SelfLinux-0.10.0

Einführung in die Bourne Again Shell (bash)

Autor: Matthias Kleine (*kleine_matthias@gmx.de*)
Formatierung: Torsten Hemm (*T.Hemm@gmx.de*)

Dieses Kapitel führt in die Grundlagen der Verwendung einer Shell ein. Es wird Wert darauf gelegt, dass der Benutzer die elementaren Mechanismen versteht und zu seinem Vorteil anwenden kann. Als Demonstrationsobjekt dient die Bourne Again Shell, da sie die meistverbreitete Shell unter Linux und insbesondere da sie kompatibel zu der Mutter aller Unix-Shells, der Bourne Shell ist.

Inhaltsverzeichnis

1 Funktionsweise

Sie möchten ein Kommando ausführen. Das Kommando liegt im Binärformat auf einem Ihrer Datenträger. Da Sie das Kommando nicht mit Ihrem Finger anschnippen können, muß es einen Weg geben, das Kommando aufzurufen. Vielleicht halten Sie das für einen einfachen Vorgang, aber das ist es nicht.

Aus der Sicht des Betriebssystems ist ein Kommando ein Prozeß wie jeder andere auch. Auf einem Multitasking-Betriebssystem wie Linux laufen zu jeder Zeit eine große Zahl von Prozessen **gleichzeitig**. Bei der Besprechung des Linux-Kernels werden wir später noch darauf eingehen, was dieses **gleichzeitig** bedeutet und welche Arbeit dabei für den Kernel anfällt. Im Augenblick genügt es uns zu wissen, dass die Prozeßverwaltung - und damit auch der Start von Prozessen - zu den wesentlichen Aufgaben des Betriebssystems selbst gehört.

In Wahrheit ist es so, dass Sie selbst gar nicht befugt sind, ein Kommando zu starten. Vielmehr müssen Sie das Betriebssystem damit beauftragen, dies für Sie zu tun. Damit tut sich aber gleich das nächste Problem auf: Der Kernel selbst ist lediglich eine Sammlung von C-Funktionen, und es dürfte wohl kaum in Ihrem Interesse liegen, mit diesen Funktionen selbst zu kommunizieren. Keine Angst - das müssen Sie auch nicht.

Wie Sie auf dieser schematischen Abbildung sehen können, bildet die Shell eine Schale um den Systemkern. Daher trägt sie auch ihren Namen - im Englischen bedeutet **shell** soviel wie **Schale** oder **Muschel**. Damit ist ausgedrückt, dass ein Zugriff auf Betriebssystem - Routinen üblicherweise über eine Shell erfolgt - die Shell vermittelt zwischen dem Benutzer und dem Betriebssystem. Um dem Betriebssystem also beispielsweise den Auftrag zu geben, ein bestimmtes Kommando für Sie zu starten, kommunizieren Sie zunächst mit der Shell, und diese reicht Ihren Auftrag in geeigneter Form weiter.

In der Unix - Welt gibt es viele Shells, von denen einige sicher zweckmäßiger als andere sind. Die meisten dieser Shells sind in einer freien Version auch unter Linux verwendbar. Die meist verbreitete Shell unter Linux, die von praktisch allen Distributoren als die Standardshell verwendet wird, ist die Bourne Again Shell, eine verbesserte Version der alten Bourne Shell. Alle weiteren Ausführungen in diesem Abschnitt werden sich daher auf die *Bash* beziehen.

Bei der Eingabe von Kommandos werden Sie von der Shell auf vielfältige Art und Weise unterstützt. Einen Teil dieser Unterstützung stellen Editierhilfen dar, die Ihnen einfach etwas Tipparbeit abnehmen sollen. Diesen Hilfen werden wir uns in nächsten Absatz widmen. Alle weiteren Absätze widmen sich den Shell - Mechanismen, die einen flexiblen und effizienten Gebrauch von Kommandos ermöglichen. Diese Mechanismen stellen gleichzeitig auch die Grundlage der Shellprogrammierung dar, die wir jedoch erst in einem späteren Kapitel behandeln möchten.

2 Hinweise zur Benutzung

2.1 Die History

Wie viele andere Shells verfügt die Bash über eine Liste der zuletzt abgesetzten Kommandos, eine sogenannte History. Selbst wenn Sie nur gelegentlich die Kommandozeile verwenden, erweist sich die History als ein ausgesprochen nützlicher Helfer. Das gilt umso mehr, wenn Sie ausgiebigen Gebrauch von der Shell machen. Die Möglichkeiten zur Nutzung der History entsprechen der Benutzung eines effizienten Editors und werden in ihrem vollen Umfang nur von den Wenigsten benutzt. Sie gehen weit über die Möglichkeiten beispielsweise von doskey hinaus, das Sie vielleicht noch aus DOS-Zeiten kennen. Wir möchten an dieser Stelle wieder nur die beiden Möglichkeiten herausgreifen, die für den alltäglichen Gebrauch die wichtigste Rolle spielen, und verweisen für weitere Details auf das fortgeschrittene Shellkapitel.

Eine Übersicht über die aktuelle History erhalten Sie mit dem Kommando

```
user@linux ~/ # history
```

ohne die Angabe eines Parameters. Das Kommando gibt eine nummerierte Liste aus, die alle abgesetzten Kommandos inklusive ihrer Parameter enthält. Standardmäßig werden bis zu 500 Kommandos verwaltet, dies übrigens unabhängig davon, ob ein Kommando syntaktisch richtig war oder nicht. Die Kommandos werden als Strings in exakt der eingegebenen Form gespeichert, genau so, wie sie bei der Eingabe am Bildschirm erscheinen. Einzige Voraussetzung für die Aufnahme in die Liste ist die Bestätigung des Kommandos durch die Enter-Taste. Die Zahl der verwalteten Kommandos kann verändert werden - mehr zu Konfigurationsfragen später. Die für uns interessante Frage lautet nun, in welcher Weise wir von der History möglichst effizient Gebrauch machen können.

Der häufigste Gebrauch der History besteht in der Verwendung der Pfeiltasten HOCH und RUNTER. Die HOCH- Taste läßt Sie das zuletzt abgesetzte Kommando in die aktuelle Kommandozeile zurückholen. Sie brauchen danach nur noch Enter zu drücken, um das Kommando nochmals abzusetzen. Sie können die Kommandozeile aber auch wie gewöhnlich editieren und erst dann bestätigen. Wiederholtes Drücken der HOCH - Taste läßt Sie jeweils um einen Schritt weiter in der Liste zurückgehen. Mit der RUNTER - Taste gehen Sie wieder den umgekehrten Weg nach vorne. Auf diese Weise können Sie sehr schnell in den zuletzt abgesetzten Kommandos blättern.

Je komplexer Ihre Kommandos werden, desto sinnvoller kann es werden, auch weiter zurückliegende Kommandos wiederzuholen. Nun ist einfaches Herumblättern nicht gerade ein effizienter Suchalgorithmus. Neben einer Reihe weiterer Methoden leistet hier insbesondere die inkrementelle Rückwärtssuche gute Dienste. Sie wird durch die Tastenkombination `STRG + R` eingeleitet, die zu dem folgenden Prompt führt:

```
(reverse-i-search)`':
```

Sie können nun damit beginnen, einen beliebigen String einzugeben, der in dem Kommando enthalten ist, das Sie aus der History zurückholen wollen. Zu dem von Ihnen angegebenen String wird das letzte Kommando herausgesucht, in dem der von Ihnen eingegebene String vorkommt, und hinter dem Doppelpunkt angezeigt. Sie müssen die Eingabe nun lediglich so lange verfeinern, bis das gewünschte Kommando erscheint. Danach können Sie es entweder mit Enter sofort absetzen oder eingeleitet durch ESC das Kommando zuvor noch editieren.

2.2 Tastenkürzel

In den obigen Absätzen war gelegentlich vom Editieren der Kommandozeile die Rede. Nun scheiden sich bei persönlichen Vorlieben für bestimmte Editoren gewöhnlich die Geister. Unter Linux fällt die Entscheidung meist für einen der Editoren vi oder Emacs in einer ihrer Implementationen. Wenn Sie einen dieser Editoren als Ihren Leib- und Mageneditor bezeichnen, können Sie die folgende Tabelle ruhig überspringen. Alle anderen können dieser Tabelle die elementarsten Tastenkürzel entnehmen, welche die Bash im Emacs-Modus zur Verfügung stellt. Der Emacs-Modus ist gleichzeitig die Standardeinstellung der Bash.

`Pfeiltasten`	VOR und ZURÜCK dienen wie üblich dem Verändern der Cursorposition
`Pos1`, `Ende`	an den Beginn / an das Ende der Zeile bewegen
`ALT + b`, `ALT + f`	je ein Wort rückwärts (»backward«) oder vorwärts (»forward«)bewegen
`Backspace`, `Entf`	Zeichen rückwärts / vorwärts löschen
`STRG + k`	bis zum Ende der Zeile löschen
`STRG + t`	die beiden vorangehenden Zeichen vertauschen (Dreher beseitigen)
`ALT + t`	die beiden vorangehenden Wörter vertauschen
`STRG + l`	löscht den Bildschirm

2.3 Die Nutzung der Tabulator-Taste

Die Tabulator-Taste stellt Ihnen einen Mechanismus zur Verfügung, den Sie gar nicht hoch genug einschätzen können: die Vervollständigung von Namen. Es handelt sich hier lediglich um einen Hilfsmechanismus für die Eingabe von Kommandozeilen und nicht etwa um einen mit der sogenannten Dateinamensexpansion verwandten Mechanismus. Sollte der Teufel es wollen, dass Sie von Berufs wegen mit der Kommandozeile arbeiten werden, können wir hier getrost festhalten, dass diese eine Taste ihnen viele Kilometer an Tastatureingaben ersparen wird.

Wozu die bash den begonnenen Namen zu vervollständigen sucht, hängt von Ihrer Eingabe ab. Beginnt die Eingabe mit einem `$`, versucht sie, einen Variablennamen daraus zu machen. Beginnt die Eingabe mit `~`, versucht sie einen Benutzernamen zu bilden. Beginnt sie mit `@`, versucht sie die Eingabe zu einem Hostnamen zu vervollständigen. Wenn keine dieser Bedingungen zutrifft, sucht die bash nach einem Alias- oder Funktionsnamen. Und last but not least (dies ist tatsächlich der häufigste Fall), bildet die bash einen Pfadnamen aus.

Selbstverständlich muß der Name, zu dem die bash vervollständigt, sei es nun eine Variable, ein Benutzername, ein Hostname, ein Alias, eine Funktion oder ein Pfad, auch wirklich existieren. Wenn Sie mit einigen der genannten Begriffe im Augenblick noch nichts anfangen können, machen Sie sich nichts draus. Alle diese Themen werden wir später noch ausführlich behandeln.

Schlagen alle Versuche, eine passende Vervollständigung zu erreichen, fehl, ertönt ein kurzer Piepston. Wenn Sie die Tabulator-Taste nun nocheinmal betätigen, zeigt die bash Ihnen alle möglichen Vervollständigungen an. Sie können dann die Eingabe so weit ergänzen, bis sie eindeutig ist, um den Namen schließlich wieder mit der Tabulator-Taste vervollständigen zu lassen. Wenn es überhaupt keine mögliche Vervollständigung Ihrer Eingabe gibt, quittiert die bash das wiederholte Drücken der Tabulator-Taste mit einem weiteren Piepston.

Die häufigste Anwendung dieses Mechanismus ist sicher das Navigieren im Dateibaum. Erstens kann man auf diese Weise auch lange Pfade in beachtlich kurzer Zeit eingeben (wenn Sie den Mechanismus erst einmal beherrschen, vergleichen Sie dies einmal mit der Klickerei in einem der grafischen Dateimanager). Und zweitens hilft es auch Ihrem Gedächtnis auf die Sprünge, wenn Sie einen Datei- oder Verzeichnisnamen nur noch ungefähr im Kopf haben. Tippen Sie dann einfach die ersten 2 oder 3 Buchstaben, die Sie noch im Kopf haben, und lassen Sie sich dann die möglichen Vervollständigungen anzeigen. Eine hübsche Sache.

2.4 Dateinamenexpansion

Obwohl der Begriff ebenfalls gut auf den gerade beschriebenen Mechanismus passen könnte, bezeichnet er doch etwas völlig anderes. Bei dieser Form der Expansion betätigen Sie keine Taste, sondern Sie geben ein Muster (engl. pattern) ein, nach dem die bash suchen soll. Ist die Suche erfolgreich, ersetzt die Shell das Muster durch jeden einzelnen gefundenen Dateinamen.

Sie können Muster bilden, indem Sie neben den üblichen Zeichen, welche einen Dateinamen bilden können, eines der Zeichen `*`, `?`, oder `[ bzw. ]` verwenden. Findet die Shell ein Wort, das eines dieser Zeichen enthält, betrachtet sie es automatisch als Muster und sucht nach passenden Dateinamen. Die Bedeutung der einzelnen Zeichen wird in folgender Tabelle ersichtlich:

`*`	eine beliebige Zeichenfolge, auch die leere
`?`	ein beliebiges einzelnes Zeichen
`[...]`	eines der in `[...]` aufgeführten Zeichen
`[!...]`	keines der in `[!...]` aufgeführten Zeichen (das Ausrufezeichen wirkt als Negation)

Die genannten Zeichen werden auch als Wildcards oder Jokerzeichen bezeichnet. Den Mechanismus, Wildcards auf alle Dateinamen aus einem Verzeichnis anzuwenden und aus den passenden Dateinamen eine Liste zu bilden, nennt man Globbing. Häufig will man eine Aktion für viele Dateien eines Verzeichnisses durchführen. Z.B. könnten Sie alle Dateien, die auf .gif enden, in ein anderes Verzeichnis verschieben wollen. In einem solchen Fall matchen Sie diese Dateien durch das Muster *.gif und benutzen das entsprechende Kommando, um die sich ergebende Dateiliste zu verschieben. Ähnliches gilt für das Fragezeichen.

Wenn Sie die Verwendung der eckigen Klammern noch nicht kennen, verdienen diese noch einige Erklärung. Durch eckige Klammern können Sie eine sogenannte Zeichenklasse definieren. Alle in den eckigen Klammern stehenden Zeichen stehen im Gesamtmuster für ein einzelnes Zeichen, ebenso wie das Fragezeichen. Während das Fragezeichen aber ein beliebiges Zeichen matcht, können Sie durch die eckigen Klammern ganz bestimmte Zeichen auswählen. Das Muster `[aeiou]` matcht einen beliebigen Vokal. Das Gesamtmuster `s[aeiou]x` paßt also auf **sax**, **sex**, **six**, **sox** und **sux**. Merken Sie sich, dass eine Zeichenklasse immer für ein einzelnes Zeichen steht. Die Datei saeioux würde beispielsweise nicht gematcht, da zwischen **s** und **x** mehr als ein Zeichen vorkommt.

Häufig ist es sinnvoller, die Zeichen anzugeben, die nicht gematcht werden sollen. Dann verwendet man die eckigen Klammern mit einem führenden Ausrufezeichen. Es handelt sich hier ebenso um eine Zeichenklasse wie ohne Ausrufezeichen, d.h., es wird genau ein Zeichen gematcht. Die Verwendung erfolgt also analog.

Die bash kennt weitere Mechanismen zur Expandierung, die gelegentlich nützlich sein können. Wir möchten es jedoch an dieser Stelle bei unseren Betrachtungen zum Thema belassen, um nicht zu tief in die Details einzusteigen und den Überblick zu wahren. Wenn Sie zum Expandierungsspezialisten aufsteigen wollen, lege ich Ihnen das fortgeschrittene Shellkapitel ans Herz, das auch die letzten chirurgischen Kunstgriffe der bash aufdecken wird.

3 Pipes und Verwandtes

3.1 Eingabe und Ausgabe von Daten

Programme verhalten sich meist so, dass sie bestimmte Daten aufnehmen, diese Daten auf irgendeine Weise verwenden, um schließlich wieder Daten auszugeben. Dieses Schema verdeutlicht sich noch bei interaktiven Programmen, die immer wieder Informationen vom Benutzer annehmen und ihm andere Informationen zurückliefern. Eine Shell ist ein typisches interaktives Programm. Zu diesem Zweck muß sie über einen Eingabekanal verfügen, über den sie Information aufnehmen kann. Dieser Eingabekanal existiert tatsächlich und erhält unter Linux die Bezeichnung **Standardeingabe**. Sie dürfen getrost bei der Vorstellung von einem Kanal oder Rohr bleiben, über das die Shell Daten entgegennimmt.

Womit aber ist die Standardeingabe verbunden? Ganz klar, üblicherweise wird dies Ihre Tastatur sein. Die Shell nimmt Zeichen für Zeichen von Ihrer Tastatur entgegen und gibt diese Zeichen auch sofort auf dem Bildschirm aus. Damit haben wir in technischer Hinsicht jedoch bereits einen großen Sprung getan, denn die Ausgabe von Zeichen auf dem Bildschirm kann selbstverständlich nicht von der Standardeingabe erledigt werden. Die Shell verfügt also über einen weiteren Kanal, der folgerichtig mit **Standardausgabe** bezeichnet wird. Die Standardausgabe der Shell ist üblicherweise mit Ihrem Monitor verbunden, so dass Sie die eingetippten Zeichen sehen können.

Es gibt noch einen dritten Kanal, der eine besondere Aufgabe zu erfüllen hat, der sogenannte **Standardfehlerkanal**. Wie der Name schon sagt, dient der Kanal zur Ausgabe von Fehlermeldungen, wenn der Programmlauf, in unserem Fall die Arbeit der Shell, aus irgendeinem Grund nicht ordnungsgemäß fortgesetzt werden konnte. Üblicherweise ist Standardfehler ebenfalls mit dem Bildschirm verbunden und schreibt daher seine Meldungen zwischen die gewöhnliche Ausgabe. Es macht jedoch Sinn, Stardardausgabe und Standardfehler voneinander zu trennen, um die Möglichkeit zu haben, gewöhnliche Ausgaben und Fehlerausgaben getrennt zu verarbeiten. Beispielsweise könnte man die Fehlerausgabe in eine Datei umlenken, um sie später zu analysieren, während, die gewöhnliche Ausgabe weiterhin über den Bildschirm läuft.

An dieser Stelle gilt es etwas Wichtiges zu verstehen: Standardeingabe, Standardausgabe und Standardfehler sind lediglich Kanäle, die mit irgendeiner Quelle und irgendeinem Ziel verbunden sein können. Standardeingabe ist nicht gleich Tastatur. Und Standardausgabe ist nicht gleich Monitor. Es gibt viele andere Quellen und Ziele, mit denen diese Kanäle verbunden werden können, wie beispielsweise Dateien oder andere Programme. Bei einer Shell macht es jedoch Sinn, Tastatur und Monitor als Eingabe und Ausgabe zu verwenden, daher ist dies die Voreinstellung.

Die drei Standardkanäle werden von Linux wie Dateien behandelt. Für geöffnete Dateien verwaltet das System eine Liste von Dateideskriptoren, die mit fortlaufenden ganzen Zahlen bezeichnet werden. Die Zahlen von `0` bis `2` sind für die drei Standardkanäle vorbelegt:

```
Standardeingabe       (stdin) : 0
Standardausgabe       (stdout): 1
Standardfehlerausgabe (stderr): 2
```

Bei der Umlenkung der Kanäle werden wir diese Bezeichnungen noch benötigen.

3.2 Datenströme für Kommandos

Standardeingabe und -ausgabe der Shell sind wenig aufregend, denn es handelt sich um die bloße Eingabe von Zeichen durch die Tastatur und deren Ausgabe auf dem Monitor.

Ebenso wie die Shell verfügen jedoch auch viele andere Programme und Kommandos über eine Standardeingabe und eine Standardausgabe. Ein hübsches Beispiel liefert uns das kleine Programm `wc` ("word count"), das zum Zählen von Worten, Zeilen und anderem dient. Lassen Sie uns hier durch Angabe der Option `-w` lediglich nach Worten zählen:

```
user@linux ~/ # wc -w
```

Der übliche Prompt der Shell verschwindet. Scheinbar geschieht nichts, aber das ist auch verständlich. `wc` dient schließlich dem Zählen - doch was soll es hier zählen? Es wartet schlichtweg auf eine Eingabe des Benutzers. Geben wir `wc` etwas zu tun:

```
user@linux ~/ # wc -w
wort1 wort2
```

Nanu? Noch immer kein Ergebnis? Nun, wir müssen `wc` mitteilen, dass die Eingabe beendet ist. In diesem Fall genügt nicht das einfache Enter, da `wc` auch nach dem Drücken der Entertaste weitere Worte erwartet. Das Endezeichen ist hier die Kombination `STRG + d`:

```
user@linux ~/ # wc -w
wort1 wort2

2
```

Aha. Sie können hier die Betätigung von `CTRL + d` nicht erkennen, aber Sie sehen das Ergebnis: `wc` hat die eingegebenen Worte gezählt und gibt die Anzahl auf dem Bildschirm aus. `wc` verfügt also über eine Standardeingabe, die hier mit der Tastatur verbunden war, und über eine Standardausgabe, wieder verbunden mit dem Bildschirm.

Erinnern wir uns nun daran, dass die Datenströme keineswegs fest mit Tastatur und Monitor verdrahtet sind. Tatsächlich ist es sehr einfach, diese Datenströme umzuleiten. Hierfür kommt eine Reihe spezieller Symbole zum Einsatz, die wir im folgenden kennenlernen werden.

3.3 Umleitung von Datenströmen

3.3.1 Aus einer Datei lesen

Wir wollen, dass die Standardeingabe nicht mit der Tastatur verbunden wird, sondern mit einer Datei. Zum Beispiel möchten wir mittels `wc` die Anzahl der Worte in einer Datei lesen:

```
user@linux ~/ # wc -w < datei.txt

157 datei.txt
```

`wc` hat diesmal nicht auf eine Eingabe gewartet, sondern direkt aus der Datei **datei.txt** gelesen. Dies wurde durch das Umlenkungszeichen `<` erreicht. Das folgende Bild veranschaulicht dies:

3.3.2 In eine Datei schreiben

Dasselbe können wir mit der Standardausgabe unternehmen. Verbinden wir sie mit der Datei anzahl.txt und geben einige Worte ein:

```
user@linux ~/ # wc -w > anzahl.txt
wort1 wort2 wort3
```

Der Prompt der Shell erscheint wieder, ohne dass wir die Ausgabe von `wc` lesen konnten. Statt auf dem Bildschirm landete die Ausgabe in der Datei anzahl.txt, was wir mittels `cat` (ein Kommando, das den Inhalt von Textdateien ausgibt) leicht ersehen können:

```
user@linux ~/ # cat anzahl.txt

3
```

Der Mechanismus wird durch das folgende Schaubild veranschaulicht:

3.3.3 An eine Datei anhängen

Durch die vorhergehende Umleitung wird der Inhalt der Datei überschrieben, falls die Datei bereits vorhanden war. Wollen wir die neuen Daten lediglich an das Ende der Datei anhängen, benutzen wir die folgende Schreibweise:

```
user@linux ~/ # wc -w >> anzahl.txt
wort1 wort2 wort3 wort4
user@linux ~/ # cat anzahl.txt

3
4
```

Der alte Inhalt ist erhalten geblieben.

3.3.4 Fehler umleiten

Der Fehlerkanal ist dem Dateideskriptor 2 zugeordnet. Bei der Umleitung des Fehlerkanals verwenden wir nun diese interne Bezeichnung:

```
user@linux ~/ # programm 2> error.txt
```

Dies ist irgendein Programm, dessen Fehlerausgabe wir in der Datei error.txt speichern wollen. Stehen in error.txt bereits andere Fehlermeldungen, die erhalten bleiben sollen, kann die folgende Schreibweise verwendet werden:

```
user@linux ~/ # programm 2>> error.txt
```

Manchmal sollen alle Ausgaben, Standardausgabe und Standardfehler, in eine Datei umgeleitet werden. Hierzu können wir folgende Schreibweise verwenden:

```
user@linux ~/ # programm > ausgabedatei 2>&1
```

Zunächst wird die Standardausgabe mit einer Datei verbunden. Dann wird der Standarfehlerkanal mit der Standardausgabe verbunden, so dass er ebenfalls in die Datei schreibt.

3.4 Kommandos verbinden

Alles bisherige war Babykram gegen das, was jetzt kommt. Der folgende Mechanismus bringt wie kaum ein anderer die Philosophie von Unix-Betriebssystemen zum Ausdruck. Wie bereits an anderer Stelle erwähnt, ähnelt Unix (also auch Linux) einem Baukasten: Es besteht aus einer Vielzahl kleiner Elemente, die sich wundersam zusammenfügen lassen.

Der zentrale Mechanismus hierbei ist die Umlenkung der Standardausgabe eines Kommandos in die Standardeingabe eines anderen Kommandos. Dieses erreichen Sie durch die Verwendung des "Pipe"-Symbols `|`. Füttern wir `wc` einmal mit der Ausgabe eines `ls`:

```
user@linux ~/ # ls | wc -w

15

user@linux ~/ # ls

_webseiten      deepcalc        gimp            mp3             themes
bewerbung       dokumente       karteikarten    rcs             tutorials
bilder          downloads       linuxartikel    software_liste
verzeichnisse
```

Wie wir sehen, findet `ls` im aktuellen Verzeichnis genau 15 Verzeichnisse. Leiten wir seine Standardausgabe zu `wc` um, benutzt dieses den Input als Standardeingabe und zählt die Worte. Die Namen erscheinen dann nicht auf dem Bildschirm, wie sie es beim darauffolgenden Aufruf ohne Umleitung tun.

Auf diese Weise lassen sich beliebig viele Kommandos zusammenfügen, so dass sich häufig auch sehr komplizierte Aufgabenstellungen durch eine einzige Kommandozeile bewältigen lassen. Sie möchten beispielsweise alle Dateien in einem bestimmten Verzeichnis finden, die auf .html enden, aus diesen diejenigen herausfiltern, welche die Zeichenkette **projekt** enthalten, und sie in ein anderes Verzeichnis, diesmal aber mit der Endung .htm verschieben. Für diese Aufgabe können sie mehrere Kommandos verwenden, die Sie mittels mehrerer Pipes verbinden. Das gesamte Gebilde bezeichnen wir als Pipeline.

4 Der Alias-Mechanismus

Der Alias-Mechanismus dient der Ersparnis von Tipparbeit, macht Kommandos leichter erinnerbar, verschönert Kommandoausgaben und kann auch zur Absicherung gegen Tippfehler verwendet werden. Ein Alias ist eine definierte Zeichenfolge, die für eine andere Zeichenfolge steht. Welche Aliase in Ihrer aktuellen Shell definiert sind, können Sie folgendermaßen feststellen:

```
user@linux ~/ # alias

alias +='pushd .'
alias -='popd'
alias ..='cd ..'
alias ...='cd ../..'
alias ckde1='source /opt/kde2/bin/kde1'
alias ckde2='source /opt/kde2/bin/kde2'
alias dir='ls -l'
alias dos2unix='recode ibmpc:lat1'
alias l='ls -alF'
alias la='ls -la'
alias ll='ls -l'
alias ls='ls $LS_OPTIONS'
alias ls-l='ls -l'
alias md='mkdir -p'
alias o='less'
alias rd='rmdir'
alias rehash='hash -r'
alias rm='rm -i'
alias unix2dos='recode lat1:ibmpc'
alias unzip='unzip -L'
alias which='type -p'
```

Die Ausgabe zeigt die auf meinem System definierten Aliase. Die Syntax ist einfach `alias name=wert`. Da wir bislang eigentlich noch gar keine Kommandos kennengelernt haben (von den wenigen Beispielen abgesehen, die nur zur Demonstration von Shell-Mechanismen dienten), wollen wir nicht im Detail auf die Ausgabe eingehen. Wir möchten lediglich feststellen, dass es Sinn macht, das häufig verwendete `ls -l` durch die simple Eingabe von `ll` aufzurufen. Auch `dos2unix` für die Konvertierung von DOS-Texten in das UNIX-Textformat ist eingängiger als `recode ibmpc:lat1`. Und dass der Alias `rm` auf `rm -i` davor schützt, durch eine Unkonzentriertheit den kompletten Inhalt eines Verzeichnisses (oder mehr) zu löschen, ist ebenfalls keine schlechte Idee. Alles in allem also ein nützlicher Mechanismus, den Sie sinnvoll für eine höhere Effizienz bei der täglichen Arbeit einsetzen können. Wenn Sie einen Alias wieder löschen wollen, können Sie dies einfach mit `unalias name` tun.

5 Kommandosubstitution

Wie wir bereits an anderer Stelle gesehen haben, bearbeitet die Shell eine gegebene Kommandozeile in vielfacher Weise. Zu den Bearbeitungsschritten zählen diverse Substitutionen, wie beispielsweise die Ersetzung von Wildcards durch entsprechende Datei- oder Verzeichnisnamen, die Ersetzung von Aliasen durch ihren Wert oder auch die Ersetzung von Variablen, die wir später noch betrachten werden. Mit Kommandosubstitution bezeichnet man einen Mechanismus, der zur Ersetzung eines Kommandos durch dessen Standardausgabe führt. Das zu ersetzende Kommando wird also aus der Kommandozeile entfernt. Danach wird an seiner Stelle die Standardausgabe des entfernten Kommandos eingefügt. Hier ein Beispiel:

```
user@linux ~/ # echo Im Verzeichnis existieren `ls | wc -w` Einträge
Im Verzeichnis existieren 22 Einträge
```

Das Kommando `echo` gibt einfach auf die Standardausgabe aus, was es als Argumente erhält. Wie zu sehen, ist die Pipeline `ls | wc -w` in den Text eingefügt, umgeben von sogenannten Backticks oder (umständlicher) **linksgeneigten Hochkommata**. Die Backticks bewirken, dass vor dem Aufruf von echo das eingeschlossene Kommando ausgeführt und seine Ausgabe an Stelle des Kommandos eingefügt wird. Hier der Beweis:

```
user@linux ~/ # ls | wc -w
22
```

In der bash können Sie noch eine zweite Schreibweise verwenden:

```
user@linux ~/ # echo Im Verzeichnis existieren $(ls | wc -w) Einträge
Im Verzeichnis existieren 22 Einträge
```

Das Kommando ist in runde Klammern eingefaßt, denen ein Dollarzeichen voransteht. Das Ergebnis ist dasselbe.

6 Vordergrund und Hintergrund: Einführung in die Jobkontrolle

Wenn Sie ein Kommando abgesetzt haben, wartet die Shell normalerweise, bis das Kommando ordnungsgemäß beendet wurde, und gibt dann wieder einen Prompt aus, um auf das nächste Kommando zu warten. Manche Kommandos können jedoch viel Zeit benötigen oder gar während der kompletten Arbeitssitzung laufen. Wenn Sie in einer grafischen Umgebung wie dem X Window-System arbeiten, können Sie von der Shell aus beliebige Programme starten, was häufig viel komfortabler ist, als sich per Maus zum gewünschten Programm zu klicken. Damit Sie nicht für jedes Programm, das Sie starten wollen, eine eigene Shell aufmachen müssen, können Sie Programme, wie man sagt, im Hintergrund starten. Das bedeutet nichts anderes als dass die Shell nicht erst auf die Beendigung des abgesetzten Programmes wartet, sondern sofort wieder einen Prompt ausgibt, um ggf. ein weiteres Kommando zu bearbeiten. Die Ausführung eines Kommandos im Hintergrund erreichen Sie, indem Sie dem Kommando ein "Kaufmannsund" hintenanstellen:

```
user@linux ~/ # kommando &
```

oder auch

```
user@linux ~/ # kommando&
```

Der Sinn der Bezeichnungen Vordergrund und Hintergrund ist unmittelbar eingängig. In technischer Hinsicht sind Vordergrund und Hintergrund zwei Begriffe, die sich im Zusammenhang mit der Shell nur auf ein bestimmtes Terminal beziehen können. Ist die sogenannte Prozeß-Gruppen-ID eines Prozesses identisch mit der eines Terminals, so kann der Prozeß von diesem Terminal Signale empfangen. Solche Prozesse laufen im Vordergrund. Was eine Prozeß-Gruppen-Id ist, werden wir später noch ausführlich behandeln. Hintergrund-Prozesse sind solche, deren Prozeß-Gruppen-Id von der des Terminals verschieden sind. Sie sind daher auch immun gegen irgendwelche Signale, die vom Keyboard herrühren.

Der Begriff des Jobs ist eine Abstraktion, welche von der Shell zur Verwaltung eingesetzt wird. Als Job wird jede Pipeline bezeichnet, aus wievielen Kommandos oder Prozessen auch immer sie bestehen mag. Dem Job wird von der bash eine Jobnummer zugewiesen, unter welcher er angesprochen werden kann. Die komplette Liste der in einer Shell laufenden Jobs können Sie sich mit dem Kommando `jobs` anzeigen lassen:

```
user@linux ~/ # kommando1 &

[1] 5520

user@linux ~/ # kommando2 &

[2] 5521

user@linux ~/ # kommando3 &

[3] 5522

user@linux ~/ # jobs

[1] Running  kommando1
[2]-  Running kommando2 &
[3]+  Running kommando3 &
```

In den eckigen Klammern erkennen Sie die zugeteilte Jobnummer. Sie unterscheidet sich von der sogenannten Prozeßnummer, die Sie hinter der Jobnummer angegeben sehen. Das +-Zeichen bei der Ausgabe des jobs-Kommandos markiert den zuletzt gestarteten Job, das --Zeichen den als vorletztes gestarteten Job.

Es gibt eine Reihe von Möglichkeiten, sich auf einen bestimmten Job zu beziehen. Das Zeichen `%` leitet einen Jobnamen ein. Jobnummer n kann als `%n` angesprochen werden. Man kann sich auch auf einen Job beziehen, indem man dem `%` die ersten Buchstaben des Kommandos voranstellt, mit dem man den Job gestartet hat. Hat man z.B. `Kommando` gestartet, kann man sich darauf mittels `%ko` beziehen, falls kein weiterer laufender Job so beginnt. Auch eine Art von Wildcard ist erlaubt: `%?ommando` oder auch `%?mmando` bezieht sich ebenfalls auf den Job, der mittels `kommando` gestartet wurde. Wenn das angegebene Präfix oder Muster auf mehr als einen Job paßt, erfolgt eine Fehlermeldung. `%%` oder `%+` bezieht sich immer auf den letzten Job. In den Begriffen der Shell ist das der zuletzt gestoppte Vordergrundprozeß oder der zuletzt gestartete Hintergrundprozeß. `%-` bezieht sich entsprechend auf den zuvorletzt gestarteten Job.

7 fg, bg und Strg-Z

Selbstverständlich möchten Sie jederzeit bestimmen können, ob ein Kommando im Vordergrund oder im Hintergrund läuft. Sie möchten es aus dem Hintergrund wieder hervorholen oder aus dem Vordergrund in den Hintergrund schicken können, auch während es bereits läuft. Hierfür können Sie die beiden Kommandos `fg` ("foreground") und `bg` ("background") verwenden. Sie können sich dabei in der oben beschriebenen Weise auf einen beliebigen Job beziehen, z.B.:

```
user@linux ~/ # kommando1 &
[1] 5520
user@linux ~/ # kommando2 &
[2] 5521
user@linux ~/ # fg %1
kommando2
```

`kommando2` läuft jetzt wieder im Vordergrund. Wenn Sie es wieder in den Hintergrund schicken wollen, können Sie dies nicht unmittelbar mit `bg` tun, denn derzeit steht Ihnen ja gar kein Prompt zur Kommandoeingabe zur Verfügung. Sie müssen den Job daher erst mit der Tastenkombination `CTRL + Z` anhalten:

```
user@linux ~/ # CTRL + Z
[2]+  Stopped kommando2
user@linux ~/ # bg
[2]+ kommando2 &
```

Aha. Wir haben das Kommando zunächst erfolgreich gestoppt und es dann mittels `bg` in den Hintergrund geschickt. Von dort können wir es freilich jederzeit wieder mittels `fg %2` hervorholen.