Linux – Prinzipien und Programmierung

Dr. Klaus Höppner

Hochschule Darmstadt – Wintersemester 2012/2013

Shell-Programmierung

Pipes

Die Bash kennt drei Standard-Dateideskriptoren:

Standard In (stdin) Standard-Eingabe, i. A. die Tastatur,

Standard Out (stdout) Standard-Ausgabe, i. A. das aktuelle Terminal,

Standard Error (stderr) Standard-Fehler, i. A. auch das aktuelle Terminal.

Umleiten von Ein- und Ausgabe

- `wc -w < hallo.txt`
  Inhalt von „hallo.txt“ als stdin für `wc -w` (Zählen der Wörter) verwenden.
- `cat hallo.txt | wc -w`
  Inhalt von „hallo.txt“ nach stdout ausgeben, stdout aber als stdin für `wc -w` verwenden.
- `echo "Hallo Welt" > hallo.txt`
  Ausgabe in Datei „hallo.txt“ schreiben (evtl. alter Inhalt wird überschrieben).
- `echo "Hallo Welt" >> hallo.txt`
  Ausgabe an Datei „hallo.txt“ anhängen.
- `gcc -c test.c 2> error.log`
  Fehlermeldungen in Datei schreiben.
- `cmd1 | tee dateiname`
  Ausgabe parallel in Datei loggen.

Wildcards, Backticks

`*` Beliebig viele (auch null) Zeichen,

`?` genau ein (beliebiges) Zeichen,

`[afz1]` eines der angegebenen Zeichen,

`[a-f]` eines der Zeichen aus dem angegebenen Bereich,

`` `cmd` ``, `${cmd}` wird durch Ausgabe von `cmd` ersetzt,

Variablen

Die Definition von Variablen erfolgt mit
`var=value` (ohne Leerzeichen)
und der Zugriff mit `$var`.

Zu Beachten ist die unterschiedliche Expansion von Variablen innerhalb von Anführungszeichen, z. B.

```
name=Klaus
echo "Hallo $name"
echo 'Hallo $name'
```

Im ersten Fall wird die Variable expandiert, im zweiten Fall nicht.

Globale und lokale Variablen

Die Unterscheidung zwischen lokalen und globalen Variablen in der bash ist sehr missdeutig, denn diese betrifft lediglich die Frage, ob Variablen an Subshells weitergegeben werden. Siehe dazu folgendes Beispiel:

```
varglobal="Foo"
export varglobal
varlokal="Bar"

echo $varglobal
echo $varlokal

bash
echo $varglobal
echo $varlokal
exit
```


Parametersubstitution

`${var:-default}` Der Inhalt von `$var`, falls diese nicht leer ist, sonst `default`,

`${var#pattern}` Der Inhalt von `$var`, wobei am Anfang ggfs. `pattern` entfernt wird (bei Wildcards kürzestmöglich),

`${var##pattern}` dto., aber bei Wildcards größtmögliche Entfernung von `pattern`,

`${var%pattern}` Der Inhalt von `$var`, wobei am Ende ggfs. `pattern` entfernt wird (bei Wildcards kürzestmöglich),

`${var%%pattern}` dto., aber bei Wildcards größtmögliche Entfernung von `pattern`,

Kontrollstrukturen

Die bash kennt die Kontrollstrukturen, die auch in anderen Programmiersprachen üblich sind:

- `if`,
- `while`,
- `for`,
- `case`.

Besonderheit ist, dass diese Kontrollstrukturen jeweils durch den umgedrehten Namen beendet werden, `case` z. B. durch `esac`.

Shell-Skripte in einer Datei

Ein Shell-Skript kann auch in eine Datei geschrieben werden.
Ubnterscheide dabei
`./myscript` von
`. ./myscript`:

Ersteres wird in einer Subshell ausgeführt, letzteres in der aktuellen Shell. *Achtung:* Das aktuelle Verzeichnis ist meist nicht in `$PATH` enthalten, daher der Aufruf `./myscript`!

Damit das Skript direkt ausgeführt werden kann, muss es für den aktuellen User executable (Zugriffsrecht `x`) sein.

Wie wird eine Datei ausgeführt

In Linux haben Dateiendungen i. A. keine spezifische Bedeutung. Eine ausführbare Datei kann daher sowohl ein kompiliertes Programm sein, aber auch ein Skript, das mit der Bash oder aber einem Interpreter wie Perl oder Python ausgeführt werden kann.

Woher weiß Linux nun, womit ein solches Skript ausgeführt werden soll?

Hierfür existiert ein besonderer Kommentar in der ersten Zeile, durch den angegeben wird, in welchem Interpreter das Skript laufen soll, z. B.
`#! /bin/bash`
`#! /usr/bin/perl`
`#! /usr/bin/python` oder auch
`#! /usr/bin/env python` (womit `python` in `$PATH` gesucht wird).

Einfaches Shell-Skript

Im folgenden soll ein Shell-Skript entwickelt werden, das alle ausführbaren Dateien in einem Verzeichnis ausführt.
Ein erster Ansatz könnte so aussehen:

```
#! /bin/bash

dir=$1

echo "Executing all in $dir"

for f in $dir/*
do
    echo "Executing $f"
    $f
done
```

Analyse

Im Prinzip funktioniert das Skript bereits: Das Argument 1 der Kommandozeile wird als Verzeichnisname übernommen und in einer `for`-Schleife werden dann alle Dateien darin ausgeführt.
Allerdings gibt es ein paar Probleme:

- Wird das Skript ohne Argumente aufgerufen, so wird versucht, alle Skripte in `/` auszuführen.
- Wird der Verzeichnisname mit `/` am Ende angegeben, erscheint dieser in der Ausgabe doppelt.
- Es wird nicht geprüft, ob das Verzeichnis überhaupt existiert.
- Es wird auch versucht, nicht ausführbare Dateien auszuführen.
- Es findet keine Überprüfung statt, ob die einzelnen Programme erfolgreich liefen.

Prüfen auf Argument

Mit folgendem Code wird ein evtl. am Ende des ersten Argumentes vorhandenes `/`-Zeichen entfernt und anschließend geprüft, ob der Verzeichnisname leer ist:

```
dir=${1%/}

if [ -z "$dir" ]
then
    echo "Usage: $(basename $0) dirname"
    exit 1
fi
```

(NB: Hierdurch wird auch verhindert, dass das Skript mit `/` direkt aufgerufen wird!)

Argumente eines Skripts

Beim Aufruf eines Skripts stehen der Befehl, mit dem das Skript aufgerufen wurde, sowie die übergebenen Argumente als Variablen zur Verfügung:

`$0`	Name des Skriptes
`$1, $2, ...`	Übergebene Argumente
`$#`	Zahl der Argumente
`$*`	Alle Argumente in *einer* Variablen (immer in " einschließen!)

Mit `shift` kann ein Argument aus der Parameterliste entfernt werden.

Testen auf Existenz des Verzeichnisses

Nun wird geprüft, ob das Verzeichnis überhaupt existiert:

```
if [ ! -d "$dir" ]
then
    echo "Directory $dir does not exist"
    exit 2
fi
```


Selektives Ausführen von Dateien

```
for f in $dir/*
do
    [ -d "$f" ] && continue
    [ -x "$f" ] || continue
    echo "Executing $f"
    $f
    ret=$?
    if [ $ret -ne 0 ]
    then
        echo "$f failed with return status $ret"
    fi
done
```


Analyse

- Nun wird in der for-Schleife zunächst geprüft, ob es sich bei der im Verzeichnis befindlichen Datei um ein Verzeichnis handelt, falls ja, wird es übersprungen,
- dann wird geprüft, ob die Datei ausführbar ist, sonst wird sie übersprungen.
- Nach dem Ausführen einer Datei wird nun der Status der Ausführung geprüft. Falls ein Fehler auftrat (Status ungleich 0), wird eine Meldung ausgegeben. Der Ausführungsstatus des letzten Befehles befindet sich dabei jeweils in der Variablen `$?`.

Prüfen auf Optionen in der Kommandozeile

Nun soll das Skript mit einigen Optionen aufgerufen werden können:
`exec_all [--log=logfile] [--stop-on-error] dirname`
wobei mit der ersten Option ein Dateiname angegeben wird, in den die Ausgabe bei Ausführen der Programme umgeleitet wird.
Bei Angabe der zweiten Option soll das Skript beendet werden, wenn es beim Ausführen eines Programms zu einem Fehler kam, d. h. alle weiteren Programme in dem Verzeichnis werden nicht mehr ausgeführt.

Realisierung

```
#! /bin/bash

usage() {
    echo "Usage: $(basename $0) [--log logfile] [--stop-on-
}

log=
stop_error=no
while [ "$1" != "${1#-}" ]
do
    option=$1
    case $option in
        -)
            shift
            break
            ;;
```


Realisierung (Forts.)

```
        --log=*)
            log=${option#--log=}
            shift
            ;;
        --log)
            log=$2
            shift 2
            ;;
        --stop-on-error)
            stop_error=yes
            shift
            ;;
```

Realisierung (Forts.)

```
        *)
            usage
            echo "Unknown option $option"
            exit 1
            ;;
    esac
done
```


Realisierung (Forts.)

```
dir=${1%/}

if [ -z "$dir" ]
then
    usage
    exit 1
fi

if [ ! -d "$dir" ]
then
    echo "Directory $dir does not exist"
    exit 2
fi
```


Realisierung (Forts.)

```
echo "Executing all in $dir"

for f in $dir/*
do
    [ -d "$f" ] && continue
    [ -x "$f" ] || continue
    echo "Executing $f"
    if [ -z "$log" ]
    then
        $f
    else
        $f >$log
    fi
```

Realisierung (Forts.)

```
    ret=$?
    if [ $ret -ne 0 ]
    then
        echo "$f failed with return status $ret"
        [ "$stop_error" = "yes" ] && break
    fi
done
```