BERUFSAKADEMIE SACHSEN
STAATLICHE STUDIENAKADEMIE
DRESDEN
UNIVERSITY OF COOPERATIVE EDUCATION

2

OOP – Linux

Arbeiten mit der Kommandozeile

mit Skriptmaterial von Dr.-Ing. M. Feldmann

Prof. Dr.-Ing. Tenshi Hara
tenshi.hara@ba-dresden.de

Gliederung der Vorlesung

Einführung: Geschichte von Unix zu Linux

Kapitel 1: Allgemeines und Grundlagen

Kapitel 2: Arbeit mit der Kommandozeile

Kapitel 3: Boot-Vorgang und Systeminitialisierung

Kapitel 4: Ausgewählte Themen der Systemadministration

Kapitel 5: Ausgewählte Themen der Netzwerkkonfiguration

Kapitel 6: Anwendungsentwicklung unter/für Linux

Kapitel 7: Ausgewählte Themen zu Web-Servern

INHALTE

- Allgemeines zur Interaktion mit der Shell
- Grundlagen zur Verwendung der Bash
- Shell-Skripte

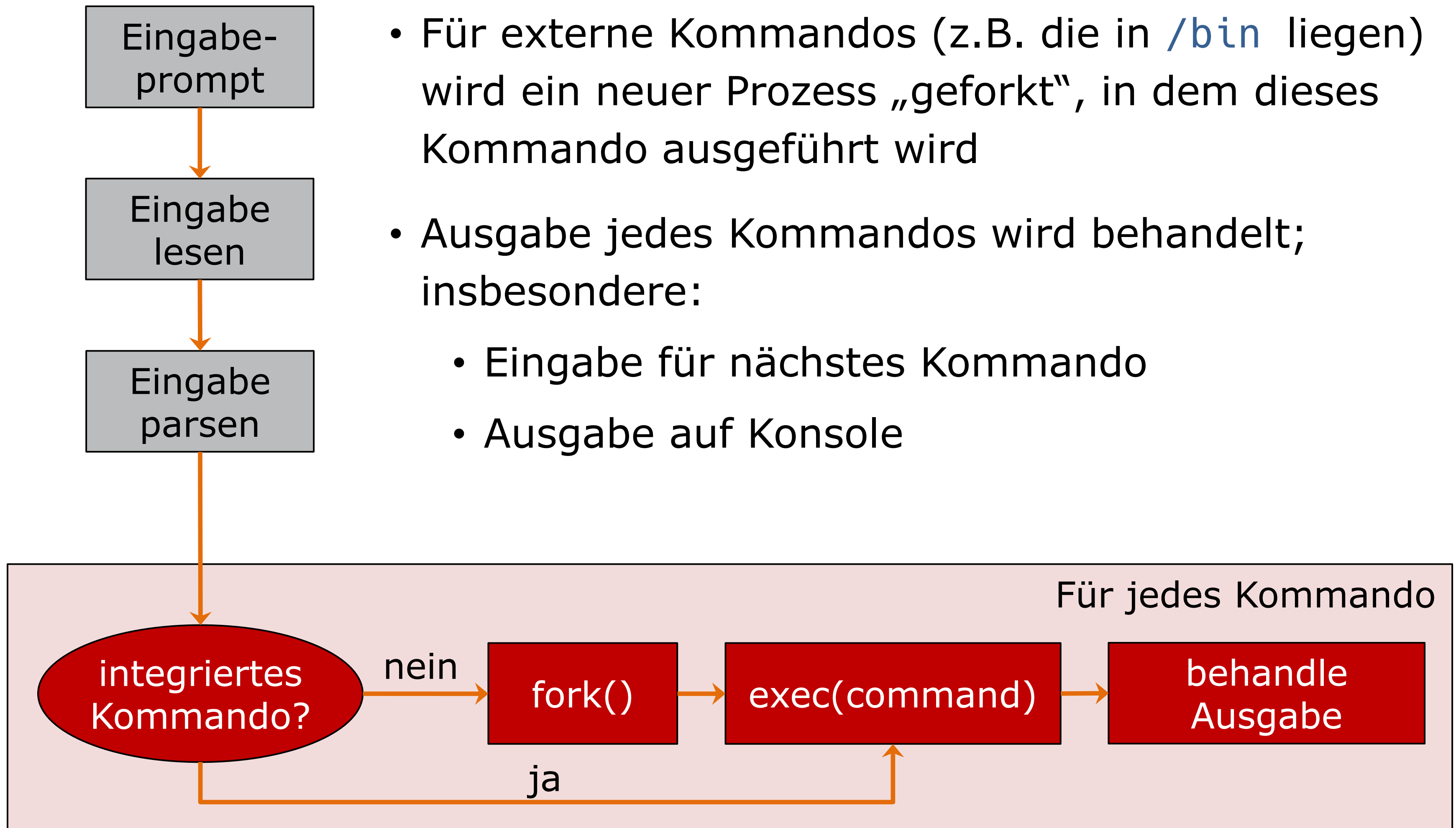
Motivation

- in GUIs sind viele Operationen nur ineffizient auslösbar
- dagegen in CLI sehr einfache Interaktion über Ein-/Ausgabe von Text
- anfangs sehr steile Lernkurve für Nutzer
  → allerdings: bei vielen Aufgaben sehr effizient
- weitere Vorteile:
  - ermöglicht über Skripte die Automatisierung zahlreicher Kommandos
  - leicht über Netzwerk/Internet bedienbar (z.B. SSH), dabei geringes Datenaufkommen

Unix-/Linux-Shells

- im Kontext dieser LV: Shell = Kommandozeileninterpreter
- traditionelle Benutzerschnittstelle unter Unix/Linux
- unterschiedliche Shells verfügbar
  - meistens gemeinsamer Minimalfunktionsumfang:
    Executables, Wildcard-Unterstützung (z.B. *), Umgebungsvariablen, Ein-/Ausgabeumleitung, Prozessparallelisierung, Kommandovervollständigung, Prozesskontrolle, Skripte, ...
  - oft gemeinsame Eigenschaften
- Verbreitete Shells:
  - Bourne-again-Shell (Bash) → Gegenstand dieser LV
  - Z-Shell (zsh)
  - Friendly Interactive Shell (Fish)

Schematische Übersicht zur Shell



- Für externe Kommandos (z.B. die in `/bin` liegen) wird ein neuer Prozess „geforkt", in dem dieses Kommando ausgeführt wird
- Ausgabe jedes Kommandos wird behandelt; insbesondere:
  - Eingabe für nächstes Kommando
  - Ausgabe auf Konsole

BASH

- **B**ourne-**a**gain **Sh**ell: weit verbreitete Shell aus dem GNU-Projekt
- Setzt IEEE 1003.2 POSIX Shell Standard um
- Projektbeschreibung verfügbar via *https://www.gnu.org/software/bash/*
- seit initialem Release 1989 in ständiger Weiterentwicklung
- Version kann abgefragt werden via

```
user@linux$ echo $BASH_VERSION
```

- Nach Start der Bash als Login-Shell wird `/etc/profile` und anschließend `~/.bash_profile` oder alternativ (falls nicht vorhanden) `~/.profile` für wichtige Konfigurationen gelesen (insbesondere für Umgebungsvariablen)
- Bei <u>interaktiver</u> Shell (gestartet aus `login`, `su`, `ssh`, …) wird `~/.bashrc` eingelesen

Bash - Umgebungsvariablen

- zur Steuerung der Funktionalität vieler Programme können Umgebungsvariablen eingesetzt werden

- Anzeige einer Variable via

```
user@linux$ echo $VARIABLENNAME
```

- bei Variablenzuweisung mittels „=“darf kein Leerzeichen zwischen Variablenname und Wert stehen

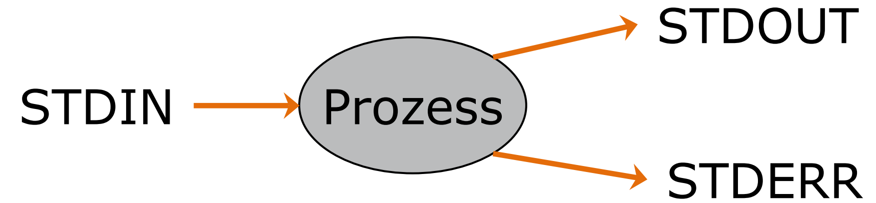
```
user@linux$ VARIABLENNAME='HELLO WORLD!'
```

- Konkatenation von Variablen mit anderen Zeichenketten durch einfaches Hintereinanderstellen möglich

```
user@linux$ echo $PATH
/usr/local/sbin:/bin
user@linux$ PATH=$PATH‘:/usr/bin‘
user@linux$ echo $PATH
/usr/local/sbin:/bin:/usr/bin
```

Bash – Pipelines und Umleitungen

- Jeder Prozess hat beim Start drei Interaktionskanäle:
  - Standardeingabekanal (0)
  - Standardausgabekanal (1)
  - Standardfehlerkanal (2)



- Pipeline:
  - Sequenz von Kommandos, die mittels „|“ voneinander separiert sind
  - Ausgabe eines Kommandos wird zur Eingabe des zweiten Kommandos
  - Gemäß Manpage: `command [ [|||&] command2 ... ]`

Bash – Pipelines und Umleitungen

- Ein-/Ausgabe von Kommandos kann durch Dateien umgeleitet werden (siehe Abschnitt „Redirection“ der Manpage der Bash)
- wichtige syntaktische Elemente für die Umleitung sind „>“ und „<“
- Umleitungen werden technisch vor der Kommandoausführung realisiert
- ausgewählte Formen von Umleitungen:

```
Kommando > Datei
```

- Leite Ausgabe in `Datei` um; falls `Datei` existiert, wird diese überschrieben

```
Kommando >> Datei
```

- Leite Ausgabe in `Datei` um; Ausgabe wird an eine bestehende `Datei` angehängt

```
Kommando < Datei
```

- Macht `Datei` zur Eingabe des Kommandos

```
Kommando n> Datei
```

- Leitet nur STDOUT (n=1) oder STDERR (n=2) um

```
Kommando n>&m
```

- Führt Kanal n mit Kanal m zusammen

Bash –Ausführung von Kommandos

- Unterstützung verschiedener Varianten der Kommandoausführung

- wichtige Formen sind:

  - Ausführung im Hintergrund durch Anfügen eines „&":

```
user@linux$ lang_laufendes_kommando &
```

  - Konditionale Ausführung:

```
user@linux$ kommando1 && kommando2
```

Wird nur ausgeführt, wenn kommando1 erfolgreich ausgeführt wurde (Rückgabe 0)

  - Kommandosubstitution (Kommandoausgabe wird gesetzt):

```
user@linux$ X=$(kommando)
```

Ausgabe wird in X gespeichert; analog: Kommando schreiben in „Backticks" (``)

  - Sequentielle Ausführung von Kommandos:

```
user@linux$ kommando1; kommando2
```

SHELL-POGRAMMIERUNG

- Shell-Programme ermöglichen
  - Automatisierung von oft benötigten Kommandofolgen (z.B. `sudo apt-get update && sudo apt-get upgrade`)
  - Erstellung von Hilfswerkzeugen für die Systemadministration
  - Durchführung von umfangreichen Backups
  - Konfiguration und Ausführung von Programmen
  - ...
- es werden auch Verzweigungen, Schleifen etc. unterstützt
  → dafür existiert eigene Programmiersprache
- "Although most users think of the shell as an interactive command interpreter, it is really a programming language in which each statement runs a command. Because it must satisfy both the interactive and programming aspects of command execution, it is a strange language, shaped as much by history as by design." (Kernighan, Pike)

Shell-Programmierung: Hello World

- Shell-Skripte sind Textdateien, die von der Bash (oder einer anderen Shell) interpretiert werden
- Shell-Skript beginnt immer mit „Shebang" (#!), das dazu führt, dass
  - das darauf folgende Programm ausgeführt wird
  - das Shell-Skript als Argument übergeben wird

Kein Kommentar! → `#!/bin/bash`

Kommentare → `#Wert in Variable Speichern`, `#Ausgabe des Variableninhaltes`

```
#!/bin/bash
#Wert in Variable Speichern
HELLO="Hello World!"
#Ausgabe des Variableninhaltes
echo $HELLO
```

verwirrende farbliche Interpretation des Programms durch CLI-Editor `pico`

- Problem mit #!/bin/bash: wenn Skript auf anderen Rechner migriert wird, liegt das Programm bash evtl. in einem anderen Verzeichnis
  - Ausweg: `#!/usr/bin/env bash` oder Verweis auf `/bin/sh` und Verwendung von Shell-übergreifender Syntax

Shell-Programmierung: Hello World

- Nach Anlegen des Skriptes wird es nicht gefunden (globaler Kontext):

```
user@linux:~/scripts$ helloworld.sh
-bash: helloworld.sh: command not found
```

- Skript wird im lokal Kontext liefert Fehler:

```
user@linux:~/scripts$ ./helloworld.sh
-bash: ./helloworld.sh: Permission denied
```

- Ausgabe von `ls -al`:

```
user@linux:~/scripts$ ls -al
total 3
drwxr-xr-x  2 user user 4096 2017-08-12 15:47 .
drwxr-xr-x 27 user user 4096 2017-08-12 15:47 ..
-rw-r--r--  1 user user   44 2017-08-12 15:47 helloworld.sh
```

- Datei muss zunächst in eine ausführbare Datei überführt werden:

```
user@linux:~/scripts$ chmod +x helloworld.sh
```

- Ausgabe von `ls -al`:

```
user@linux:~/scripts$ ls -al
total 3
drwxr-xr-x  2 user user 4096 2017-08-12 15:47 .
drwxr-xr-x 27 user user 4096 2017-08-12 15:47 ..
-rwxr-xr-x  1 user user   44 2017-08-12 15:47 helloworld.sh
```

Funktionen und Variablensichtbarkeit

- Funktionen werden durch Schlüsselwort function definiert
- Sowohl globale als auch lokale Variablen werden unterstützt

```bash
#!/bin/bash
#globale Variable
STRING="global variable"
function coolfunction {
    #lokale Variable
    local STRING="local variable"
    echo $STRING
}
echo $STRING
coolfunction
echo $STRING
```

Annotationen (Pfeile auf die Codezeilen):

- Deklaration einer Funktion → `function coolfunction {`
- Ausgabe der lokalen Variable → `echo $STRING` (innerhalb der Funktion)
- Ausgabe der globalen Variable → `echo $STRING`
- Aufruf der Funktion *coolfunction* → `coolfunction`
- Ausgabe der globalen Variable → `echo $STRING`

Argumente

- An Shell-Skripte können beim Aufruf Argumente übergeben werden
- Auf die beim Aufruf eines Skriptes angegebenen Argumente kann mittels $ARGUMENTINDEX zugegriffen werden
- $@ speichert eine Liste mit allen Argumenten als Array
- $# speichert die Anzahl der übergebenen Argumente

```
#!/bin/bash
#Ausgabe der drei Argumente
echo $1 $2 $3 ' -> echo $1 $2 $3'
#Ausgabe aller Argumente
echo $@ ' -> echo $@'
# Ausgabe der Anzahl von Argumenten
echo Number of arguments passed: $# \
' -> echo Number of arguments passed: $#'
# Speichern der Argumente in einem Array
args=("$@")
#Ausgabe des Arrays
echo ${args[0]} ${args[1]} ${args[2]} \
' -> args=("$@"); echo ${args[0]} ${args[1]} ${args[2]}'
```

Konditionale Ausführung

- Neben rein sequentieller Kommandoausführungen werden bedingte Verzweigungen (mit `if…else` und `case`) unterstützt

- Bedingungen können mit dem Kommando `test` formuliert werden (siehe für die Syntax Manpage von `test`)

- Abkürzenden Schreibweise für das Kommando `test`: `[TESTKRITERIUM]`
  → aus `test $# -ne 2;` wird `[ $# -ne 2 ];`

- Negierung mit „`!`" möglich

```
#!/bin/bash
VERZEICHNIS="./BashScripting"
# prüfen, ob das Verzeichnis mit dem in der Variable
# abgelegten Name existiert
if [ -d $VERZEICHNIS ]; then
    echo "Das Verzeichnis $VERZEICHNIS existiert..."
else
    echo "Das Verzeichnis existiert nicht..."
fi
```

Schleifen

- Um bestimmte Kommandos mehrfach auszuführen, können Schleifen verwendet werden
- Es existieren sowohl `for`- und `until-loop`- als auch `while`-Schleifen

Nächste Folie

```
#!/bin/bash
# Schreibe jede Zeile der
# Rückgabe von ls /etc/
for f in $( ls /etc/ ); do
    echo $f
done
```

```
#!/bin/bash
COUNTER=0
# bash until loop
until [ $COUNTER -gt 10 ]; do
    echo Value of count is: $COUNTER
    let COUNTER=COUNTER+1
done
```

Nutzereingaben

- Nutzereingabe können mittels read entgegengenommen werden:

```
#!/bin/bash
while :
    do read -p "Geben Sie zwei Nummern ein \
( - 1: abbrechen ) : " a b
    if [ $a -eq -1 ];
        then break
    fi
    ans=$(( a + b ))
    echo $ans
done
```

AUFGABEN

1. Schreiben Sie ein Backup-Skript, das eine gepackte Version des Heimverzeichnisses des Users „`user`“ (`/home/user`) erzeugt (mittels Werkzeug `tar`) und dieses unter dem Namen „`home-DATUM`“ in das Verzeichnis `/var/backups/` speichert – `DATUM` ist dabei das aktuelle Datum (das über das Werkzeug `date` bestimmt werden kann).

2. Schreiben Sie unter Verwendung der Befehle `tr` und `head` und des Spezial-Geräts `/dev/urandom` einen Passwortgenerator, der
   - als ersten Parameter eine Passwortlänge erhalten kann (wenn nicht angegeben: Länge = 10) (`head`)
   - als zweiten Parameter (nur wenn erster vorhanden ist) die Information als Parameter bekommt, welcher Zeichenvorrat zu verwenden ist (`tr`)

3. Erweitern Sie die Lösung aus Aufgabe 2, so dass 10 Passwörter bei einem Aufruf des Skriptes mit den angegebenen Charakteristika erzeugt werden.