Linux-Praxisbuch: Shellprogrammierung

Zurück zum Linux-Praxisbuch

1 Einleitung

Eine Shell ist ein Kommandozeileninterpreter zur Kommunikation mit einem Unix-System. Loggt man sich in ein Unix-System ein, landet man im allgemeinen in einem laufenden Shell-Programm, das darauf wartet, dass man Kommandos eingibt. Sobald man das Kommando abschließt, das heißt, nach Eingeben von Text die RETURN-Taste drückt, nimmt die Shell verschiedene Ersetzungen bei diesem Text vor und führt den endgültigen Text dann aus.

1. Beispiele für Unix-Befehle
2. Grundsätzliches
3. Aufbau eines Shell-Skriptes
4. Zeichenketten- und weitere Funktionen

2 Grundlagen

2.1 Variablen

In einem Shell-Skript hat man – genau wie bei der interaktiven Nutzung der Shell – verschiedene Möglichkeiten, Variablen einzusetzen. Anders als in vielen Programmiersprachen sind die Shellvariablen nicht an einen Datentyp wie Ganzzahl, Fließkommazahl oder Text-String gebunden. Man kann einer Variablen Text zuweisen, und im nächsten Schritt eine Zahl, mit der man rechnet.

a=hello # einer Variablen “a” den Wert “hello” zuweisen echo $a # die Variable “a” (am Bildschirm) ausgeben a=7 # der Variablen “a” einen neuen Wert zuweisen b=8 # einer Variablen “b” den Wert “8” zuweisen c=$((a+b)) # einer Variablen “c” die Summe von a+b zuweisen ("$((a+b))" wird substituiert) echo $c # die Variable “c” (am Bildschirm) ausgeben

Man muss bei der Benutzung von Variablen sehr aufpassen, wann die Variable expandiert wird und wann nicht. (Mit Expansion ist das Ersetzen des Variablennamens durch den Inhalt gemeint). Grundsätzlich werden Variablen während der Ausführung des Skriptes immer an den Stellen ersetzt, an denen sie stehen. Das passiert in jeder Zeile, unmittelbar bevor sie ausgeführt wird. Es ist also auch möglich, in einer Variable einen Shell-Befehl abzulegen. Im Folgenden kann dann der Variablenname an der Stelle des Befehls stehen. Um die Expansion einer Variable zu verhindern, benutzt man das Quoting (siehe weiter unten).

Wie aus diversen Beispielen hervorgeht, belegt man eine Variable, indem man dem Namen mit dem Gleichheitszeichen einen Wert zuweist. Dabei steht die Variable immer links, der Wert rechts: (a=5), dabei darf zwischen dem Namen und dem Gleichheitszeichen keine Leerstelle stehen.

Wenn man auf den Inhalt einer Variablen zugreifen möchte, leitet man den Variablennamen durch ein $-Zeichen ein. Vieles, was mit einem $ anfängt, wird von der Shell als Variable angesehen und dessen zuvor zugewiesener Wert verwendet. $a wird somit als “Der Wert von a” gelesen - passend zum Dollar-Zeichen.

Weitere Infos: http://www.netzmafia.de/skripten/unix/unix8.html

2.2 Vordefinierte Variablen

Es gibt eine Reihe vordefinierter Variablen, deren Benutzung ein wesentlicher Bestandteil des Shell-Programmierens ist. Einige betreffen die Parameter eines Shellskripts welche beim Aufruf des Skripts hinter dem Kommandonamen anzugeben sind. Die wichtigsten eingebauten Shell-Variablen sind:

n Aufrufparameter mit der Nummer n, 1 <= n <= 9 * Alle Aufrufparameter als ein String ("$*" == "$1 $2 $3 ...") @ Alle Aufrufparameter als einzelne Strings ("$@" == "$1” "$2” "$3” ...) # Anzahl der Aufrufparameter ? Rückgabewert des letzten Kommandos $ Prozessnummer der aktiven Shell ! Prozessnummer des letzten Hintergrundprozesses ERRNO Fehlernummer des letzten fehlgeschlagenen Systemaufrufs PWD Aktuelles Verzeichnis (wird durch cd gesetzt) OLDPWD Vorheriges Verzeichnis (wird durch cd gesetzt)

2.3 Variablen-Substitution

Unter Variablen-Substitution versteht man verschiedene Methoden, um die Inhalte von Variablen zu benutzen. Das umfasst sowohl die einfache Zuweisung eines Wertes an eine Variable als auch einfache Möglichkeiten zur Fallunterscheidung. In den fortgeschritteneren

Shell-Versionen (bash, ksh) existieren sogar Möglichkeiten, auf Substrings von Variableninhalten zuzugreifen. In der Standard-Shell sh benutzt man für solche Zwecke üblicherweise den Stream-Editor sed. Einleitende Informationen dazu finden sich im Kapitel über die Mustererkennung).

Die folgenden Mechanismen stehen in der Standard-Shell bereit, um mit Variablen zu hantieren. Bei allen Angaben ist der Doppelpunkt optional. Wenn er aber angegeben wird, muss die Variable einen Wert enthalten.

- Variable=Wert Setzt die Variable auf den Wert.
- ${Variable} Nutzt den Wert von Variable. Die Klammern müssen nicht mit angegeben werden, wenn die Variable von *Trennzeichen umgeben ist.
- ${Variable:-Wert} Nutzt den Wert von Variable. Falls die Variable nicht gesetzt ist, wird der Wert benutzt.
- ${Variable:=Wert} Nutzt den Wert von Variable. Falls die Variable nicht gesetzt ist, wird der Wert benutzt und die Variable erhält den Wert.
- ${Variable:?Wert} Nutzt den Wert von Variable. Falls die Variable nicht gesetzt ist, wird der Wert ausgegeben und die Shell beendet. Wenn kein Wert angegeben wurde, wird der Text parameter null or not set ausgegeben.
- ${Variable:+Wert} Nutzt den Wert, falls die Variable gesetzt ist, andernfalls nichts.

Beispiele:

$ h=hoch r=runter l= #Weist den drei Variablen Werte zu, #wobei l einen leeren Wert erhält. $ echo ${h}sprung #Gibt hochsprung aus. Die Klammern müssen gesetzt werden, #damit h als Variablenname erkannt werden kann. $ echo ${h:-$r} #Gibt hoch aus, da die Variable h belegt ist. #Ansonsten würde der Wert von r ausgegeben. $ echo ${tmp:-`date`} #Gibt das aktuelle Datum aus, wenn die Variable tmp #nicht gesetzt ist. (Der Befehl date gibt das Datum zurück) $ echo ${l:=$r} #Gibt runter aus, da die Variable l keinen Wert enthält. #Gleichzeitig wird l der Wert von r zugewiesen. $ echo $l #Gibt runter aus, da l jetzt den gleichen Inhalt hat wie r.

2.4 Quoting

Dies ist ein sehr schwieriges Thema, da hier mehrere ähnlich aussehende Zeichen völlig verschiedene Effekte bewirken. Unix unterscheidet allein zwischen drei verschiedenen Anführungszeichen. Das Quoten dient dazu, bestimmte Zeichen mit einer Sonderbedeutung vor der Shell zu 'verstecken' um zu verhindern, dass diese expandiert (ersetzt) werden.

Die folgenden Zeichen haben eine spezielle Bedeutung innerhalb der Shell:

; Befehls-Trennzeichen & Hintergrund-Verarbeitung ( ) Befehls-Gruppierung | Pipe < > & Umlenkungssymbole * ? [ ] ~ + - @ ! Meta-Zeichen für Dateinamen ` ` (Backticks) Befehls-Substitution (Die Backticks erhält man durch [shift] und die Taste neben dem Backspace.) $ Variablen-Substitution [newline] [space] [tab] Wort-Trennzeichen

Die folgenden Zeichen können zum Quoten verwendet werden:

- " " (Anführungszeichen) Alles zwischen diesen Zeichen ist buchstabengetreu zu interpretieren. Ausnahmen sind folgende Zeichen, die ihre spezielle Bedeutung beibehalten: $ ` " \
- ' ' (Ticks) Alles zwischen diesen Zeichen wird wörtlich genommen, mit Ausnahme eines weiteren '.
- \ (Backslash) Das Zeichen nach einem \ wird wörtlich genommen. Anwendung z. B. innerhalb von " ", um ", $ und ` zu entwerten. Häufig verwendet zur Angabe von Leerzeichen (space) und Zeilenendezeichen, oder um ein \-Zeichen selbst anzugeben.

Beispiele:

$ echo 'Ticks “schützen” Anführungszeichen' Ticks “schützen” Anführungszeichen $ echo “Ist dies ein \"Sonderfall\"?" Ist dies ein “Sonderfall"? $ echo “Sie haben `ls | wc -l` Dateien in `pwd`" Sie haben 43 Dateien in /home/rschaten $ echo “Der Wert von \$x ist $x” Der Wert von $x ist 100

2.5 Meta-Zeichen

Bei der Angabe von Dateinamen können eine Reihe von Meta-Zeichen verwendet werden, um mehrere Dateien gleichzeitig anzusprechen oder um nicht den vollen Dateinamen ausschreiben zu müssen. (Meta-Zeichen werden auch Wildcards, Joker-Zeichen oder Platzhalter genannt.)

Die wichtigsten Meta-Zeichen sind:

- * Eine Folge von keinem, einem oder mehreren Zeichen
- ? Ein einzelnes Zeichen
- [abc] Übereinstimmung mit einem beliebigen Zeichen in der Klammer
- [a-q] Übereinstimmung mit einem beliebigen Zeichen aus dem angegebenen Bereich
- [!abc] Übereinstimmung mit einem beliebigen Zeichen, das nicht in der Klammer ist

- ~ Home-Verzeichnis des aktuellen Benutzers
- ~name Home-Verzeichnis des Benutzers name
- ~+ Aktuelles Verzeichnis
- ~- Vorheriges Verzeichnis

Beispiele:

ls neu* #Listet alle Dateien, die mit 'neu' anfangen ls neu? #Listet 'neuX', 'neu4', aber nicht 'neu10' ls [D-R]* #Listet alle Dateien, die mit einem Großbuchstaben zwischen D und R anfangen. #(Natürlich wird in Shell-Skripten -- wie überall in der Unix-Welt -- zwischen Groß- und Kleinschreibung unterschieden.)

2.6 Mustererkennung

2.6.1 Einfache Muster

Wie manche andere Programme stellen auch die Shells *sh*, *bash* und *ksh* Methoden zur Erkennung von Mustern in Zeichenketten (*globs*) zur Verfügung. Allerdings unterscheiden sich die Muster für Shells von den Mustern die bei *grep*, *sed*, *awk* und *perl* verwendet werden.

zeichenmenge kann eine Aneinanderreihung der folgenden Ausdrücke sein:

2.6.2 Beispiele

- [0-2][0-9]:[0-6][0-9]

Stimmt mit einer Uhrzeit überein, bei der Stunde und Minute zweistellig angegeben sind, z.B 09:03, 17:45 aber auch 25:61, nicht aber 9:15.

- ?????*

Eine Zeichenkette mit mindestens 5 Zeichen, z. B. hallo, /etc/passwd, nicht aber abc.

- [!/]*

Eine Zeichenkette die nicht mit / beginnt, z. B. hallo, + oder ./try.sh nicht aber /tmp.

- [-+*/]

eines der Zeichen +-*/ . Am Anfang oder am Ende von [...] wird die besondere Bedeutung von - aufgehoben.

- [.,;:!?]

eines der Zeichen .,;:!? . Das Zeichen ! hat nur am Anfang von [...] eine besondere Bedeutung.

- [][<>(){}]

eines der Zeichen []<>(){} . Die besondere Bedeutung von] wird aufgehoben, wenn es am Anfang von [...] steht.[[]]

- [!]]

Stimmt mit allen Zeichenketten überein außer mit der rechten eckigen Klammer].

2.6.3 Zusammengesetzte Muster

Eine Zeichenkette die nur aus Ziffern besteht lässt sich mit den bisher dargestellten Mustern nicht beschreiben. *ksh* und *bash* bieten deshalb Erweiterungen, mit denen man auch solch ein Muster beschreiben kann. Eine *musterliste* ist eine Aneinanderreihung einer oder mehrerer Muster die durch | getrennt sind.

musterliste := muster ['|' musterliste]

2.6.4 Beispiele

- +(0|1|2|3|4|5|6|7|8|9)

Eine Zeichenkette die nur aus Ziffern besteht, also eine ganze nichtnegative Zahl. 12345, 7, 007 und 0000000 stimmen mit dem Muster überein. 1.0 und +1 stimmen nicht überein.

- +([0-9])

das Muster stimmt mit den gleichen Zeichenketten überein wie +(0|1|2|3|4|5|6|7|8|9).

- @(0|?(+|-)[1-9]*(0|1|2|3|4|5|6|7|8|9))

die ganzen Zahlen mit oder ohne Vorzeichen, 0 ohne Vorzeichen und ohne führende Nullen, also die Zeichenketten 0,1,2,3,..., +1,+2,+3,... und $-1,-2,-3,...$, nicht aber 00, -0100 oder 1.0 .

- !(!if|then|else|elif|fi|case|esac|for|while|until|do|done|{|}|function|selec

Eine Zeichenkette, die nicht eines der reservierte *ksh*-Schlüsselwörter !, if, then, else, elif, fi, case, esac, for, while, until, do, done, {, }, function, select, time, [[</code>, <code>]] ist, stimmt mit dem Muster überein.

[, (, [[], done; oder hallo stimmen mit dem Muster überein, nicht aber [[, { oder else .

2.6.5 Ausdrücke mit Muster

Muster finden bei folgenden Ausdrücken Verwendung

1. Bedingungen
2. case-Anweisung
3. Filename expansion

2.7 Arithmetische Ausdrücke

ksh und *bash* verfügen über eine einfache Möglichkeit arithmetische Ausdrücke auszuwerten:

- ((*ausdruck*))
- let "*ausdruck*"

ausdruck ist dabei ein beliebiger arithmetischer Ausdruck, der der *C*-Syntax entspricht.

2.7.1 Operatoren

Die folgende Tabelle gibt alle Operatoren, geordnet nach ihrer Priorität, an. Operatoren am Beginn der Tabelle haben höhrere Priorität als Operatoren am Ende der Tabelle. Durch Anwendung der runden Klammern (und) kann die Priorität geändert werden. Mit "*)" gekennzeichnete Operatoren sind in der *bash*, nicht aber in der *ksh* enthalten.

- *variable*++ *variable*--
  - i++ Der Wert des Ausdrucks ist gleich dem Wert von *variable*. Als Seiteneffekt wird *variable* um 1 erhöht. Der Operator kann nur auf eine Variable angewendet werden, nicht auf einen Ausdruck.
  - i-- Der Wert des Ausdrucks ist gleich dem Wert von *variable*. Als Seiteneffekt wird *variable* um 1 vermindert.
- ++*variable* --*variable*
  - ++i Der Wert des Ausdrucks ist gleich dem Wert von *variable* + 1. Als Seiteneffekt wird *variable* um 1 erhöht.
  - --i Der Wert des Ausdrucks ist gleich dem Wert von *variable* - 1. Als Seiteneffekt wird *variable* um 1 vermindert.

2.7.2 Beispiele

- Schleife mit Zählvariable in *ksh*

#!/bin/ksh ((i=0)) while ((i<m)); do echo $i # do something ((i+=1)) done

In der *bash* könnte man den Ausdruck ((i+=1)) durch ((i++)) ersetzen. Dort würde man die Schleife so verwirklichen:

#!/bin/bash m=10 for ((i=0;i<m;i++)); do echo $i # do something done

2.7.3 Arithmetik

Arithmetische Ausdrücke kann man mit $((...)) auswerten.

2.7.4 Beispiel

echo $((7*(3+4)>3+4*7))

2.8 Programmablaufkontrolle

Bei der Shell-Programmierung verfügt man über ähnliche Konstrukte wie bei anderen Programmiersprachen, um den Ablauf des Programms zu steuern. Dazu gehören Funktionsaufrufe, Schleifen, Fallunterscheidungen und dergleichen.

2.9 Kommentare (#)

Kommentare in der Shell beginnen immer mit dem Nummern-Zeichen (#). Dabei spielt es keine Rolle, ob das Zeichen am Anfang der Zeile steht, oder hinter irgendwelchen Befehlen. Alles von diesem Zeichen bis zum Zeilenende (bis auf eine Ausnahme - siehe unter Auswahl der Shell).

2.10 Auswahl der Shell (#!)

In der ersten Zeile eines Shell-Skriptes sollte definiert werden, mit welcher Shell das Skript ausgeführt werden soll. Das System öffnet dann eine Subshell und führt das restliche Skript in dieser aus, es sei denn man hat einen expliziten Interpreter angegeben: bash foo.sh

Die Angabe erfolgt über eine Zeile in der Form #!/bin/sh, wobei unter /bin/sh die entsprechende Shell (in diesem Fall die Bourne-Shell) liegt. Dieser Eintrag wirkt nur dann, wenn er in der ersten Zeile des Skripts steht. Das # Zeichen wird im Unix Slang auch als “She” bezeichnet und das Ausrufezeichen als “Bang”. Mit Hilfe der "Shebang" wird der jeweils zu verwendende Interpreter für das Skript festgelegt (in diesem Fall die Bourne-Shell).

2.11 Null-Befehl (:)

Der Doppelpunkt : als Befehl tut nichts, außer den Status 0 zurückzugeben und seine Argumente zu expandieren. Es ist also der NoOp-Befehl der Shell. Er wird benutzt, um Endlosschleifen zu schreiben (siehe unter while), oder um leere Blöcke in if- oder case-Konstrukten möglich zu machen. Beispiel: Prüfen, ob jemand angemeldet ist: checkuser.sh

if who | grep -q $1 # who: Liste der Benutzer # grep: Suche nach Muster then : # tut nichts else echo “Benutzer $1 ist nicht angemeldet” fi

2.12 Source (.)

Der Source-Befehl wird in der Form . skriptname angegeben, also ein Punkt dann ein Leerzeichen und dann der Name der Datei. Er bewirkt ähnliches wie ein #include in der Programmiersprache C.

Die Datei (auf die das Source ausgeführt wurde) wird eingelesen und ausgeführt, als ob ihr Inhalt an der Stelle des Befehls stehen würde. Diese Methode wird zum Beispiel während des Bootvorgangs in den Init-Skripten benutzt, um immer wieder benötigte Funktionen (Starten eines Dienstes, Statusmeldungen auf dem Bildschirm etc.) in einer zentralen Datei pflegen zu können (siehe Beispiel unter Ein typisches Init-Skript).

2.13 Funktionen

Es ist in der Shell auch möglich, ähnlich wie in einer anderen Programmiersprache, Funktionen zu deklarieren und zu benutzen. Mit dem Kommando return hat man die Möglichkeit, aus einer Funktion - genauso wie aus einem Skript mit exit - einen Status-Wert zurückzugeben. Beispiel: Die Funktion gibt die Anzahl der Dateien im aktuellen Verzeichnis zurück. Aufgerufen wird diese Funktion wie ein Befehl, also einfach durch die Eingabe von count.

Weil die Shell den Code zur Laufzeit interpretiert, müssen Funktionen am Anfang des Skriptes stehen, also vor ihrem erstmaligem Aufruf.

count () { ls | wc -l # ls: Liste aller Objekte im Verzeichnis # wc: Word-Count; mit Attribut -l werden Zeilen gezählt # in Verbindung mit ls werden also die (nicht versteckten) Objekte gezählt } count # Aufruf der Funktion

Beispiel-Ausgabe: 15

count2 () { if [-d "$1”]; then # überprüfen, ob der erste Parameter ein Verzeichnis ist ls $1 | wc -l # wie oben return 0 # alles OK else echo “Ungültiges Verzeichnis: $1” return 1 # Fehler fi } count2 "/gibt/es/garnicht” # Aufrufe der Funktion count2 echo “Status: $?" count2 "/etc” echo “Status: $?"

Beispiel-Ausgabe:

Ungültiges Verzeichnis: /gibt/es/garnicht

Status: 1

234

Status: 0

2.14 Bedingungen prüfen mit test ([ ] , [[ ]])

Da die Standard-Shell keine arithmetischen oder logischen Ausdrücke auswerten konnte, musste dazu ein externes Programm benutzt werden (if und Konsorten prüfen nur den Rückgabewert eines aufgerufenen Programmes -- 0 bedeutet true, alles andere bedeutet false, siehe auch Rückgabewerte). Dieses Programm heißt **test** (siehe *man test* für die manpage). der Aufruf hat die Form test *AUSDRUCK* . Üblicherweise besteht auf allen Systemen auch noch der Alternativbefehl **[** gleicher Funktionalität. Diese Variante dient dazu Skripte lesbarer zu machen. Da **[** im einfachsten Fall ein ausführbares Programm ist, muss beim Aufruf ein Leerzeichen folgen. Als letzten Aufrufparameter erwartet [das Zeichen **]** (mit einem Leerzeichen davor). Der Aufruf hat dann die Form [*AUSDRUCK*] . In modernen Shells sind test und [jedoch als *builtin*-Befehle der ausführenden Shell vorhanden (siehe help [ ), so dass die externen Befehle nicht ausgeführt werden, auch wenn sie noch z.B. als /usr/bin/[vorhanden sind (siehe man [ ) .

In neueren Shells ist als weitere Verbesserung noch das builtin [[ ]] (double brackets) vorhanden. Es hat gegenüber dem einfachen [ ] z.B. den Vorteil dass man die Primitive < > und ( ) nicht mehr escapen muss (siehe help [[ ).[1]

Der test-Befehl bietet sehr umfangreiche Optionen an. Dazu gehören Dateitests und Vergleiche von Zeichenfolgen oder ganzen Zahlen. Diese Bedingungen können auch durch Verknüpfungen kombiniert werden. Dateitests:

Bedingungen für Zeichenfolgen:

Ganzzahlvergleiche:

Kombinierte Formen: (Bedingung) Wahr, wenn die Bedingung zutrifft (wird für die Gruppierung verwendet). Den Klammern muss ein \ vorangestellt werden.

! Bedingung Wahr, wenn die Bedingung nicht zutrifft (NOT). Bedingung1 -a Bedingung2 Wahr, wenn beide Bedingungen zutreffen (AND). Bedingung1 -o Bedingung2 Wahr, wenn eine der beiden Bedingungen zutrifft (OR). Beispiele: while test $# -gt 0 Solange Argumente vorliegen. . . while [-n "$1”] Solange das erste Argument nicht leer ist. . . if ["$count” -lt 10] Wenn $count kleiner 10. . . if [-d RCS] Wenn ein Verzeichnis RCS existiert. . . if ["$Antwort” != j] Wenn die Antwort nicht “j” ist. . . if [! -r "$1” -o ! -f "$1”] Wenn das erste Argument keine lesbare oder reguläre Datei ist. . .

2.15 if

Die if-Anweisung in der Shell-Programmierung macht das gleiche wie in allen anderen Programmiersprachen, sie testet eine Bedingung auf Wahrheit und macht davon den weiteren Ablauf des Programms abhängig.

Die Syntax der if-Anweisung lautet wie folgt:

if-beispiel.sh if Bedingung1 then Befehle1 [elif Bedingung2 then Befehle2] ... [else Befehle3] fi

Wenn die Bedingung1 erfüllt ist, werden die Befehle1 ausgeführt; andernfalls, wenn die Bedingung2 erfüllt ist, werden die Befehle2 ausgeführt. Trifft keine Bedingung zu, sollen die Befehle3 ausgeführt werden.

Bedingungen werden normalerweise mit dem Befehl test formuliert. Es kann aber auch der Rückgabewert jedes anderen Kommandos ausgewertet werden. Für Bedingungen, die auf jeden Fall zutreffen sollen steht der Null-Befehl (:) zur Verfügung. Beispiele:

test-beispiele.sh #!/bin/sh # Füge eine 0 vor Zahlen kleiner 10 ein: if [$counter -lt 10]; then number=0$counter else number=$counter fi # Erstelle ein Verzeichnis, wenn es noch nicht existiert: if [! -d "$dir"]; then mkdir "$dir" # mkdir: Verzeichnis erstellen fi

Noch eine Anmerkung: Es hat sich durchgesetzt, dass man das then in die gleiche Zeile schreibt, wie die Bedingung. Dabei darf man aber den Befehlstrenner, das Semikolon, nicht vergessen.

2.16 case

Auch die case-Anweisung ist vergleichbar in vielen anderen Sprachen vorhanden. Sie dient, ähnlich wie die if-Anweisung, zur Fallunterscheidung. Allerdings wird hier nicht nur zwischen zwei Fällen unterschieden (Entweder / Oder), sondern es sind mehrere Fälle möglich. Man kann die case-Anweisung auch durch eine geschachtelte if-Anweisung völlig umgehen, allerdings ist sie ein elegantes Mittel um den Code lesbar zu halten.

Die Syntax der case-Anweisung lautet wie folgt:

case-beispiel-simpel.sh #!/bin/sh case Wert in Muster1) Befehle1;; Muster2) Befehle2;; ... esac

Wenn der Wert mit dem Muster1 übereinstimmt, wird die entsprechende Befehlsgruppe (Befehle1) ausgeführt, bei Übereinstimmung mit Muster2 werden die Kommandos der zweiten Befehlsgruppe (Befehle2) ausgeführt, usw. Der letzte Befehl in jeder Gruppe muss mit ;; gekennzeichnet werden. Das bedeutet für die Shell soviel wie springe zum nächsten esac, so dass die anderen Bedingungen nicht mehr überprüft werden.

In den Mustern sind die gleichen Meta-Zeichen erlaubt wie bei der Auswahl von Dateinamen. Wenn in einer Zeile mehrere Muster angegeben werden sollen, müssen sie durch ein Pipezeichen (|, bitweises ODER) getrennt werden. Mit *) als letzter Option werden alle auf sonst keine Zeile passenden Argumentwerte eingefangen.

Beispiele:

case-beispiel-fortgeschritten.sh #!/bin/sh # Mit dem ersten Argument in der Befehlszeile # wird die entsprechende Aktion festgelegt: case $1 in # nimmt das erste Argument Ja|Nein) response=1;; -[tT]) table=TRUE;; *) echo “Unbekannte Option"; exit 1;; esac # Lies die Zeilen von der Standardeingabe, bis eine # Zeile mit einem einzelnen Punkt eingegeben wird: while : # Null-Befehl (immer wahr, siehe unter 3.11) do echo “Zum Beenden . eingeben ==> \c” read line # read: Zeile von StdIn einlesen case "$line” in .) echo “Ausgefuehrt” break;; *) echo "$line” >> $message ;; esac done

2.17 for

Mit einer for-Schleife wird eine Zählvariable über einen bestimmten Zahlenbereich iteriert (for i = 1 to 100...next) oder mit einzelnen Elementen aus einer anzugebenden Liste.

Die Syntax der for-Schleife mit Liste lautet wie folgt:

for-syntax.sh #!/bin/sh for x [in Liste] do Befehle done

Die Befehle werden ausgeführt, wobei der Variablen x nacheinander die Werte aus der Liste zugewiesen werden. Wie man sieht ist die Angabe der Liste optional, wenn sie nicht angegeben wird, nimmt x der Reihe nach alle Werte aus $@ (in dieser vordefinierten Variablen liegen die Aufrufparameter - siehe unter Datenströme) an. Wenn die Ausführung eines Schleifendurchlaufs bzw. der ganzen Schleife abgebrochen werden soll, müssen die Kommandos continue bzw. break benutzt werden.

Beispiele:

for-beispiele.sh #!/bin/sh # Seitenweises Formatieren der Dateien, die auf der # Befehlszeile angegeben wurden, und speichern des # jeweiligen Ergebnisses: for file in "$@"; do pr "$file” > "$file.tmp” # pr: Formatiert Textdateien done # Durchsuche Kapitel zur Erstellung einer Wortliste (wie fgrep -f): for item in $(cat program_list) # cat: Datei ausgeben do echo “Pruefung der Kapitel auf” echo “Referenzen zum Programm $item ...” grep -c "$item.[co]" chap* # grep: nach Muster suchen done # Ermittle einen Ein-Wort-Titel aus jeder Datei und # verwende ihn als neuen Dateinamen: for file in "$@"; do name=$(sed -n 's/NAME: //p' "$file”) # sed: Skriptsprache zur # Textformatierung mv "$file” "$name” # mv: Datei verschieben # bzw. umbenennen done

Um die for-Schleife wie in anderen Programmiersprachen mit Zählvariablen zu nutzen, funktioniert beispielsweise folgendes:

for-zaehlschleife.sh #!/bin/sh #gibt die Zaehlvariable aus for ((i = 0; i <= 5; i++)); do echo $i done # oder: for i in {0..5}; do echo $i done # oder echo {0..5}

2.18 while

Die while-Schleife ist wieder ein Konstrukt, das einem aus vielen anderen Sprachen bekannt ist: Die kopfgesteuerte Schleife.

Die Syntax der while-Schleife lautet wie folgt:

while-syntax.sh #!/bin/sh while Bedingung do Befehle done

Die Befehle werden so lange ausgeführt, wie die Bedingung erfüllt ist. Dabei wird die Bedingung vor der Ausführung der Befehle überprüft. Die Bedingung wird dabei üblicherweise, genau wie bei der if-Anweisung, mit dem Befehl test) formuliert. Wenn die Ausführung eines Schleifendurchlaufs bzw. der ganzen Schleife abgebrochen werden soll, müssen die Kommandos continue bzw. break benutzt werden.

Beispiel:

while-beispiel01.sh #!/bin/sh # Zeilenweise Ausgabe aller Aufrufparameter: while [-n "$1"]; do echo $1 shift # mit shift werden die Parameter nach # Links geshiftet (aus $2 wird $1) done

2.19 until

Die until-Schleife ist das Gegenstück zur while-Schleife: Die ebenfalls aus vielen anderen Sprachen bekannte Schleife.

Die Syntax der until-Schleife lautet wie folgt:

until-syntax.sh #!/bin/sh until Bedingung do Befehle done

Die Befehle werden ausgeführt, bis die Bedingung erfüllt ist. Die Bedingung wird dabei üblicherweise, genau wie bei der if-Anweisung, mit dem Befehl test formuliert. Wenn die Ausführung eines Schleifendurchlaufs bzw. der ganzen Schleife abgebrochen werden soll, müssen die Kommandos continue bzw. break benutzt werden. Beispiel: Hier wird die Bedingung nicht per test, sondern mit dem Rückgabewert des Programms grep formuliert.

until-beispiel.sh #!/bin/sh # Warten, bis sich der Administrator einloggt: until who | grep “root"; do # who: Liste der Benutzer # grep: Suchen nach Muster sleep 2 # sleep: warten done echo “Der Meister ist anwesend”

2.20 continue

Die Syntax der continue-Anweisung lautet wie folgt:

continue-syntax.sh #!/bin/sh continue [n]

Man benutzt continue um die restlichen Befehle in einer Schleife zu überspringen und mit dem nächsten Schleifendurchlauf anzufangen. Wenn der Parameter n angegeben wird, werden n Schleifenebenen übersprungen.

2.21 break

Die Syntax der break-Anweisung lautet wie folgt:

break-syntax.sh break [n]

Mit break kann man die innerste Ebene (bzw. n Schleifenebenen) verlassen ohne den Rest der Schleife auszuführen.

2.22 exit

Die Syntax der exit-Anweisung lautet wie folgt:

exit-syntax.sh exit [n]

Die exit-Anweisung wird benutzt, um ein Skript vorzeitig zu beenden. Wenn der Parameter n angegeben wird, wird er von dem Skript als Exit-Code zurückgegeben.

2.23 Befehlsanordnungen

Es gibt eine Reihe verschiedener Möglichkeiten, Kommandos auszuführen:

Befehl & Ausführung von Befehl im Hintergrund Befehl1 ; Befehl2 Befehlsfolge, führt mehrere Befehle in einer Zeile nacheinander aus (Befehl1 ; Befehl2) Subshell, behandelt Befehl1 und Befehl2 als Befehlsfolge Befehl1 | Befehl2 Pipe, verwendet die Ausgabe von Befehl1 als Eingabe für Befehl2 Befehl1 $(Befehl2) Befehls-Substitution, verwendet die Ausgabe von Befehl2 als Argumente für Befehl1 Befehl1 && Befehl2 AND, führt zuerst Befehl1 und dann (wenn Befehl1 erfolgreich war) Befehl2 aus Befehl1 || Befehl2 OR, entweder Befehl1 ausführen oder Befehl2 (Wenn Befehl1 nicht erfolgreich war) { Befehl1; Befehl2;} Ausführung der Befehle in der momentanen Shell (Wichtig: Leerzeichen am Anfang und Semikolon hinter dem letzten Befehl) Beispiele: nroff Datei & Formatiert die Datei im Hintergrund cd; ls Sequentieller Ablauf (date; who; pwd) > logfile Lenkt alle Ausgaben um (Subshell) sort Datei | lp Sortiert die Datei und druckt sie vi `grep -l ifdef *.c` Editiert die mittels grep gefundenen Dateien grep XX Datei && lp Datei Druckt die Datei, wenn sie XX enthält grep XX Datei || lp Datei Druckt die Datei, wenn sie XX nicht enthält { date; who; pwd;} > logfile Lenkt alle Ausgaben um (momentane Shell)

2.24 Datenströme

Eines der markantesten Konzepte, das in Shell-Skripten benutzt wird, ist das der Datenströme. Die meisten der vielen Unix-Tools bieten die Möglichkeit, Eingaben aus der Standard-Eingabe entgegenzunehmen und Ausgaben dementsprechend auf der Standard-Ausgabe zu machen. Es gibt noch einen dritten Kanal für Fehlermeldungen, so dass man eine einfache Möglichkeit hat, fehlerhafte Pro-

grammdurchläufe zu behandeln indem man die Fehlermeldungen von den restlichen Ausgaben trennt.

Es folgt eine Aufstellung der drei Standardkanäle:

Datei-Deskriptor Name Gebräuchliche Abkürzung Typischer Standard 0 Standardeingabe stdin Tastatur 1 Standardausgabe stdout Terminal 2 Fehlerausgabe stderr Terminal

Die standardmäßige Eingabequelle oder das Ausgabeziel können wie folgt geändert werden: Einfache Umlenkung:

Befehl > Datei Standardausgabe von Befehl in Datei schreiben. Die Datei wird überschrieben, wenn sie schon bestand. Befehl >> Datei Standardausgabe von Befehl an Datei anhängen. Die Datei wird erstellt, wenn sie noch nicht bestand. Befehl < Datei Standardeingabe für Befehl aus Datei lesen. Befehl1 | Befehl2 Die Standardausgabe von Befehl1 wird an die Standardeingabe von Befehl2 übergeben. Mit diesem Mechanismus können Programme als Filter für den Datenstrom eingesetzt werden. Das verwendete Zeichen heißt Pipe.

Umlenkung mit Hilfe von Datei-Deskriptoren:

Befehl >&n Standard-Ausgabe von Befehl an den Datei-Deskriptor n übergeben. Befehl m>&n Der gleiche Vorgang, nur wird die Ausgabe, die normalerweise an den Datei-Deskriptor m geht, an den Datei- Deskriptor n übergeben. Befehl >&- Schließt die Standard-Ausgabe. Befehl <&n Standard-Eingabe für Befehl wird vom Datei-Deskriptor n übernommen. Befehl m<&n Der gleiche Vorgang, nur wird die Eingabe, die normalerweise vom Datei-Deskriptor m stammt, aus dem Datei-Deskriptor n übernommen. Befehl <&- Schließt die Standard-Eingabe.

Mehrfach-Umlenkung:

Befehl 2> Datei Fehler-Ausgabe von Befehl in Datei schreiben. Die Standard-Ausgabe bleibt unverändert (z. B. auf dem Terminal). Befehl > Datei 2>&1 Fehler-Ausgabe und Standard-Ausgabe von Befehl werden in die Datei geschrieben. (Befehl > D1) 2>D2 Standard-Ausgabe erfolgt in die Datei D1; Fehler-Ausgabe in die Datei D2. Befehl | tee Dateien Die Ausgaben von Befehl erfolgen an der Standard-Ausgabe (in der Regel: Terminal), zusätzlich wird sie vom Kommando tee in die Dateien geschrieben.

Zwischen den Datei-Deskriptoren und einem Umlenkungssymbol darf kein Leerzeichen sein; in anderen Fällen sind Leerzeichen erlaubt.

Beispiele:

cat Datei1 > Neu Schreibt den Inhalt der Datei1 in die Datei Neu. cat Datei2 Datei3 >> Neu Hängt den Inhalt der Datei2 und der Datei3 an die Datei Neu an. mail name < Neu Das Programm mail liest den Inhalt der Datei Neu. ls -l | grep “txt” | sort Die Ausgabe des Befehls ls -l (Verzeichnisinhalt) wird an das Kommando grep weitergegeben, das darin nach txt sucht. Alle Zeilen, die das Muster enthalten, werden anschließend an sort übergeben und landen dann sortiert auf der Standardausgabe.

Gerade der Mechanismus mit dem Piping sollte nicht unterschätzt werden. Er dient nicht nur dazu, relativ kleine Texte zwischen Tools hin- und herzureichen. An dem folgenden Beispiel soll die Mächtigkeit dieses kleinen Zeichens gezeigt werden:

Es ist mit den passenden Tools unter Unix möglich, eine ganze Audio-CD mit zwei Befehlen an der Kommandozeile zu duplizieren. Das erste Kommando veranlaßt, dass die TOC (Table Of Contents) der CD in die Datei cd.toc geschrieben wird. Das dauert nur wenige Sekunden. Die Pipe steckt im zweiten Befehl. Hier wird der eigentliche Inhalt der CD mit dem Tool cdparanoia ausgelesen. Da kein Dateiname angegeben schreibt cdparanoia die Daten auf seine Standardausgabe. Diese wird von dem Brennprogramm cdrdao übernommen und in Verbindung mit der TOC on the fly auf die CD geschrieben.

cd-kopieren.sh #!/bin/sh cdrdao read-toc --datafile - cd.toc cdparanoia -q -R 1- - | cdrdao write --buffers 64 cd.toc

3 Speziellere Befehle

3.1 Farbe von Text

Um bestimmte Textpassagen einer Skriptausgabe farbig hervorzuheben, existieren Steuerzeichen. Sie beginnen mit einem Escape-Zeichen (␛, Unicode u001b), gefolgt von einer öffnenden eckigen Klammer, einer Zahl und einem kleinen m. Die Zahl gibt dabei die Farbe an. 30…39 und 90…96 ändert die Vordergrundfarbe, 40…49 und 100…106 die Hintergrundfarbe. 0 schaltet die Formatierung wieder aus.

Das Escape-Zeichen kann in Vim im Editiermodus entweder mit **Ctrl-v Esc** oder mit **Ctrl-v** gefolgt vom Unicode des Zeichens eingefügt werden. Alternativ kann ein '\e' verwendet werden. Beispiel:

echo -e '\e[33m\e[4mtest\e[0m'

4 Anhang A: Beispiele

4.1 Schleifen und Rückgabewerte

Man kann mit einer until- bzw. mit einer while-Schleife schnell kleine aber sehr nützliche Tools schreiben, die einem lästige Aufgaben abnehmen.

4.1.1 Schleife, bis ein Kommando erfolgreich war

Angenommen, bei der Benutzung eines Rechners tritt ein Problem auf, bei dem nur der Administrator helfen kann.

Dann möchte man informiert werden, sobald dieser an seinem Arbeitsplatz ist. Man kann jetzt in regelmäßigen Abständen das Kommando who ausführen, und dann in der Ausgabe nach dem Eintrag root suchen. Das ist aber lästig.

Einfacher geht es, wenn wir uns ein kurzes Skript schreiben, das alle 30 Sekunden automatisch überprüft, ob der Admin angemeldet ist. Wir erreichen das mit dem folgenden Code:

auf-root-warten.sh #!/bin/sh until who | grep "^root "; do sleep 30 done echo Big Brother is watching you!

Das Skript führt also so lange das Kommando aus, bis die Ausführung erfolgreich war. Dabei wird die Ausgabe von who mit einer Pipe in das grep-Kommando umgeleitet. Dieses sucht darin nach einem Auftreten von root am Zeilenanfang. Der Rückgabewert von grep ist 0 wenn das Muster gefunden wird, 1 wenn es nicht gefunden wird und 2 wenn ein Fehler auftrat. Damit der Rechner nicht die ganze Zeit mit dieser Schleife beschäftigt ist, wird im Schleifenkörper ein sleep 30 ausgeführt, um den Prozess für 30 Sekunden schlafen zu schicken. Sobald der Admin sich eingeloggt hat, wird eine entsprechende Meldung ausgegeben.

4.1.2 Schleife, bis ein Kommando nicht erfolgreich war

Analog zum vorhergehenden Beispiel kann man auch ein Skript schreiben, das meldet, sobald sich ein Benutzer abgemeldet hat. Dazu ersetzen wir nur die until-Schleife durch eine entsprechende while-Schleife:

warten-bis-root-verschwindet.sh #!/bin/sh while who | grep "^root "; do sleep 30 done echo Die Katze ist aus dem Haus, Zeit, dass die Mäuse tanzen!

Die Schleife wird nämlich dann so lange ausgeführt, bis grep einen Fehler (bzw. eine erfolglose Suche) zurückmeldet.

4.2 Ein typisches Init-Skript

Das folgende Skript *beispiel.sh* dient dazu, den Apache HTTP-Server zu starten. Es wird während des Bootvorgangs gestartet, wenn der dazugehörige Runlevel initialisiert wird.

Das Skript muss mit einem Parameter aufgerufen werden. Möglich sind hier start, stop, status, restart und reload. Wenn falsche Parameter übergeben wurden, wird eine entsprechende Meldung angezeigt.

Das Ergebnis der Ausführung wird mit Funktionen dargestellt, die aus der Datei /etc/rc.d/init.d/functions stammen. Ebenfalls in dieser Datei sind Funktionen, die einen Dienst starten oder stoppen.

Zunächst wird festgelegt, dass dieses Skript in der Bourne-Shell ausgeführt werden soll (Auswahl der Shell).

beispiel.sh #!/bin/sh

Dann folgen Kommentare, die den Sinn des Skriptes erläutern.

beispiel.sh (Fortsetzung) ## Startup script for the Apache Web Server # # chkconfig: 345 85 15 # description: Apache is a World Wide Web server. It is \ # used to serve HTML files and CGI # # processname: httpd # pidfile: /var/run/httpd.pid # config: /etc/httpd/conf/access.conf # config: /etc/httpd/conf/httpd.conf # config: /etc/httpd/conf/srm.conf

Jetzt wird die Datei mit den Funktionen eingebunden.

beispiel.sh (Fortsetzung) # Source function library. . /etc/rc.d/init.d/functions

Hier werden die Aufrufparameter ausgewertet.

beispiel.sh (Fortsetzung) # See how we were called. case "$1" in start) echo -n "Starting httpd: "

Nachdem eine Meldung über den auszuführenden Vorgang ausgegeben wurde, wird die Funktion daemon aus der Funktionsbibliothek ausgeführt. Diese Funktion startet das Programm, dessen Name hier als Parameter übergeben wird. Dann gibt sie eine Meldung über den Erfolg aus.

beispiel.sh (Fortsetzung) daemon httpd echo

Jetzt wird ein Lock-File angelegt. (Ein Lock-File signalisiert anderen Prozessen, dass ein bestimmter Prozess bereits gestartet ist. So kann ein zweiter Aufruf verhindert werden.)

beispiel.sh (Fortsetzung) touch /var/lock/subsys/httpd ;; stop) echo -n "Shutting down http: "

Hier passiert im Prinzip das gleiche wie oben, nur dass mit der Funktion killproc der Daemon angehalten wird.

beispiel.sh (Fortsetzung) killproc httpd echo

Danach werden Lock-File und PID-File gelöscht. (In einem sogenannten PID-File hinterlegen einige Prozesse ihre Prozess-ID, um anderen Programmen den Zugriff zu erleichtern, z.B. um den Prozess anzuhalten etc.)

beispiel.sh (Fortsetzung) rm -f /var/lock/subsys/httpd rm -f /var/run/httpd.pid ;; status)

Die Funktion status stellt fest, ob der entsprechende Daemon bereits läuft, und gibt das Ergebnis aus.

beispiel.sh (Fortsetzung) status httpd ;; restart)

Bei Aufruf mit dem Parameter restart ruft sich das Skript zwei mal selbst auf (in $0 steht der Aufrufname des laufenden Programms). Einmal, um den Daemon zu stoppen, dann, um ihn wieder zu starten.

beispiel.sh (Fortsetzung) $0 stop $0 start ;; reload) echo -n "Reloading httpd: "

Hier sendet die killproc-Funktion dem Daemon ein Si-

gnal das ihm sagt, dass er seine Konfiguration neu einlesen soll.

beispiel.sh (Fortsetzung) killproc httpd -HUP echo ;; *) echo “Usage: $0 {start|stop|restart|reload|status}"

Bei Aufruf mit einem beliebigen anderen Parameter wird eine Kurzhilfe ausgegeben. Dann wird dafür gesorgt, dass das Skript mit dem Exit-Code 1 beendet wird. So kann festgestellt werden, ob das Skript ordnungsgemäß beendet wurde (exit).

beispiel.sh (Fortsetzung) exit 1 esac exit 0

4.3 Parameterübergabe in der Praxis

Es kommt in der Praxis sehr oft vor, dass man ein Skript schreibt, dem der Anwender Parameter übergeben soll. Wenn das nur eine Kleinigkeit ist (zum Beispiel ein Dateiname), dann fragt man einfach die entsprechenden vordefinierten Variablen ab. Sollen aber richtige Parameter eingesetzt werden, die sich so einsetzen lassen wie man es von vielen Kommandozeilentools gewohnt ist, dann benutzt man das Hilfsprogramm getopt. Dieses Programm parst die originalen Parameter und gibt sie in standardisierter Form zurück.

Das soll an folgendem Skript verdeutlicht werden. Das Skript kennt die Optionen -a und -b. Letzterer Option muss ein zusätzlicher Wert mitgegeben werden. Alle anderen Parameter werden als Dateinamen interpretiert.

getopt.sh #!/bin/sh set -- $(getopt “ab:" "$@") || {

Das set-Kommando belegt den Inhalt der vordefinierten Variablen neu, so dass es aussieht, als ob dem Skript die Rückgabewerte von getopt übergeben wurden. Man muss die beiden Minuszeichen angeben, da sie dafür sorgen, dass die Aufrufparameter an getopt und nicht an die Shell selbst übergeben werden. Die originalen Parameter werden von getopt untersucht und modifiziert zurückgegeben: a und b werden als Parameter markiert, b sogar mit der Möglichkeit einer zusätzlichen Angabe.

Wenn dieses Kommando fehlschlägt ist das ein Zeichen dafür, dass falsche Parameter übergeben wurden. Also wird nach einer entsprechenden Meldung das Programm mit Exit-Code 1 verlassen.

getopt.sh (Fortsetzung) echo “Anwendung: $(basename $0) [-a] [-b Name] Dateien” 1>&2 exit 1 } echo “Momentan steht in der Kommandozeile folgendes: $*" aflag=0 name=NONE while : do

In einer Endlos-Schleife, die man mit Hilfe des Null-Befehls (:) baut, werden die neuen Parameter der Reihe nach untersucht. Wenn ein -a vorkommt, wird die Variable aflag gesetzt. Bei einem -b werden per shift alle Parameter nach Links verschoben, dann wird der Inhalt des nächsten Parameters in der Variablen name gesichert.

getopt.sh (Fortsetzung) case "$1” in -a) aflag=1 ;; -b) shift; name="$1” ;; --) break ;;

Wenn ein -- erscheint, ist das ein Hinweis darauf, dass die Liste der Parameter abgearbeitet ist. Dann wird per break) die Endlosschleife unterbrochen. Die Aufrufparameter enthalten jetzt nur noch die eventuell angegebenen Dateinamen, die von dem restlichen Skript wie gewohnt weiterverarbeitet werden können.

getopt.sh (Fortsetzung) esac shift done shift

Am Ende werden die Feststellungen ausgegeben.

getopt.sh (Fortsetzung) echo “aflag=$aflag / Name = $name / Die Dateien sind $*"

4.4 Fallensteller: Auf Traps reagieren

Ein laufendes Shell-Skript kann durch Druck auf die Interrupt-Taste (normalerweise [CTRL+C]) unterbrochen werden. Durch Druck auf diese Taste wird ein Signal an den entsprechenden Prozess gesandt, das ihn bittet sich zu beenden. Dieses Signal heißt SIGINT (für SIGnal INTerrupt) und trägt die Nummer 2. Das kann ein kleines Problem darstellen, wenn das Skript sich temporäre Dateien angelegt hat, da diese nach der Ausführung nur noch unnötig Platz verbrauchen und eigentlich gelöscht werden sollten. Man kann sich sicher auch noch wichtigere Fälle vorstellen, in denen ein Skript bestimmte Aufgaben auf jeden Fall erledigen muss, bevor es sich beendet.

Es gibt eine Reihe weiterer Signale, auf die ein Skript reagieren kann. Alle sind in der Man-Page von signal beschrieben. Hier die wichtigsten:

Nummer Name Bedeutung 0 Normal Exit Wird durch das exit-Kommando ausgelöst. 1 SIGHUP Wenn die Verbindung abbricht (z.B. wenn das Terminal geschlossen wird). 2 SIGINT Zeigt einen Interrupt an ([CTRL+C]). 15 SIGTERM Wird vom kill-Kommando gesendet.

Wie löst man jetzt dieses Problem? Glücklicherweise verfügt die Shell über das trap-Kommando, mit dessen Hilfe man auf diese Signale reagieren kann. Die Anwendung soll in folgendem Skript beispielhaft dargestellt werden.

Das Skript soll eine komprimierte Textdatei mittels zcat in ein temporäres File entpacken, dieses mit pg seitenweise anzeigen und nachher wieder löschen.

zeige-komprimierte-datei.sh #!/bin/sh stat=1 temp=/tmp/zeige$$

Zunächst werden zwei Variablen belegt, die im weiteren Verlauf benutzt werden sollen. In stat wird der Wert abgelegt, den das Skript im Falle eines Abbruchs als Exit-Status zurückliefern soll. Die Variable temp enthält den Namen für eine temporäre Datei. Dieser setzt sich zusammen aus /tmp/zeige und der Prozessnummer des laufenden Skripts. So soll sichergestellt werden, dass noch keine Datei mit diesem Namen existiert.

zeige-komprimierte-datei.sh (Fortsetzung) trap 'rm -f $temp; exit $stat' 0 trap 'echo "$(basename $0): Ooops...” 1>&2' 1 2 15

Hier werden die Traps definiert. Bei Signal 0 wird die temporäre Datei gelöscht und der Wert aus der Variable stat als Exit-Code zurückgegeben. Dabei wird dem rm-Kommando der Parameter -f mitgegeben, damit keine Fehlermeldung ausgegeben wird, falls die Datei (noch) nicht existiert. Dieser Fall tritt bei jedem Beenden des Skriptes auf, also sowohl bei einem normalen Ende, als auch beim Exit-Kommando, bei einem Interrupt oder bei einem Kill. Der zweite Trap reagiert auf die Signale 1, 2 und 15. Das heißt, er wird bei jedem unnormalen Ende ausgeführt. Er gibt eine entsprechende Meldung auf die Standard-Fehler-Ausgabe aus. Danach wird das Skript beendet, und der erste Trap wird ausgeführt.

zeige-komprimierte-datei.sh (Fortsetzung) case $# in 1) zcat "$1" > $temp pg $temp stat=0 ;;

Jetzt kommt die eigentliche Funktionalität des Skriptes: Das case-Kommando testet die Anzahl der übergebenen Parameter. Wenn genau ein Parameter übergeben wurde, entpackt zcat die Datei, die im ersten Parameter angegeben wurde, in die temporäre Datei. Dann folgt die seitenweise Ausgabe mittels pg. Nach Beendigung der Ausgabe wird der Status in der Variablen auf 0 gesetzt, damit beim Skriptende der korrekte Exit-Code zurückgegeben wird.

zeige-komprimierte-datei.sh (Fortsetzung) *) echo “Anwendung: $(basename $0) Dateiname” 1>&2 esac

Wenn case eine andere Parameterzahl feststellt, wird eine Meldung mit der Aufrufsyntax auf die Standard-Fehlerausgabe geschrieben.

4.5 Beispiel: Großschreibung des Dateinamens auf Kleinschreibung umändern

Wenn man von Windows auf Linux umsteigt und kopiert alle Dateien des Windowssystem auf CD und von dort ins Linuxsystem, kann es passieren, dass alle Dateien groß geschrieben sind. Wie kann man das mittels eines Scriptes ändern? Man kann folgendes Script benutzen:

for i in * ; do mv -i "$i” $(echo "$i” |tr '[A-ZÄÖÜ]' '[a-zäöü]') ; done

Erklärung dazu:

- Schleife über alle Dateinamen als Variable $i des aktuellen Verzeichnis. mv = umbenennen
- -i falls 2 Dateien existieren , die sich nur in der Klein und Großschreibung unterscheiden dann nachfragen statt überschreiben
- $(...) : Ersetze den Ausdruck durch die Ausgabe des/der Befehls/e in den runden Klammern
- echo $i gibt Inhalt der Variablen i aus und leite es weiter (| = Pipe) an tr
- tr = Austausch der Buchstaben in der jeweils korrespondierenden Position der 2 Argumente
  - also alle großen in die jeweiligen kleinen Buchstaben

done = Abschluss der Schleife

Vorsicht: Probieren Sie diesen Befehl zunächst in einem unwichtigen Verzeichnis aus. Verzeichnisnamen werden nicht geändert.

Der '*' als Wildcard in der 'for'-Anweisung ersetzt natürlich alle Dateien im aktuellen Verzeichnis. Stattdessen können auch andere Ersetzungen, wie z.B: '*.dat' benutzt werden.

Der Parameter '-i' des mv-Befehls verhindert ein versehentliches Überschreiben, wenn beim Konvertieren doppelte Dateinamen entstehen. Die Ersetzung der deutschen Umlaute oder des ß funktioniert allerdings so nicht auf jedem UNIX-Derivat. Manchmal kann man stattdessen Oktalzahlen oder 'Ae' usw. als solches benutzen. Eigentlich sollte man Umlaute in Dateinamen aber sowieso nicht verwenden ;). Selbstverständlich ist es angebracht, dass man das o.a. Kommando bei häufigem Gebrauch in ein Shellskript packt. Statt '*' wird dann '$*' dort eingetragen, damit Dateinamen als Argument an das Script übergeben werden können.

Statt tr '[A-ZÄÖÜ]' '[a-zäöü]' können für diesen Fall auch direkt 2 spezielle Parameter von tr verwendet werden: tr '[:upper:]' '[:lower:]'

5 Literatur zum Thema Shellprogrammierung

- Shell-Programmierung Einführung, Praxis, Referenz; Galileo Computing 782 S., 2005, geb., mit CD 44,90 Euro, ISBN 3-89842-683-1
  - Bourne-, Korn- und Bourne-Again-Shell (Bash)
  - Inkl. grep, sed und awk
  - von Jürgen Wolf
- Gancarz, Mike. The UNIX Philosophy. Digital Press, 1995.
  - “Die Neun Programmiergrundsätz von Cancarz”
- Newham, Cameron. Learning the bash Shell : [covers bash 3.0] /. 3. ed. UNIX Shell programming. Beijing ; Köln [u.a.] :: O’Reilly,, 2005.
- Levithan, Steven; Goyvaerts. Reguläre Ausdrücke Kochbuch /. Köln: O’Reilly, 2010
  - über sog. Reguläre Ausdrücke zum Programmieren mit Suchmustern
- Sobell, Mark G. ; A practical guide to Linux commands, editors, and shell programming. Upper Saddle River, N.J.: Prentice Hall, c2013.

6 Links zum Thema Shellprogrammierung

- http://openbook.galileocomputing.de/unix_guru/node303.html
  - Sehr einfache Beispiele, gut erklärt.
- http://www.selflinux.de/selflinux/html/shellprogrammierung.html (Bourne-kompatible)
  - Basiert auf einer alten Version des Textes von Ronald Schaten: http://www.schatenseite.de/unixshell.html
- http://www.freeos.com/guides/lsst/index.html *(engl.)*
- http://bash.cyberciti.biz/guide/Main_Page *(engl.)*
  - Shell Scripting Tutorial
- http://tldp.org/LDP/abs/html/index.html *(engl.)*
- „Bash Reference Manual“. Zugegriffen 30. Dezember 2013. http://www.gnu.org/software/bash/manual/bashref.html.
- „/bin/bash - Komfortfunktionen in der Bash“. Zugegriffen 30. Dezember 2013. http://www.bin-bash.de/komfort.php.
- „Returning Values from Bash Functions | Linux Journal“. Zugegriffen 30. Dezember 2013. http://www.linuxjournal.com/content/return-values-bash-functions.

↑ Inhaltsverzeichnis ↑

[1] http://mywiki.wooledge.org/BashFAQ/031

7 Text- und Bildquellen, Autoren und Lizenzen

7.1 Text

- **Linux-Praxisbuch: Shellprogrammierung** *Quelle:* https://de.wikibooks.org/wiki/Linux-Praxisbuch%3A_Shellprogrammierung?oldid=807407 *Autoren:* Mkleine, A.Heidemann, Jan~dewikibooks, Gneer~dewikibooks, Bodhi-Baum, E^(nix), ThePacker, Gronau~dewikibooks, Shelmton, Yuuki Mayuki, Hilefoks, Guenter, Stefan Majewsky, LivingShadow, FeG, Günter Nowak, Prog, Cinhtau, Matthias M., Itu, WissensDürster, Christoph Franzen, Steavor, Juetho, Peter Littmann, IP-Sichter, BuSchu, Hanskell, Philip1991~dewikibooks, Buchfreund~dewikibooks, Mr N, Qwertz84, Ludki und Anonyme: 86

7.2 Bilder

7.3 Inhaltslizenz

- Creative Commons Attribution-Share Alike 3.0