Shell Scripting Crash Kurs

Was ist eine Shell?

Shell = Hülle um den Betriebsystemkern

Benutzer gibt eine Anweisung ein, diese wird von der Shell interpretiert und in einen Betriebsystemaufruf (auch System-Call genannt) umgewandelt.

Eine Eingabe der Kommandozeile durchläuft einen Interpreter und ist somit nichts anderes als ein Shellscript, das von der Tastatureingabe aus, ohne zwischenzuspeichern, in einem Script ausgeführt wird.

Unix war das erste Betriebsystem das ein vom Betriebsystemkern unabhängiges Programm als Shell implementierte. Diese Shell läuft also als simpler einfacher Prozess. das Kommando `ps` bestätigt dies:

```
rda@deb > ps
  PID TTY      TIME     CMD
 5890 pts/40   00:00:00 bash
 5897 pts/40   00:00:00 ps
```

Bei Unix Systemen ist gewöhnlich die BASH-Shell (`bash`) als Standard-Interpreter aktiviert.

Was ist ein Shellscript?

Ein Kommando oder eine Kommandoverkettung die in der Shell eingegeben wird, zum Beispiel:

```
rda@deb > ls /etc | sort | less
acpi
adduser.conf
adjtime
```

```
aliases
...
```

stellt tatsächlich schon ein Script dar.

Aliase in der Shell

```
rda@deb > alias gohome="cd ~"
rda@deb > cd /usr
rda@deb > pwd
/usr
rda@deb > gohome
rda@deb > pwd
/home/rda
```

Shellscript erstellen und ausführen

Der Editor

Es kann ein beliebiger editor wie `vi`, `vim`, `emacs` oder ein grafischer Editor verwendet werden.

Der Name des Shellscripts

Den Namen unter dem ein Shellscript abespeichert wird kann fast willkürlich frei gewählt werden.

- Es darf kein Name verwendet werden, von dem es ein Kommando mit dem selben Namen gibt. Dies kann mit dem Kommando `which` getestet werden (`which scriptname`)
- Es sollten nur die Zeichen `A` bis `Z`, `0` bis `9` und _ verwendet, auf jeden fall sollten Sonderzeichen vermieden werden.
- Es sollte ein sinnvoller Name verwendet werden.

Ein Script erstellen

Das folgende Listing stellt ein simples Beispiel Shellscript dar:

```
echo "ich bin ein shellscript..."
echo
echo "shellscript ende"
```

Das Script ausführbar machen

```
rda@deb:~/scr$ ls -l
total 4
-rw-r--r-- 1 rda rda 63 2009-09-03 18:30 script
rda@deb:~/scr$ chmod u+x script
rda@deb:~/scr$ ls -l
total 4
-rwxr--r-- 1 rda rda 63 2009-09-03 18:30 script
```

Das Script ausführen

```
rda@deb:~/scr$ ./script
ich bin ein shellscript...

shellscript ende
rda@deb:~/scr$
```

Ein Script bei der Ausführung

Dies soll Anhand des folgenden Scripts demonstriert werden:

```
# Script-Name: finduser

# gibt alle Dateien des Users rda auf dem Bildschirm aus
# leitet die standard Fehlerausgabe nach /dev/null um
find / -user rda -print 2>/dev/null
```

Die Laufenden Prozesse vor dem Ausführen des Scripts:

```
rda@deb:~/scr$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
rda      13785 13784  0 17:30 pts/0    00:00:00 -bash
rda      14098 13785  0 18:52 pts/0    00:00:00 ps -f
```

Das Script in den Hintergrund schicken und die Standardausgabe auf `/dev/null` umleiten:

```
rda@deb:~/scr$ ./finduser 1>/dev/null &
[1] 14101
rda@deb:~/scr$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
rda      13785 13784  0 17:30 pts/0    00:00:00 -bash
rda      14101 13785  0 18:58 pts/0    00:00:00 -bash
rda      14103 14101  3 18:58 pts/0    00:00:00 find / -user rda -print
rda      14104 13785  0 18:58 pts/0    00:00:00 ps -f
```

Anschliessend wird wieder mit `ps -f` die Liste der Prozesse angezeigt. Wie man anhand der PPID (Parent Process ID) sieht wird ein neuer Prozess (PID 14101) von unserer Shell gestartet in dem jetzt das Shellscript abläuft, welches wiederum find (PID 14103) ausführt.
Ist das Shellscript jetzt ein Prozess? Jein, das Shellscript selber ist nicht eigenständig auf einem Prozessor lauffähig, es benötigt ein Interpreter-Programm, nämlich eine Shell, welche das Script in die Systemsprache übersetzt.

Ausführende Shell festlegen

Vorher wurde das Script in einer Bash als (Sub-)Shell ausgeführt. Falls das Script in einer bestimmten Shell ausgeführt werden soll, kann entweder die (Sub-)Shell direkt mit dem Shellscript als Argument aufgerufen oder die auszuführende (Sub-)Shell im Script selbst festgelegt werden.

Aufruf als Argument der Shell

Aufruf mit einer Bourne-Shell (sh):

```
rda@deb:~/scr$ sh finduser
```

Aufruf mit einer Korn-Shell (ksh):

```
rda@deb:~/scr$ ksh finduser
```

Ebenso kann auch eine ash oder zsh werwendet werden. Es könnte auch testweise mit einer C-Shell (bspw. csh) aufgerufen werden, um festzustellen, dass schon hier erste Probleme auftreten.

Ausführende Shell im Script festlegen (She-Bang-Zeile)

Der gwöhnliche Weg besteht darin, die auszuführende Shell im Script selbst festzulegen. Dabei muss in der ersten Zeichenfolge `#!` gefolgt von der absoluten Pfadangabe der entsprechenden Shell befinden:

In einer Korn-Shell:

```
#!/bin/ksh
```

oder

```
#!/usr/bin/ksh
```

In einer Bash-Shell:

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

oder

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

In einer Bourne-Shell:

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

oder

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

Shellscript ohne Subshell ausführen

Um ein Shellscript direkt auf der aktuellen Shell ohne Subshell auszuführen muss ledigilich ein Punkt plus Leerzeichen vorangestellt werden. Dieser Vorgang wird bspw. verwendet, wenn im Script Veränderungen an den Umgebungsvariablen vorgenommen werden sollen. Für diesen Vorgang benötigt der Benutzer nicht einmal Ausführrechte.

```
rda@deb:~/scr$ . ./script
```

Variablen

Dem Script ein Argument übergeben

Einem Shellscript kann wie einem Programm auch ein oder mehrere Argumente übergeben werden, dies wird mit folgendem Script demonstriert:

```
#!/bin/sh
# Script-Name: ux

# $1 ist das erste Argument welches dem Script uebergeben wurde
chmod u+x $1
```

Einen Benutzereigenen Shell Befehl erstellen

Überprüfen ob `ux` als Befehl schon existiert:

```
rda@deb:~/scr$ which ux
```

Der Inhalt von `$PATH` auf den Bildshirm ausgeben:

```
rda@deb:~/scr$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/games
```

Zusätzlichen Suchpfad für benutzereigene Befehle zur Variable `$PATH` hinzufügen:

```
rda@deb:~/scr$ tmppath=$PATH":/home/rda/bin:"
rda@deb:~/scr$ PATH=$tmppath
rda@deb:~/scr$ export PATH
```

`bin` Verzeichnis im Benutzer Home erstellen und `ux` dorthin kopieren:

```
rda@deb:~/scr$ mkdir ../bin
rda@deb:~/scr$ cp ux ../bin
rda@deb:~/scr$ cd ..
```

Den Befehl testen:

```
rda@deb:~$ touch test
rda@deb:~$ ls -l test
-rw-r--r-- 1 rda rda 0 2009-09-03 23:03 test
rda@deb:~$ ux test
rda@deb:~$ ls -l test
-rwxr--r-- 1 rda rda 0 2009-09-03 23:03 test
```

Es muss nicht unbedingt ein bin verzeichnis im home des Benutzers angelegt werden, sondern das Script kann direkt in ein bereits vorhandenes Verzeichnis, der Variable `$PATH`, verschoben werden. (Zum Beispiel: `/usr/local/bin`)

Programmausgaben an Variablen übergeben

Das folgende Listing stellt ein Beispiel Shellscript dar, das den Benutzer und die Zeit ausgibt:

```
#!/bin/sh
# Script-Name: usertime

datum=$(date +%d.%m.%Y)
benutzer=$(whoami)
echo "Das Script wird vom Benutzer" $benutzer "am" $datum "ausgeführt."
```

Das Script bei der Ausführung:

```
rda@deb:~$ vim test
rda@deb:~$ ux test
rda@deb:~$ ./test

Das Script wird vom Benutzer rda am 03.09.2009 ausgefuehrt

rda@deb:~$
```

Ein Script mit einem Script erstellen

Mit folgendem Script wird ein weiteres Script namens `ux` erstellt, welches das User Ausführrecht hinzufügt. Anschliessend wird ein bin Verzeichnis erstellt und `ux` dort hin verschoben und schliesslich wird noch der Pfad zum neuen bin Verzeichnis zur Variable `$PATH` hinzugefügt.

```
#!/bin/sh
# Script-Name: installux

cd $HOME
cat > ux << EOF
#!/bin/sh
# Script-Name: ux
chmod u+x \$1
EOF

chmod u+x ux
mkdir $HOME/bin
mv ux $HOME/bin

tmppath=$PATH":$HOME/bin:"
PATH=$tmppath
export PATH
```

Login-Shell?

Eine Login-Shell ist eine normale Shell, die als erstes Kommando (Prozess) beim Einloggen gestartet wird und beim Hochfahren der kompletten Umgebung (jede Shell hat seine Umgebung) viele andere Kommandos startet. Eine echte Login-Shell erkennt man an der Ausgabe des Kommandos `ps` daran, dass sich vor dem Namen der Shell ein `'-'` befindet (bspw. `-bash`; `-sh` oder `-ksh`). Der Bindestrich ist eine Eigenheit der Funktion `execl()` unter C.

```
login as: rda
rda@192.168.1.10's password:
Linux deb 2.6.26-2-686 #1 SMP Mon May 11 19:00:59 UTC 2009 i686
You have new mail.
Last login: Fri Sep  4 20:33:48 2009 from 192.168.1.2
rda@deb:~$ ps
  PID TTY          TIME CMD
16434 pts/0    00:00:00 bash
16447 pts/0    00:00:00 ps
rda@deb:~$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
rda      16434 16433  5 20:45 pts/0    00:00:00 -bash
rda      16448 16434  0 20:45 pts/0    00:00:00 ps -f
rda@deb:~$ bash
rda@deb:~$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
rda      16434 16433  3 20:45 pts/0    00:00:00 -bash
rda      16449 16434 13 20:45 pts/0    00:00:00 bash
rda      16460 16449  0 20:45 pts/0    00:00:00 ps -f
rda@deb:~$ logout
bash: logout: not login shell: use `exit'
rda@deb:~$ exit
exit
rda@deb:~$ ps -f
UID        PID  PPID  C STIME TTY          TIME CMD
rda      16434 16433  1 20:45 pts/0    00:00:00 -bash
rda      16461 16434  0 20:45 pts/0    00:00:00 ps -f
rda@deb:~$ logout
```

Datenströme

Kanäle

- Standardeingabe (kurz `stdin`; Kanal 0) - gewöhnlich Eingabe von der Tastatur.
- Standardausgabe (kurz `stdout`; Kanal 1) - normalerweise Ausgabe auf dem Bildschirm.
- Standardfehlerausgabe (kurz `stderr`; Kanal 2) - in der Regel auch der Bildschirm.

Kanäle umlenken

1 (stdout)	`cmd > file`	Standardausgabe in Datei umlenken
1 (stdout)	`cmd >> file`	Standardausgabe ans Ende einer Datei umlenken
2 (stderr)	`cmd 2> file`	Standardfehlerausgabe in eine Datei umlenken
2 (stderr)	`cmd 2>> file`	Standardfehlerausgabe ans Ende einer Datei umlenken
1 (stdout) 2 (stderr)	`cmd > file 2>&1`	Standardfehlerausgabe und Standardausgabe in die gleiche Datei umlenken
1 (stdout) 2 (stderr)	`cmd > file 2> file2`	Standardfehlerausgabe und Standardausgabe jeweils in eine extra Datei umlenken
0 (stdin)	`cmd < file`	Datei in die Standardeingabe eines Kommandos umleiten

Pipes

```
cmd | cmd | cmd
```

Leitet `stdout` eines Kommandos auf `stdin` des nächsten Kommandos um.

-- SvenMaeder - 2009-09-04