Erich Ehses • Lutz Köhler • Petra Riemer
Horst Stenzel • Frank Victor

Betriebssysteme

Ein Lehrbuch mit Übungen zur Systemprogrammierung in UNIX/Linux

ein Imprint von Pearson Education
München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Script-Programmierung in UNIX/Linux

3

ÜBERBLICK

3.1 Was versteht man unter Script-Programmen?

Script-Programme ergänzen das Konzept von UNIX, seine Benutzerschnittstelle durch das Bereitstellen von vielen, insbesondere *kombinierbaren Hilfsprogrammen* zu bilden. Damit steht deren Aufruf und Verbindung im Vordergrund.

- Bei *Filtern* dient der Ausgabestrom des einen als Eingabe des nächsten.
- Bei *Systemprogrammen* werden die Rückgabewerte und Errorcodes weiterbenutzt.

Irgendwann gelangt man an den Punkt, wo Kontrollstrukturen und programmiersprachliche Konstrukte benötigt werden. Hier setzen die Script-Programme an, die interpretiert werden und daher eine einfache Möglichkeit der Programmierung bieten.

Definition Scripte sind Textdateien, die interpretierbaren Progammcode in einer der Sprachen der Kommandointerpreter, der Bourne-Shell `sh` oder der `csh`, enthalten.

Ihre Ausführbarkeit wird mit dem *Execute-Bit* der Zugriffsrechtemaske angedeutet. Der dazugehörende *Interpeter* wird in der ersten Zeile eines Scripts festgelegt, und zwar als Kommentar: `#!/bin/sh` oder `#!/bin/csh`

Die verwendbaren Programmiersprachen sind imperativ, kennen *Variablen*, *Kontrollstrukturen*, *Prozeduren oder Funktionen* und nutzen den *Aufruf anderer Programme* des Betriebssystems. Die Sprache der `csh` ist sehr C-ähnlich. Natürlich können sie auch mit *Optionen und Argumenten* versehen werden. Um diese einfach und einem gewissen Standard folgend zu verarbeiten, bietet das Betriebssystem das Hilfsprogramm `getopt` an.

Das *Einsatzgebiet* von Scripten ist vielfältig. Sie kontrollieren wichtige Systemvorgänge: das Hochfahren des gesamten UNIX-Systems, das Starten und Stoppen essentieller Dienste – wie zum Beispiel den des Webservers – und sehr häufig das Konfigurieren und Installieren von Programmpaketen. Da gerade dort die Notwendigkeit zur Anpassung häufig auftritt, sollte man sich damit auskennen.

3.2 Programmierung von UNIX/Linux-Scripten

In diesem Abschnitt widmen wir uns den Programmiermöglichkeiten, die uns der Kommandointerpreter `sh` bietet. Dies geschieht vor dem Hintergrund, dass fast alle Script-Dateien, die zum Betriebssystem gehören, in der Sprache der Bourne-Shell verfasst sind. Um diese zu verstehen oder zu verändern, ist das Beherrschen der Programmiersprache der `sh` notwendig. Der damit verbundene Minimalismus, das Beschränken auf die einfache `sh`, verschafft dem System Effizienz und eine enorme Stabilität. Im Folgenden verwenden wir einige Beispiele aus [Kernighan et. al. 1986], die wir modifiziert haben.

Erzeugen eigener Kommandos

Angenommen, ein Systemadministrator führt sehr oft das folgende Kommando aus, um zu sehen, wie viele User angemeldet sind:

```
$ who | wc -l              # -l steht für lines
```

Es wäre sinnvoll, wenn man hierfür ein eigenes Kommando `anz` für *Anzahl der User* hätte. Als Erstes erzeugt man dazu eine Datei mit dem Namen `anz`, die dieses Kommando enthält.

Wie wir schon am Anfang festgestellt haben, ist die Shell ein gewöhnliches Programm. In unserem Fall verwenden wir `sh`. Also können wir auch die Eingabe für dieses Programm umleiten. Damit können wir unsere Datei `anz` folgendermaßen ausführen:

```
$ sh < anz
```

Es gibt jedoch einen einfacheren Weg. Man kann die Datei *ausführbar machen* und dann akzeptiert die Shell die Datei als Kommando:

```
$ chmod +x anz
$ anz
```

In diesem Fall nennt man die Datei ein *Shell-Script*. Shell-Scripte werden als eigene Prozesse gestartet, d. h. die Shell glaubt, das folgende Kommando sei ausgeführt worden:

```
$ sh anz
```

Diesen Prozess nennt man daher auch *Sub-Shell*.

Unser Shell-Script `anz` funktioniert allerdings nur, wenn sich die Datei in unserem Arbeitsverzeichnis befindet. Wollen wir jedoch `anz` in jedem Directory ausführen können, müssen wir die Datei in unser privates `bin`-Directory verlagern [vgl. Kernighan et. al. 1986]:

```
$ pwd
/usr/koehler
$ mkdir bin                # bin erzeugen
$ echo $PATH
:/usr/koehler/bin:/bin:/usr/bin
$ mv anz bin               # anz nach bin verlagern
$ ls anz
anz not found              # anz ist wirklich nicht mehr im
                           # Arbeitsverzeichnis
$ anz                      # die Shell findet es trotzdem
10
$
```

`PATH` wird in `.profile` gesetzt.

Ein anderes Beispiel für ein praktisches Shellscript ist das folgende: `werMachtWas` (`who` und `ps -a`).

Parameter für Shell-Scripte

Lassen Sie uns ein Shell-Script konstruieren, das Programme ausführbar macht. Wir nennen es `exe`. Damit ist

```
$ exe anz
```

eine Abkürzung für:

```
$ chmod +x anz
```

Als Erstes brauchen wir eine Datei mit dem Namen `exe`, die das obige enthält. Aber wie behandeln wir die Eingabe für dieses Programm, d. h. geben die Datei an, die ausführbar gemacht werden soll?

Hierzu gibt es die Parameter für Shell-Scripte:

Definition Parameter für Shell-Scripte

- `$1` wird überall im Script durch das erste Argument ersetzt,
- `$2` durch das zweite
- usw.

Die Datei `exe` sieht dann so aus:

```
chmod +x $1
```

Schließlich führen wir das folgende Kommando aus:

```
$ exe anz
```

Betrachten wir die ganze Folge von Kommandos zum Erreichen unseres Ziels:

```
$ echo "chmod +x $1" > exe          # Datei exe mit dem Inhalt chmod und
                                    # x $1 erzeugen
$ sh exe exe                        # exe selbst ausführbar machen
$ echo echo Hier ist marc > hallo   # Testprogramm erzeugen
$ hallo
hallo: cannot execute
$ exe hallo
$ hallo
Hier ist marc
$ mv exe /usr/koehler/bin
$ rm hallo
```

Soll das Kommando `exe` für mehrere Dateien funktionieren, so würde man Folgendes schreiben:

```
chmod +x $1 $2 $3
```

Dies ist aber nur bis `$9` möglich, denn `$10` wird als `$1` gefolgt von `0` betrachtet. Jedenfalls, wenn wir `$1` bis `$9` angeben, geht es mit einer beliebigen Anzahl von Argumenten bis 9. Aber dies ist unsauber. Was machen wir, wenn wir mehr als zehn Argumente angeben wollen? In diesem Fall geben wir Folgendes an:

```
chmod +x $*
```

`$*` steht also für beliebig viele Argumente.

Das folgende Script zählt die Anzahl von Zeilen in Dateien:

```
$ cat lc
# lc (line count)
wc -l $*
$ lc /usr/koehler/bin/*
    1 /usr/koehler/bin/exe
    2 /usr/koehler/bin/lc
    usw.
```

Noch eines wollen wir nachtragen: `$0` bezeichnet das Programm, das gerade ausgeführt wird.

Als Beispiele [aus Kernighan et. al. 1986] geben wir Programme mit den Namen 2, 3, 4, usw. an, die dazu dienen sollen, die Ausgabe auf dem Bildschirm in 2, 3, 4 usw. Spalten anzuzeigen. Wir implementieren lediglich die Datei 2 und geben dann Links auf diese Datei an. Es handelt sich hier also um ein und dasselbe Programm mit den Namen 2, 3, 4, usw. Dies erledigen wir mit dem Kommando `ln`:

```
$ cat 2                     # Datei in 2, 3, 4 usw. Spalten ausgeben
pr - $0 -t -l1 $*           # -$0 gibt die Anzahl der Spalten für pr an.
                            # -t unterdrückt die Überschrift.
                            # -l1 definiert die Länge einer Seite als 1.
$ ln 2 3; ln 2 4; ln 2 5
$ ls /usr/koehler/bin | 4   # Anzeige des Directorys in vier Spalten.
```

Shell-Variablen

Die Zeichenfolgen `$1`, …, `$9` sind *positionelle Parameter (Shell-Variablen)*, die wir bereits kennen gelernt haben. `PATH` ist die Liste von Directories, in denen die Shell nach Kommandos sucht. `HOME` bezeichnet das Heimatdirectory.

Mit Ausnahme der positionellen Parameter können Shell-Variablen verändert werden. Mit dem Gleichheitszeichen wird einer Variablen ein Wert zugewiesen. Stellen wir ein `$` vor den Namen der Variablen, können wir auf ihren Wert lesend zugreifen.

```
$ PATH=/bin:/usr/bin        # WICHTIG: keine Blanks bei der Zuweisung erlaubt
$ PATH=$PATH:/usr/games
$ echo $PATH
/bin:/usr/bin:/usr/games
```

Sie können auch eigene Variablen definieren. Diese schreibt man meistens klein, da solche mit besonderer Bedeutung groß geschrieben werden (wie `PATH`). Die Zuweisung an die Variable benutzt auf der rechten Seite die so genannten *Backticks*, um das Kommando `pwd` zu klammern. Damit wird veranlasst, dass das Ergebnis der Ausführung des Kommandos der Variable zugewiesen wird. Ohne die Backticks würde lediglich die Zeichenkette `pwd` zum Inhalt der Variable. Im Folgenden wird mehrfach auf diese Notation zurückgegriffen, z. B. `set date` oder `echo $PATH` im Gegensatz zu `echo "Hallo"`.

Es muss darauf geachtet werden, dass exakt diese Backticks, also die von links oben nach rechts unten verlaufenden Anführungsstriche, benutzt werden.

```
$ pwd
/usr/koehler/bin
$ dir=`pwd`           # merken, wo wir sind.
$ cd /usr/meier/bin
$                     # etwas im fremden Directory arbeiten
$ cd $dir             # und zurück ins eigene Directory
$ pwd
/usr/koehler/bin
```

Mit dem UNIX-Kommando `set` zeigen Sie die Werte aller Shell-Variablen an:

```
$ set
HOME = ...
PATH = ...
...
dir = ...
```

Wenn Sie nicht alle sehen wollen, benutzen Sie am besten `echo`:

```
$ echo $dir
```

Der Wert einer Variablen gehört immer zu der Shell, in der sie erzeugt wurde. Die Werte vererben sich also nicht an Sub-Shells:

```
$ name=marc
$ sh
$ echo $name
               # name nicht bekannt
$ control-d    # Sub-Shell beenden
$ echo $name
marc
```

Hier ergibt sich also das Problem, dass ein Shell-Script die Werte von Variablen nicht ändern kann, denn es wird ja als eigener Prozess ausgeführt (Sub-Shell). Um dies zu ermöglichen, hat man das Kommando . `(Punkt)` eingeführt. Eine Datei, die mit `Punkt` ausgeführt wird, ist eigentlich kein Script und muss auch nicht ausführbar sein. Es wird einfach die Standardeingabe umgelenkt und somit werden die Kommandos der Datei so behandelt, als würden sie interaktiv eingegeben. Ursprünglich wurde das Kommando `Punkt` erfunden, um die Profile-Datei auszuführen, ohne das System verlassen zu müssen (`.profile`).

```
$ cat /usr/koehler/bin/.games
PATH=$PATH:/usr/games
$ echo $PATH
                                # PATH ohne den Zusatz /usr/games
$ .games
$ echo $PATH
                                # PATH mit dem Zusatz /usr/games
```

Es gibt noch eine bequemere Möglichkeit, Shell-Variablen in Sub-Shells verfügbar zu machen, und zwar durch *Exportieren*. Hierzu dient das Kommando `export`.

```
$ name=marc
$ export name
$ sh
$ echo $name
marc            # name ist bekannt.
$ control-d
$ echo $name
marc            # name ist immer noch bekannt.
```

Regel Exportieren Sie nur Variablen, die in *allen* Sub-Shells benutzt werden sollen (wie `HOME` oder `PATH`).

Umlenken der Ein- und Ausgabe

Betrachten wir noch einmal ein Beispiel [vgl. Kernighan et al. 1986]:

```
$ time wc kapitel3 > anzahl.out
    real 1.2
    user 0.4
    system 0.4
```

In diesem Fall erscheint die Diagnoseausgabe trotz der Verletzung auf dem Bildschirm. Möchte man dies vermeiden, so kann man schreiben:

```
$ time wc kapitel3 > anzahl.out 2>zeiten.out
$                    # Prima. Das funktioniert.
$ cat zeiten.out
    real 1.2
    user 0.4
    system 0.4
```

Möchte man die gesamte Ausgabe in eine Datei packen, so kann man Folgendes schreiben:

```
$ time wc kapitel3 > anzzeit.out 2>&1
```

Hier wird sowohl Ausgabe als auch Diagnose in die Datei umgelenkt. `2>&1` bewirkt, dass die Diagnoseausgabe in den gleichen Strom gelenkt wird wie die Standardausgabe.

Definition Umlenkungsoperatoren

`>dat`	Standardausgabe in Datei `dat`
`fd>dat`	Ausgabe von Filedeskriptor `fd` in Datei `dat`
`fd>>dat`	Ausgabe von Filedeskriptor `fd` an Datei `dat` anfügen
`>>dat`	Standardausgabe an Datei `dat` anfügen
`<dat`	Standardeingabe aus Datei `dat`
`kom1 | kom2`	Standardausgabe von Kommando `kom1` wird Standardeingabe von `kom2`

Kontrollstrukturen

Nachdem wir im vorherigen Abschnitt die grundlegenden Konzepte der UNIX-Shell besprochen haben, wenden wir uns nun der Thematik der Shellprogrammierung zu. Wir tun dies am Beispiel der Shell `sh`. Dies ist nur eine Einführung. Um Genaueres über einzelne Kommandos zu erfahren, empfehle ich Ihnen, mit dem Kommando `man sh` das UNIX-Manual zu lesen.

Die for-Anweisung

Die Syntax der *for-Anweisung* lautet:

Definition Syntax der `for`-Anweisung

```
for <variable> in <Liste von Worten>
do
<Anweisungen>
done
```

Als Beispiel geben wir eine Schleife an, die Dateinamen ausgibt:

```
for i in *
do
   echo $i
done
```

Sehr häufig wird als Laufvariable `i` verwendet (es gibt übrigens keine Variablendeklarationen). Mit `*` sprechen wir im obigen Beispiel alle Dateien des aktuellen Directorys an.

Das nächste Beispiel zeigt, wie wir Dateien vergleichen können. Und zwar nehmen wir an, wir haben unser Vorlesungsmanuskript in Form von mehreren Dateien im aktuellen Directory und wollen dies mit den Dateien in einem Directory mit dem Namen `alteVersion` vergleichen (aus [Kernigahn et al. 1986]):

```
for i in vorles.*          # vorles.1 vorles.2 usw.
do
   echo $i
   diff -b alteVersion/$i $i
   echo                    # neue Zeile wegen besserer Lesbarkeit
done | pr -h "Differenz zwischen `pwd`/alteVersion `pwd`" | lpr &
```

In diesem Fall erzeugen wir einen Seitentitel mit der Option `-h`. Außerdem erzeugen wir einen Hintergrundprozess mittels `&`, wobei dies sich auf die ganze Schleife bezieht. Der Vorteil ist, dass wir nicht auf jeden einzelnen Vergleich warten müssen, sondern das Ergebnis in einem Lauf erhalten.

Zur Syntax ist noch zu sagen, dass `do` und `done` nur erkannt werden, wenn sie gleich nach einem Zeilentrenner oder einem Semikolon auftreten. Wir könnten also kürzer schreiben:

```
for <variable> in <Liste von Worten>; do <Anweisungen>; done
```

Aufpassen müssen Sie, wenn Sie innerhalb einer Schleife Anweisungen verwenden, die selbst wieder eine Schleifenverarbeitung erlauben, wie zum Beispiel:

```
chmod +w $*   # die Dateien des Directorys
              # als Argumente eines Shellscripts
```

Falls Sie in diesem Fall Folgendes schreiben:

```
for i in $*
do
   chmod +w $i
done
```

verlieren Sie unnötig Zeit. Besser wäre in diesem Fall nur `chmod +w $*` anstatt der Schleife zu schreiben.

Zum Schluss dieses Abschnitts noch ein größeres Beispiel aus [Kernighan et al. 1986]: Wir nehmen an, Sie möchten einem Freund oder einer Freundin Ihre geschriebenen Shell-Scripts über E-Mail zukommen lassen. Am einfachsten ist es, wenn Sie alle Dateien Ihres Directorys zum Beispiel `/usr/koehler/bin` aneinander hängen und per E-Mail verschicken. Unsere erste Lösung könnte also lauten:

```
$ cd /usr/koehler/bin
$ cat dateienVersenden              # Diese Datei dann ausführen.
for i in *
do
   echo Hi! Hier kommt Datei $i :
   cat $i
done | mail ia4711@advm2
```

Was ist schlecht an dieser Lösung? Nun ja, für den Sender ist sie akzeptabel, aber für den Empfänger nicht, denn er muss alles mit einem Editor auseinander pflücken. Schöner wäre es, wenn wir ein Script verschicken würden, das in der Lage ist, die Dateien selbst auszupacken. Hierzu benötigen wir jedoch noch einen speziellen Mechanismus, und zwar so genannte *Here-Dokumente*.

Exkurs (Here-Dokumente) Dieser Mechanismus sorgt dafür, dass man die Standardeingabe für ein Kommando zusammen mit dem Kommando in eine Datei schreiben kann. Man braucht also keine zweite Datei und alles steht schön zusammen in einem Script. Betrachten wir dazu das folgende Telefonauskunftsprogramm:

```
$ cat 01188
grep "$*" << ENDE   # von hier an bis zum Wort ENDE ist alles
frechen 02234       # die Standardeingabe
bonn 0228
gummersbach 02261
koeln 0221
ENDE
$
```

Zurück zu unserem Shellscript zum Versenden von Dateien (aus [Kernighan et al. 1986]):

```
$ cat dateienVersenden
# Dateien zusammenpacken und versenden

echo "# Zum Auspacken, sh auf diese Datei anwenden"
for i
do
   echo "echo $i 1>&2"            # Anführungszeichen kennzeichnen
                                  # Zeichenfolgen, wobei aber $
                                  # ausgewertet wird.
   echo "cat >$i <<'Ende von $i'" # Beginn des Here-Documents
   cat $i
   echo "Ende von $i"
done
```

Nun probieren wir das zuerst einmal bei uns aus, ehe wir es verschicken:

```
$ dateienVersenden exe anz >schrott
$ cat schrott
                          # Zum Auspacken, sh auf diese Datei anwenden
echo exe 2>&1             # Wir verwenden die Diagnose-Ausgabe.
cat exe <<'Ende von exe'
chmod +x $*
Ende von exe
                          # Das gleiche folgt nun für anz.
                          # Ich liste das hier aber nicht auf!
$
```

Jetzt probieren wir aus, ob wir die Dateien auspacken können:

```
$ mkdir test
$ cd test
```

```
$ sh ../schrott   # Shell anwenden.
exe
anz
$ ls
exe
anz
$ cat exe #       Schön! Funktioniert also!
chmod +x $*
```

Das letzte Shell-Script ist sehr interessant, weil es selbst ein Programm generiert, Gebrauch macht von der Ein-/Ausgabeumlenkung und Here-Dokumente benutzt. Die Kürze des Programms ist beeindruckend.

Die case-Anweisung

Definition Syntax der `case`-Anweisung

```
case <wort> in
<muster>)   <anweisungen>;;
<muster>)   <anweisungen>;;
...
esac
```

Es wird `<wort>` mit `<muster>` verglichen und `<anweisungen>` dort und nur dort ausgeführt, wo das erste Mal der Vergleich positiv ist.

Als Beispiel für die Anwendung von `case` wollen wir das Standard-Kalenderprogramm `cal` verbessern.

```
$ cal
usage: cal [month] year
$ cal march 2005
bad argument
$ cal 3 2005              # Wir sehen den Monatskalender für 03/2005
```

Vier Dinge wollen wir ändern:

1. Der Monat soll als Text angegeben werden können.
2. Geben wir zwei Argumente an, so soll sich `cal` so verhalten wie das originale `cal`.
3. Mit einem Argument soll `cal` den entsprechenden Monat des laufenden Jahres oder den aktuellen Monat des eingegebenen Jahres ausgeben.
4. Wenn kein Argument angegeben wird, soll der aktuelle Monat im laufenden Jahr angezeigt werden.

Sehr wichtig ist hier, dass wir eine eigene Version von `cal` anlegen können und dies zum Beispiel in `/usr/victor/bin` ablegen. Dann wird immer die eigene Version auf-

gerufen und man braucht das originale *cal* nicht anzufassen (aus [Kernighan et al. 1986]).

```
$ cat cal
# eine angepasste Schnittstelle zu /usr/bin/cal

case $# in                      # $# gibt die Anzahl der Argumente an,
                                # mit denen cal aufgerufen wurde.
0) set `date`; m=$2; j=$6 ;;    # kein Argument, Backticks!
1) m=$1; set `date`; j=$6 ;;    # ein Argument: dieses Jahr
*) m=$1; j=$2 ;;                # zwei Argumente
esac

case $m in
jan* | Jan*) m=1 ;;
feb* | Feb*) m=2 ;;
...
dez* | Dez*) m=12 ;;
[1-9] |10|11|12) ;;             # Monatszahl
*) j=$m; m="" ;;                # nur ein Jahr
esac
/usr/bin/cal $m $j              # Original cal
```

Wir müssen nur noch etwas zu der trickreichen Anwendung von `set` und `date` sagen:

```
$ date
Wed March 19 17:30:00 EDT 2005
$ set `date`
$ echo $1
Sat
$ echo $2
March
```

`set` ist ein sehr mächtiges Kommando. Ohne Argumente zeigt es die Variablenbelegung an. Hat `set` Argumente, so werden dadurch die Variablen `$1`, `$2` usw. definiert.

Hier einmal zusammengefasst:

Definition Vordefinierte *Shell-Variablen*

`$#`	Anzahl der Argumente
`$*`	Alle Argumente für die Shell
`$?`	Resultatwert des letzten Kommandos
`$$`	Prozessnummer der Shell
`$!`	Prozessnummer des letzten Kommandos mit &
`$PS1`	Erstes Promptzeichen
`$PS2`	Zweites Promptzeichen

Die if-Anweisung

Definition Syntax der `if`-Anweisung

```
if <anweisung>
then
    <anweisungen>;;
else                  # der else-Teil kann auch entfallen
    <anweisungen>;;
fi
```

Jede Anweisung liefert einen `exit`-*Code*:

0	erfolgreich
Wert ungleich 0	nicht erfolgreich

Achtung: Dies ist genau umgekehrt zu C und C++!

In diesem Zusammenhang sollten wir das Kommando `test` erwähnen, dessen einzige Aufgabe es ist, einen `exit`-Code zu liefern. Das Kommando gibt nichts aus und ändert keine Dateien. Es können verschiedene Flags eingesetzt werden:

`test -r datei`	existiert `datei` und ist sie lesbar?
`test -f datei`	existiert `datei` und ist sie kein Directory?

usw. (vgl. hierzu `man test`).

Wie oben schon erwähnt, können Sie sich den `exit`-Code des letzten Kommandos mit

```
$ echo $?
```

anschauen.

Normalerweise ist `case` schneller als `if`, da bei `case` im ersten Fall nur Mustervergleiche durchgeführt werden, während bei `if` im zweiten Fall immer eine Anweisung ausgeführt wird, um einen Wahrheitswert zu erhalten. Daher wird in Shell-Scripts meist `case` dort bevorzugt, wo wir in C, C++ oder Java ein `if` verwenden würden.

Wir wollen uns nun als Beispiel für `if` das Kommando `which` anschauen, was dazu dient, herauszufinden, welche Datei einem Kommando entspricht. Um `which` zu implementieren, müssen wir die Variable `PATH` untersuchen. Und zwar schauen wir für jeden Directory-Eintrag nach, ob wir dort eine Datei mit dem gleichen Namen wie unser Kommando finden (aus [Kernighan et al. 1986]).

```
$ cat which
# Welches Kommando in PATH wird ausgeführt?

case $# in
0)   echo "usage: which command" 1>&2; exit 2
esac
```

```
for i in `echo $PATH | sed s/:/\ /g`      # dies ist nervig. Wir
                                          # formulieren es nicht aus.
                                          # Hier soll lediglich aus PATH
                                          # durch Ersetzen eine Liste von
                                          # Directory-Namen entstehen.
do
   if test -f $i/$1
   then
   echo $i/$1
   exit 0                                 # erfolgreich. Pfad ausgeben.
   fi
done
exit 1                                    # nicht gefunden.
```

Testen wir nun das Ganze:

```
$ exe which                    # ausführbar machen
$ which ed
/bin/ed
$ mv which /usr/koehler/bin
$ which which
/usr/koehler/bin/which
```

Weitere Anweisungen

Sicherlich wird `for` als Schleife am meisten verwendet. Es gibt jedoch noch `while` und `until`. Mit diesen Konstrukten lässt sich gut programmieren, was passieren soll, wenn ein bestimmtes Ereignis eintritt.

Definition Syntax der `while`-Anweisung

```
while <anweisung>
do
    <anweisungen>;; # ausgeführt, solange <anweisung> wahr liefert
done
```

Definition Syntax der `until`-Anweisung

```
until <anweisung>
do
    <anweisungen>;; # ausgeführt, solange <anweisung> falsch liefert
done # <
```

Als Beispiel geben wir ein Programm an, das beobachtet, ob sich jemand anmeldet:

```
while sleep 60
do
    who | grep marc
done
```

Diese Version hat einige *Nachteile*:

- Wenn *Marc* angemeldet ist, muss man eine Minute warten, ehe man es erfährt.
- Bleibt *Marc* angemeldet, bekommt man die Meldung jede Minute.

Formulieren wir die ganze Sache besser:

```
until who | grep marc
do
    sleep 60
done
```

Es gilt hier: Wenn *Lutz* angemeldet ist, erhalten wir das Ergebnis sofort.
Verpacken wir dies nun als eigenes Kommando:

```
$ cat watchfor
# Warten bis sich jemand anmeldet

case $# in
0)   echo "usage: watchfor person" 1>&2; exit 1
esac

until who | grep $1
do
    sleep 60
done
```

Zum Schluss dieses Abschnitts geben wir noch ein Script aus [Kernighan et al. 1986] an, das beobachtet, wie sich die Anmeldesituation jede Minute ändert. Wir führen jede Minute `who` aus und berichten über die Unterschiede zu vorher. Die Ausgabe von `who` bewahren wir in einer Datei in `tmp` auf. Um unsere Dateien von anderen zu unterscheiden, machen wir `$$` (die Prozessnummer des Shell-Kommandos) zum Bestandteil des Dateinamens. Dies ist ein *übliches Verfahren und kommt oft vor*!

```
$ cat watchfor
# Beobachten, wer sich an- und abmeldet

new=/tmp/watchfor1.$$
old=/tmp/watchfor2.

>$old                  # leere Datei anlegen
```

```
while true              # Endloschleife: true ist eine Anweisung,
                        # die immer true liefert. Man kann
                        # auch : verwenden.
do
who > $new
diff $old $new
mv $new $old
sleep 60
done | awk '    />/ { $1 = "in: "; print }       # dient zur schönen
                /</ { $1 = "out: "; print }'     # Darstellung der Ausgabe
```

Ändert man `> $old` in `who > $old` ab, so bekommt man wirklich nur die Änderungen und nicht alle Benutzer, die angemeldet sind.

Eine Anwendung, die ähnlich wie `watchfor` funktioniert, kennen Sie: Sie heißt: *You have new mail!*

3.3 Praxisbeispiele

Standardisierte Parameterübergabe

Alle UNIX-Programme kennen per Konvention die Angabe und Annahme von Optionen und deren Argumenten in der Form `-x wert`, dem Bindestrich gefolgt von einem repräsentativen Zeichen und einem Argumentwert. Zur einfacheren Weiterverarbeitung sollte das Hilfsprogramm `getopt` benutzt werden, da es gleichzeitig die Einhaltung der Syntax und die Zulässigkeit der gewählten Optionen überprüft.

Definition Das *Hilfsprogramm* `getopt` *– parse command options*

- Die Utility `getopt` wird benutzt, um beim Aufruf von Shell-Scripten deren Optionen aus der Kommandozeileneingabe herauszufiltern. Ein Test auf zulässige Optionen ist eingeschlossen.
- Die Parameter bedeuten:
  - `args` Options- und Argumentfolge `$1 ... $n`, durch `set -- $args` zugewiesen.
  - `optstring` Zeichenkette. Enthält die zulässigen Optionszeichen. Folgt ein ‚:', besitzt die Option ein Argument (mit oder ohne Leerzeichen anzugeben).
  - `$*` Kommandozeileneingabe beim Shell-Scriptaufruf.
- SYNOPSIS

```
args=`getopt optstring $*` ; errcode=$?; set -- $args
```

Das folgende Programmfragment demonstriert, wie die Argumente eines Kommandos verarbeitet werden. In diesem Fall sind die Optionen `-a` und `-b` sowie `-o` mit einem Argument erlaubt, neben weiteren Argumenten `file`. Folgende Programmaufrufe könnten so realisiert werden:

```
cmd -aoarg file file
cmd -a -o arg file file
cmd -oarg -a file file
cmd -a -oarg -- file file
```

Auch die typische `usage`-Ausgabe fehlerhafter Angaben wird hier berücksichtigt.

Das Programm beginnt mit dem „Parsen“ der Kommandozeile hinsichtlich der korrekten Syntax. Der Aufruf von `getopt` untersucht die Zeichenkette `$*` auf das Vorkommen von Optionen aus der möglichen Optionsmenge `abo`.

Beispiel 1

Verarbeiten von Optionsangaben in der Kommandozeile

```
args=`getopt abo: $*`

   if [ $? != 0 ]
   then
           echo 'Usage: ...'
           exit 2
   fi
   set -- $args

   for i
   do
           case "$i"
           in
                   -a|-b)
                           echo flag $i set; sflags="${i#-}$sflags";
                           shift;;
                   -o)
                           echo oarg is "'"$2"'"; oarg="$2"; shift;
                           shift;;
                   --)
                           shift; break;;
           esac
   done
   echo single-char flags: "'"$sflags"'"
   echo oarg is "'"$oarg"'"
```

Mit der Abfrage des Ausführungsstatus in `$?` können fehlerhafte Angaben bemerkt werden.

Sind diese ausgeschlossen, werden die Optionen und Argumente mit `set` aus der Variablen `$args` gelesen und den einzelnen *Shell-Argumenten* `$1, $2` ... zugewiesen, jeweils angeführt mit einem Strich. Das `getopt`-Programm garantiert, dass nach den Optionen und deren Argumenten in der darauffolgenden Argumentvariablen der Wert `--` vorgefunden wird.

So kann mit einer `for`-*Schleife* in Verbindung mit `shift`, einem `case` und `break` die Menge der Optionen durchgegangen werden, bis die Kennung `--` auftritt. Die

Fallunterscheidung im `case` wird genutzt, um die Optionen oder ihre Werte spezifisch zu verarbeiten. Hier werden sie mit `echo` zur Illustration ausgegeben und dann in den Variablen `sflags` und `oarg` gesammelt.

Startup- und Shutdownscripte

Das *Starten* und *Herunterfahren*, aber auch die *Reinitialisierung* von Diensten aller Art wird mittels Scripten realisiert. Dafür werden diesen als Argumente standardmäßig die Kennungen `start, stop, reload, reload-modules, restart, force-reload` mitgegeben. In ihnen werden dann wiederum Scripte oder Programme aufgerufen, die applikationsspezifisch sind.

Als typische Verzeichnis-Orte, an denen diese Scripte vorzufinden sind, gelten:

- `/etc/init.d`
- `/etc/rc1.d, /etc/rc2.d, /etc/rc3.d` (runlevel-spezifisch).

Wir zeigen nun das Script `apache` aus `/etc/init.d`, das sich auf das Script `apachectl` aus dem Installationsverzeichnis `/usr/local/apache` und dort aus `bin` bezieht. Letzteres kennt nicht alle und auch andere Kennungen, so dass hier ein Übersetzungsvorgang stattfindet.

Zuerst werden bei solchen Scripten nach der Wahl des Kommandointerpreters und einem selbstverständlichen Kommentar mit beschreibendem Text einige Variablen initialisiert. Dazu gehört das `PIDFILE`. In ihm ist die Prozess-Identifikationsnummer gespeichert: das macht `apachectl` selbst. Existiert es, kann davon ausgegangen werden, dass der Dienst bereits läuft.

Beispiel 2

Startup und Shutdown des Webservers Apache

```
#! /bin/bash
#
# apache Start the apache HTTP server.
#
NAME=apache
PATH=/bin:/usr/bin:/sbin:/usr/sbin:/usr/local/apache/bin
DAEMON=/usr/local/apache/bin/apache
PIDFILE=/var/run/$NAME.pid
CONF=/usr/local/apache/conf/httpd.conf
APACHECTL=/usr/local/apache/bin/apachectl

trap "" 1
export LANG=C

test -f $APACHECTL || exit 0
...
```

Der `trap`-Befehl der Shell bewirkt, dass Unterbrechungen ignoriert werden: hier das Signal `SIGHUP`. Der `export`-Befehl gehört auch zu den *builtins*, den eigenen Befehlen

der Shell. Er stellt die daraufhin definierte Variable mit ihrem Wert in weiteren Sub-Shells zur Verfügung. Jeder weitere Script- oder Programmaufruf erhält seine eigene Sub-Shell zu seiner Ausführung.

Beispiel 3

Startup und Shutdown des Webservers Apache (Fortführung)

```
...
case "$1" in
   start)
   echo -ne "Starting web server: $NAME.\n"
   $APACHECTL start
   ;;
stop)
   echo -ne "Stopping web server: $NAME.\n"
   $APACHECTL stop
   ;;
reload)
   echo -ne "Reloading $NAME configuration.\n"
   $APACHECTL graceful
   ;;
reload-modules)
   echo -ne "Reloading $NAME modules.\n"
   if [ -f $PIDFILE ]
   then
      $APACHECTL stop
      sleep 4
   fi
   $APACHECTL start
   ;;
restart)
   $0 reload-modules
   ;;
force-reload)
   $0 reload-modules
   ;;
*)
   echo "Usage: /etc/init.d/$NAME {start|stop|reload|reload-
modules|force-reload|restart}"
   exit 1
   ;;
esac
exit 0
```

Das nachfolgende `case`-Statement bewirkt je nach erstem Argument einen adäquaten Aufruf des eigentlichen Steuerscripts `apachectl` des Webservers. Im Falle von `reload-modules` muss ein Neustart erfolgen. Wird aufgrund der Existenz des `PIDFILES` ein Aktivsein des Dienstes festgestellt, wird dieser gestoppt, eine Weile gewartet und dann

der Neustart veranlasst. Genauso verfährt man bei `restart` und `force-reload`, wobei sich das Script dann rekursiv aufruft – veranlasst durch den Eintrag `$0` stellvertretend für den ursprünglichen Programmaufruf, nun allerdings mit neuem Argument. Eine *Usage*-Klausel vervollständigt das Script.

ZUSAMMENFASSUNG

Shell-Scripte begegnen uns an vielen Stellen des Betriebssystems und sind integraler Bestandteil von UNIX/Linux. Mit ihnen werden alle möglichen Dienste Runlevel-spezifisch, also zum richtigen Zeitpunkt, gestartet oder gestoppt. Der Administrator hat überdies die Möglichkeit zu reinitialisieren (*reload*). Auch im Bereich der – teilweise automatischen – Konfiguration von Softwarepaketen werden Shell-Scripte häufig eingesetzt (*configure*).

Es gibt unterschiedliche Shells, die Kommandointerpreter-Aufgaben übernehmen. Für die traditionelle UNIX-Script-Programmierung wird vorzugsweise die Bourne-Shell genutzt.

Ihr Sprachumfang bietet Variablen, Parameter, keine Typunterstützung, aber Funktionen und die gängigen Kontrollstrukturen. Eine standardisierte Verarbeitung von Optionen und Kommandozeilenargumenten wird mit Hilfe des Hilfsprogramms `getopt` gewährleistet.

Datenströme zwischen Script-Komponenten werden leicht über die Konzepte der Umlenkung, Pipeline-Verbindung oder auch des Here-Dokuments verwirklicht.

ZUSAMMENFASSUNG

3.4 Übungen

1 Schreiben Sie ein Shell-Script mit dem Namen `myquestion`. Dieses Script hat als Parameter den Text einer Frage, die mit *Ja* oder *Nein* beantwortet werden kann. Antwortet der Benutzer mit *Ja*, so liefert das Script den Rückgabewert 0, bei *Nein* wird 1 geliefert, andernfalls erscheint die Frage noch einmal.

2 Erstellen Sie ein Shell-Script `mydelete`, das als Parameter eine Liste von Dateien hat. Diese Dateien sollen nun gelöscht werden. Das Shell-Script fragt aber für jede Datei, ob sie wirklich gelöscht werden soll. Nur bei der Antwort *Ja* wird sie gelöscht.

Hinweis: Sie sollen hier das Shell-Script `myquestion` aufrufen.

3 Schreiben Sie ein Shell-Script `chatsituation`, das für einen Chat-Room jede Minute ausgibt, wie sich die Anmeldesituation verändert hat und wie viele Personen sich im Raum befinden. Nehmen Sie an, alle, die sich auf unserem UNIX-Rechner befinden, sind in diesem Chat-Room.

Tipp: `who` benutzen.

4 Schreiben Sie ein Shell-Script, mit dem Sie eine Datei löschen können. Die zu löschende Datei soll als Parameter beim Aufruf mitgegeben werden. Legen Sie noch ein zweites Script an, in dem Sie fragen, ob die Datei wirklich gelöscht werden soll. Diese Übung soll Ihnen unter anderem zeigen, wie Scripte untereinander aufgerufen werden.

5 Schreiben Sie ein Shell-Script, das die ersten n Files eines Ordners löscht. Das Verzeichnis und die Anzahl der zu löschenden Dateien sollen als Parameter beim Aufrufen angegeben werden.

6 Schreiben Sie ein Programm, das Daten aus einem Verzeichnis von einer FTP-Adresse in ein angegebenes Verzeichnis laden kann. Sie können die Variablen als Parameter bei der Eingabe übergeben oder fest im Script eintragen. Nutzen Sie zur Übertragung `mget`.

7 Schreiben Sie ein Script, das mit Traps arbeitet. Nach dem Starten soll immer ein eigenes Prompt angezeigt werden, das alle Eingaben entgegennimmt, aber nicht ans System weiterleitet. Wenn der Benutzer „stop“ oder „Stop“ eingibt, soll das Programm beendet werden. Wird Strg + c oder Strg + z oder Strg + \ eingegeben, wird das Programm nicht beendet, es wird lediglich eine Meldung ausgegeben.

8 Schreiben Sie ein Script, das sich die Systemzeit holt, sich aus dieser die aktuelle Stunde holt und eine entsprechende Meldung ausgibt. Von 9.00 bis 11.00 Uhr wird „Guten Morgen“ ausgeben, um 12.00 Uhr „Mahlzeit“, von 13.00 bis 17.00 Uhr „Zeit für eine Pause“ und bei allem anderen „Schönen Abend und süße Träume“! Geben Sie die Meldungen in einem Intervall aus, das Sie sich aussuchen können!

9 Schreiben Sie ein Script, das die `root`-Partition nach Dateien durchsucht, die in den letzen x Tagen geändert wurden und eine Größe von y Blöcken haben (512 Byte pro Block). Die Angaben sollen beim Programmaufruf mit übergeben werden.

10 Schreiben Sie ein Script `doku`, das Ihnen wesentliche Konfigurationsdateien des UNIX-Systems formatiert mit `pr` (zum Druck) ausgibt. Darunter fallen z. B.:

```
/etc/defaultrouter (Solaris)
/etc/fstab
/etc/hosts
/etc/inetd.conf
/etc/nsswitch.conf
/etc/passwd
/etc/resolv.conf
```

Der Initialbuchstabe soll als Optionskennzeichen dienen. So erzeugt der Aufruf `doku -dfhinpr` die Auflistung aller genannten Konfigurationsdateien. Bei Angabe der Option `-h` oder einer nichtspezifizierten Option erfolgt eine Usage-Ausgabe mit detaillierter Hilfestellung zur Benutzung des Scripts. Hierzu setzen Sie (analog zum Script `apachctl`) das Konzept des Here-Dokuments ein, um die auszugebenden Teste mit in das Script zu integrieren.