Desktop ade:
Me and my shell

Erkan Yanar
erkmerk@gmx.de

casus belli

Warum auf einem Linuxtag mit Schwerpunkt Desktop?

Beherrschung der Shell := freie Wahl des Desktops

Beherrschung der Shell := Unabhängigkeit vom Desktop

Üblichen Verdächtigen

Ziel des Vortrags

Meistern der großen Shellstolpersteine

Übersicht

- Quoting
- Subshells
- SUSV3
- Kür

"Quoting"

Quoting

Die Shell ist ein Makroprozessor
Die Kommandozeile wird mehrmals durchlaufen
Danach werden die Programme aufgerufen

Problem

Der mehrmalige Durchlauf

oder

Die Shell mischt mit

Tokenizing/Expansion

- Tokenizing
  - Die Shell unterteilt die Kommandozeile in einzelne Tokens
- Parameter Expansion
  - ${Variablen} werden durch ihren Wert ersetzt
- Command Substitution
  - $() ‘‘ Kommandos werden ausgeführt und das Ergebnis gesetzt
- Word Splitting (Field-Splitting)
  - Die Zeile wird aufgrund vom $IFS (Internal Field Separator) in einzelne Tokens zerlegt)

Hilfe

Was kann ich dagegen tun?

Quoten!!!

Durch Quoten wird die Shell angewiesen auf (eine) Expansion/Tokenizing zu verzichten.

Drei Quoting-Möglichkeiten

Wie kann ich Quoten?

- Backslash \
  - maskiert das folgende Zeichen
  - \$hasch_mich
- Singlequotes ’ ’
  - Maskiert den eingeschlossenen Bereich
- Doublequotes " "
  - Maskieren den eingeschlossenen Bereich "nur" vor dem Word Splitting
  - rm "$wort"
  - rm "mich auch"

Beispiele

wort=hallo welt
wort="platz da"
echo/touch $wort
echo/touch "$wort"
Vorführen?!

```
for i in *
  do  mach_was "$i"
done
```

Frage:

Warum wird * nicht beim Word Splitting zerlegt?

Pathname Expansion kommt nach dem Splitting ;-)

Kurzzusammenfassung

- Das Tokenizing kategoriesiert einzelne Worte
- Parameter Expansion und Command Substitution ersetzen Tokens
- Word Splitting unterteilt die Ersetzungen in Wörter ($IFS)
- Backslash \ und Singlequotes ’ ’ verhindern beim Tokenizing die "Kategorisierung"
- Doublequotes " " ermöglichen beim Tokenizing Wörter mit Trennern
- Doublequotes " " bewahren vor dem Word Splitting

Subshells

Exkurs Prozess/Verzeichnisbaum

Mal ein Tafelbild!

Prozesse rufen via fork-exec andere Prozesse auf

Ein gefork-execter Prozess bekommt eine Kopie der exportierten Variablen

Kindprozesse haben keine Zugriff mehr auf den Mutterprozess

Subshells

Subshells sind nichts weiter als Kindprozesse

Subshells bekommen nur die exportierten Variablen als Umgebung mit

Subshells haben eine eigene Umgebung

Subshells können keine Variablen des Mutterprozesse ändern

Vorführung

chdir Bsp
Shellscript (exp. Variablen)
Shellscript (Addition)

Lösung

Builtins

- Shell-Kommandos
- Erzeugen keinen neuen Prozess
- Performen

Builtin-Liste

Skripte in der aktuellen Shell ausführen:

- . oder source

  Diese Anweisung führt das Skript in der aktuellen, anstatt einer Subshell aus
  source tolles_skript.sh

Weitere Builtins

- cd

  Jetzt kommen wir endlich weg vom Fleck
- export, echo, read, alias, exit

  denk mal darüber nach

Vorführung

Skripte von vorhin mit . und source starten

Brainstorming zu export, echo, read, alias, exit?

Subshells in der Shellprogrammierung

Wo treffen wir auf Subshells?

- Command Substitution
- Pipes
- ()

Subshells-Command Substitution

Command Substitution geschieht in einer Subshell

Es kann nur mit der Rückgabe gearbeitet werden

```
> c=4
> echo $c
4
> $(c=10)
> echo $c
4
```

Subshells-Pipes

Pipes

Die Bash führt die Seiten einer Pipe in einer Subshell aus.

```
> size=0 find /tmp -user erkan -print0 | xargs  -0 stat -c ’%s’| \
  while read c; do size=$((size+c)); echo Zwischensumme: $size;done \
  echo Am Ende: $size
...
Zwischensumme: 35052639
Zwischensumme: 36113595
Zwischensumme: 37174551
Zwischensumme: 38235507
Zwischensumme: 38372267
Zwischensumme: 38509027
Zwischensumme: 38513123
Zwischensumme: 38513123
Zwischensumme: 38513123
Zwischensumme: 38517219
Zwischensumme: 38521315
Zwischensumme: 38521932
Am Ende:
```

Subshells-Pipes

Pipes

Die Bash führt die Seiten einer Pipe in einer Subshell aus.

```
> size=0 find /tmp -user erkan -print0 | xargs  -0 stat -c ’%s’| \
  while read c; do size=$((size+c)); echo Zwischensumme: $size;done \
  echo Am Ende: $size
```

...
Zwischensumme: 35052639
Zwischensumme: 36113595
Zwischensumme: 37174551
Zwischensumme: 38235507
Zwischensumme: 38372267
Zwischensumme: 38509027
Zwischensumme: 38513123
Zwischensumme: 38513123
Zwischensumme: 38513123
Zwischensumme: 38517219
Zwischensumme: 38521315
Zwischensumme: 38521932
Am Ende:

Subshells-Pipes

Etwas einfacher:
$ a=manno
$ echo hallo | read a
$ echo $a
manno

Subshells-()

() gruppiert Anweisungen und führt Sie in einer Subshell aus

nicht verwechseln mit { }

```
$ pwd
/home/erkan
$ (cd /tmp)
$ pwd
/home/erkan
$
```

exec macht sie dauerhaft

```
   exec 1>/tmp/log
```

Kurzzusammenfassung

- Subshells sind neue Prozesse mit eigener Umgebung
- Subshells haben keinen Zugriff auf die Muttershell
- Builtins . und source führen Skripte in der ("Mutter")Shell aus
- Rechne bei Command Substitution und Pipes mit Subshells!

SUSV3

SUSV3

Definiert Standardverhalten für Shells

Größtmögliche Kompatibilität zwischen den Shells

SUSV3+bash

- echo
  - echo verhält sich nicht POSIX-Konform
  - -e der bash ist bei POSIX standard
  - printf benutzen?!
- Pipes
  - Kein Standard zu Subshells und Pipes
  - Ergo: Nur mit der Rückgabe arbeiten

Kür

Kür

- MAX_ARG
- Unangenehme Dateinamen (Newlines)
- Debugging

MAX_ARG

Maximale Länge der Argumentenliste (exec)

Subshells werden via fork-exec erzeugt.

Argumentenliste > MAX_ARG => Fehlermeldung

Achtung, gilt nicht für builtins

$ /bin/ls *
Useless Use of ls *
bash: /bin/ls: Argument list too long
$ echo *
....

find schafft Abhilfe

find ... -exec {} \;

Umgeht via -exec ARG_MAX

Erzeugt für jeden Treffer einen weiteren Prozess
(Subshell)

Kommt auch mit Newlinezeichen zurecht
Die in einer "POSIX"-Pipe verloren gehen.

GNU-find

GNU-find kennt -print0

Kombination mit GNU-xargs -0

find .. -print0 | xargs -0 command
POSIX-find/xargs kennt die Optionen nicht

Schnell und sicher
Gerade bei Dateien mit \newline(Trennzeichen)

Vorführen!

Debugging

Hilfsmittel für die Schnelle

set -x schreibt jedes Kommando nach dem Expandieren und vor der Ausführung auf STDERR

```
$ a=hallo
$ set -x
$ echo $a
+ echo hallo
hallo

$ wort="halasd  asdfaf"
+ wort=halasd  asdfaf
$ touch $wort
+ touch halasd asdfaf
$ touch "$wort"
+ touch ’halasd  asdfaf’
```

set -v schreibt jede Zeile auf STDERR, bevor es expandiert wird

```
$ g="hallo"
$ set -v -x
set -v -x
+ set -v -x
$ eval a=’$g’
eval a=’$g’
+ eval ’a=$g’
a=$g
++ a=hallo
```

Gute Nacht

locale

Achten Sie auf die locale-Settings
Ermöglichen:
Sprachtypische Sortierung -> sort
Reguläre Ausdrücke

```
$LANG=C
$echo  HALLOÄÄÄÄ| tr ’[:upper:]’ ’[:lower:]’
halloÄÄÄÄ
$LANG=de_DE
$ echo HALLOÄÄÄÄ| tr ’[:upper:]’ ’[:lower:]’
 halloääää
```

Letzte Seufzer

rechnen

IFS

read