Einführung Unix/Linux

Kapitel: Arbeiten in der Shell

Jens Roesen <jens@roesen.org>

Würzburg, März 2010
Version: 0.1.5 – **sehr beta** –

Dieses Dokument wurde in vim (`http://www.vim.org`) bzw. TEXnicCenter (`http://www.texniccenter.org/`) geschrieben und mit LaTeX (`http://www.latex-project.org/`) formatiert und gesetzt.
Die jeweils aktuelle Version ist unter `http://www.roesen.org` erhältlich.

Inhaltsverzeichnis

Vorwort

Motivation

Im Rahmen interner Schulungsmassnahmen kam die Frage nach geeigneten Schukungsmaterialien bzw. einem Skript für Unix-Neulinge auf. Schulungsunterlagen und Skripte für Einsteiger, aber letztendlich hat mir bei den meisten entweder etwas gefehlt, oder es war für unseren Zweck viel zu viel irrelevanter Stoff. Da wir in der Hauptsache mit Solaris und Linux-Systemen arbeiten und dabei Themen wie X-Windows oder Druckerverwaltung komplett ausklammern können, aber auf Themen wie Netzwerke, Troubleshooting, Mailserver oder DNS-Server Wert legen, habe ich mich irgendwann hingesetzt und angefangen dieses Skript zu schreiben.

Es ist der Versuch Systemadministratoren mit Grundkenntnissen in Unix den Arbeitsalltag zu erleichtern und dabei zwei verschiedene Unix-Geschmacksrichtungen, nämlich Solaris[1] und Red Hat Enterprise Linux, gleichermassen zu betrachten.

Mir ist durchaus klar, dass nicht alles im Folgenden beschriebene „state of the art“ ist bzw. sein kann und sicher auch noch etliche Fehler übersehen wurden. Wer einen solchen findet, weiss wie man einige Aufgaben besser lösen kann oder bessere Beispiele kennt ist herzlich eingeladen mir eine Mail an <jens@roesen.org> zu schicken.

Zielgruppe

Dieses Kurzscript ist als Crashkurs zur Einführung in die Administration von Unix und Linux Systemen gedacht. Es wird dabei ausschliesslich auf der Konsole und ohne grafische Oberfläche gearbeitet. Die gezeigten Beispiele beziehen sich auf Systeme unter Sun Solaris und RedHat Enterprise Linux.

Ohne Vorkenntnisse und Erfahrung mit Unix und/oder Linux Systemen wird der angesprochene Stoff teils nur schwer zu verstehen sein. Als alleiniges Lehrskript für blutige Anfänger ist es daher nicht geeignet obwohl in einigen Kapiteln vereinzelt kurz auf Grundlagen eingegangen wird (z.B. Kapitel 2).

Aufbau des Skripts

Wirr. Durch und durch. Aber zu mehr ist momentan keine Zeit. Ich habe versucht die Kapitel und Themen in eine halbwegs sinnvolle Reihenfolge zu bringen. Im Lauf der Zeit wird da sicherlich noch einiges umgestellt werden.

[1] In der vorliegenden Version des Skripts nur bis Version 9.

Typographisches

Da sich alle Beispiele, Kommandos und Ausgaben auf der Konsole abspielen, werden diese Bereiche entsprechend formatiert um sich vom regulären Text abzusetzen. Nach dem Login als User root, mit dem wir in diesem Skript hauptsächlich arbeiten werden, landet man in einem Command Prompt der so aussehen koennte:

```
[root@server1 root]#
```

Da dieser Prompt ja nach name des Systems oder aktuellem Verzeichnis mal kürzer aber auch sehr viel länger sein kann, wird der root Prompt auf

```
#
```

verkuerzt. Bitte den Hash (`#`) hier **nicht** als Kommentarcharakter verstehen, der unter Linux/Unix z.B. in Shellskripten die folgende Zeile vor der Ausführung durch die Shell schützt. Der normale User-Prompt, falls er uns wirklich einmal begegnen sollte, wird analog dazu auf

```
$
```

zusammengestrichen.

Für Konsolenausgaben, Konfigurationsdateien oder Teile von Skripten wird eine nichtproportionale Schrift verwendet:

```
if [ -n "$_INIT_NET_IF" -a "$_INIT_NET_STRATEGY" = "dhcp" ]; then
        /sbin/dhcpagent -a
fi
```

Werden in einem Beispiel Konsoleneingaben vom Benutzer erwartet, wird die in einer nichtproportionale Schrift dargestellt, wobei die Benutzereingaben fett gedruckt sind:

```
# uname -a
SunOS zoidberg 5.9 Generic_118558-21 sun4u sparc SUNW,Sun-Blade-100
```

Kommandos, Dateinamen oder Benutzerkennungen im laufenden Text werden ebenfalls in einer nichtproportionalen Schrift dargestellt: „Mit dem Befehl `pwd` kann überprüft werden, in welchem Verzeichnis man sich gerade befindet.“.

Müssen in einem Beispiel noch Teile der erwarteten Benutzereingaben durch die richtigen Werte ersetzt werden, so wird dieser Teil in kursiver Nichproportionalschrift dargestellt: „Für jedes Interface welches beim boot konfiguriert werden soll muß eine Datei **/etc/hostname.*interface*** existieren.“.

Eigennamen, Personen oder Organisationen erscheinen manchmal (ich bin gerade zu faul alle Vorkommen entsprechend zu formatieren) in Kapitälchen: „Eine sehr große Rolle hierbei hat die University of California in Berkley (UCB) gespielt, an der Thompson im Winter 76/77 eine Vorlesung zum Thema Unix abhielt.“.

1 Arbeiten in der Shell

> We have persistent objects,
> they're called files.
>
> *(Ken Thompson)*

1.1 Die Shell als Schnittstelle zum Kernel

Wie schon in Kapitel 2 erwähnt wurde, umgibt die Shell den Kernel und stellt somit die Schnittstelle zwischen User und Betriebssystem dar. Einfach gesprochen ist die Shell ein simples Programm welches die Befehle des Users entgegennimmt und für den Kernel übersetzt. Dabei bietet sie dem Benutzer mittels diverser grundlegender Programmierwerkzeuge (Schleifen, Variablen, bedingte Verzweigungen usw.) sogar die Möglichkeit selber kleinere „Programme“, sogenannte Shellscripte, zu schreiben. Dazu jedoch später mehr. Welche Shell nach einem erfolgreichen Login gestartet wird, wird für jeden User in der Datei `/etc/passwd` definiert. Sollte einem diese Shell nicht gefallen, kann man entweder den Systemadministrator bitten eine andere Loginshell einzutragen oder selber nach dem Login die gewünschte Shell selber starten.

1.2 Die verschiedenen Shells

1.2.1 Bourne-Shell - Das Urgestein

Die Bourne-Shell (`/bin/sh`) ist die älteste der heute noch gebräuchlichen Shells. Sie wurde von Steven Bourne bei AT&T entwickelt und erstmals mit Unix Version 7 in 1979 ausgeliefert. Für den User `root` ist sie die Standardloginshell (`/bin/sh` bzw. `/sbin/sh`). Normale User verwenden sie jedoch selten, da die Bourne-Shell im Vergleich zu den anderen Shells relativ unkomfortabel ist. Aufgrund ihrer geringen Systembelastung und großen Verfügbarkeit ist sie für die Entwicklung von Shellscripten immer noch erste Wahl. Der Defaultprompt für normale User ist das Dollarzeichen (`$`).

1.2.2 Bash

Die wohl bekannteste und am weiten verbreitetste Shell ist die BASH — Bourne Again Shell (`/bin/bash`). Sie ist im Rahmen des GNU Projekts[1] ursprünglich von Brian Fox

[1] GNU ist ein rekursives Akronym für „GNU's not UNIX“. Das GNU Projekt hat es sich seit 1984 zur Aufgabe gemacht ein freies Unix-artiges Betriebssystem zu entwickeln. Urheber dieses Projekts war Richard Stallman von der Free Software Foundation (FSF). `http://www.gnu.org`

entwickelt und am 10. Januar 1988 offiziell vorgestellt worden. Die BASH übernahm ihrerseits die besten Features der bisher bekannten Shells und fügte ihr noch einige weitere hinzu. Dabei wurde viel Wert auf eine möglichst gute Konfigurierbarkeit der Arbeitsumgebung gelegt. Bereits vorhandenen Bourne-Shell Shell-Scripte können natürlich ebenfalls weiter verwendet werden. Sie gehört allerdings immer noch nicht zur Standardinstallation eines jeden Unix.

1.2.3 C-Shell - für den C-Programmierer unter uns

Billy Joy entwickelte in Berkley die in ihrer Syntax der Programmiersprache C sehr ähnliche C-Shell (`/bin/csh`). Sie bietet gegenüber der Bourne-Shell mehr Komfort durch z.B. eine command-line history, job control und der Möglichkeit Aliase zu vergeben. Der normale Prompt ist der Hostname gefolgt von einem Prozentzeichen (`hostname%`). Mittlerweile wird sie meiner Erfahrung nach jedoch eher selten eingesetzt — jedenfalls kenne ich keinen überzeugten C-Shell User.

1.2.4 Korn-Shell

Die Korn-Shell (`/bin/ksh`) haben wir Herrn David Korn von den AT&T Bell Laboratories zu verdanken. In ihr wurden etliche Neuerungen der C-Shell übernommen und um Features wie command-line editing erweitert. Dabei ist sie weiterhin voll kompatibel zur Bourne-Shell, so daß bereits vorhandene Shell-Scripte auch weiterhin verwendet werden können. Die Korn-Shell ist auf vielen Systemen noch die Standardshell für normale User.

1.3 Metazeichen

Beim Arbeiten mit der Shell muss man die Bedeutung diverser Sonderzeichen, der Metazeichen, kennen. Metazeichen stehen bei der Verarbeitung der Eingaben nicht für sich selber sondern werden von der Shell entsprechend ersetzt.

Das bekannteste Metazeichen ist wohl das Sternchen „`*`“, oder auch Asterisk genannt. Das `*` wird von der Shell zu jedem erdenklichen und passenden Zeicher ersetzt. Das schliesst auch „kein Zeichen“ ein. Besonders oft wird das bei der Dateinamenerweiterung, dem sogenannten „Globbing“ benutzt. Das „`?`“ hingegen wird nur zu einem einzigen Zeichen erweitert.

In Tabelle 1.4 auf Seite 10 werden einige der wichtigsten Metazeichen zusammengefasst und erklärt.

1.4 Ein- und Ausgabeumlenkung in der Shell

Auf der Konsole ist es möglich die Ausgaben eines Programms bzw. die Eingaben die es erwartet zu beeinflussen. Dazu dient das Ein-/Ausgabekonzept von Unix, welches drei vordefinierte Datenkanäle besitzt: `stdin(0)`, `stdout(1)` und `stderr(2)`. Wird nichts

anderes angegeben erfolgen die Ausgaben standardmässig auf dem Bildschirm (standard out/output - `stdout`) und die benötigten Eingaben werden von der Tastatur gelesen (standard in/input - `stdin`). So gesehen verhalten sich viele Programme wie ein Filter: sie lesen Daten aus einer Eingabedatei ein, bearbeiten sie und schreiben sie in eine Ausgabedatei.

`ls` gibt den Verzeichnisinhalt normalerweise direkt auf standard out aus. Die Shell bietet uns jedoch die Möglichkeit die Ausgabe z.B. in eine Datei oder auf einen Drucker umzuleiten. Dies erreichen wir durch das '>'. Wenn ein > und ein Dateiname einem Befehl angehangen wird, werden alle Ausgaben statt auf standard out `stdout(1)` in diese Datei geschrieben.

```
# ls -l verzeichnisinhalt.txt
verzeichnisinhalt.txt: No such file or directory
# ls > verzeichnisinhalt.txt
# ls verzeichnisinhalt.txt
-rw-r--r--  1 root  root  3598 Dec 11 10:42 verzeichnisinhalt.txt
#
```

Die Datei `verzeichnisinhalt.txt` hat noch nicht existiert. Durch das Umlenken der Ausgabe von `ls` wurde sie angelegt und mit den Ausgaben von **ls -l** gefüllt.
Existiert die Datei noch nicht wird sie angelegt. Existiert die angegebene Datei bereits wird sie überschrieben!

```
# echo "Eine Zeile Text" > zeilentext.txt
# cat zeilentext.txt
Eine Zeile Text
# echo "Eine neue Zeile Text ueberschreibt die alte" > zeilentext.txt
# cat zeilentext.txt
Eine neu Zeile Text ueberschreibt die alte
#
```

Der Befehl `echo` gibt einfach den Text aus den man ihm als Argument übergibt. Im ersten Schritt wurde durch die Umlenkung der Ausgabe *Eine Zeile Text* die Datei `zeilentext.txt` erzeugt. Im zweiten Schritt wurde durch das erneute Umlenken nach `zeilentext.txt` der Inhalt überschrieben. Will man das Überschreiben verhindern, muss man mit '>>' umlenken. Dann werden die umgelenkten Ausgaben ans Ende der Datei angehangen.

Ebensoleicht kann man auch standard in `stdin(0)` umleiten bzw. sich die erwarteten Eingaben die sonst über die Tastatur erfolgen würden aus einer Datei holen:

```
# mail drizzt@netzsheriff.de < zeilentext.txt
```

So maile ich mir den Inhalt der Datei `zeilentext.txt` aus dem Beispiel von oben zu.

All diese Umlenkungen lassen sich auch kombinieren. Besonders häufig benutzt man dies z.B. bei Cronjobs:

```
1 2 * * * [ -x /usr/sbin/rtc ] && /usr/sbin/rtc -c > /dev/null 2>&1
```

Cronjobs sehen wir uns später noch genauer an. An der Stelle reicht es erstmal wenn ich frech behaupte durch diesen Eintrag würde täglich um 02:01 Uhr geschaut ob die Datei `/usr/sbin/rtc` ausführbar ist. Falls dies zutrifft wird sie mit der Option `-c` gestartet. Uns interessiert dann nur noch der Ausdruck `> /dev/null 2>&1`. Was `> /dev/null` bewirkt sollte klar sein: alle möglichen Ausgaben von `rtc` werden in das Nulldevice `/dev/null` umgeleitet. Einfach gesagt: auf den Müll damit. Mit `2>&1` zusammen wird es eine sogenannte gekoppelte Umleitung. Ein `2>` mit eintsprechender Zieldatei dahinter würde standard error `stderr(2)` umleiten:

```
# ls verzeichnisinhalt.txt
verzeichnisinhalt.txt: No such file or directory
# ls verzeichnisinhalt.txt 2> ls-error.txt
# cat ls-error.txt
verzeichnisinhalt.txt: No such file or directory
#
```

Mit `2>&1` leite ich `stderr` genauso um wie `stdout` zuvor: in unserem Cronjob-Beispiel von oben also nach `/dev/null`.

Getrennte Umlenkungen sind ebenso möglich:

```
# cat zeilentext.txt nichtvorhanden.txt
Eine neu Zeile Text ueberschreibt die alte Zeile
cat: cannot open nichtvorhanden.txt
# cat zeilentext.txt nichtvorhanden.txt > stdout.txt 2> stderr.txt
# cat stdout.txt
Eine neu Zeile Text ueberschreibt die alte
# cat stderr.txt
cat: cannot open nichtvorhanden.txt
#
```

Tabelle 1.1 auf Seite 5 fasst die Möglichkeiten noch einmal zusammen.

1.5 Pipes

Das Konzept der Pipelines unter Unix ermöglicht es, die Ausgaben (`stdout`) eines Prozesses direkt als erwartete Eingabe (`stdin`) eines anderen Prozesses zu nutzen. Dabei wird der vertikale Strich „`|`“, auch „Pipe Zeichen“ oder schlicht „Pipe“ genannt, als Signal für das Erstellen einer Pipe zwischen zwei Prozessen genutzt. Das einfachste Beispiel und der wohl am häufigsten auftretende Fall ist, die Ausgabe eines Prozesses an `less` zu übergeben:

```
# ps aux | less
```

Damit wird die unter Umständen mehrere Bildschirmseiten lange Prozessliste an `less` übergeben und so seitenweise dargestellt.

Ein anderes Beispiel aus dem Admin-Alltag:

```
# mailq | grep MAILER-DAEMON | awk '{print $1}' | xargs postsuper -d
```

Ein-/Ausgabeumleitung in der Shell	
Umlenken von...	**Bourne/Korn/Bourne-Again-Shell (Bash)**
`stdin`	`< file`
`stdout`	`> file`
`stderr`	`2> file`
`stdout` und `stderr` in selbe Datei	`> file 2>&1` (Reihenfolge beachten)
getrennte Umlenkung von `stdout` und `stderr` in verschiedene Dateien	`> file 2> file2`
Ausgabe an bestehende Datei anhängen	`>> file`

Tabelle 1.1: Ein-/Ausgabe Umlenkungsmöglichkeiten

Hier wird die aktuelle Mailqueue eines Mailservers mit `mailq` abgefragt. In der `mailq` Ausgabe wird mit `grep` nach allen Zeilen mit `MAILER-DAEMON` gesucht und diese werden an `awk` übergeben. `awk` gibt schliesslich nur das erste Feld dieser Zeilen, die Postfix Queue-ID der Mails, aus und diese werden mittels `xargs` an `postsuper -d` übergeben und so aus der Queue gelöscht.

1.6 Verknüpfen von Kommandos

In der Shell können mehrere Kommandos miteinander verknüpft werden um diese z.B. stur hintereinander ablaufen zu lassen oder aber auch die Verarbeitung der einzelnen Kommandos an Bedingungen knüpfen. Der einfachste Weg zwei Kommandos miteinander zu verknüpfen geht über ein einfaches `;`.

```
# echo "echo 1"; echo "gefolgt von echo 2"
echo 1
gefolgt von echo 2
#
```

Hier werden die beiden `echo`s einfach hintereinander weg ausgeführt, egal was kommt bzw. egal ob das erste `echo` erfolgreich oder mit einem Fehler beendet wurde.

In manchen Fällen ist es aber sinnvoll das nächste Kommando nur dann auszuführen, falls das vorhergehende erfolgreich (bzw. eben nicht erfolgreich) war. Ob ein Kommando erfolgreich war bzw. ohne Fehler abgeschlossen wurde wird in einer Variablen[2], dem

[2]Mit Variablen beschäftigen wir uns später - so dieses Script je fertiggestellt wird - noch genauer. Fuer den Moment reicht es aus sich eine Variable als eine Art Speicher für Informationen vorzustellen und zu wissen das Variablen mit einem `$` beginnen. In Variablen sind u.a. wichtige Werte für die Shell wie der Path/Pfad (`$PATH`) oder das aktuelle Verzeichnis (`$PWD`) gespeichert. Man kann den Wert/Inhalt einer Variablen mit `echo $<`*`variablenname`*`>` abfragen.

sogenannten Rückgabewert `$?`, gespeichert. Wenn `$?` den Wert 0 hat wurde das letzte Kommando ohne Fehler ausgefuehrt. Ist der Wert von `$?` ungleich 0 trat ein Fehler auf.

Im folgenden Beispiel war das `echo` erfolgreich und das `cat` nicht:

```
# echo Test
Test
# echo $?
0
# cat nichtexistentedatei.txt
cat: cannot open nichtexistentedatei.txt
# echo $?
2
#
```

Um zu erreichen das ein Kommando nur dann ausgeführt wird falls das vorausgehende erfolgreich war, verknüpfe ich beide mittels `&&`. Soll das zweite Kommando nur ausgeführt werden falls das erste fehlschlägt verknüpft man beide mittels `||`. Hier ein paar Beispiele:

```
# echo Test && echo Test bestanden
Test
Test bestanden
#

# cat nichtexistentedatei.txt || echo ERROR
cat: cannot open nichtexistentedatei.txt
ERROR
e nich#

# cat nichtexistentedatei.txt && echo ERFOLG
cat: cannot open nichtexistentedatei.txt
#
```

So einfach ist das ;) In Tabelle 1.2 auf Seite 7 findet sich nochmal eine Übersicht zum Thema.

1.7 Quoting und Escapen

Im Abschnitt 1.3 haben wir schon einige Sonder- oder Metazeichen kennengelernt. Die Shell ersetzt diese Zeichen automatisch. Nun kommen aber im täglichen Betrieb durchaus Situationen vor, in denen wir nicht wollen das diese Zeichen ersetzt werden. In einem solchen Fall müssen wir diese Sonderzeichen vor der Shell schützen indem wir sie quoten oder escapen.

Quoten kommt vom englischen to quote, etwas/jemanden Zitieren bzw. etwas in Anführungszeichen setzen, und genau das wird auch gemacht. Man kann mit doppelten (`"..."`) oder einfachen Anführungszeichen (`'...'`) sowie mit dem Backslash (\) quoten. Je nach Methode ändert sich auch das Ergebnis.

Der Backslash (\) schützt nur das unmittelbar folgende Zeichen. Abgesehn vom Zeilenumbruch newline werden alle Zeichen geschützt.

Kommandoverknüpfungen in der Shell	
Verknüpfung mit...	**Auswirkung**
`;`	– das `;` bewirkt, daß die einzelnen dadurch getrennten Kommandos nacheinander ausgeführt werden – Der Rückgabewert (`$?`) des gesamten Kommandos entspricht dem Rückgabewert des zuletzt ausgeführten
`&`	– durch das Anhängen eines `&` an ein Kommando wird bewirkt, daß dieses Kommando im Hintergrund ausgeführt wird – die Shell meldet sich nach Absetzen des Kommandos sofort wieder mit einem Prompt, bereit weitere Kommandos entgegenzunehmen und auszuführen, während das mit `&` gestartete Kommando noch läuft – `stdout` und `stderr` eines mit `&` gestarteten Kommandos werden – falls nicht umgeleitet – ganz normal ausgegeben
`k1 || k2`	– `k2` wird nur dann ausgeführt wenn der Rückgabewert `$?` von `k1` ungleich `0` ist – `k1` also nicht fehlerfrei abgelaufen ist – Der Rückgabewert (`$?`) des gesamten Kommandos entspricht dem Rückgabewert des zuletzt ausgeführten
`k1 && k2`	– `k2` wird nur dann ausgeführt wenn der Rückgabewert `$?` von `k1` gleich `0` ist – `k1` also fehlerfrei abgelaufen ist – Der Rückgabewert (`$?`) des gesamten Kommandos entspricht dem Rückgabewert des zuletzt ausgeführten

Tabelle 1.2: Kommandoverknüpfungen

Mit den doppelten Anführungszeichen (`"..."`) (auch „weak quotes“ genannt) schützt man bis zum nächsten doppelten Anfürungszeichen alle Sonderzeichen abgesehen von doppelten Anführungszeichen, dem Backslash, dem Dollarzeichen und den „backticks“ (`‘...‘`).

Die einfachen Anführungszeichen (`'...'`) (auch „strong quotes“) schützen bis zum nächsten einfachen Anführungszeichen alles ausser einfachen Anführungszeichen.

Will man auf die filename generation, das Ersetzen von `*` zu Dateinamen, ganz verzichten, kann man es mit `set -f` abschalten und bei Bedarf mit `set +f` wieder aktivieren.

1.8 Umgebungsvariablen

Wie bei allen anderen Betriebssystemen werden auch unter Unix viele Eigenschaften der Arbeitsumgebung durch Variablen, den Umgebungsvariablen, festgelegt. Sie legen

Quoting in der Shell	
\	– das direkt rechts vom Backslash stehende Zeichen verliert seine besondere Bedeutung – Bsp.: `echo \*` oder `echo \$LOGNAME`
`'...'`	– Text innerhalb dieser einfachen Anführungszeichen wird von der Shell nicht verändert, **alle** Sonder-/Metazeichen verlieren ihre besondere Bedeutung – Variablen werden **nicht** durch ihren Wert ersetzt
`"..."`	– alle Sonderzeichen **außer** `$` verlieren ihre besondere Bedeutung – Variablen innerhalb von `"..."` werden durch ihren Wert ersetzt

Tabelle 1.3: Quoting in der Shell

fest in welchen Verzeichnissen nach ausführbaren Dateien gesucht werden soll, welche Terminalvariante zum Einsatz kommt oder welcher Texteditor genutzt wird.

Welche Umgebungsvariablen aktuell gesetzt sind, kann man sich mit `env` oder `set`anzeigen lassen bzw. sie gezielt mit `echo $<variablenname>` abfragen.

```
# env
TERM=vt100
SHELL=/bin/bash
USER=root
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games
PWD=/root
HOME=/root
LOGNAME=drizzt
_=/usr/bin/env

# echo $SHELL
/bin/bash
```

Umgebungsvariablen werden bei Start einer Shell automatisch durch das einlesen und ausführen einiger Scripte gesetzt. Da wir hauptsächlich mit der Bash arbeiten beschränke ich mich auf die Dateien die für die Bash relevant sind. Bei anderen Shells läuft das alles jedoch in einem ähnlichen Schema ab.

In jedem Homedir sollte es eine Datei `.bash_profile` geben („sollte“, nicht „muss“). Diese wird bei jedem Start der Bash eingelesen und die darin enthaltenen Anweisungen werden ausgeführt. Hier einmal die `.bash_profile` von einem RHEL3:

```
# .bash_profile

# Get the aliases and functions
if [ -f ~/.bashrc ]; then
        . ~/.bashrc
fi
```

```
# User specific environment and startup programs

PATH=$PATH:$HOME/bin
BASH_ENV=$HOME/.bashrc
USERNAME="root"

export USERNAME BASH_ENV PATH
```

In diesem Beispiel werden hauptsächlich zwei Funktionen ausgeführt. Im oberen Teil wird mit einer einfachen `if`-Abfrage geprüft ob eine Datei `.bashrc` im Homedir existiert. Sollte dies der Fall sein, so wird diese Datei „gesourced“. Sie wird eingelesen und ausgeführt, genauso wie die `.bash_profile` grade eingelesen wird. Im unteren Teil werden dann einige Umgebungsvariablen angepasst und anschliessend durch `export` gesetzt.

Oft gibt es Dateien wie `/etc/bashrc` oder `/etc/profile` über die systemweite Umgebungsvariablen gesetzt werden. Im Beispiel oben wurde getestet ob die Datei `/.bashrc` existiert. In dieser wiederum gab es eine Anweisung, über die eine eventuell vorhandene `/etc/bashrc` eingelesen wird.

Alle Umgebungsvariablen lassen sich natürlich auch per Hand setzen. In der sh setzt man sie mit `<variablenname>=<variablenwert>` und macht sie danach mit `export <variablenname>` bekannt bzw. kombiniert man das in der bash gleich zu `export <variablenname>=<variablenwert>`. Unter csh oder tcsh reicht zum setzen und bekanntmachen `setenv <variablenname> <variableninhalt>`.

Metazeichen der Shell	
*	– der Asterisk – wird zu **keinem bzw. einem oder mehreren** beliebigen Zeichen für Dateinamen, ausgenommen Dateien mit einem „.“ am Beginn – versteckte Dateien werden also von diesem Metazeichen nicht erfaßt
?	– wird zu **einem** beliebigen Zeichen eines Dateinamens expandiert
[s]	– wird mit **einem** Zeichen aus der Zeichenkette `s` entsprechend den verfügbaren Dateinamen ersetzt Bsp.: `t[mu]p` würde auf die Dateien `tmp` und `tup` zutreffen – sofern sie existieren
[!s]	– wird durch ein Zeichen ersetzt, welches nicht in `s` enthalten ist
[c1-cn]	– wird durch genau ein Zeichen aus den Zeichen `c1-cn` ersetzt – Bsp.: `[a-z]` wäre ein Kleinbuchstabe von a bis z
{s1,s2,...}	– das Ergebnis besteht aus Zeichenketten in denen die in den geschweiften Klammern und durch Komma getrennten Zeichenketten enthalten sind – Bsp.: bei `ls h{an,un}d` würde der Ausdruck zu `ls hand hund` ersetzt werden sofern diese Dateien vorhanden sind – *nicht in der Bourne- und Korn-Shell verfügbar*
$	– das Zeichen beschreibt den Beginn einer Variablen deren Wert an dieser Stelle eingesetzt werden soll – Bsp.: `echo $HOME`
‘kommando‘	– man nennt sie Backticks – der Befehl innerhalb der Backticks wird ausgeführt und der Ausdruck ‘`kommando`‘ wird durch die Ausgabe von `kommando` ersetzt – Bsp.: `echo ‘pwd‘`
SPACE, TAB	– `SPACE` (Leertaste) und `TAB` (Tabulatortaste) werden in der Shell als Trennzeichen (Whitespaces) benutzt – mehrere Trennzeichen hintereinander werden von der Shell zu einem verkürzt
$(kommando)	– `kommando` wird in einer Sub-Shell ausgeführt – die neu gestartete Sub-Shell wird direkt nach Beendigung von `kommando` wieder geschlossen – da `kommando` in einer extra Shell gestartet wird, wirken sich mögliche Änderungen an Shell- und Umgebungsvariablen durch `kommando` nur auf diese neu gestartete Shell aus

Tabelle 1.4: Metazeichen in der Shell