der Datei (eventuell mit einem langen Pfad) erfordern. Für die Standarddateien werden in solchen Fällen spezielle Platzhalter verwendet:

&0 Standardeingabe
&1 Standardausgabe
&2 Standard-Fehlerausgabe

Beispiele :

```
Kommando > ausgabe 2>&1
```

Das Kommando

```
cat <a <b >c >d
```

kopiert den Inhalt der Datei b in die Datei d. Als Seiteneffekt legt es eine leere Datei c an oder löscht den Inhalt von c, falls die Datei schon existierte.

Es ist somit auch möglich, denselben Kanal in einem Befehl mehrmals umzuleiten. Dabei gilt, dass die jeweils zuletzt ausgeführte Ein- oder Ausgabeumleitung Vorrang hat.

1.3.10.2 Pipes

Eine Pipe verbindet zwei Kommandos über einen temporären Puffer, d. h. die Ausgabe vom ersten Programm wird als Eingabe vom zweiten Programm verwendet. Alles, was das erste Programm in den Puffer schreibt, wird in der gleichen Reihenfolge vom zweiten Programm gelesen. Pufferung und Synchronisation werden vom Betriebssystem vorgenommen. Der Ablauf beider Prozesse kann verschränkt erfolgen. In einer Kommandofolge können mehrere Pipes vorkommen.

Die Bildung einer Pipe erfolgt durch das Zeichen |, das zwischen zwei aufeinander folgende Befehle eingefügt wird. Dadurch wird die Standardausgabe des einen zur Standardeingabe des anderen Kommandos.

```
Kommando 1 | Kommando 2
```

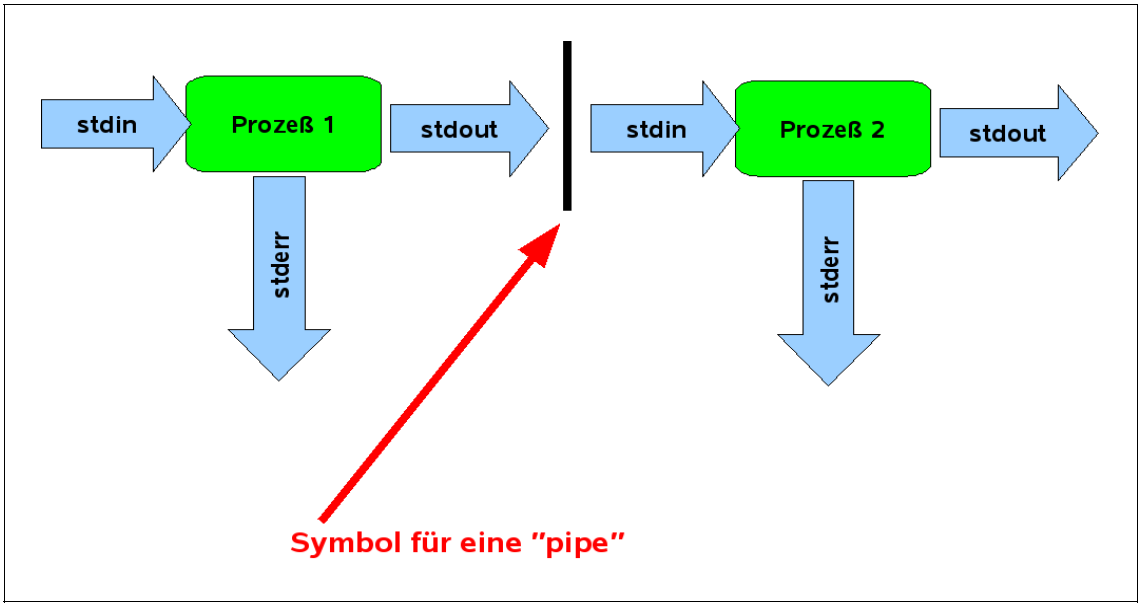

Abbildung 4: Pipes unter LINUX

Beispiele:

ls -al | less

```
ps -ax | grep wohlrab
```

1.3.10.3 FIFOs

Die Idee, die hinter **first in first out** steckt, ist im Grunde dieselbe wie bei einer Pipe. Sie wird verwendet, damit zwei voneinander unabhängige Programme miteinander kommunizieren können.

Beispiel:

```
mkfifo fifo
ls -l > fifo &
less < fifo
```

1.3.10.4 Ausgabevervielfältigung

Das Kommando tee bewirkt, dass zwei Kopien der Programmausgabe erzeugt werden. Auf diese Weise kann eine Kopie am Bildschirm angezeigt werden, während die zweite Kopie in eine Datei geschrieben wird.

Beispiele:

```
ls | tee inhalt
ls | tee inhalt1 inhalt2
ls -l | tee inhalt1 | sort +4 > inhalt2
```

1.3.11 Kommandoausführung

Bevor ein Kommando von der bash ausgeführt wird, interpretiert diese zunächst eventuell eingegebene Steuerzeichen. Auf diese Weise ist es möglich, Programme im Hintergrund zu starten, Jokerzeichen zu verwenden und Ergebnisse eines Kommandos in die Parameterliste eines anderen Kommandos zu übernehmen.

1.3.11.1 Hintergrundprozesse

Kommandos können im Hintergrund ausgeführt werden, wenn beim Aufruf ein & angehängt wird. Das unter Umständen zeitaufwendige Kommando wird dann von der bash verarbeitet, während im Vordergrund bereits die Eingabeaufforderung zur Verfügung steht.

Beispiel:

```
find / -name   *sh   > ergebnis &
```

Hinweis

Mit **Strg + Z** kann ein Programm zunächst unterbrochen und dann in den Hintergrund versetzt werden.

Mit **Strg + C** wird die Ausführung gänzlich abgebrochen.

1.3.11.2 Ausführung mehrerer Kommandos

Kommando	Funktion
Kommando1; kommando2	Führt die Kommandos nacheinander aus
Kommando1 && Kommando2	Führt kommando2 aus wenn kommando1 erfolgreich War
Kommando1 —— kommando2	Führt kommando2 aus wenn kommand1 einen Fehler Liefert
Kommando1 & kommando2	Startet kommando1 im Hintergrund, Kommando2 im Vordergrund
(kommando1; kommando2)	Führt beide Kommandos in der gleichen shell aus

1.3.12 Substitutionsmechanismen

Der Begriff Substitutionsmechanismus klingt zunächst sehr abstrakt. Gemeint ist, dass vor der Ausführung eines Kommandos, die durch Sonderzeichen gebildeteten Ausdrücke von der bash ausgewertet werden. Dies geschieht bei der Verwendung von Jokerzeichen zur Bildung von Dateinamen, bei der Zusammensetzung von Zeichenketten, bei der Berechnung arithmetischer Klammern, bei der Verwendung von umgekehrten Apostrophen zur Kommandosubstitution usw.

*	Der Stern steht für eine beliebige Zeichenfolge - oder für überhaupt kein Zeichen. Dazu ein Beispiel: **`ab*`** steht für alle Dateinamen, die mit "ab" anfangen, auch für "ab" selbst ("ab", "abc", "abcd", "abxyz", usw.).
?	Das Fragezeichen steht für genau ein beliebiges Zeichen. Zum Beispiel: **`?bc`** steht für alle Dateinamen mit 3 Zeichen, die auf "bc" enden ("abc", "bbc", "1bc", "vbc", "xbc", usw.), nicht jedoch für "bc".
[ ]	Die eckige Klammer wird ersetzt durch **eines** der in der Klammer stehenden Zeichen. Auch ein Bereich ist möglich, z. B. [a-k] = [abcdefghijk]. Beispiel: **`a[bcd]`** wird ersetzt durch "ab", "ac" und "ad". Soll das Minuszeichen selbst in die Zeichenmenge aufgenommen werden, muss es an erster Stelle stehen (gleich nach der öffnenden Klammer).
[!]	Die eckige Klammer mit Ausrufezeichen wird ersetzt durch eines der **nicht** in der Klammer stehenden Zeichen, zum Beispiel: **`[!abc]`** wird ersetzt durch ein beliebiges Zeichen außer a, b oder c. Soll das Ausrufezeichen selbst in die Zeichenmenge aufgenommen werden, muss es an letzter Stelle stehen.
\	Der Backslash hebt den Ersetzungsmechanismus für das folgende Zeichen auf. Beispiel: **`ab\?cd`** wird zu "ab?cd" - das Fragezeichen wird übernommen. **Wichtig:** Bei der Umleitung von Ein- und Ausgabe werden Metazeichen in den Dateinamen hinter dem Umleitungszeichen nicht ersetzt.

Beispiele für die Anwendung:

`ls -l a*`	listet alle Dateien, die mit "a" anfangen
`ls test?`	listet alle Dateien die mit "test" anfangen und 5 Zeichen lang sind ("test1", "test2", "testa")
`ls /dev/tty1[0-9]`	listet alle Terminalbezeichnungen mit einer 1 in der Zehnerstelle ("tty11", "tty12", ... , "tty19")

1.3.13 Dateinamenbildung mit Jokerzeichen

Wie bereits mehrfach angesprochen, ist nicht das jeweilige Kommando für die Auswertung der Jokerzeichen wie in Tabelle 7.11 auf Seite 68 erläutert, zuständig, sondern die bash selbst. Diese übergibt nach der Expansion des Ausdrucks eine Liste mit den Dateien bzw. Verzeichnissen an das Kommando.

Da oft das Ergebnis eines mit Jokerzeichen versehenen Ausdrucks nicht den Erwartungen entspricht, sollte man zunächst einmal die Expansion genauer betrachten. Außerdem ist zu beachten, was das Kommando dann mit der übergebenen Liste an Datei- und Verzeichnisnamen anstellt.

Beispiele:

```
echo /*
ls /*
```

Lebenswichtig:
Der * ist ein gefährliches Zeichen, Tippfehler können zum Fiasko führen, wenn aus Versehen ein Leerzeichen zuviel getippt wird.

`rm a*`	löscht beispielsweise alle Dateien, die mit "a" anfangen.
`rm a *`	löscht dagegen erst die Datei "a" und dann alle Dateien im Verzeichnis.

Anmerkungen:

- Der Punkt am Anfang von Dateinamen stellt eine Ausnahme dar, er muss explizit angegeben werden (wegen der Verzeichnisreferenzen **"."** bzw. **".."** und der Tatsache, dass Dateien, die mit einem Punkt beginnen, normalerweise nicht angezeigt werden).
- Der **"\"**am Zeilenende unterdrückt auch das Return-Zeichen - das Kommando kann in der folgenden Zeile fortgesetzt werden (es erscheint dann der Prompt ">" anstelle von "$").

1.3.14 String-Ersetzungen (Quoting)

Um bestimmte Sonderzeichen (z. B. *, ?, [ ], Leerzeichen, Punkt) zu übergeben, ohne dass sie von der Shell durch Dateinamen ersetzt werden, werden Anführungszeichen verwendet, die auch ineinander geschachtelt werden können. Dabei haben die drei verschiedenen Anführungszeichen Doublequote ("), Quote (') und Backquote (`) unterschiedliche Bedeutung:

" "	Keine Ersetzung der Metazeichen * ? [ ], jedoch Ersetzung von Shellvariablen (siehe unten) und Ersetzung durch die Ergebnisse von Kommandos (Backquote). Auch \ funktioniert weiterhin. Dazu ein Beispiel: **`echo Der * wird hier durch alle Dateinamen ersetzt`** **`echo "Der * wird hier nicht ersetzt"`**
' '	Das einfache Anführungszeichen unterdrückt jede Substitution. Zum Beispiel: **`echo 'Weder * noch `pwd` werden ersetzt'`**
` `	Zwischen Backquote (Accent Grave) gesetzte Kommandos werden ausgeführt und das Ergebnis wird dann als Parameter übergeben (d. h. die Ausgabe des Kommandos landet als Parameter in der Kommandozeile). Dabei werden Zeilenwechsel zu Leerzeichen. Braucht dieses Kommando Parameter, tritt die normale Parameterersetzung in Kraft. Zum Beispiel: **`echo "Aktuelles Verzeichnis: `pwd`"`** Weil die verschiedenen Quotes manchmal schwer zu unterscheiden sind, wurde bei der *bash* eine weitere Möglichkeit eingeführt. Statt in Backquotes wird die Kommandofolge in $(...) eingeschlossen., z. B.: **`echo "Aktuelles Verzeichnis: $(pwd)"`**

1.3.15 Berechnung arithmetischer Ausdrücke

Die bash ist in der Lage, bei der Eingabe eines korrekten Ausdrucks diesen zu berechnen. Hierbei sind die meisten aus der Programmiersprache C bekannten Operatoren erlaubt.:

- Die vier Grundrechenarten (+,-, *,/),
- die Modulofunktion %,
- für Vergleich (==, !=, <, <=, >, >=),
- für Bitverschiebungen (<<, >>) und schließlich
- die logischen Operatoren Nicht (!), Und (& &) und Oder (——).

Hierbei gültig sind auch die üblichen Rechenregeln (z.B. Punkt vor Strich).

Beispiel:

```
echo `expr 2 + 3`
```

1.3.16 Reguläre Ausdrücke

Reguläre Ausdrücke (engl.: regular expressions)sind ein Begriff aus der Theoretischen Informatik – Reguläre Ausdrücke beschreiben Mengen aus Zeichenketten. Wichtig werden Sie daher immer dann, wenn Sie wie oben beschrieben Dateien an der Kommandozeile spezifizieren wollen. Doch auch bei der Suche von Mustern in Textdateien sind Reguläre Ausdrücke nützlich.

1.3.16.1 Syntax von Regulären Ausdrücken

Reguläre Ausdrücke setzt man aus einzelnen Zeichen zusammen. Die Bedeutung dieser entnehme man der folgenden Tabelle der man-Page von grep. Diese Bausteine können Sie beliebig zusammensetzen, um Reguläre Ausdrücke zu bilden. Weitere Besonderheiten von Regulären Ausdrücken entnehmen Sie bitte der man-Page von grep.

1.3.16.2 grep

grep wird gebraucht, um Dateien, oder die Ausgabe von Programmen, zu durchsuchen, und nach Stichworten und Mustern zu filtern. Als Parameter dienen dabei ein Regulärer Ausdruck wie oben beschrieben und eine Datei. Alternativ kann grep auch über so genannte Pipes an “normale” Programme und Anweisungen “angehängt” werden. Dann wird die Ausgabe dieser Programme nach dem angegebenen Muster gefiltert.

1.3.16.3 Bausteine regulärerer Ausdrücke:

Zeichen	Bedeutung
.	Beliebiges Zeichen
\w	Beliebiges alphanumerisches Zeichen
\W	Beliebiges nicht alphanumerisches Zeichen
[xyz]	Eines der Zeichen x, y oder z
[x-z]	Ein Zeichen aus x - z
[^q-v]	Alle Zeichen außerhalb des Bereichs q-v

1.3.16.4 Multiplikatorbausteine:

Zeichen	Bedeutung
?	Muster kommt einmal oder gar nicht vor
*	Muster kommt nicht oder beliebig oft vor
+	Muster kommt mindestens einmal vor
{n}	Muster kommt exakt n mal vor
{n,}	Muster kommt mindestens n mal vor
{n,m}	Muster kommt zwischen n und m mal vor

Beispiele:

Durchsuche die Liste der aktuellen Prozesse nach denen des Users “michael”

```
ps -eadf | grep michael
```

Durchsuche die `/etc/fstab` nach Master-Festplatten am Primary-Port:

```
grep "\/dev\/hda*" /etc/fstab
```

1.3.16.5 Limits

Mit dem Befehl ulimits ist es möglich, Limits für die Nutzung von Systemressourcen zu setzen bzw. sich diese anzeigen zu lassen. Allerdings wirken sich die Beschränkungen nur auf Prozesse aus, die durch die Shell gestartet werden. Die zahlreichen Optionen und die Bedeutung von Hard- und Softlimit können in der manpage zur Bash nachgeschlagen werden.

Die gesetzten Limits werden übrigens durch
```
ulimit  a
```
sichtbar.

Systemweite Einstellungen sollten in **/etc/profile** bzw. **/etc/profile.local** gesetzt werden. Die Angaben erfolgen in kB.

1.3.16.6 Parameterübergabe

Shell-Skripts können mit Parametern aufgerufen werden, auf die über ihre Positionsnummer zugegriffen werden kann. Die Parameter können zusätzlich mit vordefinierten Werten belegt werden.

Positionsparameter	Bedeutung
$#	Anzahl der Argumente
$0	Name des Kommandos
$1	1. Argument
. . . .	. . . .
$9	9. Argument
$@	alle Argumente (z. B. für Weitergabe an Subshell)
$*	alle Argumente konkateniert (--> ein einziger String)

Zur Verdeutlichung soll ein kleines Beispiel-Shell-Skript **testparam** dienen:

```
#!/bin/sh
echo "Mein Name ist $0"
echo "Mir wurden $# Parameter übergeben"
echo "1. Parameter = $1"
```

```
echo "2. Parameter = $2"
echo "3. Parameter = $3"
echo "Alle Parameter zusammen: $*"
echo "Meine Prozessnummer PID = $$"
```

Nachdem dieses Shell-Skript mit einem Editor erstellt wurde, muss es noch ausführbar gemacht werden

```
chmod 766 testparam
```

Anschließend wird es gestartet und erzeugt die folgenden Ausgaben auf dem Bildschirm:

```
$ ./testparam eins zwei drei vier
Mein Name ist ./testparam
Mir wurden 4 Parameter übergeben
1. Parameter = eins
2. Parameter = zwei
3. Parameter = drei
Alle Parameter zusammen: eins zwei drei vier
Meine Prozessnummer PID = 3212
$
```

1.3.16.7 Anmerkungen

- So, wie Programme und Skripts des UNIX-Systems in Verzeichnissen wie **/bin** oder **/usr/bin** zusammengefasst werden, ist es empfehlenswert, im Home-Directory ein Verzeichnis **bin** einzurichten, das Programme und Skripts aufnimmt.
- Die Variable PATH wird dann in der Datei **.profile** durch die Zuweisung
  PATH=$PATH:$HOME/bin
  erweitert.
- Damit die Variable **PATH** auch in Subshells (d. h. beim Aufruf von Skripts) auch wirksam wird, muss sie exportiert werden:
  export PATH
- Alle exportierten Variablen bilden das Environment für die Subshells. Information darüber erhält man mit dem Kommandos:

	set	Anzeige von Shellvariablen und Environment-Variablen
oder	**env**	Anzeige der Environment-Variablen

In Shellskripts kann es sinnvoll sein, die Variablen in Anführungszeichen ("...") zu setzen, um Fehler zu verhindern. Beim Aufruf müssen Parameter, die Sonderzeichen enthalten ebenfalls in Anführungszeichen (am besten '...') gesetzt werden. Dazu ein Beispiel. Das Skript "zeige" enthält folgende Zeile:

```
grep $1 dat.adr
```

Der Aufruf

```
zeige 'Hans Meier'
```

liefert nach der Ersetzung das fehlerhafte Kommando

```
grep Hans Meier dat.adr
```

das nach dem Namen 'Hans' in der Adressdatei **dat.adr** und einer (vermutlich nicht vorhandenen) Datei namens 'Meier' sucht.

Die Änderung von "zeige"

```
grep "$1" dat.adr
```

liefert bei gleichem Parameter die korrekte Version

```
grep "Hans Meier" dat.adr.
```

Die zweite Quoting-Alternative

```
grep '$1' dat.adr
```

ersetzt den Parameter überhaupt nicht und sucht nach der Zeichenkette "$1".

Das Skript "zeige" soll nun enthalten:

```
echo "Die Variable XXX hat den Wert $XXX"
```

Nun wird eingegeben:

```
$ XXX=Test
$ zeige
```

Als Ausgabe erhält man:

```
Die Variable XXX hat den Wert
```

Erst wenn die Variable "exportiert" wird, erhält man das gewünschte Ergebnis:

```
$ XXX=Test
$ export XXX
$ zeige
Die Variable XXX hat den Wert Test
```

Das Skript "zeige" enthalte nun die beiden Kommandos:

```
echo "zeige wurde mit $# Parametern aufgerufen:"
echo "$*"
```

Die folgenden Kommandoaufrufe zeigen die Behandlung unterschiedlicher Parameter:

```
$ zeige
zeige wurde mit 0 Parametern aufgerufen:

$zeige eins zwei 3
zeige wurde mit 3 Parametern aufgerufen:
eins zwei 3

$ zeige eins "Dies ist Parameter 2" drei
zeige wurde mit 3 Parametern aufgerufen:
eins Dies ist Parameter 2 drei
```

Die Definition von Variablen (und Shell-Funktionen) kann man mit **unset** wieder Rückgängig machen.

1.3.17 Gültigkeit von Kommandos und Variablen

Jeder Kommandoaufruf und somit auch der Aufruf einer Befehlsdatei (Shellskript) hat einen neuen Prozess zur Folge. Es wird zwar das Environment des Elternprozesses "nach unten" weitergereicht, jedoch gibt es keinen umgekehrten Weg. Auch der Effekt der Kommandos (z. B. Verzeichniswechsel) ist nur innerhalb des Kindprozesses gültig. Im Elternprozess bleibt alles beim alten. Das gilt natürlich auch für Zuweisungen an Variablen.

Die Kommunikation mit dem Elternprozess kann aber z. B. mit Dateien erfolgen. Bei kleinen Dateien spielt sich fast immer alles im Cache, also im Arbeitsspeicher ab und ist somit nicht so ineffizient, wie es zunächst den Anschein hat. Außerdem liefert jedes Programm einen Rückgabewert, der vom übergeordneten Prozess ausgewertet werden kann.

```
0: O. K.
> 0: Fehlerstatus
```

Es gibt außerdem ein Kommando, das ein Shell-Skript in der aktuellen Shell und nicht in einer Subshell ausführt:

```
. zeige
```

Das Dot-Kommando erlaubt die Ausführung eines Skripts in der aktuellen Shell-Umgebung, z. B. das Setzen von Variablen usw.

Damit die Variable auch in Subshells (d. h. beim Aufruf von Skripts auch wirksam wird, muss sie exportiert werden:

```
export $PATH
```

Alle exportierten Variablen bilden das Environment für die Subshells.

1.3.18 Interaktive Eingaben in Shellscripts

Es können auch Shellskripts mit interaktiver Eingabe geschrieben werden, in dem das read-Kommando verwendet wird.

```
read variable [variable ...]
```

read liest eine Zeile von der Standardeingabe und weist die einzelnen Felder den angegebenen Variablen zu. Feldtrenner sind die in $IFS definierten Zeichen. Sind mehr Variablen als Eingabefelder definiert, werden die überzähligen Felder mit Leerstrings besetzt. Umgekehrt nimmt die letzte Variable den Rest der Zeile auf.

Wird im Shell-Skript die Eingabe mit < aus einer Datei gelesen, bearbeitet read die Datei zeilenweise.

Anmerkung:
Da das Shell-Skript in einer Sub-Shell läuft, kann $IFS im Skript umdefiniert werden, ohne dass es nachher restauriert werden muss. Die Prozedur "zeige" enthält beispielsweise folgende Befehle:

```
IFS=','
echo "Bitte drei Parameter, getrennt durch Komma eingeben:"
read A B C
echo Eingabe war: $A $B $C
```

Aufruf (Eingabe rot, Ausgabe blau):

```
$ zeige
Bitte drei Parameter, getrennt durch Komma eingeben:
eins,zwei,drei
Eingabe war: eins zwei drei
```

1.3.19 Weitere wichtige Shell-Strukturen

- Bedingungen
- Eigenschaften von Dateien
- Vergleiche und logische Verknüpfungen
- Bedingte Anweisungen
- case-Anweisung
- for-Anweisung
- while-Anweisung
- until-Anweisung
- exit
- break

2. Beispiele für Shell-Scripts

2.1 Datei verlängern

"Wie hänge ich den Inhalt einer Variablen an eine Datei an?":

```
( cat file1 ; echo "$SHELLVAR" ) > file2
```

2.2 Telefonbuch

Telefonverzeichnis mit Hier-Dokument. Aufruf tel Name [Name ..]:

```
if [ $# -eq 0 ]
then
   echo "Usage: `basename $0` Name [Name ..]"
   exit 2
fi
for SUCH in $*
do
   if [ ! -z $SUCH ] ; then
        grep $SUCH << "EOT"
        Hans  123456
        Fritz 234561
        Karl  345612
        Egon  456123
        EOT
   fi
done
```

2.3 Auflistung des Directory-Trees

2.3.1 Kompakte Auflistung des Inhalts eines ganzen Dateibaums

Dies ist ein rekursives Programm. Damit es klappt, muss das folgende Skript **dir** über **$PATH** erreichbar sein.

Aufruf: dir Anfangspfad.

```
if test $# -ne 1
then
    echo "Usage: dir Pfad"
    exit 2
fi
cd $1
echo
pwd
ls -CF
for I in *
do
    if test -d $I
    then
        dir $I
    fi
done
```

2.3.2 Auflisten des Dateibaums in grafischer Form

Analog zur vorherigen Aufgabenstellung, wobei zur Dateiauswahl alle Optionen des find-Kommandos zur Verfügung stehen. Aufruf z.B. tree . für alle Dateien oder tree . -type d für Directories ab dem aktuellen Verzeichnis.

```
# find liefert die vollständigen Pfadnamen (temporäre Datei).
# Mit ed werden die "/"-Zeichen erst zu "|---- " expandiert
# und dann in den vorderen Spalten aus "|---- " ein Leerfeld
# "     |" gemacht.
if test $# -lt 1
then
    echo "Usage: tree Pfadname [Pfadname ...] [find-Options]"
else
   TMPDAT=$0$$
   find $@ -print > $TMPDAT 2>/dev/null
   ed $TMPDAT << "EOT" >/dev/null 2>/dev/null
1,$s/[^\/]*\//|---- /g
1,$s/---- |/     |/g
w
q
EOT
   cat $TMPDAT
    rm $TMPDAT
   fi
```

Mit dem Stream-Editor sed kann das sogar noch kompakter formuliert werden:

```
if [ $# -lt 1 ]
then
    echo "Usage: tree Pfadname [Pfadname ...] [find-Options]"
else
    find $@ -print 2>/dev/null | \
    sed -e '1,$s/[^\/]*\//|---- /g' -e '1,$s/---- |/     |/g'
  fi
```

2.4 Argumente mit J/N-Abfrage ausführen

Das folgende Skript führt alle Argumente nach vorheriger Abfrage aus. Mit "j" wird die Ausführung bestätigt, mit "q" das Skript abgebrochen und mit jedem anderen Buchstaben (in der Regel "n") ohne Ausführung zum nächsten Argument übergegangen. Ein- und Ausgabe erfolgen immer über das Terminal(-fenster), weil /dev/tty angesprochen wird. Das Skript wird anstelle der Argumentenliste bei einem anderen Kommando eingesetzt, z. B. Löschen mit Nachfrage durch rm $(pick *)

```
# pick - Argumente mit Abfrage liefern
for I ; do
   echo "$I (j/n)? \c" > /dev/tty
   read ANTWORT
   case $ANTWORT in
     j*|J*) echo $I ;;
     q*|Q*) break ;;
   esac
done </dev/tty
```

2.5 Sperren des Terminals während man kurz weggeht

Nach Aufruf von lock wird ein Kennwort eingegeben und das Terminal blockiert. Erst erneute Eingabe des Kennwortes beendet die Prozedur. Die Gemeinheit dabei ist, dass sich bei jeder Fehleingabe die Wartezeit verdoppelt.

```
echo "Bitte Passwort eingeben: \c"
stty -echo  # kein Echo der Zeichen auf dem Schirm
read CODE
stty echo
tput clear  # BS loeschen
trap "" 2 3
banner " Terminal "
banner " gesperrt "
MATCH=""
DELAY=1
while [ "$MATCH" != "$CODE" ]
do
     sleep $DELAY
     echo "Bitte Passwort eingeben: \c"
     read MATCH
     DELAY='expr $DELAY \* 2`  # doppelt so lange wie vorher
done
echo
```

2.6 Dateien im Pfad suchen

Das folgende Skript bildet das Kommando "which" nach. Es sucht im aktuellen Pfad (durch PATH spezifiziert) nach der angegebenen Datei und gibt die Fundstelle aus. An diesem Skript kann man auch eine Sicherheitsmaßnahme sehen. Für den Programmaufruf wird der Pfad neu gesetzt, damit nur auf Programme aus /bin und /usr/bin zugegriffen wird. Bei Skripts, die vom Systemverwalter für die Allgemeinheit erstellt werden, sollte man entweder so verfahren oder alle Programme mit absolutem Pfad aufrufen.

```
#!/bin/sh
# Suchen im Pfad nach einer Kommando-Datei
OPATH=$PATH
PATH=/bin:/usr/bin
if [ $# -eq 0 ] ; then
   echo "Usage: which kommando" ; exit 1
fi
for FILE
do
   for I in `echo $OPATH | sed -e 's/^:/.:/' -e 's/::/:.:/g \ -e
's/:$/:./'`
   do
      if [ -f "$I/$FILE" ] ; then
        ls -ld "$I/$FILE"
       fi
   done
done
```

2.7 Berechnung von Primfaktoren einer Zahl

```
echo "Zahl eingeben: \c"; read ZAHL
P=2
while test `expr $P \* $P` -le $ZAHL; do
     while test `expr $ZAHL % $P` -ne 0; do
        if test $P -eq 2; then
           P=3
       else
            P=`expr $P + 2`
       fi
    done
    ZAHL=`expr $ZAHL / $P`
    echo $P
done
if test $ZAHL -gt 1; then
   echo $ZAHL
fi
echo ""
```

2.8 Berechnung des Osterdatums nach C.F. Gauss

```
if [ $# -eq 0 ] ; then
    echo "Osterdatum fuer Jahr: \c"; read JAHR
else
   JAHR="$1"
fi
G=`expr $JAHR % 19 + 1`
C=`expr $JAHR / 100 + 1`
X=`expr \( $C / 4 - 4 \) \* 3`
Z=`expr \( $C \* 8 + 5 \) / 25 - 5`
D=`expr $JAHR \* 5 / 4 - $X - 10`
E=`expr \( $G \* 11 + $Z - $X + 20 \) % 30`
if test $E -lt 0; then
   $E=`expr $E + 30`
fi
if [ $E -eq 25 -a $G -gt 11 -o $E -eq 24 ] ; then
   E=`expr $E + 1`
fi
TAG=`expr 44 - $E`
if [ $TAG -lt 21 ] ; then
   TAG=`expr $TAG + 30`
fi
TAG=`expr $TAG + 7 - \( $D + $TAG \) % 7`
if [ $TAG -gt 31 ] ; then
   TAG=`expr $TAG - 31`
   MON=4
else
   MON=3
fi
echo "Ostern $JAHR ist am ${TAG}.${MON}.\n"
```

Statt des expr-Befehls kann bei der Bash auch das Konstrukt $((...)) verwendet werden. Das Programm sieht dann so aus:

```
if [ $# -eq 0 ] ; then
    echo "Osterdatum fuer Jahr: \c"; read JAHR
else
   JAHR="$1"
fi
G=$(($JAHR % 19 + 1))
C=$(($JAHR / 100 + 1))
X=$((\( $C / 4 - 4 \) \* 3))
Z=$((\( $C \* 8 + 5 \) / 25 - 5))
D=$(($JAHR \* 5 / 4 - $X - 10))
E=$((\( $G \* 11 + $Z - $X + 20 \) % 30))
if test $E -lt 0; then
   $E=$(($E + 30))
fi
if [ $E -eq 25 -a $G -gt 11 -o $E -eq 24 ] ; then
   E=$(($E + 1))
fi
TAG=$((44 - $E))
if [ $TAG -lt 21 ] ; then
   TAG=$(($TAG + 30))
fi
TAG=$(($TAG + 7 - \( $D + $TAG \) % 7))
if [ $TAG -gt 31 ] ; then
   TAG=$(($TAG - 31))
   MON=4
else
   MON=3
fi
echo "Ostern $JAHR ist am ${TAG}.${MON}.\n"
```

2.9 Wem die Stunde schlägt

Wer im Besitz einer Soundkarte ist, kann sich eine schöne Turmuhr, einen Regulator oder Big Ben basteln. Die folgende "Uhr" hat Stunden- und Viertelstundenschlag. Damit das auch klappt, ist ein Eintrag in der crontab nötig:

```
0,15,30,45 * * * * /home/sbin/turmuhr
```

So wird das Skript turmuhr alle Viertelstunden aufgerufen. Es werden zwei Sounddateien verwendet, hour.au für den Stundenschlag und quater.au für den Viertelstundenschlag. Statt des Eigenbau-Programms audioplay kann auch der sox verwendet werden oder man kopiert die Dateien einfach nach /dev/audio. Die Variable VOL steuert die Lautstärke.

```
#!/bin/sh
BELL=/home/local/sounds/hour.au
BELL1=/home/local/sounds/quater.au
PLAY=/usr/bin/audioplay
VOL=60
DATE=`date +%H:%M`
MINUTE=`echo $DATE | sed -e 's/.*://'`
HOUR=`echo $DATE | sed -e 's/:.*//'`

if [ $MINUTE = 00 ]
then
    COUNT=`expr \( $HOUR % 12 + 11 \) % 12`
    BELLS=$BELL
    while [ $COUNT != 0 ];
    do
        BELLS="$BELLS $BELL"
        COUNT=`expr $COUNT - 1`
    done
    $PLAY -v $VOL -i $BELLS
elif [ $MINUTE = 15 ]
then    $PLAY -v $VOL -i $BELL1
elif [ $MINUTE = 30 ]
then    $PLAY -v $VOL -i $BELL1 $BELL1
elif [ $MINUTE = 45 ]
then    $PLAY -v $VOL -i $BELL1 $BELL1 $BELL1
else
    $PLAY -v $VOL -i $BELL1
fi
```

3. Abbildungsverzeichnis

Abbildungsverzeichnis