Linux und Shell-Programmierung – Teil 5

Prof. Dr. Christian Baun

Fachhochschule Frankfurt am Main
Fachbereich Informatik und Ingenieurwissenschaften
christianbaun@fb2.fh-frankfurt.de

Heute

- Shell-Skripting (Teil 1)
  - Die Shell
  - Varianten der Shell
  - Kommentare
  - Auswahl der Shell
  - Shell-Skripte testen (`sh`)
  - Feste Variablen (`${var}`, `$0`, `$# $*`, `$@`, `$$`, `$-`, `$_`, `$?`, `&!`)
  - Kommandozeilenparameter verarbeiten
  - Tests für Zeichenketten, Zahlen und Dateien (`test`, `[`)
  - Rückgabewert setzen (`true`, `false`)
  - Shell-Skripte vorzeitig beenden (`exit`)
  - Kontrollstrukturen in Shell-Skripten
    - `if`-Anweisung, `case`-Anweisung
    - `while`-Schleife, `until`-Schleife, `for`-Schleife
  - Schleifen vorzeitig verlassen (`break`)
  - Schleifen erneut durchlaufen (`continue`)
  - Endlosschleifen
  - Zählschleifen

Die Shell

- Die Shell ist ein Programm, durch das das System die Anweisungen (Befehle) des Benutzers verstehen kann
- Wegen Ihrer Funktion wird die Shell häufig als Befehls- oder Kommandointerpreter bezeichnet
- Die Shell hat drei Hauptaufgaben:
  - Interaktive Anwendung (Dialog)
  - Anwendungsspezifische Anpassung des Systemverhaltens (Umgebungsvariablen definieren)
  - Programmierung (Shell-Skripting)
- Die Shell kennt einige Mechanismen, die aus Hochsprachen bekannt sind. Diese sind u.a. Variablen, Datenströme, Funktionen,...

Einfache und komfortable Varianten der Shell

- Es gibt nicht *die* eine Shell, sondern es existieren mehrere Varianten
- Jede Variante hat ihre Vor- und Nachteile
- Es ist unter Linux/UNIX kein Problem den Kommandointerpreter auszutauschen
  - Aus diesem Grund stehen auf fast allen Systemen mehrere unterschiedliche Shells zur Verfügung
- Welche Variante der Shell ein Benutzer verwenden möchte, ist reine Geschmackssache
- Die existierenden Varianten der Shell können in eher einfache und eher komfortable Shells unterschieden werden

Einfache Varianten der Shell

- **Bourne**- oder **Standard-Shell** (`sh`):
  - Kompakteste und einfachste Form
  - Bietet u.a. Umlenkung der Ein- oder Ausgaben, Wildcards zur Abkürzung von Dateinamen, Shell-Variablen, usw.
  - Steht auf praktisch allen Systemen zur Verfügung
  - Shell-Skripte für die Standard-Shell sind sehr portabel
- **Korn-Shell** (`ksh`):
  - Weiterentwicklung der Bourne-Shell
  - Bietet u.a. History-Funktionen, eine Ganzzahl-Arithmetik und Aliase
- **C-Shell** (`csh`):
  - Bietet ähnliche Features wie die Korn-Shell
  - Syntax ist sehr stark an die Programmiersprache C angelehnt
  - Geringe Portabilität
    - Darum eher ungeeignet für Shell-Skripte

Komfortable Varianten der Shell

- **Bourne-Again-Shell** (`bash`):
  - Voll abwärtskompatibel zur Standard-Shell
  - Bietet aber von allen Shells die komfortabelsten Funktionen für das interaktive Arbeiten
  - Standard-Shell auf allen Linux-Systeme
  - Steht auf den meisten anderen UNIX-Systemen zur Verfügung
- **TENEX-C-Shell** (`tcsh`):
  - Verhält sich zur C-Shell wie die Bourne-Again-Shell zur Standard-Shell
  - Voll kompatibel zur C-Shell. Bietet aber zusätzliche Komfort-Funktionen
- Es existieren noch viele weitere Varianten der Shell
- Eine Übersicht: `http://de.wikipedia.org/wiki/Unix-Shell`
- Für Shell-Skripte optimal: Standard-Shell oder Bourne-Again-Shell
- Aus Gründen der Portabilität sollte für Shell-Skripte die Standard-Shell oder die Bourne-Again-Shell verwendet werden

Warum schreibt man Shell-Skripte?

- Shell-Skripte sind immer da hilfreich, wo:
  - Ständig wiederkehrende Kommandos zusammengefasst werden sollen
    - Diese können dann mit einem einzelnen Aufruf gestartet werden
  - Schnell kleine Programme entwickelt werden sollen
  - Regelmäßige Systemüberwachung notwendig ist
  - Umfangreiche Protokoll- und Servicedaten (Log-Daten) anfallen, die überwacht werden müssen
  - Automatisierung ist notwendig, um Fehler zu vermeiden und Ressourcen zu sparen
- Typische Einsatzgebiete für Shell-Skripte sind Administrationsaufgaben (z.B. Backup)

Wie schreibt man Shell-Skripte?

- Um ein einfaches Shell-Skript zu erzeugen, startet man einen beliebigen Editor und führt ein paar Kommandos hintereinander zeilenweise auf
- Ein einfaches Beispiel:

```
# Mein erstes Shell-Skript
echo "Test"
date
whoami
```

- Diese Zeilen werden unter dem Namen `shellskript` gespeichert
- Die Datei muss noch ausführbar gemacht werden:

```
$ chmod u+x shellskript
$ ls -l shellskript
-rwxr--r-- 1 user user 51 2009-10-15 14:12 shellskript
```

Das erste Shell-Skript

- Ergebnis der Ausführung des ersten Shell-Skripts:

```
$ ./shellskript
Test
Di 23. Okt 17:09:40 CEST 2007
testuser
```

Die Shell auswählen

- In der ersten Zeile eines Shell-Skriptes sollte immer definiert werden, mit welcher Shell das Skript ausgeführt werden soll
- In diesem Fall öffnet das System eine Subshell und führt das restliche Shell-Skript in dieser aus
- Die Angabe der Shell erfolgt über eine Zeile in der Form:
  - Für die Standard-Shell

```
#!/bin/sh
```

  - Für die Bourne-Again-Shell

```
#!/bin/bash
```

- Der Eintrag wirkt nur, wenn er in der ersten Zeile des Shell-Skripts steht

Kommentare

- Kommentare in der Shell beginnen immer mit dem Zeichen #
- Es spielt keine Rolle, ob das Zeichen am Anfang der Zeile steht, oder hinter Kommandos
- Alles ab dem Zeichen # bis zum Zeilenende wird beim interpretieren von der Shell ignoriert

```
# Das ist eine Kommentarzeile!
```

- Beim Schreiben von Shell-Skripten sollte man nicht mit Kommentaren geizen, um die Lesbarkeit zu erhöhen
- Der Einzige Fall, in dem der Text hinter dem # nicht ignoriert wird, ist bei der Auswahl der Shell

Shell-Skripte testen

- Zum Testen eines Shell-Skripts empfiehlt sich das Kommando `sh -x`
- Beim Aufruf mit `sh -x` wird jedes Kommando im Shell-Skript ausgeführt und das Ergebnis direkt ausgegeben

```
$ cat shellskript
#!/bin/bash
# Mein erstes Shell-Skript

echo "Test"
date
whoami
```

```
$ sh -x shellskript
+ echo Test
Test
+ date
Mi 31. Okt 10:28:46 CET 2007
+ whoami
bauni
```

Shell-Skripte vorzeitig beenden mit `exit`

- Shell-Skript beenden sich automatisch, sobald ihre letzte Zeile ausgeführt wurde
  - Es ist aber auch möglich, ein Shell-Skript vorzeitig selbst beenden zu lassen
- Zum vorzeitigen Abbruch eines Shell-Skripts existiert das Kommando `exit`

```
exit
```

- `exit` kann eine ganze Zahl als Argument mitgegeben werden, um den Rückgabewert (Exit-Status) festzulegen
  - `exit 0` bedeutet so viel wie *alles ok*
  - `exit 1` bedeutet *Fehler!*
- Der Rückgabewert kann später über die Variable $? ausgelesen werden

Feste Variablen bei Shell-Skripten

`${var}`	Wert der Variablen `var`
`$0`	Name der Programms (Shell-Skripts)
`$#`	Anzahl der Argumente auf der Kommandozeile
`$1 $2 $2`	Erstes, zweites, drittes ... Argument
`$*`	Alle Argumente auf der Kommandozeile (`$1 $2 $3 ...`)
`$@`	Wie `$*`
`"$@"`	Expandiert im Unterschied zu `$*` zu `"$1" "$2" "$3" ...`
`$$`	Prozessnummer (PID) der Shell
`$-`	Die aktuellen Shell-Optionen
`$_`	Name der Datei, für die diese Shell gestartet wurde
`$?`	Rückgabewert des zuletzt ausgeführten Kommandos (Normalerweise `0` bei erfolgreicher Durchführung)
`&!`	Prozessnummer des zuletzt gestarteten Prozesses

Feste Variablen und Kommandozeilenparameter

```
$ cat variablen_skript
echo Anzahl der Übergabeparameter: $#
echo Übergabeparameter: $*
echo Benutzer ist: $USER
echo Shell ist eine: $SHELL
echo Erster Parameter: $1
echo Zweiter Parameter: $2
echo Dateiname des Shell-Skripts: $0
echo Prozessnummer \(PID\): $$

$ ./variablen_skript eins zwei drei
Anzahl der Übergabeparameter: 3
Übergabeparameter: eins zwei drei
Benutzer ist: testuser
Shell ist eine: /bin/bash
Erster Parameter: eins
Zweiter Parameter: zwei
Dateiname des Shell-Skripts: ./variablen_skript
Prozessnummer (PID): 9444
```

Vergleichsoperationen

- Das Kommando `test` ist Bestandteil der Shell und wertet einfache Boolesche Ausdrücke aus, die aus Zahlen und Strings bestehen könnten
- Entsprechend der Auswertung von `test` ist der Rückgabewert (Exit-Status) 0 (true) oder 1 (false)
- Für `test` gibt es den Alias `[`
  - Wenn Sie diesen Alias verwenden, müssen Sie als letztes Argument von `test` ein `]` angeben
- Die Vergleichsoperatoren müssen von Leerzeichen umgeben sein, sonst werden Sie von der Shell nicht erkannt
  - Das Gilt auch für die Klammern

Ist 10 größer als 5?

```
$ test 10 -gt 5
$ echo $?
0
```

Ist 10 größer als 5?

```
$ [ 10 -gt 5 ]
$ echo $?
0
```

Vergleichsoperationen (Tests für Zeichenketten)

`"s1" == "s2"`	Wahr, wenn die Zeichenketten gleich sind
`"s1" != "s2"`	Wahr, wenn die Zeichenketten ungleich sind
`-z "s1"`	Wahr, wenn die Zeichenkette leer ist (Länge $= 0$)
`-n "s1"`	Wahr, wenn die Zeichenkette nicht leer ist (Länge > 0)

Sind die beiden Strings gleich?

```
$ test "TEST" == "TEST"
$ echo $?
0
```

Sind die beiden Strings ungleich?

```
$ test "String" != "TEST"
$ echo $?
0
```

Vergleichsoperationen (Tests für Ganze Zahlen)

`n1 -eq n2`	Wahr, wenn die Zahlen gleich sind
`n1 -ne n2`	Wahr, wenn die Zahlen ungleich sind
`n1 -gt n2`	Wahr, wenn `n1` $>$ `n2` sind
`n1 -ge n2`	Wahr, wenn `n1` $\geq$ `n2` sind
`n1 -lt n2`	Wahr, wenn `n1` $<$ `n2` sind
`n1 -le n2`	Wahr, wenn `n1` $\leq$ `n2` sind

Ist 15 kleiner als 10?

```
$ test 15 -lt 10
$ echo $?
1
```

Hat der String `Test` eine Länge $>$ 0?

```
$ test -n "Test"
$ echo $?
0
```

Vergleichsoperationen (Tests für Dateien)

`-d Name`	Wahr, wenn `Name` ein Verzeichnis ist
`-f Name`	Wahr, wenn `Name` eine reguläre Datei ist
`-L Name`	Wahr, wenn `Name` ein symbolischer Link ist
`-r Name`	Wahr, wenn `Name` existiert und lesbar ist
`-w Name`	Wahr, wenn `Name` existiert und schreibbar ist
`-x Name`	Wahr, wenn `Name` existiert und ausführbar ist
`-s Name`	Wahr, wenn `Name` existiert und die Größe > 0 ist
`f1 -nt f2`	Wahr, wenn `f1` jünger als `f2` ist
`f1 -ot f2`	Wahr, wenn `f1` älter als `f2` ist

Vergleichsoperationen (Sonstige Tests)

`!`	Negation.
`-a`	Logisches *und*
`-o`	Logisches *oder*
`\(...\)`	Gruppierung. Die Klammern müssen jeweils durch einen Backslash geschützt werden

Wahr (`true`) und Falsch (`false`)

- In der Shell existieren die Kommandos `true` und `false`, die ihren Rückgabewert (Exit-Status) entsprechend setzen

```
$ true
$ echo $?
0
```

```
$ false
$ echo $?
1
```

- Die Kommandos `true` und `false` sind besonders bei Bedingungen und Schleifen nützliche Werkzeuge

Kontrollstrukturen in Shell-Skripten

- Für Shell-Skripte stehen verschiedene Kontrollstrukturen zur Verfügung
- **Bedingte Programmausführung**
  - `if`
  - `case`
- **Schleifen**
  - `while`
  - `until`
  - `for`

Bedingte Ausführung mit `if`

- Mit der `if`-Anweisung ist es möglich, Bedingungen zu realisieren
- Struktur der `if`-Anweisung:

```
if [ Bedingung ]
then
  Anweisungsblock
fi
```

- Die Bedingung entspricht der Schreibweise von `test`.
- Das `fi` bedeutet *end if*.

Beispiel zur if-Anweisung

```
# cat if
#!/bin/bash
# Beispiel zur if-Anweisung

if [ `whoami` == "root" ]
then
  echo "Sie sind der Admin."
fi
```

```
# ./if
Sie sind der Admin.
```

Bedingte Ausführung mit `if-else`

- Die `if`-Anweisung kann um einen `else`-Zweig erweitert werden.
- Struktur der `if-else`-Anweisung:

```
if [ Bedingung ]
then
  Anweisungsblock
else
  Anweisungsblock
fi
```

Beispiel zur if-else-Anweisung

```
$ cat ifelse
#!/bin/bash
# Beispiel zur if-else-Anweisung

if [ `whoami` == "root" ]
then
  echo "Sie sind der Admin."
else
  echo "Sie sind nicht der Admin."
fi
```

```
$ ./ifelse
Sie sind nicht der Admin.
```

Bedingte Ausführung mit `if-elif-else`

- Die `if-else`-Anweisung kann um einen oder mehrere `elif`-Zweige erweitert werden
- Struktur der `if-elif-else`-Anweisung:

```
if [ Bedingung ]
then
  Anweisungsblock
elif [ Bedingung ]
then
  Anweisungsblock
else
  Anweisungsblock
fi
```

Beispiel zur if-elif-else-Anweisung

```
$ cat ifelifelse
#!/bin/bash
# Beispiel zur if-elif-else-Anweisung

if [ `whoami` == "root" ]
then
  echo "Sie sind der Admin."
elif [ `whoami` == "alice" ]
then
  echo "Sie sind Alice."
elif [ `whoami` == "bob" ]
then
  echo "Sie sind Bob."
else
  echo "Keine Ahnung, wer Sie sind."
fi
```

Weiteres Beispiel zur if-elif-else-Anweisung

```
$ cat umfrage
#!/bin/bash
# Beispiel zur if-elif-else-Anweisung

echo "Finden Sie Shell-Skripting schwer? (ja/nein)"
read antwort
echo "Ihre Antwort war: $antwort"
if [ "$antwort" = "ja" ]
then
  echo "üben üben üben."
elif [ "$antwort" = "nein" ]
then
  echo "weiter so."
else
  echo "Diese Antwort habe ich nicht verstanden."
fi
```

Die case-Anweisung

- Die case-Anweisung wertet einen einzelnen Wert aus und verzweigt zu einem passenden Code-Abschnitt
- Struktur der case-Anweisung:

```
case Variable in
  Muster) Anweisungsblock ;;
  Muster) Anweisungsblock ;;
  Muster) Anweisungsblock ;;
  *) Default-Anweisungsblock ;;
esac
```

- Die Anweisungsblöcke werden durch ;; abgeschlossen, denn ein einzelnes Semikolon ist das Trennzeichen zwischen Kommandos auf der selben Zeile
- Das esac bedeutet *end case*

Beispiel zur case-Anweisung

```
#!/bin/bash
# Beispiel zur case-Anweisung

echo "Finden Sie Shell-Skripting schwer?"
read antwort
case "$antwort" in
  j*|J*|y*|Y*)
    echo "üben üben üben."
  ;;
  n*|N*)
    echo "weiter so."
  ;;
  *)
    echo "Diese Antwort habe ich nicht verstanden."
  ;;
esac
```

Die `while`-Schleife

- Eine `while`-Schleife wiederholt eine Menge von Befehlen, **so lange eine Bedingung erfüllt ist**
- Bei `while` erfolgt die Prüfung der Bedingung **vor** der Abarbeitung der Schleife
- Struktur der `while`-Schleife:

```
while [ Bedingung ]
do
  Anweisungsblock
done
```

Beispiel zur `while`-Schleife

```
#!/bin/bash
# while-Schleife

i=1

while [ $i -le 5 ]
do
  echo $i
  i=‘expr $i + 1‘
done
```

```
$ ./while
1
2
3
4
5
```

- Achtung: Die Hochkommata, die `expr` umschließen, sind diejenigen, neben der Backspace-Taste (Shift nicht vergessen!)

Die `until`-Schleife

- Eine `until`-Schleife wiederholt eine Menge von Befehlen, **so lange bis eine Bedingung erfüllt ist**
- Bei `until` erfolgt die Prüfung der Bedingung **nach** der Abarbeitung der Schleife
- Struktur der `until`-Schleife:

```
until [ Bedingung ]
do
  Anweisungsblock
done
```

Beispiel zur until-Schleife

```
#!/bin/bash
# until-Schleife

i=1

until [ $i -gt 5 ]
do
  echo $i
  i=‘expr $i + 1‘
done
```

```
$ ./until
1
2
3
4
5
```

Die `for`-Schleife

- Die `for`-Schleife iteriert über Werte aus einer Liste
- Struktur der `for`-Schleife:

```
for variable in Liste_der_Parameter
do
  Anweisungsblock
done
```

Beispiel zur for-Schleife

```
$ cat for
#!/bin/bash
# for-Schleife

for nummer in eins zwei drei
do
  echo "Parameter $nummer"
done
```

```
$ ./for
Parameter eins
Parameter zwei
Parameter drei
```

Einsatzbeispiele von `for`-Schleifen

- Alle Dateien mit der Endung `.zip` im aktuellen Verzeichnis entpacken:

```
for i in *.zip; do unzip -o $i; done
```

- Bei allen Dateien mit der Endung `.JPG` im aktuellen Verzeichnis, die Endung in `.jpg` ändern:

```
for i in *.JPG; do mv $i ${i%.JPG}.jpg; done
```

- Alle Dateien mit der Endung `.jpg` im aktuellen Verzeichnis so umbenennen, dass der Dateiname mit `SYS_SS2009_` anfängt, danach kommt eine fortlaufende Nummer und am Ende die Dateiendung `.jpg`:

```
let a=0; for i in *.jpg; do let a=a+1;
mv $i SYS_SS2009_${a}.jpg; done
```

break und continue

- Um eine Schleife vorzeitig zu verlassen, existiert das Kommando `break`
- Ein Aufruf von `break` weist die Shell an, zur nächsten Anweisung hinter der Schleife zu springen
- Das Gegenstück von `break` ist `continue`
- Ein Aufruf von `continue` weist die Shell an, zum Anfang der Schleife zurückzukehren und gegebenenfalls einen neuen Durchlauf zu starten
- Sind mehrere Schleifen ineinander verschachtelt, kann mit einem zusätzlichen Argument ausgewählt werden, welche Schleifen die Shell abbrechen bzw. wiederholen soll
  - `break` oder `break 1` beendet die direkt umgebende Schleife
  - `break 2` beendet die zweite von innen umgebende Schleife usw.
  - `continue 2` beendet die innere Schleife und startet die äußere neu

Einsatzbeispiele von break

- break und sleep in Warteschleifen

```
while true
do
  [ -f datei.tmp ] && break
  sleep 60
done
```

- break bei Benutzereingaben

```
while true
do
  read eingabe
  [ $eingabe == "q" ] && break
  ...
done
```

Endlosschleifen

- Endlosschleifen können auf zwei Arten einfach realisiert werden:
  - Mit einer `while`-Schleife und der Bedingung `true`
  - Mit einer `until`-Schleife und der Bedingung `false`

```
$ cat while_endlos
#!/bin/bash

while true
do
  echo "Endlosschleife"
done
```

```
$ cat until_endlos
#!/bin/bash

until false
do
  echo "Endlosschleife"
done
```

- Solche Schleifen werden in der Regel auf Grund einer Bedingung mit `break` oder `exit` beendet

Endlosschleifen mit `break` beenden

```
while true
do
  Kommandos
  ...
  if [ Bedingung ]
    then break
  fi
  ...
done
```

- Oder einfacher:

```
  ...
[ Bedingung ] && break
```

Zählschleifen (1/2)

- Der Umgang mit Zahlen und numerische Variablen ist in Shell-Skripten nicht intuitiv
  - Die Skriptsprache ist für den Umgang mit Strings angelegt
- Um Zählschleifen der Art `for i=0; i<n; i++` zu realisieren, muss man sich mit einer `while`-Schleife behelfen:

```
#!/bin/sh
n=8;
i=1;

while [ $i -lt $n ]
do
  echo test $i
  i=$((i+1))
done
```

```
$ ./schleife.bat
test 1
test 2
test 3
test 4
test 5
test 6
test 7
```

Zählschleifen (2/2)

- Einfacher ist es, Zählschleifen in Shell-Skripten mit `seq` zu realisieren

```
$ for i in ‘seq 5‘ ; do echo Durchlauf Nr.$i; done
Durchlauf Nr.1
Durchlauf Nr.2
Durchlauf Nr.3
Durchlauf Nr.4
Durchlauf Nr.5
```

```
$ for i in ‘seq 4 7‘ ; do echo Durchlauf Nr.$i; done
Durchlauf Nr.4
Durchlauf Nr.5
Durchlauf Nr.6
Durchlauf Nr.7
```