4

Praktische Shellskripte

Inhalt

Lernziele

- Einige Beispiele für Shellskripte analysieren und verstehen
- Grundlegende Techniken der praktischen Shellprogrammierung kennen und anwenden können

Vorkenntnisse

- Vertrautheit mit der Kommandooberfläche von Linux
- Umgang mit Dateien und einem Texteditor
- Shell-Vorkenntnisse (etwa aus Kapitel 3 und den vorigen Kapiteln)

4.1 Shellprogrammierung in der Praxis

Die vorigen Kapitel haben große Teile der Shell-Syntax vorgestellt. Vermutlich wundern Sie sich inzwischen, was das alles soll: Erklären wir Ihnen das komplette Inventar der Küche und Speisekammer und erwarten jetzt, dass Sie ein sternverdächtiges Fünfgang-Menü kochen? Keine Angst. Programmieren lernt man nicht aus einer Syntaxbeschreibung, sondern durch das Studium vorbildlicher Programme und vor allem durch eigene Experimente. In diesem Kapitel haben wir darum einige Shellskripte für Sie zusammengetragen, an denen Sie viele gängige Techniken kennenlernen und schon Gesehenes vertiefen können; vor allem sollen diese Skripte Sie aber dazu inspirieren, selbst Hand anzulegen und die Möglichkeiten der Shell in der Praxis kennenzulernen.

4.2 Rund um die Benutzerdatenbank

Benutzerdatei

Auf »normalen« Linux-Systemen sind die Informationen über Benutzer – Benutzernamen und -IDs, primäre Gruppe, bürgerlicher Name, Heimatverzeichnis und so weiter – in der Datei `/etc/passwd` abgelegt.[1] Hier ein Beispiel zur Erinnerung:

```
tux:x:1000:100:Tux der Pinguin:/home/tux:/bin/bash
```

Der Benutzer `tux` hat also die UID 1000 und als primäre Gruppe die mit der GID 100. Im wirklichen Leben heißt er »Tux der Pinguin«, sein Heimatverzeichnis ist `/home/tux` und seine Login-Shell ist die Bash.

Gruppendatei

Die Gruppendatei `/etc/group` enthält für jede Gruppe den Namen, ein optionales Kennwort, die GID und eine Mitgliederliste, die üblicherweise nur diejenigen Benutzer aufführt, die die betreffende Gruppe als zusätzliche Gruppe haben:

```
users:x:100:
pinguine:x:101:tux
```

Wer ist in dieser Gruppe? Unser erstes Skript soll zu einem Gruppennamen alle Benutzer aufzählen, die diese Gruppe als primäre Gruppe führen. Die Herausforderung besteht darin, dass in `/etc/passwd` nur die GID der primären Gruppe steht und wir uns diese also zuerst aus `/etc/group` anhand des Gruppennamens holen müssen. Unser Skript übernimmt die gefragte Gruppe als Kommandoparameter.

```
#!/bin/bash
# pgroup -- erste Version

# Hole die GID aus /etc/group
gid=$(grep "^$1:" /etc/group | cut -d: -f3)

# Suche nach Benutzern mit der betreffenden GID in /etc/passwd
grep "^[^:]*:[^:]*:[^:]*:$gid:" /etc/passwd | cut -d: -f1
```

Beachten Sie die Verwendung von `grep` zum Finden der richtigen Zeile und von `cut` zum Extrahieren der passenden Spalte in dieser Zeile. Im zweiten `grep` ist der Suchausdruck etwas umständlich, aber wir müssen aufpassen, dass wir nicht irrtümlich eine UID selektieren, die aussieht wie unsere gesuchte GID – darum achten wir darauf, erst mal drei Doppelpunkte von links abzuzählen und dann nach der GID zu schauen.

Fehler abfangen

Bei der Programmierung von Shellskripten gilt das Gleiche wie bei der Programmierung überhaupt: Der meiste Aufwand geht in das Abfangen von Benutzer-

[1] Unter nicht ganz normalen Linux-Systemen greifen Methoden wie LDAP zur Speicherung für Benutzerdaten um sich. Sollten Sie ein solches System vor sich haben, können Sie sich in der Regel mit Kommandos wie »`getent passwd >$HOME/passwd`« und »`getent group >$HOME/group`« die Dateien besorgen, die für die folgenden Experimente nötig sind.

und anderen Fehlern. Unser Skript kümmert sich zum Beispiel weder um Probleme beim Aufrufen – Weglassen des Gruppennamens oder das Angeben zusätzlicher, überflüssiger Parameter – noch über Probleme bei der Ausführung. Es ist relativ unwahrscheinlich, dass `/etc/passwd` nicht zur Verfügung steht (das wäre Ihnen dann wohl schon anderweitig aufgefallen), aber es kann durchaus sein, dass der Benutzer des Skripts einen Gruppennamen angibt, den es nicht wirklich gibt. Aber fangen wir vorne an.

Als allererstes sollten wir uns davon überzeugen, dass das Skript mit der richtigen Anzahl Parameter aufgerufen wurde, nämlich einem. Prüfen können Sie das zum Beispiel, indem Sie den Wert von `$#` inspizieren: Syntaxprüfung

```
if [ $# -ne 1 ]
then
    echo >&2 "usage: $0 GROUP"
    exit 1
fi
```

Dieses Stückchen Shell-Code illustriert gleich einige wichtige Techniken. Ist die Anzahl der Parameter (in `$#`) nicht 1, wollen wir das Skript mit einer Fehlermeldung beenden. Die Fehlermeldung schreibt `echo`, wobei wir darauf achten, dass sie schön ordentlich nicht auf der Standard-Ausgabe, sondern auf der Standard-Fehlerausgabe erscheint (das `>&2` – 2 ist ja der Standard-Fehlerausgabe-Kanal). `$0` ist der Name, unter dem das Skript aufgerufen wurde; es ist üblich, diesen in der Fehlermeldung anzugeben, und so stimmt er immer, auch wenn die Skript-Datei umbenannt wurde. Fehlermeldung

`$0` liefert den Namen des Skripts so, wie der Aufrufer ihn angegeben hat, also möglicherweise mitsamt Überflüssigem wie einer Pfadangabe. Für Fehlermeldungen ist das nicht notwendigerweise schön; Sie können etwas verwenden wie

```
myself=${0##*/}
<<<<<
echo >&2 "usage: $myself GROUP"
```

und so die Ausgabe auf den tatsächlichen Skriptnamen beschränken.

Mit `exit` wird das Skript (vorzeitig) beendet, wobei wir als Rückgabewert eine 1 liefern, für »generischen Misserfolg«. Rückgabewert

Wenn Sie `exit` ohne Argument aufrufen oder einfach am Ende eines Shellskripts ankommen, beendet sich die Shell auch. In diesem Fall ist der Rückgabewert der Shell der Rückgabewert des letzten ausgeführten Kommandos (`exit` zählt nicht mit). Vergleichen Sie

```
$ sh -c "true; exit"; echo $?
0
$ sh -c "false; exit"; echo $?
1
```

Dann müssen wir uns natürlich noch um den Fall der nicht existierenden Gruppe kümmern. Im Abschnitt 3.4.2 haben Sie gehört, wie `grep` mit seinen Rückgabewerten umgeht: 0 bedeutet »etwas Passendes wurde gefunden«, 1 steht für »alles im Grunde OK, aber keine passende Zeile gefunden« und 2 für »irgendein Fehler ist aufgetreten« (möglicherweise war der reguläre Ausdruck nicht in Ordnung, oder beim Lesen der Eingabe ist irgendetwas schief gegangen). Das wäre schon sehr nützlich; wir könnten prüfen, ob der Rückgabewert von `grep` 1 oder 2 ist und das Skript gegebenenfalls mit einer Fehlermeldung beenden. Beim kritischen Kommando

```
#!/bin/bash
# pgroup -- verbesserte Version

# Prüfe die Parameter
if [ $# -ne 1 ]
then
    echo >&2 "usage: $0 GROUP"
    exit 1
fi

# Hole die GID aus /etc/group
gid=$(grep "^$1:" /etc/group | cut -d: -f3)
if [ -z "$gid" ]
then
    echo >&2 "$0: group $1 does not exist"
    exit 1
fi

# Suche nach Benutzern mit der betreffenden GID in /etc/passwd
grep "^[^:]*:[^:]*:[^:]*:$gid:" /etc/passwd | cut -d: -f1
```

Bild 4.1: Welche Benutzer haben eine bestimmte primäre Gruppe? (Verbesserte Version)

```
gid=$(grep "^$1:" /etc/group | cut -d: -f3)
```

Rückgabewert einer Pipeline

gibt es da nur leider ein kleines Problem: Der Rückgabewert einer Pipeline ist der Rückgabewert des *letzten* Kommandos, und das `cut` klappt grundsätzlich eigentlich immer, auch wenn das `grep` ihm eine leere Eingabe liefert (die `cut`-Argumente sind ja fundamental in Ordnung). Wir müssen uns also etwas Anderes einfallen lassen.

Was passiert aber, wenn `grep` keine passende Zeile findet? Richtig, die Ausgabe von `cut` ist leer, ganz im Gegensatz zu dem Fall, dass `grep` die Gruppe finden kann (wenn wir eine syntaktisch korrekte `/etc/group`-Datei voraussetzen, dann hat die gefundene Zeile auch eine dritte Spalte mit einer GID). Wir müssen also nur prüfen, ob unsere Variable `gid` einen »echten« Wert hat:

```
if [ -z "$gid" ]
then
    echo >&2 "$0: group $1 does not exist"
    exit 1
fi
```

(Bemerken Sie auch hier `$0` als Teil der Fehlermeldung.)

Damit ergibt sich als »vorläufig endgültige« Version unseres Skripts der Inhalt von Bild 4.1.

In welchen Gruppen ist ein Benutzer Mitglied? Unser nächstes Beispiel ist ein Skript, das die Gruppen ausgibt, in denen ein Benutzer Mitglied ist – ähnlich dem Kommando `groups`. Hierbei müssen wir beachten, dass es nicht ausreicht, nur `/etc/group` zu betrachten, denn normalerweise sind Benutzer bei ihrer primären Gruppe nicht in `/etc/group` eingetragen. Wir verfolgen also den folgenden Plan:

1. Gib den Namen der primären Gruppe des Benutzers aus
2. Gib die Namen der zusätzlichen Gruppen des Benutzers aus

Der erste Teil sollte uns leicht fallen – er ist im Prinzip das vorige Skript »andersherum«:

```
# Primäre Gruppe
gid=$(grep "^$1:" /etc/passwd | cut -d: -f4)
grep "^[^:]*:[^:]*:$gid:" /etc/group | cut -d: -f1
```

Der zweite Teil scheint eher noch einfacher zu sein: Ein simples

```
grep $1 /etc/group
```

bringt uns schon in die Nähe des Nirwana. Oder nicht? Überlegen Sie, was dieses `grep` alles finden könnte:

- Zunächst mal den Benutzernamen in der Mitgliederliste einer Gruppe. Das ist das, was wir wollen.
- Außerdem Benutzernamen in der Mitgliederliste, die den gesuchten Benutzernamen als Teilzeichenkette enthalten. Wenn wir nach `john` suchen, bekommen wir auch alle Zeilen von Gruppen, in denen der Benutzer `johnboy` Mitglied ist, `john` aber nicht. Schon falsch.
- Dasselbe Problem gilt für Benutzernamen, die Teilzeichenketten von Gruppennamen sind. Eine Gruppe `staff` hat mit einem Benutzer `taf` erst mal nichts zu tun, passt aber trotzdem.
- Ziemlich absurd, aber möglich: Der gesuchte Benutzername könnte als Teilzeichenkette eines verschlüsselten Gruppenkennworts auftreten, das nicht in `/etc/gshadow`, sondern in `/etc/group` steht (absolut erlaubt).

Auch hier ist also Sorgfalt angesagt, wie immer bei `grep` – Sie sollten sich angewöhnen, beim Erstellen von regulären Ausdrücken so bösartig und negativ zu denken, wie Sie irgend können. Dann kommen Sie bei etwas an wie

```
grep "^[^:]*:[^:]*:[^:]*:.*\<$1\>" /etc/group | cut -d: -f1
```

Das heißt, wir suchen den Benutzernamen nur im vierten Feld von `/etc/group`. Die »Wortklammern« `\<...\>` (eine Spezialität von GNU-`grep`) helfen uns gegen den `johnboy`-Fehler. Insgesamt sind wir dann bei

```
#!/bin/bash
# lsgroups -- erste Version

# Primäre Gruppe
gid=$(grep "^$1:" /etc/passwd | cut -d: -f4)
grep "^[^:]*:[^:]*:$gid:" /etc/group | cut -d: -f1

# Zusätzliche Gruppen
grep "^[^:]*:[^:]*:[^:]*:.*\<$1\>" /etc/group | cut -d: -f1
```

Probieren wir das Skript mal auf einem Debian-GNU/Linux-System aus:

```
$ ./lsgroups tux
tux
dialout
fax
voice
cdrom
floppy
<<<<<
```

```
#!/bin/bash
# lsgroups -- endgültige Version

# Primäre Gruppe
( gid=$(grep "^$1:" /etc/passwd | cut -d: -f4)
  grep "^[^:]*:[^:]*:$gid:" /etc/group | cut -d: -f1

# Zusätzliche Gruppen
  grep "^[^:]*:[^:]*:[^:]*:.*\<$1\>" /etc/group \
    | cut -d: -f1 ) | sort -u
```

Bild 4.2: In welchen Gruppen ist Benutzer *x*?

```
src
tux
scanner
```

Hierbei sollten uns zwei Dinge auffallen. Zunächst ist die Liste unsortiert, was unschön ist; ferner taucht die Gruppe `tux` in der Liste zweimal auf. (Debian GNU/Linux ist eine der Distributionen, die standardmäßig jeden Benutzer in eine eigene gleichnamige Gruppe tun.) Letzteres rührt daher, dass `/etc/group` eine Zeile der Form

```
tux:x:123:tux
```

enthält – ungewöhnlich, aber absolut erlaubt.

Wir sollten die Ausgabe also noch sortieren und dabei Dubletten entfernen (»`sort -u`«). Die Frage ist: Wie? Ein explizites »`lsgroups | sort -u`« liefert uns die gewünschte Lösung, ist aber unbequem; das Sortieren sollte Teil des Skripts sein. Dabei stören wiederum die beiden logisch getrennten Pipelines. Eine Möglichkeit, das zu beheben, wäre die Verwendung einer Zwischendatei:

```
grep … >/tmp/lsgroups.$$
grep … >>/tmp/lsgroups.$$
sort -u /tmp/lsgroups.$$
```

(das `$$` wird durch die PID der Shell ersetzt und macht den Namen der Zwischendatei eindeutig). Dieser Ansatz ist unappetitlich, da er potentiell Schrott liegenlässt, wenn aufgrund eines Fehlers die Zwischendatei am Ende des Skripts nicht gelöscht werden kann (es gibt Mittel und Wege, das zu verhindern). Außerdem stellt das Anlegen von Zwischendateien mit solch simplen Namen eine mögliche Sicherheitslücke dar. Viel bequemer ist es, die beiden Pipelines in einer expliziten gemeinsamen Subshell auszuführen und deren Ausgabe dann nach `sort` zu leiten:

```
( grep …
  grep … ) | sort -u
```

Auf diese Weise landen wir bei unserer endgültigen Version (Bild 4.2).

Übungen

4.1 [1] Ändern Sie das Skript `pgroup` so ab, dass es zwischen den Fehlersituationen »Syntaxfehler in der Eingabe« und »Gruppe existiert nicht« differenziert, indem es verschiedene Rückgabewerte liefert.

4.3 Dateioperationen

Das Automatisieren von Dateioperationen ist ein dankbares Einsatzgebiet für Shellskripte – das Verschieben, Umbenennen, Sichern von Dateien in Abhängigkeit verschiedenster Kriterien ist oft vielschichtiger, als sich das in einfachen Kommandos ausdrücken lässt. Shellskripte sind also eine bequeme Möglichkeit für Sie, Ihre eigenen Kommandos zu definieren, die genau das tun, was Sie brauchen.

Umbenennen mehrerer Dateien Das `mv`-Kommando ist nützlich, um eine Datei umzubenennen oder mehrere Dateien in ein anderes Verzeichnis zu verschieben. Was damit nicht geht, ist das Umbenennen von mehreren Dateien auf einmal, so ähnlich wie Sie das vielleicht von MS-DOS kennen:

```
C:\> REN *.TXT *.BAK
```

Das funktioniert, weil bei DOS das `REN`-Kommando selbst sich um die Bearbeitung der Suchmuster kümmert – bei Linux ist die Bearbeitung der Suchmuster dagegen Sache der Shell, die nicht weiß, was `mv` mit den Namen anschließend vor hat.

Den allgemeinen Fall der Mehrfachumbenennung per Shellskript zu lösen ist möglich, aber wir beschäftigen uns an dieser Stelle mit einer Einschränkung, nämlich dem Ändern der »Endung« einer Datei. Genauer gesagt möchten wir ein Shellskript `chext` aufstellen, das per

```
$ chext bak *.txt
```

alle aufgezählten Dateien so umbenennt, dass sie die als ersten Parameter angegebene Endung bekommen. In unserem Beispiel würden also sämtliche Dateien, deren Namen auf »`.txt`« enden, so umbenannt, dass die Namen auf »`.bak`« enden.

Die Hauptaufgabe des `chext`-Skripts ist es offenbar, die passenden Argumente für `mv` zu konstruieren. Wir müssen in der Lage sein, die Endung eines Dateinamens zu entfernen und eine andere Endung anzuhängen – und wie Sie das in der Bash realisieren können, haben Sie schon gelernt: Erinnern Sie sich an die `${…%…}`-Konstruktion bei der Variablensubstitution und betrachten Sie etwas wie

```
$ f=../a/b/c.txt; echo ${f%.*}
../a/b/c
```

Alles ab dem letzten Punkt wird entfernt.

Und daraus ergibt sich der erste Ansatz für unser `chext`-Skript:

```
#!/bin/bash
# chext -- Dateiendung ändern, erste Version

suffix="$1"
shift

for f
do
    mv "$f" "${f%.*}.$suffix"
done
```

Beachten Sie zunächst den Einsatz von Anführungszeichen, um Probleme mit Leerzeichen in Dateinamen zu vermeiden. Ebenfalls interessant ist der Umgang mit der Kommandozeile: Das erste Argument – gleich hinter dem Skriptnamen – ist die gewünschte neue Endung. Diese legen wir in der Variablen `suffix` ab und rufen anschließend das Kommando `shift` auf. `shift` sorgt dafür, dass alle Positionsparameter einen Schritt »nach links« machen: aus `$2` wird `$1`, aus `$3` wird `$2`

`shift`

und so weiter. Das alte `$1` wird verworfen. Nach unserem `shift` besteht die Kommandozeile also nur noch aus den Namen der umzubenennenden Dateien (die die Shell freundlicherweise aus allfälligen Suchmustern für uns aufgestellt hat), so dass wir bequem mit »`for f`« darüber iterieren können.

Das `mv`-Kommando sieht möglicherweise etwas ehrfurchtgebietend aus, aber es ist eigentlich nicht schwer zu verstehen. `f` ist einer unserer zu ändernden Dateinamen, und mit

```
${f%.*}.$suffix
```

wird, wie schon oben angedeutet, das alte Suffix entfernt und das neue – das in der Shellvariable `suffix` steht – textuell angehängt.

Die Sache ist natürlich nicht ganz ungefährlich, wie Sie sich leicht klar machen können, wenn Sie sich Dateinamen wie `../a.b/c` vorstellen, wo der letzte Punkt nicht in der letzten Pfadkomponente steht. Es gibt mehrere Möglichkeiten, dieses Problem zu lösen. Eine davon involviert den *stream editor*, `sed`, den wir in Kapitel 6 kennen lernen werden, eine andere benutzt die Kommandos `basename` und `dirname`, die auch in manch anderem Zusammenhang nützlich werden können. Sie dienen dazu, einen Dateinamen in einen Verzeichnis- und einen Dateianteil zu zerlegen:

```
$ dirname ../a/b.c/d.txt
../a/b.c
$ basename ../a/b.c/d.txt
d.txt
```

Sie können den Bash-Operator `%` also »entschärfen«, indem sie ihm nur den Dateianteil des zu verändernden Namens vorlegen. Das `mv`-Kommando wird dann zu etwas wie

```
d=$(dirname "$f")
b=$(basename "$f")
mv "$f" "$d/${b%.*}.$suffix"
```

(in der `${…%…}`-Konstruktion sind leider nur Variablennamen erlaubt, keine Kommandosubstitutionen).

Außerdem gehört zu einem anständigen Shellskript natürlich auch die Syntaxprüfung der Kommandozeile. Unser Skript braucht mindestens zwei Parameter – das neue Suffix und einen Dateinamen –, nach oben sind keine Grenzen gesetzt (naja fast). Bild 4.3 zeigt die fertige Version.

Übungen

4.2 [!2] Schreiben Sie ein Shellskript, das einen Dateinamen übernimmt und als Ausgabe die Namen der übergeordneten Verzeichnisse liefert, etwa so:

```
$ hierarchy /a/b/c/d.e
/a/b/c/d.e
/a/b/c
/a/b
/a
/
```

4.3 [2] Benutzen Sie das in der vorherigen Übung erstellte Skript, um ein Skript zu schreiben, das dasselbe tut wie »`mkdir -p`« – das Skript bekommt den Namen eines anzulegenden Verzeichnisses übergeben und soll außer diesem Verzeichnis auch alle möglicherweise nicht existierenden Verzeichnisse weiter oben im Dateibaum erzeugen.

```
#!/bin/bash
# chext -- Dateiendung ändern, verbesserte Version

if [ $# -lt 2 ]
then
    echo >&2 "usage: $0 SUFFIX NAME ..."
    exit 1
fi

suffix="$1"
shift

for f
do
    mv "$f" "${f%.*}.$suffix"
done
```

Bild 4.3: Massen-Suffixänderung für Dateinamen

4.4 Protokolldateien

Größenkontrolle von Protokolldateien Ein Linux-System erzeugt im laufenden Betrieb diverse Protokolldaten, die zum Beispiel über den Syslog-Daemon in Dateien geschrieben werden. Typische Protokolldateien können schnell wachsen und im Laufe der Zeit beachtliche Größen annehmen. Eine gängige Aufgabe bei der Systemadministration ist darum die Größenkontrolle und gegebenenfalls das Kürzen bzw. Neuansetzen von Protokolldateien. – Die meisten Linux-Distributionen benutzen dafür heuer ein standardisiertes Werkzeug namens `logrotate`.

Als nächstes wollen wir ein Shellskript `checklog` entwickeln, das prüft, ob eine Protokolldatei eine gewisse Länge erreicht oder überschritten hat, und sie gegebenenfalls umbenennt und unter ihrem alten Namen neu anlegt. Als Grundgerüst wäre etwas denkbar wie

```
#!/bin/bash
# checklog -- Prüfe eine Protokolldatei
#             und erneuere sie, falls nötig

if [ $# -ne 2 ]
then
    echo >&2 "usage: $0 FILE SIZE"
    exit 1
fi

if [ $(ls -l "$1" | cut -d' ' -f5) -ge $(( 1024*$2 )) ]
then
    mv "$1" "$1.old"
    > "$1"
fi
```

Die interessante Zeile in diesem Skript ist die mit dem Ausdruck

```
$(ls -l "$1" | cut -d' ' -f5) -ge $(( 1024*$2 ))
```

Hier bestimmen wir die Länge der als Parameter angegebenen Datei (fünfte Spalte der Ausgabe von »`ls -l`«) und vergleichen diese mit der ebenfalls als Parameter angegebenen Maximallänge. Den Längenparameter interpretieren wir dabei als Anzahl von Kibibytes.

Ist die Datei so lang wie die Maximallänge oder länger, wird sie umbenannt und unter dem alten Namen eine leere Datei erzeugt. Im wirklichen Leben ist das nur die halbe Miete: Ein Programm wie der `syslogd` öffnet die Protokolldatei beim Start und schreibt dann fröhlich hinein, egal wie die Datei später heißt – unser Skript kann sie zwar umbenennen, aber das bedeutet noch lange nicht, dass der `syslogd` dann in die neue Datei schreibt. Dazu muss er erst ein `SIGHUP` geschickt bekommen. Eine Möglichkeit, dies zu realisieren, ist über einen (optionalen) dritten Parameter:

```
<<<<<
> "$1"
[ -n "$3" ] && killall -HUP "$3"
<<<<<
```

Unser Skript könnte dann etwa so aufgerufen werden:

```
checklog /var/log/messages 1000 syslogd
```

Mehrere Protokolldateien überwachen Unser Skript aus dem vorigen Abschnitt ist vielleicht ganz nett, aber im wirklichen Leben haben Sie es normalerweise mit mehr als einer Protokolldatei zu tun. Sie können `checklog` natürlich *x*-mal mit verschiedenen Argumenten aufrufen, aber wäre es nicht möglich, dass das Programm sich um mehrere Dateien auf einmal kümmert? Im Idealfall könnten wir eine Konfigurationsdatei haben, die die zu erledigende Arbeit beschreibt und ungefähr so aussehen könnte:

Konfigurationsdatei

```
SERVICES="apache syslogd"
FILES_apache="/var/log/apache/access.log /var/log/apache/error.log"
FILES_syslogd="/var/log/messages /var/log/mail.log"
MAXSIZE=100
MAXSIZE_syslogd=500
NOTIFY_apache="apachectl graceful"
```

Im Klartext: Das Programm soll sich um die »Dienste« `apache` und `syslogd` kümmern. Zu jedem dieser Dienste gibt es eine Konfigurationsvariable, die mit »`FILES_`« anfängt und eine Liste der interessanten Protokolldateien enthält, und optional eine, die mit »`MAXSIZE_`« anfängt und die gewünschte Maximalgröße angibt (die Variable `MAXSIZE` gibt einen Standardwert vor, falls für einen Dienst keine eigene `MAXSIZE_`-Variable definiert ist). Ebenfalls optional ist die »`NOTIFY_`«-Variable, die ein Kommando angibt, mit dem der Dienst über eine neue Protokolldatei benachrichtigt wird (ersatzweise wird wie oben »`killall -HUP` ⟨*Dienst*⟩« ausgeführt).

Damit haben wir die Arbeitsweise unseres neuen Skripts – nennen wir es `multichecklog` – schon fast festgelegt:

1. Die Konfigurationsdatei einlesen
2. Für jeden Dienst in der Konfigurationsdatei (`SERVICES`):
3. Bestimme die gewünschte Maximalgröße (`MAXSIZE`, `MAXSIZE_*`)
4. Prüfe jede Protokolldatei gegen die Maximalgröße
5. Benachrichtige gegebenenfalls den Dienst

Der Anfang von `multichecklog` könnte zum Beispiel so aussehen:

```
#!/bin/bash
# multichecklog -- Prüfe mehrere Protokolldateien

conffile=/etc/multichecklog.conf
[ -e $conffile ] || exit 1
. $conffile
```

Wir prüfen, ob unsere Konfigurationsdatei – hier `/etc/multichecklog.conf` – existiert; wenn nein, gibt es nichts für uns zu tun. Wenn sie existiert, lesen wir sie einfach ein und definieren dadurch *in der aktuellen Shell* die Variablen `SERVICES` usw. (wir haben die Syntax der Konfigurationsdatei trickreich so festgelegt, dass das geht). Konfigurationsdatei lesen

Anschließend betrachten wir die einzelnen Dienste:

```
for s in $SERVICES
do
    maxsizevar=MAXSIZE_$s
    maxsize=${!maxsizevar:-${MAXSIZE:-100}}
    filesvar=FILES_$s
    for f in ${!filesvar}
    do
        checklonger "$f" "$maxsize" && rotate "$f"
    done
done
```

Wenn Sie aufmerksam mitgelesen haben, fällt Ihnen sicherlich die etwas verquere Konstruktion

```
maxsizevar=MAXSIZE_$s
maxsize=${!maxsizevar:-${MAXSIZE:-100}}
```

auf. Es geht uns darum, den Wert von `maxsize` wie folgt zu bestimmen: Zunächst wollen wir prüfen, ob »`MAXSIZE_`⟨*Dienst*⟩« existiert und nichtleer ist; wenn ja, wird der Wert dieser Variablen übernommen. Wenn nein, prüfen wir, ob `MAXSIZE` existiert; wenn ja, wird der Wert dieser Variablen übernommen, sonst 100. Das Problem bei der Sache ist der tatsächliche Name von »`MAXSIZE_`⟨*Dienst*⟩«, der sich erst in der Schleife ergibt. Dafür verwenden wir eine bisher nicht erklärte Eigenschaft der Variablensubstitution: In einem Variablenbezug der Form »`${!`⟨*Name*⟩`}`« wird der *Wert* von ⟨*Name*⟩ als Name der Variablen interpretiert, deren Wert schließlich eingesetzt wird, etwa so: Indirekte Substitution

```
$ DE=Hallo
$ CH=Grüezi
$ AT=Servus
$ land=CH
$ echo ${!land} Welt
Grüezi Welt
```

Der ⟨*Name*⟩ muss trotzdem ein gültiger Variablenname sein – in unserem Skript würden wir gerne etwas sagen wie

```
maxsize=${!MAXSIZE_$s:-${MAXSIZE:-100}}
```

aber das ist nicht erlaubt, also die Extraumleitung mit `maxsizevar`. (Derselbe Trick ist auch für »`FILES_`⟨*Dienst*⟩« nötig.)

Bemerken Sie ferner, dass wir den Größentest und das Umbenennen nicht in der inneren Schleife ausformuliert haben. Damit unser Skript übersichtlicher wird, realisieren wir diese beiden Aktionen als Shellfunktionen: Übersichtlichkeit

```
# checklonger FILE SIZE
function checklonger () {
    test $(ls -l "$1" | cut -d' ' -f5) -ge $(( 1024*$2 ))
}

# rotate FILE
function rotate () {
    mv "$1" "$1.old"
```

```
#!/bin/bash
# multichecklog -- Prüfe mehrere Protokolldateien

conffile=/etc/multichecklog.conf
[ -e $conffile ] || exit 1
. $conffile

# checklonger FILE SIZE
function checklonger () {
    test $(ls -l "$1" | cut -d' ' -f5) -ge $(( 1024*$2 ))
}

# rotate FILE
function rotate () {
    mv "$1" "$1.old"
    > "$1"
}

for s in $SERVICES
do
    maxsizevar=MAXSIZE_$s
    maxsize=${!maxsizevar:-${MAXSIZE:-100}}
    filesvar=FILES_$s
    notify=0
    for f in ${!filesvar}
    do
        checklonger "$f" "$maxsize" && rotate "$f" && notify=1
    done
    notifyvar=NOTIFY_$s
    [ $notify -eq 1 ] && ${!notifyvar:-killall -HUP \$s}
done
```

Bild 4.4: Mehrere Protokolldateien überwachen

```
    > "$1"
}
```

Als letztes fehlt uns nur noch die Benachrichtigung des Dienstes (wir haben sie in der ersten Version unserer Schleife unterschlagen). Wir müssen den Dienst nur einmal benachrichtigen, egal wie viele seiner Protokolldateien wir »rotiert« haben. Am geschicktesten geht das so:

```
notify=0
for f in ${!filesvar}
do
    checklonger "$f" "$maxsize" && rotate "$f" \
        && notify=1
done
notifyvar=NOTIFY_$s
[ $notify -eq 1 ] && ${!notifyvar:-killall -HUP $s}
```

Auch hier kommt wieder der Trick mit der indirekten Substitution zur Anwendung. Die Variable `notify` hat genau dann den Wert 1, wenn eine Protokolldatei rotiert werden musste.

Insgesamt sieht unser Skript jetzt so aus wie in Bild 4.4. – Die in diesem Skript verwendete Technik, eine »Konfigurationsdatei« zu lesen, die Zuweisungen an

Shellvariable enthält, ist sehr verbreitet. Linux-Distributionen verwenden sie gerne, die SUSE-Distributionen zum Beispiel für die Dateien in `/etc/sysconfig` und Debian GNU/Linux für die Dateien in `/etc/default`. Grundsätzlich kann in diesen Konfigurationsdateien alles auftreten, was in der Shell möglich ist; Sie tun jedoch gut daran, sich auf Variablenzuweisungen zu beschränken, die Sie möglicherweise durch Kommentarzeilen erklären können (einer der Hauptvorteile des Ansatzes).

Übungen

4.4 [1] Ändern Sie das Skript `multichecklog` so ab, dass der Name der Konfigurationsdatei wahlweise auch in der Umgebungsvariablen `MULTICHECKLOG_CONF` stehen kann. Vergewissern Sie sich, dass Ihre Änderung das tut, was sie soll.

4.5 [2] Wie würden Sie dafür sorgen, dass die Maximallänge der Protokolldateien bequem in der Form »`12345`« (Bytes), »`12345k`« (Kilobytes), »`12345M`« (Megabytes) angegeben werden kann? (*Tipp:* `case`)

4.6 [3] Bisher benennt die `rotate`-Funktion die aktuelle Protokolldatei `$f` um in `$f.old`. Definieren Sie eine alternative `rotate`-Funktion, die beispielsweise 10 alte Versionen der Datei wie folgt verwaltet: Beim Rotieren wird `$f` umbenannt zu `$f.0`, eine etwa schon vorhandene Datei `$f.0` wird umbenannt in `$f.1` usw.; eine etwa vorhandene Datei `$f.9` wird gelöscht. (*Tipp:* Das Kommando `seq` erzeugt Folgen von Zahlen.) Wenn Sie besonders gründlich sein wollen, machen Sie die Anzahl der alten Versionen konfigurierbar.

4.7 [2] Bei vielen Protokolldateien sind der Eigentümer, die Gruppe und der Zugriffsmodus wichtig. Erweitern Sie die `rotate`-Funktion so, dass die neu angelegte leere Protokolldatei für Eigentümer, Gruppe und Zugriffsmodus dieselben Einstellungen hat wie die alte.

Wichtige Ereignisse Ab und zu landen Meldungen im Protokoll, über die Sie als Systemverwalter möglichst bald informiert werden wollen. Natürlich haben Sie Dringenderes zu tun, als ständig das Protokoll zu beobachten – was läge also näher, als ein Shellskript damit zu beauftragen? Natürlich ist es unklug, einfach `/var/log/messages` periodisch mit `grep` zu durchsuchen, weil Sie dann mitunter mehrmals durch dasselbe Ereignis alarmiert werden. Sie können eher von einer Eigenschaft des Linux-`syslogd` profitieren, nämlich dass dieser auf Wunsch in eine *named pipe* schreibt, wenn Sie vor deren Namen einen vertikalen Balken setzen:

```
# syslog.conf
<<<<<
*.*;mail.none;news.none     |/tmp/logwatch
```

Die *named pipe* legen Sie natürlich vorher mit `mkfifo` an.

Ein einfaches Protokoll-Mitleseskript könnte also so aussehen:

```
#!/bin/bash

fifo=/tmp/logwatch
[ -p $fifo ] || ( rm -f $fifo; mkfifo -m 600 $fifo )

grep --line-buffered ALARM $fifo | while read LINE
do
    echo "$LINE" | mail -s ALARM root
done
```

Wir erzeugen zuerst die *named pipe*, falls sie nicht existiert. Anschließend wartet das Skript darauf, dass im Strom der Protokollzeilen eine auftaucht, die die

Zeichenkette »ALARM« enthält. Diese Zeile wird dann an den Systemverwalter geschickt.

Statt per Mail könnten Sie die Nachricht natürlich auch per SMS, Piepser, ... verschicken, je nachdem, wie dringend sie ist.

Kritisch dafür, dass das Skript wirklich funktioniert, ist die GNU-grep-Erweiterung `--line-buffered`. Sie sorgt dafür, dass `grep` seine Ausgabe zeilenweise schreibt, statt größere Mengen Ausgabe zu puffern, wie er das sonst aus Effizienzgründen beim Schreiben in Pipes tut. Eine Zeile könnte sonst eine halbe Ewigkeit brauchen, bis das `read` sie tatsächlich zu sehen bekommt.

Wenn Sie das `expect`-Paket von Don Libes installiert haben, verfügen Sie über ein Programm namens `unbuffer`, das die Ausgabe beliebiger Programme »entpuffert«. Sie könnten dann etwas schreiben wie

```
unbuffer grep ALARM $fifo | while read LINE
```

auch wenn `grep` die `--line-buffered`-Option nicht unterstützte.

Warum benutzen wir nicht einfach etwas wie

```
grep --line-buffered ALARM $fifo | mail -s ALARM root
```

? Ganz klar: Wir wollen ja, dass die Benachrichtigung möglichst umgehend erfolgt. Bei der einfachen Pipeline würde `mail` aber darauf warten, dass `grep` ihm das Dateiende signalisiert, um sicherzugehen, dass es alles Mailenswerte bekommen hat, bevor es die Mail tatsächlich verschickt. Die umständlichere »`while-read-echo`«-Konstruktion dient zur »Vereinzelung« der Nachrichten.

Statt dass Sie sich mühevoll einen regulären Ausdruck überlegen, der alle Ihre interessanten Protokollzeilen erfasst, können Sie übrigens auch die `-f`-Option von `grep` verwenden. Damit liest `grep` eine Reihe von regulären Ausdrücken aus einer Datei (einen pro Zeile) und sucht nach allen gleichzeitig:

```
grep --line-buffered -f /etc/logwatch.conf $fifo | …
```

entnimmt die Suchmuster der Datei `/etc/logwatch.conf`. Mit einem Trick können Sie die Suchmuster aber auch in die `logwatch`-Datei selber aufnehmen:

```
grep <<ENDE --line-buffered -f - $fifo | while read LINE
ALARM
WARNUNG
KATASTROPHE
ENDE
do
    echo ...
done
```

Hier stehen die Muster in einem Hier-Dokument, das dem `grep`-Prozess auf dessen Standardeingabe zur Verfügung steht; der spezielle Dateiname »`-`« bringt die `-f`-Option dazu, die Musterdatei von der Standardeingabe zu lesen.

Übungen

4.8 [2] Welche andere Möglichkeit gibt es, `grep` nach mehreren regulären Ausdrücken gleichzeitig suchen zu lassen?

4.5 Systemadministration

Shellskripte sind ein wichtiges Werkzeug für die Systemadministration – sie erlauben es, stumpfsinnig wiederholte Vorgänge zu automatisieren oder selten Gebrauchtes in bequemer Form zugänglich zu machen, damit Sie es sich nicht immer von neuem überlegen müssen. Außerdem ist es möglich, sich zusätzliche Funktionalität zusammenzubauen, die das System nicht von sich aus mitbringt.

df mit Nachbrenner Das `df`-Kommando gibt aus, wieviel Platz auf den Dateisystemen noch frei ist. Leider ist die Ausgabe auf den ersten Blick recht unübersichtlich – es wäre oft schön, zum Beispiel die prozentuale Auslastung der Dateisysteme etwa als »Balkengrafik« visualisieren zu können. Nichts leichter als das: Hier ist die Ausgabe von `df` auf einem typischen System:

```
$ df
Filesystem           1K-blocks      Used Available Use% Mounted on
/dev/hda3              3842408   3293172    354048  91% /
tmpfs                   193308         0    193308   0% /dev/shm
/dev/hda6             10886900   7968868   2364996  78% /home
/dev/hdc                714808    714808         0 100% /cdrom
```

»Grafisch« etwas aufbereitet könnte das so aussehen:

```
$ gdf
Mounted      Use%
/             91% ##########################################################------
/dev/shm       0% ------------------------------------------------------------------
/home         78% ###############################################------------------
/cdrom       100% ##################################################################
```

Wir müssen uns also die 5. und 6. Spalte der `df`-Ausgabe holen und sie als 1. und 2. Spalte in die Ausgabe unseres `gdf`-Skripts schreiben. Der einzige Haken an der Sache ist, dass `cut` sich im Feld-Ausschneidemodus nicht am Leerzeichen als Feldtrenner orientieren kann: Zwei aneinandergrenzende Leerzeichen führen zu einem leeren Feld in der Zählung (versuchen Sie mal »`df | cut -d' ' -f5,6`«). Statt mühselig die Zeichen in der Zeile zu zählen, um den Spalten-Ausschneidemodus von `cut` benutzen zu können, machen wir es uns einfach und ersetzen mit `tr` jede Folge von Leerzeichen durch ein Tabulatorzeichen. Unser `gdf`-Skript sieht dann – noch ohne die grafische Ausgabe – erst einmal so aus:

```
#!/bin/bash
# gdf -- "Grafische" Ausgabe von df (Vorversion)

df | tr -s ' ' '\t' | cut -f5,6 | while read pct fs
do
    printf "%-12s %4s " $fs $pct
done
```

Neu sind hier nur zwei Dinge: Die `read`-Anweisung liest das erste von `cut` ausgeschnittene Feld in die Variable `pct` und das zweite in die Variable `fs` (mehr über `read` erfahren Sie in Abschnitt 5.2). Und das `printf`-Kommando erlaubt die Ausgabe von Zeichenketten und Zahlen gemäß der Angaben in einem »Format« – hier »`%-12s %4s` « oder »eine linksbündige Zeichenkette in einem Feld von genau 12 Zeichen Breite (gegebenenfalls abgeschnitten oder mit Leerzeichen aufgefüllt) gefolgt von einem Leerzeichen und einer rechtsbündigen Zeichenkette in einem Feld von genau 4 Zeichen Breite (dito)«.

```
#!/bin/bash
# gdf -- "Grafische" Ausgabe von df (Fertige Version)

hash="############################################################"
dash="------------------------------------------------------------"

df "$@" | tr -s ' ' '\t' | cut -f5,6 | while read pct fs
do
    printf "%-12s %4s " $fs $pct
    if [ "$pct" != "Use%" ]
    then
        usedc=$((${#hash}*${pct%\%}/100))
        echo "${hash:0:$usedc}${dash:$usedc}"
    else
        echo ""
    fi
done
```

Bild 4.5: `df` mit Balkengrafik für die Plattenauslastung

`printf` steht als externes Programm zur Verfügung, ist aber (in leicht erweiterter Form) auch fest in die Bash eingebaut. Es gibt Dokumentation entweder als Handbuchseite (`printf(1)`), als Info-Dokument oder in der Bash-Anleitung; Details über die erlaubten Formate müssen Sie allerdings in der Dokumentation der C-Bibliotheksfunktion `printf()` nachschlagen (in `printf(3)`). Von `printf` wird offenbar angenommen, dass es so tief im kollektiven Unterbewusstsein der Unix-Benutzer verankert ist, dass es nicht mehr groß erklärt werden muss ...

Zur Lösung der Aufgabe fehlen uns nur noch die grafischen Balken, die sich natürlich aus der prozentualen Auslastung der Dateisysteme gemäß Spalte 5 (vulgo `pct`) ergeben. Die Balken selbst schneiden wir der Einfachheit halber aus vorgegebenen Zeichenketten der gewünschten Länge aus; wir müssen nur aufpassen, dass wir uns nicht an der Titelzeile (»`Use%`«) verschlucken. Nach dem `printf` steht also etwas wie

```
if [ "$pct" != "Use%" ]
then
    usedc=$((${#hash}*${pct%\%}/100))
    echo "${hash:0:$usedc}${dash:$usedc}"
else
    echo ""
fi
```

Dabei sind `hash` eine lange Kette von »`#`«-Zeichen und `dash` eine ebenso lange Kette von Strichen. `usedc` ergibt sich aus der Länge von `hash` – erhältlich über die spezielle Expansion `$#hash` – multipliziert mit dem Prozentsatz der Auslastung geteilt durch 100, gibt also die Anzahl von »`#`«-Zeichen an, die im Balken angezeigt werden sollen. Den Balken selbst bekommen wir, indem wir `$usedc` viele Zeichen aus `hash` ausgeben und gerade genug Zeichen aus `dash` anhängen, dass der Balken genauso lang ist wie `hash`. Das gesamte Skript steht in Bild 4.5; `hash` und `dash` sind je 60 Zeichen lang, was bei einer Standard-Terminalfensterbreite von 80 Zeichen zusammen mit dem Format den Platz gerade gut ausnutzt.

Übungen

4.9 [1] Warum steht in Bild 4.5 »`df "$@"`«?

4.10 [3] Schreiben Sie eine Version von `gdf`, die die Länge des Balkens abhängig von der Größe des Dateisystems macht. Das größte Dateisystem soll über die ganze ursprüngliche Breite gehen und die anderen Dateisysteme einen proportional kürzeren Balken haben.

Kommandos in diesem Kapitel

`logrotate`	Verwaltet, kürzt und „rotiert" Protokolldateien	`logrotate`(8)	66
`mkfifo`	Legt FIFOs (benannte Pipes) an	`mkfifo`(1)	71
`printf`	Gibt Zahlen und Zeichenketten formatiert aus	`printf`(1), `bash`(1)	73
`seq`	Erzeugt Folgen von Zahlen auf der Standardausgabe	`seq`(1)	71
`tr`	Tauscht Zeichen in der Standardeingabe gegen andere aus oder löscht sie	`tr`(1)	73
`unbuffer`	Unterdrückt Ausgabepufferung eines Programms (Bestandteil des `expect`-Pakets)	`unbuffer`(1)	72

Zusammenfassung

- `grep` und `cut` sind nützlich, um bestimmte Zeilen und Spalten aus Dateien zu extrahieren.
- Sie sollten Fehlersituationen sorgfältig abfangen – sowohl beim Aufruf Ihres Skripts als auch während des Ablaufs.
- Ein- und Ausgabeumleitung für Folgen von Kommandos ist über explizite Subshells möglich.
- Die Kommandos `dirname` und `basename` erlauben Dateinamen-Manipulationen.
- Dateien mit Zuweisungen an Shellvariable können als bequeme »Konfigurationsdateien« für Shellskripte fungieren.
- Protokollnachrichten lassen sich einzeln verarbeiten, indem Sie sie mit `read` aus einer *named pipe* lesen.
- Das `printf`-Kommando erlaubt die formatierte Ausgabe von textuellen oder numerischen Daten.