ZUSAMMENFASSUNG – IT-BETRIEB & MONITORING

Kreienbühl Mika – Studierender Techniker HF System- & Netzwerktechnik mit Cyber-Security

Mika Kreienbühl
mika.kreienbuehl@student.ipso.ch

Inhalt

Übersicht IT-Betrieb & Monitoring

Im Rahmen der Techniker HF an der IFA (ipso Bildung) werden im 2. Semester Kenntnisse des Skriptings unter Bas erarbeitet, welche im Laufe der weiteren Weiterbildung wichtig für weitere Fächer wie Pen Testing unter Kali Linux sind.

Als Referenz dieser Zusammenfassung dienen die Unterrichtmaterialien, welche im Rahmen der Techniker HF im Herbst/Winter Semester 2021/22 an der IFA ausgehändigt wurden. Ausserdem werden die Notizen sowie selbst erarbeitete Informationen verwendet. Sämtliche Angaben in diesem Dokument sind ohne Gewähr und jegliche Haftung wird abgelehnt.

Fächerinfos

Schule	IFA, Bern
Lehrgang	Techniker HF System- & Netzwerktechnik mit Cyber-Security
Dozierender	Peter Christen
Unterrichtszeitraum	05.2022 bis 08.2022

Grundlagen

VI(M) handling

Der Editor VI(M) wird oft von Profis eingesetzt. Er zeichnet sic dadurch aus, dass, bei beherrschen der Bedienung, die Arbeit sehr schnell gelingt. Jedoch ist das handling deutlich schwieriger als z.B. eim nano.

Wichtige Tastenbefehle

- q -> quit
- w -> write/save
- ! -> force
- x -> löscht ein Zeichen
- d0 -> löscht bis Zeilenanfang
- dd -> löscht die ganze Zeile
- r -> ersetzt ein definiertes Zeichen
- u -> undo

Skript erstellen

Ein Skript kann grundsätzlich in einer einfachen Datei geschrieben werden. Jedoch muss zu Beginn des Skripts die Shell mittels sogenanntem Shebang definiert werden. Dieser wird zu Beginn des Skripts in die erste Zeile geschrieben.

Ein Shebang sieht folgendermassen aus:

#!/bin/bash

Damit wird definiert, dass mit der Shell in /bin/bash das Skript ausgeführt werden soll. Natürlich könnte auch eine andere Shell angegeben werden.

Skript starten

Um ein Skript zu starten, muss dies zuvor noch ausführbar gemacht werden, dies geschieht durch die Anpassung der Rechte:

chmod 755 <script.sh>

Um das Skript dann auch zu starten, muss man eine der nachfolgenden Möglichkeiten verwenden:

Möglichkeit	**Besonderheiten**
bash <script.sh>	Wird in einer neuen Shell ausgeführt, überschreibt den Shebang
./<script.sh>	Wird in einer neuen Shell ausgeführt
. <script.sh>	Wird in der bestehenden Shell ausgeführt
Source <script.sh>	Wird in der bestehenden Shell ausgeführt

Variabeldefinition

Variabeln sind casesensitive die Variabel a und A sind nicht dieselbe. Im gleichen Zuge bringt der Vergleich von Variabel A mit Inhalt b und Variabel C mit Inhalt B ebenfalls das Ergebnis, dass der Inhalt unterschiedlich ist.

Neben Zahlen sowie Buchstaben sind in Variabelnamen noch _ erlaubt. Der Name einer Variabel darf maximal 256 Zeichen haben.

Es gibt auch Varibeln welche vom System kommen. Diese werden mit «*printenv*» abgerufen.

Wichtige Umgebungsvariabeln

- $PWD -> Workingdirectory
- $OLDPWD -> Previous Workingdirectory

Dabei sind vordefinierte Variabeln immer in GROSSBUCHSTABEN, Uservariabeln jedoch in kleinbuchstaben.

Um ein Parameter eines Befehls mit einer Variabel zu ergänzen kann man den Variabelname in {} setzen. Bsp:

Ls ${pre}mk*

{} definiert dabei, dass eine Variabel mit dem Namen «pre» eingesetzt werden soll.

Variabeln sind an Shells gebunden. Heisst also, bei einer neuen Shell, gibt es auch eine neue Variabelverwaltung.

Variabeln können jedoch exportiert werden. Dadurch werden die Variabeln beim Öffnen einer neuen Shell in der bestehenden Shell übernommen. Werden dann die Variabeln in der neuen verändert, so wird die Variabel in der bestehenden Shell NICHT verändert. Eine Rückwärtsvererbung gibt es also nicht.

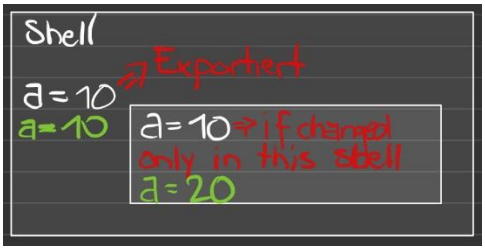


Figure 1 Variabelvererbung 1

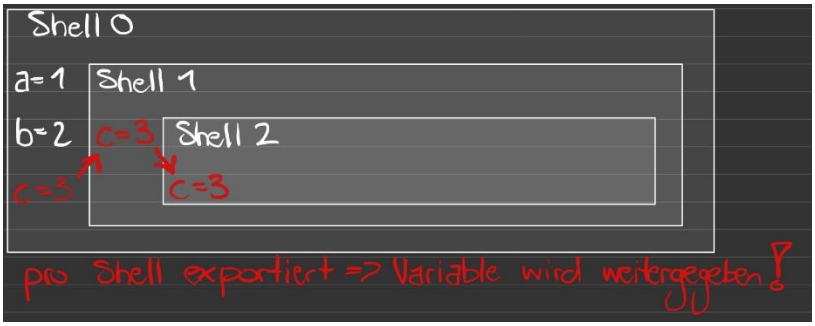


Figure 2 Variabelvererbung 2

Variabelwerte

Um einen Standardwert einer Variabel zu definieren, wird folgendes definiert:

Echo ${b:=10}

Wobei nun die Variabel b den Standardwert 10 hat

Um z.B. das Datum in einer gewissen Formatierung zu erhalten wird «*date '+%d.%m.%Y'*» ausgeführt.

Soll das Datum in eine Variabel geschrieben werden, so wird «*datum =* $(*date '+%d.%m.%Y')*» definiert.

Die Substitution «$(*date '+%d.%m.%Y')*» kann z.B. auch beim Speichern oder Kopieren von Dateien verwendet werden um z.B. eine Sicherheitskopie mit dem Datum zu versehen.

Positionsvariabeln

Positionsvariabeln sind Variabeln welche «eingebaut» sind und beim Ausführen von Skripts hinter dem Skriptnamen noch ergänzt werden können. Diese tragen die Namen $0 bis $9. Wobei $0 der Name des Skriptes ist.

Mittels «*shift*» kann die Positionsvariabel eingerückt werden. Also $2 wird zu $1, $2 zu $2 usw.

Befehlsablauf

Kommando auf Tastaturbuffer lesen
↓
Kommando in Array zerlegen
↓
Meta-Charakter auflösen
↓
Alias? Yes → Kommando ausführen
↓ No
Funktion? Yes → Kommando ausführen
↓ No
Shell-Builtin? Yes → Kommando ausführen
↓ No
Error

Stdout & Stderr

Der Standardout definiert die normale Fehlerlose Ausgabe. Der Standarderror gibt die Errors aus. Die Ausgaben können folgendermassen umgelenkt werden:

ls -l > anna	Ausgabe in Datei anna
ls -l > liste 2>liste.error	Ausgabe stdout in liste, Ausgabe stderr in liste.error
ls -l > liste 2>>liste.error	Ausgabe stdout in liste, Ausgabe stderr ergänzend in liste.error
ls -l te.sh fred >> liste.txt 2>&1	Ausgabe stdout & stderr in liste.txt

Will eine Ausgabe ignoriert und ins «Nichts» geschrieben werden, so kann das file /dev/null verwendet werden. Dies ist ein «Schwarzes Loch» und Ausgaben werden da nicht gespeichert.

Stdin

Der Stdin ist das Gegenteil von Stdout. Wobei der Stdin als Tastaturbuffer definiert wird. Man kann über diesen auch den Inhalt einer Datei einlesen. Dabei ergänzt man den Befehl folgendermassen:

«<kommando> <in.txt»

Der Inhalt des files in.txt wird somit in das Kommando eingegeben.

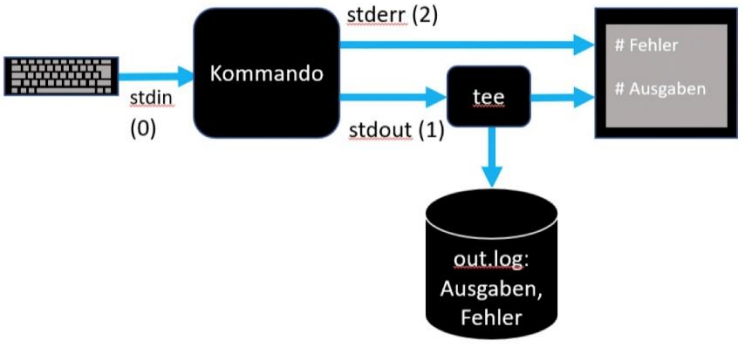
Sollen mehrere Afragen erfolgen so kann man dies mittels «<kommando> <<ENDE» eingeben. Somit werden neue Abfragen getätigt bis das Wort «ENDE» eingegeben wird.

Pipe

Das Pipe verhält sich so, dass es die Ausgaben des Stdout (Kanal 1) vom ersten Kommando an das nachfolgende Kommando übergeben wird. Also:
ps | ls (Macht keinen Sinn würde jedoch die Ausgabe vom ps an ls übergeben)

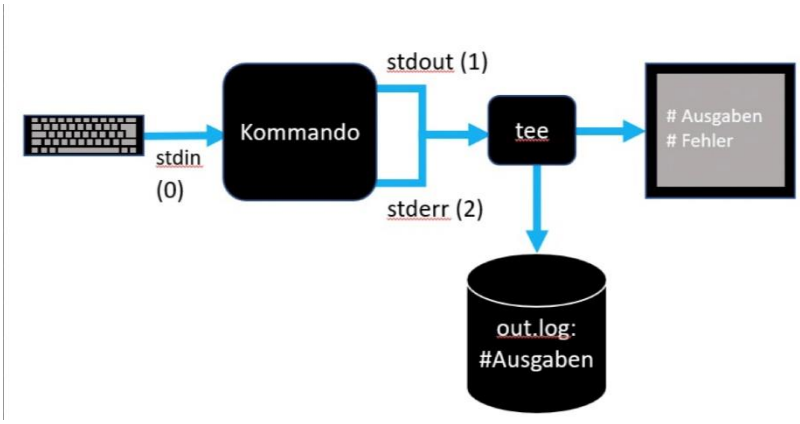
Tee

Der Tee Command lässt zwischenzeitliche Ausgaben zu. So kann man z.B. etwas testen oder aber einen Zwischenschritt in eine Datei schreiben.



Tee & Merge

Die Ausgaben können in denselben Kanal umgeleitet werden und anschliessend mit dem tee in ein Logfile geschrieben werden.



Das Merging funktioniert mittel «*2>&1*» wobei hier der Kanal 2 in den Kanal 1 geschrieben wird.

Git

Direkt vom bash kann mit git eine Versionsverwaltung geführt werden. Die wichtigsten Versuche werden nachfolgend aufgezeigt.

Command	**Bedeutung**
git init	Create new local repository in current dir
git status	Print git status (Files changed,…)
git add <script.sh>	Add one or more files to staging
git commit -m <script.sh>	Commit changes
git checkout master	Back to master commit
git checkout <commit nr>	Back to specific commit
git diff <commit nr1> \ <commit nr2>	Differenz zweiter commits vergleichen
git config –global user.name «user.name»	User in Konfig hinterlegen
git config –global user.email «email»	E-Mail in Konfig hinterlegen
echo «filename» > .gitignore	File exkludieren
git git config –list	git Konfiguration anschauen

Lebenszyklus einer Datei in Git

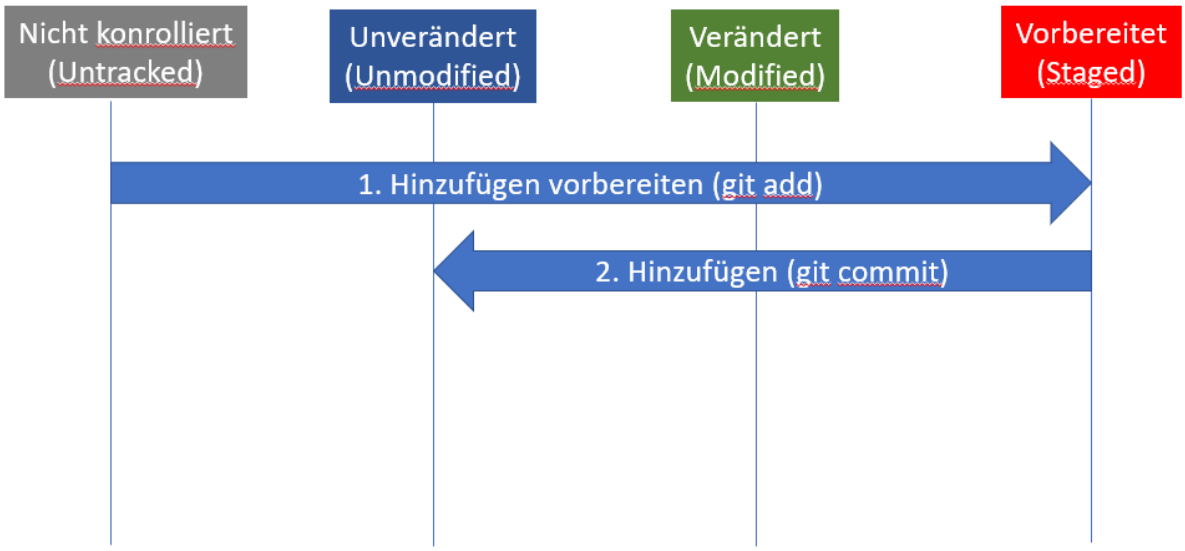


Figure 3 Lebenszyklus einer Datei in Git

Bash-Skripting

Kontrollstrukturen

Es gibt verschiedene Konstrollstrukturen, um Aktionen aufzuführen. Es wird zwischen Sequenz, Selektion und Iteration unterschieden. Eine Sequenz sind Aktionen welche nacheinander abgearbeitet werden, bei einer Selektion wird geprüft, ob eine Bedingung erfüllt ist und anhand des Ergebnisses werden dann Aktionen ausgeführt oder ignoriert. Bei einer Iteration wird eine oder mehrere Aktionen solang ausgeführt, wie eine Bedingung erfüllt ist.

Grafische Darstellung

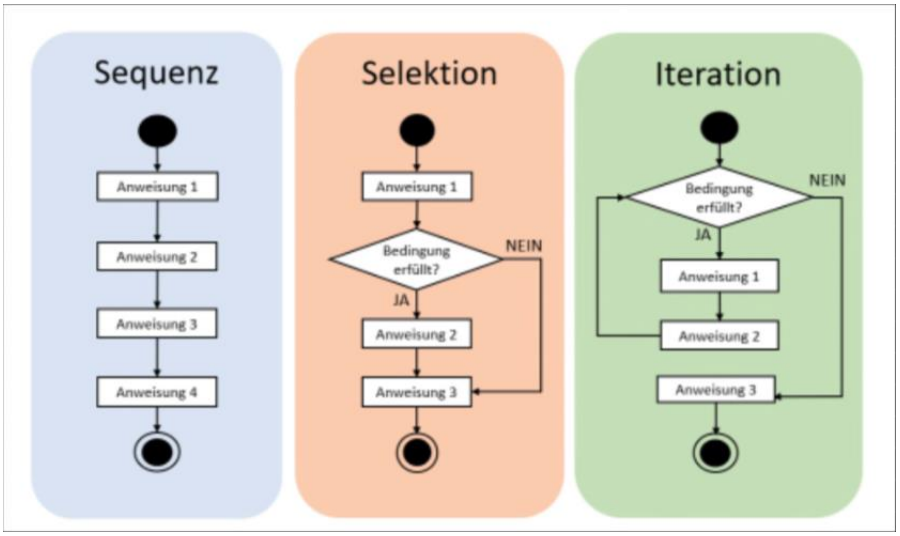


Figure 4 Kontrollstrukturen

Zahlenverleich

Um Zahlen zu vergleichen, gibt es verschiedene Prüfmechanismen, um Zahlen zu vergleichen und somit Selektionsaktionen durchzuführen.

Ausdruck	**Bedeutung**	**Liefert wahhr (0) zurück, wenn…**
test $var1 -eq $var1	eq = equal	$var1 eine Zahl ist
test $var1 -eq $var2	eq = equal	$var1 gleich $var2 ist
test $var1 -ne $var2	ne = not equal	$var1 ungleich $var2 ist
test $var1 -lt $var2	lt = less than	$var1 kleiner als $var2 ist
test $var1 -gt $var2	gt = greater than	$var1 grösser als $var2 ist
test $var1 -le $var2	le = less equal	$var1 kleiner oder gleich $var2 ist
test $var1 -ge $var2	ge = greater equal	$var1 grösser oder gleich $var2 ist

Zu beachten ist, dass beide zu vergleichenden Variablen ganze Zahlen enthalten müssen, ansonsten wird immer 0 zurückgegeben.

Dateiprüfung

Um Dateien und Pfade zu prüfen, gibt es die Möglichkeit anhand von Parametern Dateien und Pfade auf ihre Eigenschaften zu prüfen.

Ausdruck	**Bedeutung**	**Liefert wahr (0) zurück, wenn…**
[-e $name]	Existenz	$name existiert (Verzeichnis oder Datei)
[-f $name]	Normale Datei	$name ist eine normale Datei
[-x $name]	Ausführbarkeit	$name ist ausfürbar, wenn Verzeichnis -> cd möglich
[-d $name]	Verzeichnis	$name ist ein Verzeichnis
[-w $name]	Schreibrecht	$name ist schreibbar, wenn Verzeichnis -> Dateien können angelegt werden
[-r $name]	Leseberechtigung	$name ist lesbar, wenn Verzeichnis -> ls möglich

Variabelvergleich

Um Inhalte von Variablen zu vergleichen, werden nachfolgende Möglichkeiten unter bash geboten:

Ausdruck	**Operator**	**Liefert wahr (0) zurück, wenn…**
[«$var1» = «$var2»]	=	$var1 gleich $var2
[«$var1» != «$var2»]	!=	$var1 ungleich $var2
[-z «$var»]	-z	$var leer ist
[-n «$var»]	-n	$var nicht leer ist

Als Alternative zu [..] wird unter bash, zsh und ksh folgende Möglichkeit geboten:

Ausdruck	**Bedeutung**	**Liefert wahr (0) zurück, wenn…**
[[$a == $b]]	gleich	$a und $b den gleichen Inhalt haben
[[$a != $b]]	Ungleich	$a und $b nicht den gleichen Inhalt haben
[[$a == muster]]	entspricht muster	$a entspricht muster mit Wildcards
[[$a =~ regexp]]	entspricht regexp	Regexp trifft auf $a zu
[[$a =- wordpart]]	Enthält wordpart	$a enthält einen Wortteil ([[$a =- Haus]] -> Hausmauer)

Ambersand

Um Bedingungen zu verketten, gibt es die Möglichkeit mittels ambersand and und or Verknüpfungen einzusetzen.

Syntax

ls test.sh && echo «File ist vorhanden» -> And-Verknüpfung, erste Bedingung ist wahr, also wird auch die 2. Ausgeführt.

ls test.sh || echo «File ist nicht vorhanden» -> Or-Verknüpfiung, erste Bedingung ist falsch, also wird die 2. Ausgeführt.

In der if-Schlaufe kann auch noch mittels -a & -o die Verknüpfung getätigt werden.

Rechnen mit Bash

Es gibt auch die Möglichkeit mit Bash zu rechnen folgende Möglichkeiten gibt es dazu:

a=2
b=3

c=$(($a+$b))

let c=$a+$b

c=$[$a+$b]

Modulo

Auch gibt es die Möglichkeit das Modulo zu berechnen. Die macht man folgendermasse:
((c=b%a)) z.B. also ((5%2)) | 5/2 = 2 Rest 1

Modulo gibt in diesem Fall die Zahl 1 aus. Modulo gibt den Rest aus, welcher nicht in ganze Zahlen geteilt werden kann.

While-Schleife

Die While-Schleife bildet die Funktion Aktionen auszuführen, bis eine Bedingung nicht mehr erfüllt ist.

Syntax

```
while [$a -lt 10]
do
        ((a++))
        echo $a
done
```

Ablaufgrafik

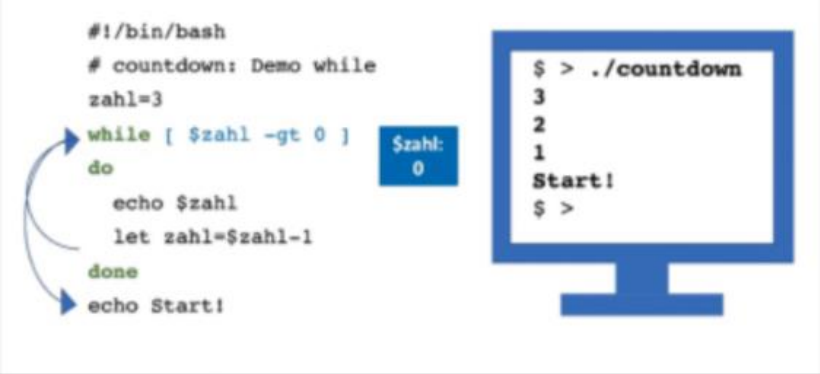


Figure 5 While-Ablaufgrafik

For-Schleife

Die For-Schleife definiert die Funktion Aktionen auszuführen für eine gewisse Anzahl Werte z.B. in einem Array.

Syntax 1. Form

```
for i in 1 2 3 Hause 5 6
do
        echo $i
done
```

Ausgabe

```
root@ifadebian:/home/user/IFA/ITBM/Vorbereitung2/aufgabe-6# for i in 1 2 3 Hause 5 6
> do
> echo $i
> done
1
2
3
Hause
5
6
root@ifadebian:/home/user/IFA/ITBM/Vorbereitung2/aufgabe-6#
```

Figure 6 Ausgabe 1. Form

Syntax 2. Form

```
for ((zahl=1;zahl<=3;zahl++))
do
echo $zahl
done
```

Ausgabe

```
root@ifadebian:/home/user/IFA/ITBM/Vorbereitung2/aufgabe-6# for ((zahl=1;zahl<=3;zahl++))
> do
> echo $zahl
> done
1
2
3
root@ifadebian:/home/user/IFA/ITBM/Vorbereitung2/aufgabe-6# _
```

Figure 7 Ausgabe 2. Form

Until-Schleife

Die Until-Schleife definiert, dass eine Aktion ausgeführt wird, solange eine Bedingung NICHT erfüllt ist.

```
until [$a -gt 10]
do
        ((a++))
        echo $a
done
```

Eingaben

Benutzereingaben

Um Benutzereingaben anzufordern, gibt es den command read.

Command	Bedeutung
read -p «Hilfstext» <variabel>	Liest eine Benutzereingabe in die definierte Variabel ein mit Hilfstext
read <variabel>	Liest eine Benutzereingabe in die definierte Variabel ein
read -sp	Zeigt die Eingabe nicht an z.B. für Passworteingabe

Fileeingaben

Um Informationen von einem File einzulesen, kann mit einer While-Schleife die Datei eingelesen werden und mit den eingelesenen Variablen Aktionen durchgeführt werden.

Syntax mit Eingabeumlenkung

```
datei=$1
while read zeile
do
        echo $zeile
done < $datei
```

Syntax mit Pipe

```
datei=$1
cat $datei | while read zeile
do
        echo $zeile
done
```

Ausgaben

Ausgaben formatieren

Um Ausgaben zu formatieren kann man printf anwenden

Syntax

```
Printf «%-20s %6s %10s\n» $a $b $c
```

Eine Liste für weitere Formatierungsmöglichkeiten findet man hier.

Mehrfachauswahl

Um eine Möglichkeit zu bieten aus mehreren verschiedenen Optionen auswählen zu können, wird die Funktion eines cases verwendet.

Syntax

```
case $option in
        a) echo «Option a erkannt»;;
        b) echo «Option b erkannt»;;
        C) echo «Option C erkannt»;;
        ?) echo «Unbekannte Option»;;
esac
```

Kombiniert wird dies in der Regel mit einer getopts while-Schleife.

```
while getopts abC: option 2>/dev/null;
do
        case $option in
                a) echo «Option a erkannt»;;
                b) echo «Option b erkannt»;;
                C) echo «Option C erkannt»;;
                ?) echo «Unbekannte Option»;;
        esac
done
```

Ausgabe

```
user1@host:~$ ./getopts_simple.sh -a -b -C foo
Option -a erkannt
Option -b erkannt
Option -C mit foo erkannt

user1@host:~$ ./getopts_simple.sh -C foo -b
Option -C mit foo erkannt
Option -b erkannt
```

Figure 8 Ausgabe GetOpts

Funktion definieren

Um den Skript-Code agiler und übersichtlicher zu gestalten, wird oft mit Funktionen gearbeitet. Unter Umständen sogar mit einer Funktions Bibliothek. Zu beachten ist, dass Variablen in Funktionen global in der Bash-Shell sind. Eine Variabel kann als Lokal definiert werden indem die Variabeldefinition mit «local» angeführt wird. Also: local s=0

Syntax

```
function summe () {
        s=$1+$2
        echo «Summer ist $s»
}
summe 3 5
```

Um eine Funktion zu exportieren wird er Command «export -f summe» ausgeführt. So steht die Funktion auch in Subshells zur Verfügung, wie man dies bereit von Variablen kennt.

Funktionsbibliothek

Eine Funktionsbibliothek wird in der modernen Programmierung sowie auch dem modernen Skripting oft eingesetzt.

Einfach gesagt, ist es eine Sammlung von Funktionen, welche dann in Skripten eingebunden wird und somit einmal geschriebene Funktion wiederholt und von verschiedenen Skripten «zentral» benutzt werden kann.

```
#Functionlibrary aufgabe-6

function is_writeable_dir ()
{
        if [[ -d $1 && -w $1  ]] 2> /dev/null
then
        return 0
fi
return 1
}

export -f is_writeable_dir
~
```

Figure 9 Funktionsbibliothek

```
#!/bin/bash

#Read Funclib
source ./lib/myfunc 1

#Progress check

for name in $@
do
        if is_writeable_dir $name 2
        then
                echo "$name: create file(s) permitted"
        else
                echo "$name: create file(s) not permitted"
        fi
done
~
```

Figure 10 Skript mit Funktionsbibliothek (1=Eingebundene Bibliothek, 2= Funktionsaufruf)

Arrays

Ein Array ist einfach gesagt eine Variabel welche mit mehreren Werten. Das Handling ist folgendermassen:

Command	**Bedeutung**
arr1=(10 «Haus» 30 50 Türe)	Definition eines Arrays
echo ${arr1[0]}	Aufruf Array «arr1» 0. Wert
arr1[0]=40	Definition des 0. Wertes von arr1
arr1+=(hallo)	Ergänzung des Arrays mit dem Wert hallo
echo ${arr1[*]}	Abrufen aller Array-Werte
echo ${#arr[*]}	Abrufen der Anzahl Values

Assoziative Arrays

Ein Assoziatives Array ist ein Array welches mehrere Inhalte sowie einen Key speichern. Also definiert man z.B. ein Array «Lebensmittel» mit dem Key «Frucht» definiert man den Wert «Apfel» ruft man nun die Position «Frucht» auf so bekommt man den Wert «Apfel» zurück.

lebensmittel

Key	Frucht	Suppe	Getränk	Dessert
Wert	Apfel	Boullion	Wein	Glace

Figure 11 Assoziatives Array "lebensmittel"

Definition

declare -A arr1
arr1[Apfel]=Früchte
arr1[Birne]=Früchte
arr1[Lauch]=Gemüse
arr1[Sellerie]=Gemüse
Key, Wert

Wichtig ist, dass bei der Deklaration ein grosses A mitgegeben wird also -A , ein kleines a (-a) definiert ein normales Array.

Testing

Arten von Tests

- Unit-Test
  - Prüfung einer Funktion oder eines Skripts
- Integrations-Test
  - Prüfung der Zusammenarbeit mehrerer Funktionen oder Skripts
- System-Test
  - Test des gesamten Systems

Was kann getestet werden?

- Direkte Wirkungen
  - Rückgabewerte von Funktionen und Skripts (Return-Code, Exit-Code)
  - Terminal-Ausgaben
- Seiteneffekte
    - Änderungen am Zustand des Systems (z.B. Inhalt oder Erzeugung von Variablen und Dateien)
    - Änderungen am Verhalten des Systems (Reagiert es wie erwartet auf Ereignisse?)

Test-Pfade

- Positivtest
  - Testing über Verhalten bei «Sachgemässer» Verwendung
- Negativtest
  - Testing über Verhalten bei Falscher Verwendung
- Destruktiver Test
  - Versuche das Skript völlig anders als vorgesehen zu nutzen und versuchen ein Absturz zu provozieren

Testumgebung

Damit Testergebnisse reproduzierbar sind, muss die Testumgebung immer gleich sein. Damit gemeint sind Betriebssystem sowie Benutzerumgebung. Es bietet sich ein Snapshot des Systems an.

Um die Funktion noch nicht fertiggestellter Elemente zu simulieren, können sogenannte «Stubs» oder «Mock» gebaut werden. Heisst also; ein Platzhalter einbauen, bei welchem der Return-Code fest codiert ist.

Testfälle, Testdaten

Um Eingaben zu testen, müssen Eingabewerte definiert werden. Da nicht alle möglichen Eingaben getestet werden können, werde sogenannte Äquivalenzklassen mit einigen Werten definiert.

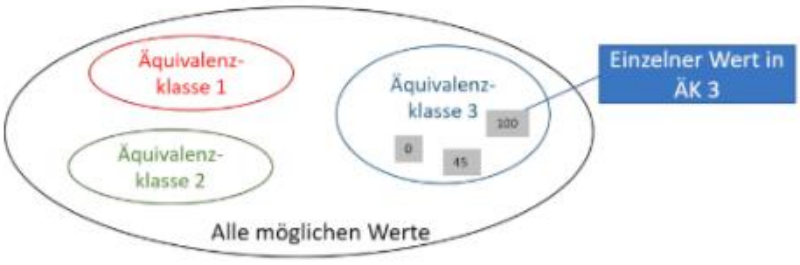


Figure 12 Äquivalenzklassen

Prozessmanagement

Fork

Ein Fork erzeugt eine Kopie des Prozesses, beide Prozesse laufen weiter wobei nur einer «Bearbeitet» wird.

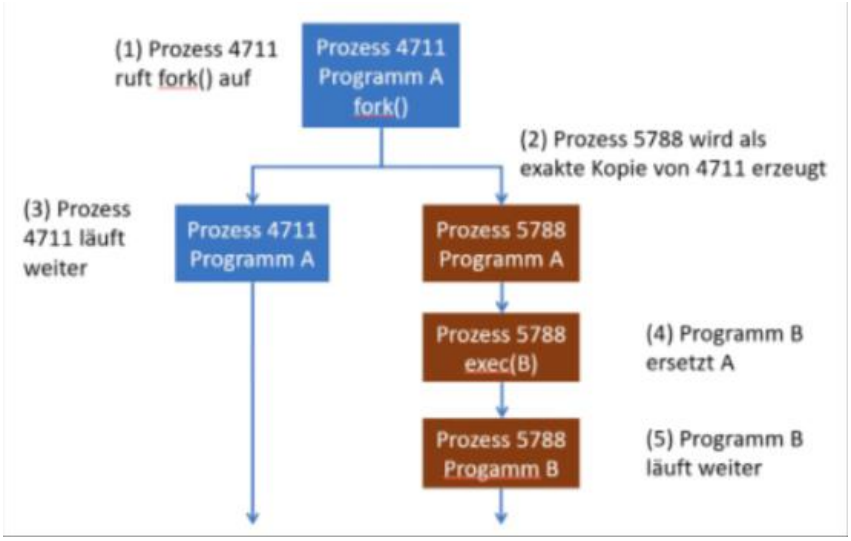


Figure 13 Forkhandling

Wait

Wait erzeugt ebenfalls eine exakte Kopie des Prozesses. Das Original wird pausiert, die Kopie wird prozessiert und die Ergebnisse werden in den Ursprungsprozess returned.

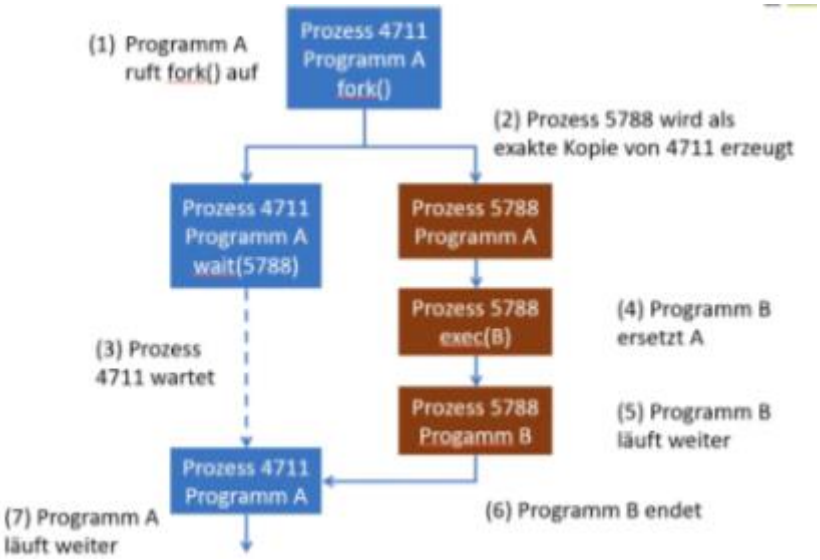


Figure 14 Waithandling

Hintergrundausführung

Mit & wird der Prozess im Hintergrund ausgeführt. Mit fg kann dann der Prozess wieder in den Vordergrund geholt werden:

Entkopplung

Um einen Prozess von seinem Parentprozess zu entkoppeln, wird vor den Kommand ein «*nohup*» gesetzt. Als Parent wird dann der INIT-Prozess geführt.

Ausgabeumleitung

Die Ausgaben können umgeleitet werden. Die wohl populärsten Umleitungen sind folgende:

Umleitung	**Bedeutung**
1>&2	Stdout in Stderr
2>&1	Stderr in Stdout
2>/dev/null	Errors ins Nichts
1> log.log	Stdout in Datei
2> log.log	Stderr in Datei

Signale

Signale sind ein Mechanismus zur Kommunikation zwischen Prozessen und zwischen Kernel und Prozessen. Die vordefinierten Signale lassen sich in mehrere Kategorien einteilen:

- Systemsignal
  - Werden bei Hard- und Softwarefehlern vom Kernel an Prozesse gesendet

Signal	Nr.	Auslöser	Standardreaktion
SIGILL	4	Illegale Instruktion z.B. 1 / 0	Core Dump + Ende
SIGABRT	6	Abnormale Beendigung	Core Dump + Ende
SIGFPE	8	Fehler bei Gleitkommaoperation	Core Dump + Ende
SIGSEGV	11	Unerlaubter Speicherzugriff	Core Dump + Ende

Figure 15 Systemsignale

- Gerätesignale
  - Werden bei Änderungen von Geräten an Prozesse gesendet
- Benutzerdefinierte Signal

Signal	Nr.	Auslöser	Standardreaktion
SIGHUP	1	Konfigfile neueinlesen	Ende
SIGINT	2	Benutzer drückt (Strg)+(C)	Ende
SIGKILL	9	Benutzer gibt kill -9 PID ein	Ende (unbedingt)
SIGTERM	15	Benutzer gibt kill -15 PID oder kill PID ein	Ende
SIGCONT	18	Prozess wird fortgesetzt (Eingabe bg JOB oder fg JOB)	Prozess läuft weiter
SIGSTOP	19	Prozess wurde angehalten	Anhalten
SIGTSTP	20	Benutzer drückt (Strg)+(Z)	Anhalten

Figure 16 Benutzerdefinierte Signale

Signale senden

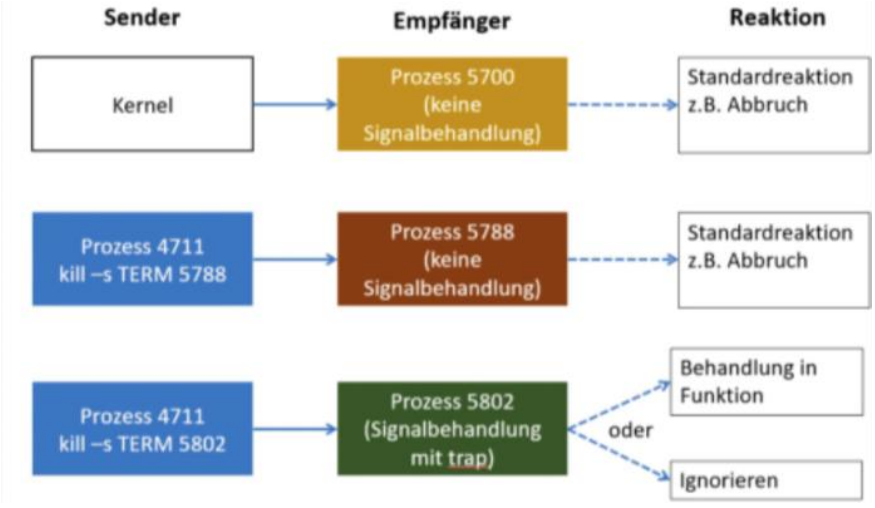


Figure 17 Signale senden

Signale abfangen

Signale können ebenfalls abgefangen werden z.B. kann so ein einfaches Abbrechen von Skripten verhindert werden.
Dies geschieht mit einer trap. Wobei im Skript mit «trap 'command' <Signal>» das Standardverhalten verändert/unterdrückt wird.

Beispiele

Trap	**Verhalten**
trap '' SIGTERM	SIGTERM wird ignoriert
trap 'echo Signal SIGTERM erhalten ' SIGTERM	Ausgabe «*Signal SIGTERM erhalten*»
trap cleanup SIGTERM	Funktio *cleanup* wird ausgeführt
trap SIGTERM	Reset zu Standard

Wobei beachtet werden muss, dass SIGKILL/Kill -9 nicht abgefangen werden kann, da nicht dem Prozess «Bitte beenden» gesagt wird, sondern ihm einfach den Zugriff auf die Computerressourcen entzogen wird. Wird ein SIGKILL ausgeführt, kann dies jedoch zu korrupten Daten führen, da der Prozess mitten im Prozessieren gekickt wird.

Named Pipes

Named pipes sind wie «Ablagen» für Ausgaben welche dann wieder abgerufen werden können. Nutzbar sind diese für das schieben von Informationen zwischen Skripts.

Syntax

Command	**Bedeutung**
echo «Guten Tag»>namedpipe	Befüllen des Namespipes
cat namedpipe	Abrufen des Namedpipe, kann z.B: über eine normale pipe dann weitergegeben werden

Eval

Mittels eval kann code aus einer Variable ausgeführt werden.

Syntax

Command	**Bedeutung**
command=«ls -l»	Die Variable wird mit «*ls -l» befüllt*
eval $command	Aus der Variable $command wird «*ls -l*» ausgeführt.