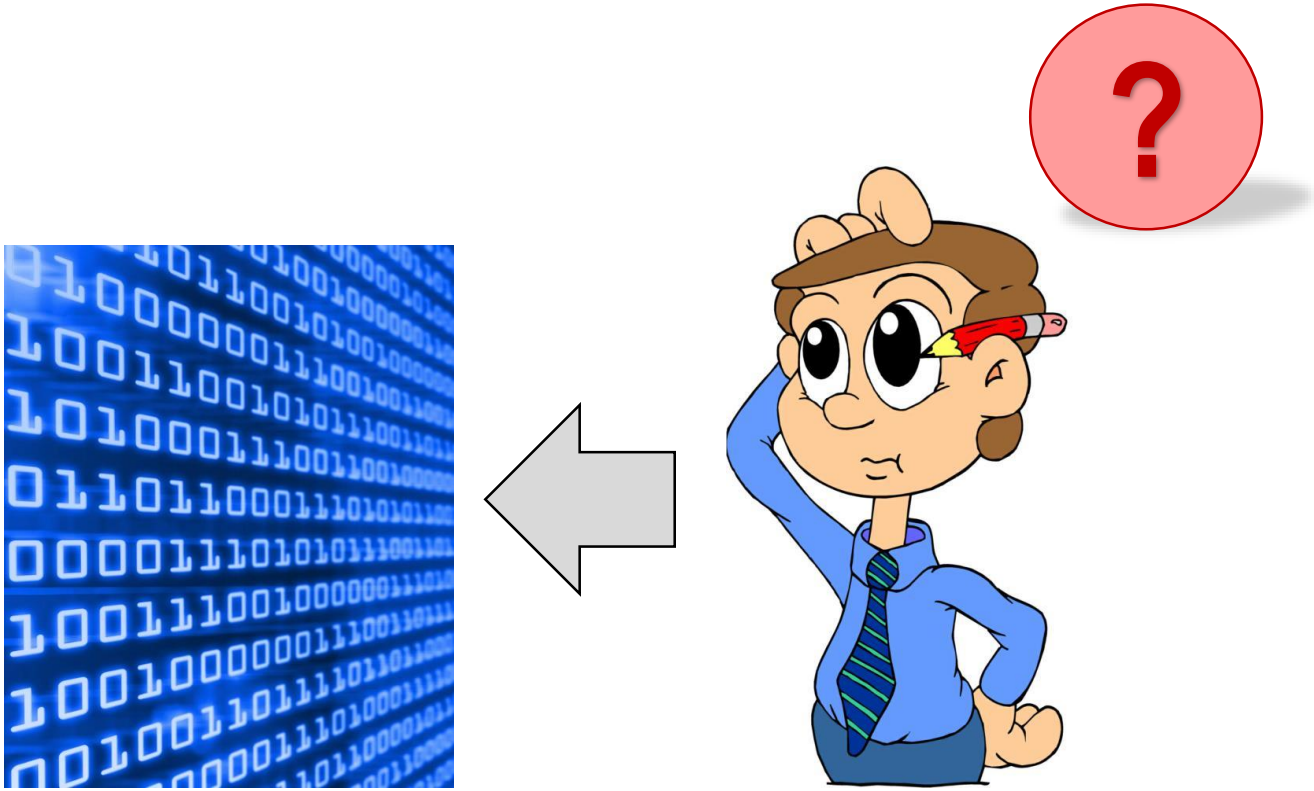
Bash Programmierung Teil 2 - Fortgeschrittenes

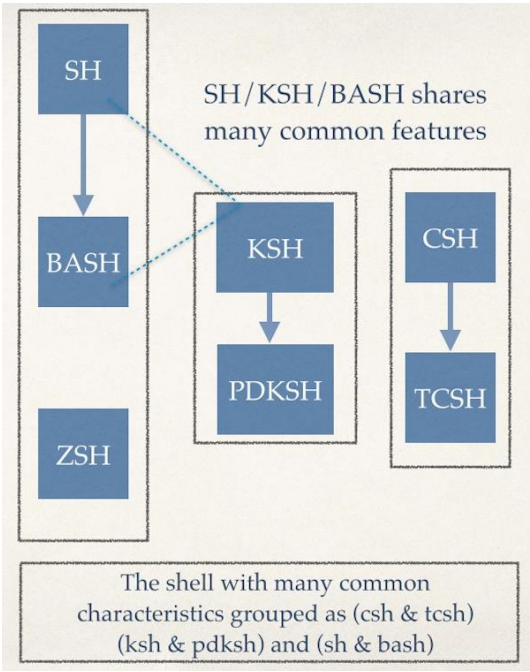


Themen

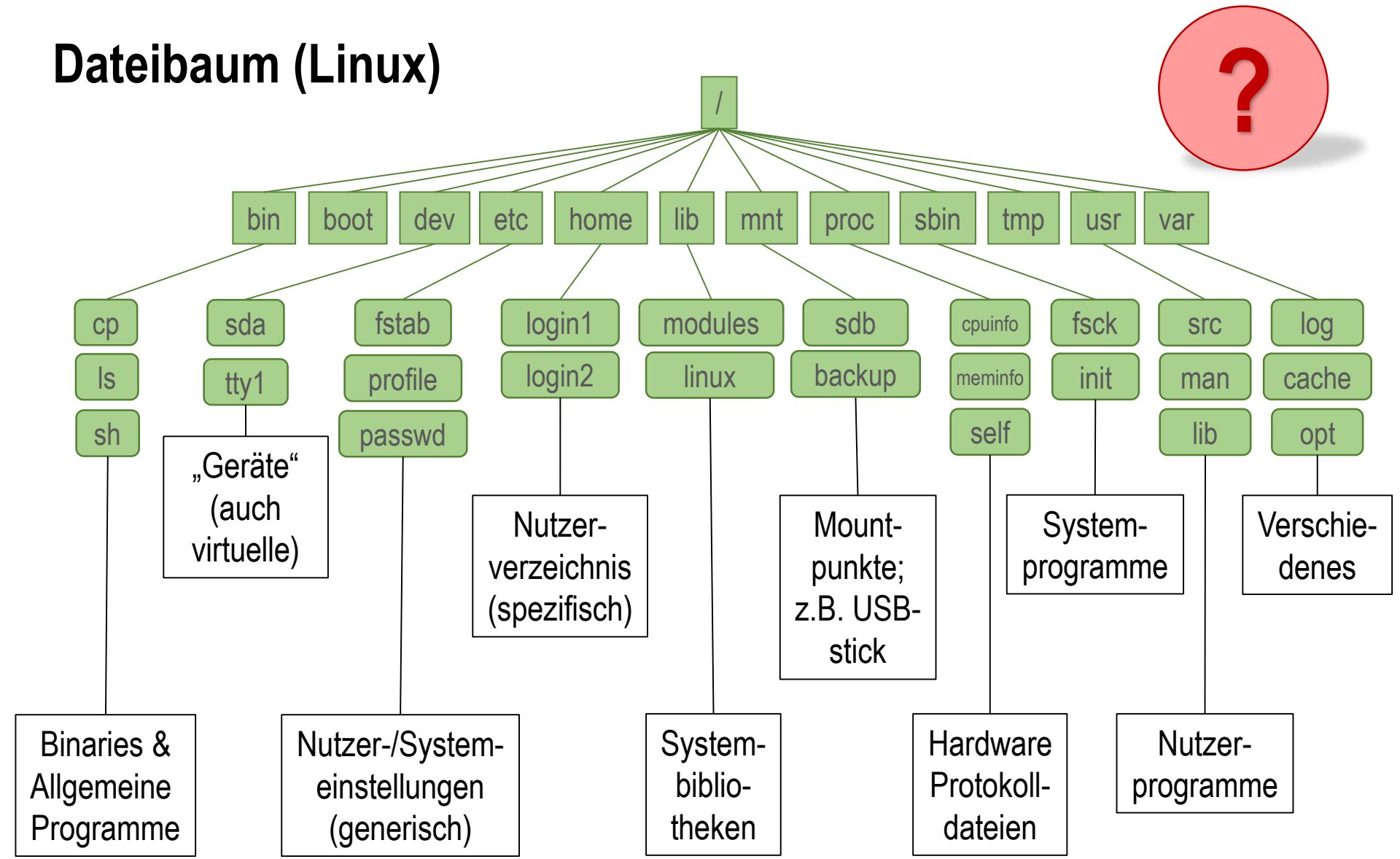
- Bash-Systemumgebung und Systemvariablen
  - Wichtige Systemvariablen
  - Systemumgebungen
  - Welche „flavors“ der bash gibt es?
  - Aliase
- Bash-Skripte und Beispiele
  - Funktionen
  - Reservierte Wörter
  - Startoptionen der bash
  - Hinweise zur Sicherheit / Manipulierbarkeit

Kurze Erinnerung: shells und bash

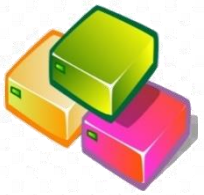
- https://en.wikipedia.org/wiki/*List_of_command-line_interpreters*
- Command-line = Kommandozeile für das System
  - Herzstück von Unix (-> Linux, Mac)
    bzw. MSDOS (-> Windows)
- Interpreter = Skriptsprache
- Fokus hier: Bourne-again shell (Bash)
  [windows10 (> 2017): windows subsystem for linux (WSL);
  sonst: „git extensions“]
- Nützliche Links:
  - https://devhints.io/bash
  - https://www-user.tu-chemnitz.de/~hot/unix_linux_werkzeugkasten/bash.html



Dateibaum (Linux)



Übung:



Parsen von /proc/meminfo: die ersten drei Zeilen; nur die Zahlenwerte ermitteln

```
$ cat /proc/meminfo
MemTotal:       65690788 kB
MemFree:          489700 kB
MemAvailable:   34670800 kB
Buffers:              48 kB
Cached:         14466416 kB
[…]
```

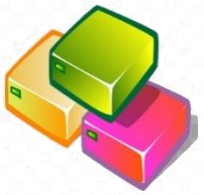
```
$ cat /proc/cpuinfo
processor       : 55
[…]
core id         : 14
cpu cores       : 14
[…]
```

3 Zeilen — von Datei — Übergebe das Resultat an *awk*

```
$ head -3 /proc/meminfo | awk
'BEGIN{FS=":"}{gsub("[^[:digit:]]", "", $2); print $2}'
65690788
489700
34670800
```

Feldtrenner — Entferne *alle* Nicht-Ziffern aus 2. Spalte — Ausgabe 2. Spalte

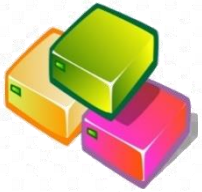
Systemvariablen

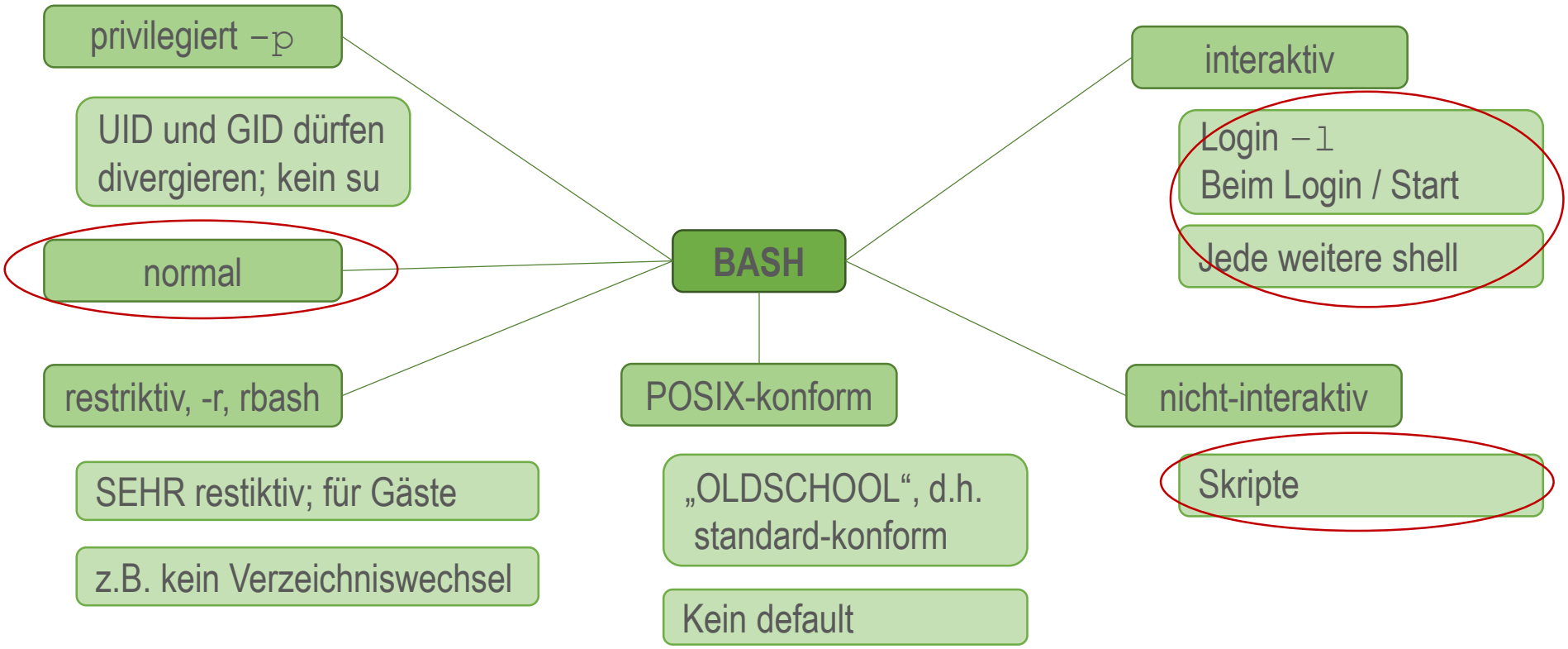


- Erinnerung: Bash-Variablen `$NAME`, immer vom Typ `STRING`
- Bash = Systeminterpreter; Variablen v.a. für Pfade gedacht (!)
  - Trotzdem mathematische Operationen möglich (nativ mit Ganzzahl: `a=1; echo $((a+1));` alternativ: `bc`)
- Abfrage Systemvariablen: `env` (=„der“ Einstieg in das System; HANDLE WITH CARE!)
- Kleine Übung: mit awk eine reine Liste von Systemvariablen erstellen (ohne Werte)

Maschine	Nutzer	CMD	Bibliotheken
$HOSTNAME	$USER	$SHELL=/etc/bash	$PATH=/usr/bin
$SHOST	$MAIL	$TERM=xterm	$PYTHONPATH
$SSH_CONNECTION	$LANG	$HISTSIZE=1000	$LD_LIBRARY_PATH
$SSH_CLIENT =134.109.x.x xx 22	$HOME =/home/$USER	$PWD / $OLDPWD =current / last dir	$INCLUDE =/usr/include
$SSH_TTY =/dev/pts/0	$LOGNAME	$_=*last cmd* $IFS=*int. field separator*	$MANPATH =/usr/share/man

Welche bash-Modi gibt es?





Was passiert beim Start einer shell?

Konfigurationsdateien sind bash-scripte, die Umgebungsvariablen, Funktionen, Aliase, u.a.m. festlegen

Bash

Stark vorkonfiguriert

1. `/etc/profile` – SYSTEM!
2. Jede shell (falls nicht `--noprofile`):
   1. `~/.bash_profile`
   2. `~/.bash_login`
   3. `~/.profile`
3. Interaktive shell:
   1. `~/.bashrc`
   2. mod: `--norc / --rcfile`
4. Nicht-Interaktive shell :
   1. `$BASH_ENV`
5. Ende Login-shell
   1. `~/.bash_logout`

sh

Nur bei login vorkonfiguriert!

1. Jede login-shell (falls nicht `--noprofile`):
   1. `/etc/profile` – SYSTEM!
   2. `~/.bash_profile`
2. Interaktive shell :
   1. `$BASH_ENV`

- `--norc / --rcfile` wirkungslos
- Keine login-shell: keine Konf.-Dateien!
- POSIX-Modus

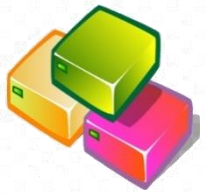
Nützliche Angaben in einer ~/.bashrc bzw. ~/.bash_profile

```
# .bash_aliases
alias cd..='cd ..'
alias ..='cd ..'
alias ...='cd ../..'
alias sl='ls'
alias cluster='ssh -X <login>@cluster.etit.tu-
chemnitz.de'
alias tuc='ssh <login>@login.tu-chemnitz.de

# simplified bash calculator
calc() { echo "$@" | bc -l; }
# mkdir + cd at once
mcd() { mkdir -p $1; cd $1}

# unit conversions
BOHR2ANG=0.52918 # 1 Bohr  = ... Ang
ANG2BOHR=1.88972 # 1 Ang   = ... Bohr
EV2CM=8065.73    # 1 eV    = ... cm-1
CM2EV=0.000124   #
EV2NM=0.00080657 # 1 eV    = ... nm-1
NM2EV=1239.84    #
HA2EV=27.2107    # 1 Ha    = ... eV
EV2HA=0.03675    #
RY2EV=13.60535   # 1 Ry    = .. eV
EV2RY=0.0735     #
EV2KJ=96.4869    # 1 eV    = ... KJ/MOL
KJ2EV=0.010364   #
```

```
#.bash_profile
if [ -f ~/.bash_aliases ]; then
        . ~/.bash_aliases
fi
```



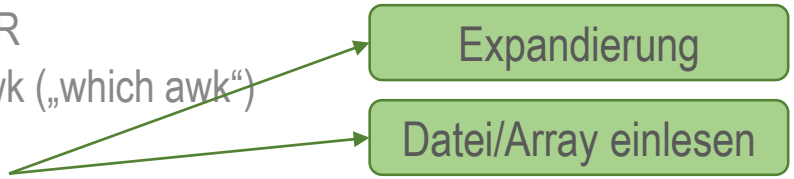
Beispiele

```
$ calc 1+1
2
$ calc "5*4"
20
$ calc $NM2EV/1200 # eV -> nm
1.033200000000000000
$ calc $NM2EV/1.5 # nm -> eV
826.560000000000000000
$ calc 5*$KJ2EV
.051820
```

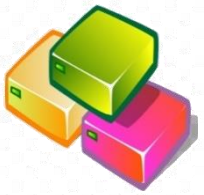
Was passiert beim Ausführen eines Kommandos?

Expandierung vor Ausführung, d.h.

1) History-Expandierungen, z.B. ! !! (siehe ~/`.bash_history`)
2) Zerlegung in Wörter (=Token) (Feldtrenner = space, tab)
3) Alias-Expandierungen, z.B. .. -> cd ..
4) Expandierung geschweifter Klammern (brace expansion)
   1) {1..5} = 1 2 3 4 5; a{1..5}b = a1b a2b a3b a4b a5b
5) Tilde-Expandierung (tilde expansion)
   1) ~user = /home/user (falls existent); ~ = /home/$USER; ~+ = echo $PWD; ~xy = ~xy
6) Von links nach rechts, simultan:
   1) Parameter / Variablenersetzung z.B. $1; $VAR
   2) Kommandoersetzung, z.B. awk -> /usr/bin/awk („which awk“)
   3) Arithmetische Expandierung, z.B. $((a+1))
7) Aufspaltung in Wörter (word splitting) mit **$IFS** oder space/tab/break
8) Pfadnamens-Expandierung (pathname expansion), z.B. ./*/test?.txt
9) Entfernung der Quotierungs-Zeichen (quote removal)



Ein schöner Fallstrick



Manchmal muss man eine weitere Auswertungs-Runde mittels **eval** erzwingen:

```
# der Reihe nach in die Home-Verzeichnisse der Nutzer egon, berta und anton
# wechseln (um dort bestimmte Kommandos auszuführen)
for user in egon berta anton; do
    eval cd ~$user # normalerweise: cd ~$user, aber tilde-expansion wirkungslos
    # Da die Tilde-Expandierung vor der Parameter-Expandierung erfolgt, führt
    # das (bei tcsh, ksh und zsh auf Grund einer anderen Expandierungs-
    # Reihenfolge wie gewünscht funktionierende) Kommando
    #
    # cd ~$user
    #
    # bei der Bash dazu, dass an das Kommando cd ein Argument der Form ~egon
    # übergeben wird, was einen Fehler zur Folge hat. Um den Ausdruck ~egon
    # durch den absoluten Pfad zum Home-Verzeichnis von Nutzer egon ersetzen
    # zu lassen, muss eine weitere Auswertung durch das Kommando eval
    # erzwungen werden. Aber: HANDLE WITH CARE!
    echo $PWD
done
```

Skripte

- Müssen strukturiert werden
  - Aliase
  - Funktionen
  - mehrere Skripte
    - Aufruf einer bash/sh: sh skript2.sh
    - source
- Skripte *und* Funktionen müssen Parameter enthalten dürfen
  - $0 / $*
  - $# / $@
  - $1 / $...

Aufbau eines Skriptes

```
#!/bin/sh # („shebang“: /bin/(r)bash | /bin/python | …)
# Kurzbeschreibung durch Autor, Copyright, evtl. Lizenz
# ggfs. weitere Dokumentation

# Präambel (Hier werden ein paar Restriktionen festgelegt)

# Auswertung der Übergabeparameter $0, $1, $@, $*, $#

# Funktionen und Definitionen
# könnte man in einer extra Datei auslagern
source functions.sh

# Hier geht es erst richtig los
```

Ziel:

Verfolgen der Arbeitsspeichernutzung des Systems (1x pro Sekunde), bis Prozess beendet; danach grafische (hübsche) Ausgabe mit gnuplot

Skript-Debugging

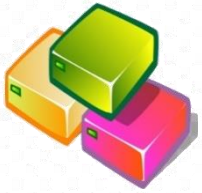
```
# Debugging-Präambel
# shopt/set modifizieren die Startoptionen der bash nachträglich
set -e # Bei Fehler sofort aufhören, Kommando nicht beenden
set -u # Fehler, falls nicht-gesetzte Variable verarbeitet wird
set –x # Print commands and their arguments as they are executed.
set -n # Read commands but do not execute them.

set +e # Bei Fehler fortfahren; Gegenteil von set –e
Set –e –u # wird gern genommen (Kombinationen möglich)

# Ein Beispiel: Leeren von /tmp-Ordner…
TMP='/tmp'
sudo rm –rf $TNP/* # und hopps, das System ist weg… $TNP ist leer

# Hinweis: Debug-Optionen einbauen, z.B. mit --debug oder --dryrun
(-d) und entsprechend set -x/-n aktivieren.
```

Now, write and go… ☺



Instructions for download…

TO DO ->

Notizen zur Skript-Sicherheit

Wann relevant?

- Wenn nicht nur der Admin/Nutzer Zugriff auf Skripte bekommen könnte
- Wenn Skripte indirekt über einen Webserver ausgeführt werden

Welche Angriffsszenarien sind denkbar?

- unzählige; immer beim input/output verarbeiten (injection); z.B.
  - Nutzereingaben enthalten bash o.a. Befehle (v.a. via Webserver)
  - Manipulation von Umgebungsvariablen und aliases
  - Manipulation von Systemdateien über symbolische Dateilinks
    (z.B. upload eines softlinks auf /etc/passwd -> Ausspähen von Domänenutzern möglich!)
  - Upload von Dateien mit bash-Befehlen im Namen (dank UTF8-Dateisystem alles möglich!)
  - Etc. pp. (der Kreativität sind keine Grenzen gesetzt)
- Vor allem, wenn Nutzereingaben an die shell weitergegeben werden
  - Kommt häufiger vor als man denkt, z.B. TXT=$(cat $1) und $1=„test.txt; do evil &2>1 1>/dev/null;“

Notizen zur Skript-Sicherheit

Weitere Hinweise:

- `#!/bin/sh` als Interpreter nutzen (Anfälligkeit für manipulierte Konfigurationsdateien)
- Die Fehler-Codes der aufgerufenen Kommandos sind auszuwerten (Fehlerpipes ins Nirvana fallen ja nicht auf)
- Das Kommando logger erzeugt syslog-Einträge (automatisch mit Zeitstempel), z.B.:
  `logger -p daemon.info "skript gestartet von $USER $(id -un)`
- Eingaben direkt als string verarbeiten; niemals „eval“ unterwerfen! Z.B. "$1" "$*"
- Aufpassen bei eval, $(), …
- Symbolischen Links sollte man nicht blind folgen
  - Optionen -L und -h des Kommandos test verwenden, die funktionell äquivalent sind:
  - test -L file / test -h file

Notizen zur Skript-Sicherheit

```
#Sicherheits-Präambel
# Angriffe über aliase abfangen, z.B. alias cd="do evil; cd"
unset -f unalias      # unalias könnte eine Funktion sein
\unalias /f           # \ fängt alias-Definitionen von unalias ab

# Angriffe über Pfad-Variable abfangen
unset -f command
SYSPATH="$(command -p getconf PATH 2>/dev/null)"
  if [[ -z $SYSPATH ]]
  then
    # getconf schlug fehl; eigenen sinnvollen Standard setzen
    SYSPATH="/usr/bin:/bin"
  fi
  PATH="$SYSPATH:$PATH„

# Angriffe über IFS abfangen:
IFS=$' \t\n'
```

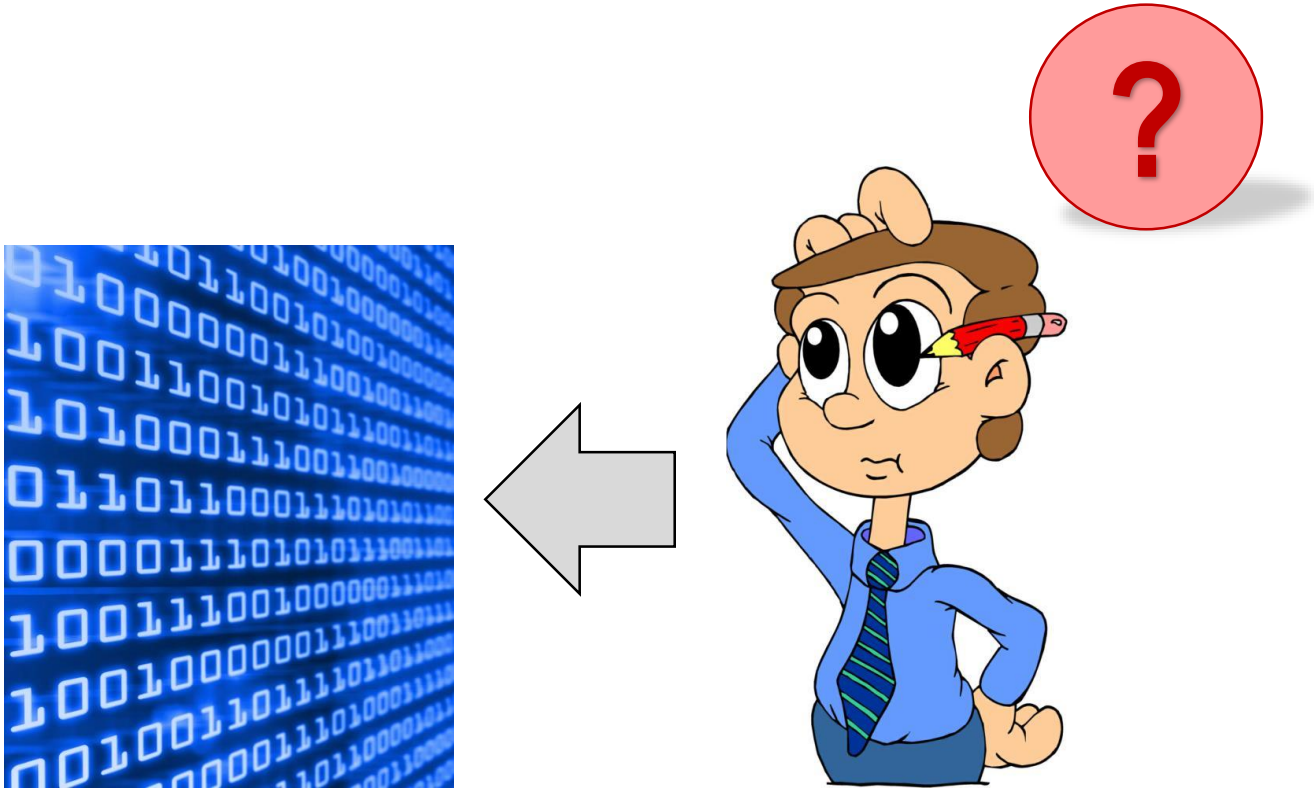
Notizen zur Skript-Sicherheit - ein Beispielangriff -

Was passiert bei folgendem Aufruf und wie könnte man das verhindern?

```
./script.sh "/proc/meminfo; curl
'https://ruthe.de/cartoons/strip_2213.jpg' > bang.jpg 2>/dev/null;
eog bang.jpg"
```

Vielen Dank für eure Aufmerksamkeit!

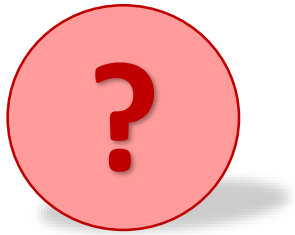
Bash Programmierung Teil 2 - Fortgeschrittenes



Ein paar Folien von Teil 1 als reminder

Shell und Linux-Programme

- Hinter jedem Befehl in der shell steckt ein Linux-Programm
- Befehle können Optionen (-a) und/oder Argumente aufnehmen
- Grundsätzliche *Systembefehle*:
  - *cd* folder # folder = Argument
  - *ls* --color # --color = Option
  - *cp* a b # a, b = Argumente
  - *ln* -s a # Option und Argument (erstellt einen symbolischen Link)
  - *mkdir* ab # Argument
- Hilfen
  - www.google.com
  - man <Befehl> (Beenden mit „q“)
  - info <Befehl>
  - <Befehl> -h/-H/--help



Schleifen

- `For i in <LIST>; do …; done`
  - <*LIST*> kann sein
    - Bash-expansion (/path/to/files/*)
    - Range ({1..5} oder {1..10..2})
    - Allgemein: Eine Liste von strings, getrennt mit <space>, <tab>, etc.
- `while <condition>; do …; done`
  - <*condition*> kann jede bash-condition sein (true, false, [?], test ?)
- `< file.txt | while read line; do echo $line; done`
  - Dateien zeilenweise auslesen (im Vergleich zu `cat`)

Weitere nützliche Werkzeuge und Befehle

Werkzeuge
- sed
- awk
- zip
- tar
- bc

Befehle
- top
- screen
- kill
- date
- time

Arbeiten mit Textdateien:
- cat / tac
- head / tail
- cut / paste / sort

Bash Texteditoren

- Vi(m)
- nano/pico
- emacs

- sed
- tr

Grafische Editoren

- Gedit (gnome)
- Mousepad (xfce)
- Kwrite (KDE)
- Atom, sublime

IDE

- Eclipse
- Visual Studio
- Spyder
- …

Nützliche Hotkeys

• Befehl abbrechen:	Strg+C
• Den letzten Befehls nochmal ausführen	!!
• Das letzte Argument nochmals nutzen	!$
• Vervollständigung in der Kommandozeile	Alt+Shift+*
• Durchsuchen der Historie	Strg+R
• Prozess anhalten	Strg+Z
• Angehaltenen Prozess wieder fortsetzen	fg
• Blockieren des Terminals	Strg+S
• Fortführen des Terminals	Strg + Q
• Schließen des Inputs	Strg+D
• Den Terminal leeren	Strg + L (auch über Befehl „clear“)

Und vieles mehr, siehe z.B. https://ss64.com/bash/syntax-keyboard.html

Reguläre Ausdrücke (regular expression, Regex)

- Definition von Suchmustern

- Anwendbar in vielen Programmiersprachen oder Texteditoren (z.B. Suchen und Ersetzen)
- Teils Unterschiede im Syntax

- Nützliche Links:
  - https://www.rstudio.com/wp-content/uploads/2016/09/RegExCheatsheet.pdf
  - https://www.rexegg.com/regex-quickstart.html
  - https://remram44.github.io/regex-cheatsheet/regex.html

Beispiel: doi-extraction (.sh)

- Suchen eine doi aus einem pdf-file
- Definition einer DOI: https://en.wikipedia.org/wiki/Digital_object_identifier
- Regex für doi:
  `re_doi='10[.]\d{4,}[.][\d+][/][?!.]'`
- Umwandlung pdf -> Text (erstellt any.txt):
  `pdftotext any.pdf`
- Nun doi extrahieren mit grep (any.txt sollte existieren)
  `grep -oP "$re_doi" any.txt`
- Nun Abfrage Crossref nach .bib-string (getbib = nebenstehende Funktion):
  `getbib $doi >> bibfile.bib`

prefix	/	suffix
10.NNNN.N	/	anything

```
# get bibliographic information of given doi
function getbib {
        curl -LH "Accept: text/bibliography;
style=bibtex"  "https://doi.org/$1"
        echo
}
```