An diesem Dokument wird aktuell noch gearbeitet!

SLAC 2025: Shell-Vortrag und -Workshop von Martin Schulte

Repository:
`git clone https://slac2025:gldt-8HifPTWs27scJ5mFrqo1@gitlab.com/martin-schulte/slac2025.git`

Wann ist es sinnvoll, in der Shell zu programmieren?

Empfehlung: Man sollte sich über die sinnvolle Einsatzgebiete von Shell-Scripten klar werden.
Eine Shell ist meist in sehr rudimentären Systemen verfügbar, zum Beispiel in frühen Phasen der Installation. Zudem ist die Shell Architektur-unabhängig. Meist ist der Einsatz auch dann sinnvoll, wenn im Wesentlichen vorhandene Programme aufgerufen werden sollen. Dann können Aufgaben auch CPU- und Speicher-schonend ausgeführt werden.

Eher ungeeignet ist die Shell, wenn die Programme viel Logik enthalten, komplexe Datenstrukturen oder binäre Daten verarbeitet werden müssen oder Laufzeit ein Thema ist.

Shell-Programmierung kann trotzdem dieser Einschränkungen sinnvoll sein, wenn bereits eine große Code-Basis vorhanden oder kein Personal mit Kenntnisssen in geeigneteren Sprachen vorhanden ist. Das relativiert auch die Anmerkungen am Anfang von Google's Shell Style Guide.

Für viele Aufgaben im Bereich der Systemadministration ist `python` gute Alternative zur Verwendung einer Shell.

Welche Shell verwenden?

Prinzipiell dürften wohl alle Shells, die in der Linux-/Unix-Welt zur Verfügung stehen, zum Erstellen von Skripten geeignet sein. In den meisten Fällen kommen allerdings die Abkömmlinge der Bourne-Shells zum Einsatz. Als kleinster gemeinsamer Nenner ist hier der POSIX-Standard zu sehen, `https://de.wikipedia.org/wiki/POSIX` gibt einen guten Überblick über die verschienden Versionen. Zumindest unter Debian 11/12 wird dieser Standard (und auch nicht mehr) von der `dash` implemtiert, auf die auch `/bin/sh` verlinkt.

Sollen Skripten auch auf älteren Systemen oder zum Beispiel in der BSD-Welt laufen, kann es sinnvoll sein, nur die Features zu nutzen, die im POSIX-Standard enthalten sind.

Den POSIX-Standard (zumindest in guter Näherung) implementieren auch die `bash`, `ksh` und `zsh` – sie bieten aber zusätzliche und untereinander ähnliche Funktionalitäten. Da die `bash` in den meisten Distributionen direkt installiert wird, ist sie eine sinnvolle Wahl, wenn Feature, die nicht im POSIX-Standard enthalten sind (wie zum Beispiel Arrays und Vergleiche mit doppelten eckigen Klammern, genutzt werden sollen. Alle weiteren Angaben in diesem Dokument beziehen sich auf die `bash`.

Die richtige Version verwenden

Sowohl der POSIX-Standard (zuletzt 2024!) als auch die anderen genannten Shells entwickeln sich weiter. Läuft auf einem Zielsystem eine ältere Shell als auf dem Entwicklungssystem, kann das zu Problemem führen. Daher sollte dokumentiert werden, für welche Versionen der ausgewählten Shell ein Script entwickelt wurde – ggf. sollte auch zu Beginn Code eingefügt werden, der auf die richtige Shell und die richtige Version prüft.

In `bash`, `ksh` und `zsh` stehen dafür die Variablen `BASH_VERSION`, `KSH_VERSION` und `ZSH_VERSION` zur Verfügung, in der `bash` gibt es auch das Array (siehe unten) `BASH_VERSINFO`.

shebang

Siehe 01-shebang im Repository

In welcher Shell läuft das aktuelle script: `readlink /proc/$$/exe # eventuell realpath statt readlink nehmen`

Double Quotes

Wird ein Variableninhalte in der Shell nicht innerhalb von Double-Quotes referenziert (`$varname`), so wird wird der Wert noch Word-Splitting und Wildcard-Expansion unterzogen. Das kann unerwartete Folgen haben (wobei das Ergebnis je nach den Verzeichnisinhalten variieren kann):

```
$ msg='ho* christmas'
$ cd /home
$ echo $msg # echo wird mit 3 Argumenten aufgerufen:
ho* Santa Claus
$ cd /etc
$ echo $msg # echo wird mit 7 Argumenten aufgerufen:
host.conf hostname hosts hosts.allow hosts.deny Santa Claus
```

Single Quotes

Immer in Single-Quotes sezten, um eine Interpretation von Sonderzeichen für das jeweilige Programm durch die Shell zu verhindern:

- Den regulären Ausdruck bei `grep`
- Den Wert hinter `-name` bei `find`
- (`useradd ... -c 'Hans Meyer' ...`)

String-Verarbeitung

POSIX IEEE 1003 seit mindestens der 2004 Edition (siehe Parameter-Expansion unter `https://pubs.opengroup.org/onlinepubs/9799919799.2024edition/`).

```
  ${parameter:-word}  ${parameter-word}
  ${parameter:=word}  ${parameter=word}
  ${parameter:?word}  ${parameter?word}
  ${parameter:+word}  ${parameter+word}
  ${#parameter}
  ${parameter%word}   ${parameter%%word}
  ${parameter#word}   ${parameter##word}
  ${#parameter}
  ${parameter%[word]}
  ${parameter%%[word]
  ${parameter#[word]}
  ${parameter##[word]}
```

bash >= ?.?

```
  ${parameter:offset}            ${parameter:offset:length}
  ${parameter/pattern/string}    ${parameter//pattern/string}
  ${parameter/#pattern/string}   ${parameter/%pattern/string}
  ${parameter^pattern}           ${parameter^^pattern}
  ${parameter,pattern}           ${parameter,,pattern}
```

bash >= 4.4

```
  ${parameter@Q}
  ...
```

Hiermit lassen sich Konstrukte wie `echo "$var" | cut -c 1-3` performant und – zumindest in der `bash` – auch bei entsprechender Setzung der Variablen `LANG` auch UTF-8-aware ersetzen.

Code-Formatierung

Folgende Punkte sind die wichtigsten im Bezug auf die Formatierung von Shell-Scripten:

Einrücktiefe

Das Innere von `if`-Blöcken oder `while`-Schleifen sollte aus Gründen der Lesbarkeit eingerückt werden, typischerweise werden hier 2, 4 oder 8 Leerzeichen beziehungsweise eine Tabulator genutzt.

Platzierung von then und do

```
# then/do in eigene Zeile
if cmd1
then
  cmd2
  cmd3
fi

while cmd4
do
  cmd5
  cmd6
done
```

```
# then/do mit Semicolon abgetrennt hinter if/while
if cmd1; then
  cmd2
  cmd3
fi

while cmd4; do
  cmd5
  cmd6
done
```

Hier sollte ein einheitlicher Standard innerhalb der Firma, Organisation oder Behörde etabliert werden, siehe dazu dazu den Abschnitt zum Style Guide, insbesondere `https://google.github.io/styleguide/shellguide.html#s5-formatting`.

`[ ]` und `[[ ]]`

`[` ist ein normales (POSIX-)Kommando, das in die meisten Shells eingebaut ist. Abgesehen davon, dass es als letztes Argument eine `]` benötigt, ist es identisch zum Programm `test`.

Ausflug: Warum werden Programme in die Shell eingebaut?

Es gibt ein Reihe von Gründen, warum kleine(!) Programme in eine Shell eingebaut sind:

- Programme, die nur funktionieren, wenn sie in die Shell eingebaut sind: cd, exit, type, source, jobs, set
- Programme, die oft in Scripten aufgerufen werden, werden aus Performance-Gründen in die Shell eingebaut, zum Beispiel `[`.
- Programme, die in die Shell eingebaut sind, können möglicherweise mehr (siehe zum Beispiel die Option `-v` unter `help test`)

Dieser Absatz gilt nicht für Shells wie sie von busybox zur Verfügung gestellt werden.

$(( )) und (( ))

(integer) indexed Arrays

Associative Arrays

Shellcheck

`shellcheck` analysiert Shell-Scripten in verschiedenen Dialekten (POSIX, bash, ksh, zsh) und findet dabei erstaunlich viele Hinweise auf (potentielle) Probleme. Unter Debian kann es über apt install shellcheck installiert werden.

Mehr unter `https://www.shellcheck.net/`, alle Warnungen sind sehr gut auf `https://www.shellcheck.net/wiki/` dokumentiert.

Warnungen von shellcheck können für ein ganzes Script oder für einzelne Zeilen ausgeschaltet werden:
`# shellcheck disable=SC....`

Empfehlung: `shellcheck` verwenden!

`shellcheck` kann über die asynchronous lint engine mit dem vim (aber auch anderen IDEs) verheiratet werden, Infos dazu in der Datei `vimrc` im Repository.

Die Verwendung von shellcheck ist ein must-have!

(Google's) Shell Style Guide

Die zuvor genannten Themen werden in Google's Shell Style Guide angesprochen, der unter `https://google.github.io/styleguide/shellguide.html` zu finden ist.

Empfehlung: Google's Shell Style Guid gründlich zu lesen und – ggf. in angepasster Form – für die eigene Firma, Behörde, Organisation verbindlich zu übernehmen!