Montag: Linux II

2 Linux II
- Rechte
- Batch-Verarbeitung
- Encoding and Locale
- AWK

Linux II

Was sind Rechte?

- Unix-Systeme sind Mehrbenutzersysteme:
  Mehrere Benutzer sind am System aktiv
- Rechte (Permissions) verhindern, dass Benutzer Daten lesen bzw. ändern, die sie nicht lesen bzw. ändern sollen
- Rechte werden auf Datei- bzw. Verzeichnisebene vergeben
- Rechte gelten für den gerade angemeldeten User

Welche Rechte gibt es?

- `read (r)`
- `write (w)`
- `execute (x)`

Rechte: Read

- Das Read-Recht wird für jede Art von nur lesendem Zugriff benötigt
- Ist nur das Read-Recht verfügbar, sind Änderungen ausgeschlossen
- Bedeutung:

Datei	Lesen des Dateiinhalts
Verzeichnis	Lesen des Verzeichnisinhalts (= Auflisten der Dateien im Verzeichnis)

Rechte: Write

- Das Write-Recht wird benötigt, um Änderungen vorzunehmen
- Wer über Write-Rechte verfügt, kann Dinge löschen
- Bedeutung:

Datei	Schreiben in die Datei, Ändern der Dateirechte
Verzeichnis	Anlegen oder Löschen von Dateien und Unterverzeichnissen

Rechte: Execute

- Das Execute-Recht wird benötigt um Programme auszuführen
- Der Prozess erbt die Rechte des Users, der den Prozess startet (= der Prozess darf, was der User darf, der ihn gestartet hat)
- Um ein Programm wirklich ausführen zu können, muss es *auch* lesbar sein
- Bedeutung:

Datei	Verfügt ein User über das execute-Recht bei einer Datei, kann er diese Datei starten. Wenn es sich um Code (z.B. Bash oder Maschinencode) handelt, wird er dann ausgeführt. Wenn nicht, gibt es eine Fehlermeldung
Verzeichnis	Wechseln in ein Verzeichnis

Wie werden diese Rechte auf Dateien verteilt?

Getrennt für drei Benutzer"typen"

- Eigentümer (User, üblicherweise der Ersteller der Datei)
- Gruppe (Group)
- Alle anderen (Others)

Notation

- Jedes Recht kann für jeden Benutzertyp unabhängig festgelegt werden
- 9 Zeichen – 3 für User, 3 für Group, 3 für Others

Beispiel

- `rw-rw-r--`
  Eigentümer und Gruppe dürfen lesen und schreiben, andere nur lesen

Wie werden diese Rechte auf Dateien verteilt?

Getrennt für drei Benutzer"typen"

- Eigentümer (User, üblicherweise der Ersteller der Datei)
- Gruppe (Group)
- Alle anderen (Others)

Notation

- Jedes Recht kann für jeden Benutzertyp unabhängig festgelegt werden
- 9 Zeichen – 3 für User, 3 für Group, 3 für Others

Beispiel

- `rw-rw-r--`
  Eigentümer und Gruppe dürfen lesen und schreiben, andere nur lesen

Wie werden diese Rechte auf Dateien verteilt?

Getrennt für drei Benutzer"typen"

- Eigentümer (User, üblicherweise der Ersteller der Datei)
- Gruppe (Group)
- Alle anderen (Others)

Notation

- Jedes Recht kann für jeden Benutzertyp unabhängig festgelegt werden
- 9 Zeichen – 3 für User, 3 für Group, 3 für Others

Beispiel

- `rw-rw-r--`
  Eigentümer und Gruppe dürfen lesen und schreiben, andere nur lesen

Ändern von Dateirechten

- **`chmod`** ändert die Rechte einer Datei
  Darf: Dateieigentümer
- **`chown`** ändert den Dateieigentümer
  Darf: root
- **`chgrp`** ändert die Gruppe einer Datei
  Darf: Dateieigentümer, wenn er in der neuen Gruppe ist

Beispiele

- `$ chmod u+x datei.sh`
- `$ chmod g-w datei.txt`

Ändern von Dateirechten

- **`chmod`** ändert die Rechte einer Datei
  Darf: Dateieigentümer
- **`chown`** ändert den Dateieigentümer
  Darf: root
- **`chgrp`** ändert die Gruppe einer Datei
  Darf: Dateieigentümer, wenn er in der neuen Gruppe ist

Beispiele

- `$ chmod u+x datei.sh`
- `$ chmod g-w datei.txt`

Linux II

Linux II

Variablen in der Shell

In der Shell können Variablen verwendet werden.

- Deklaration: Variablenname, Gleichheitszeichen, Wert (keine Leerzeichen!)
- Benutzung: Dollarzeichen, Variablenname
- Variablen sind grundsätzlich *ungetypt*!

Beispiel

```
$ VARNAME=Hallo
$ echo $VARNAME
Hallo
$ echo $VARNAME Welt!
Hallo Welt!
$ echo $VARNAME2

$ echo ${VARNAME}2
Hallo2
```

Variablen in der Shell

In der Shell können Variablen verwendet werden.

- Deklaration: Variablenname, Gleichheitszeichen, Wert (keine Leerzeichen!)
- Benutzung: Dollarzeichen, Variablenname
- Variablen sind grundsätzlich *ungetypt*!

Beispiel

```
$ VARNAME=Hallo
$ echo $VARNAME
Hallo
$ echo $VARNAME Welt!
Hallo Welt!
$ echo $VARNAME2

$ echo ${VARNAME}2
Hallo2
```

Variablen in Strings

" und ' verhalten sich unterschiedlich!

- Double quotes: Variablenreferenzen werden ersetzt
- Apostroph/Single quote: Variablenreferenzen werden nicht ersetzt

Beispiele

```
$ TEST="A B"
$ echo "$TEST"
A B
$ echo '$TEST'
$TEST
$ TEST="A     B"
#Zeichenkette wird an Whitespace gesplittet
$ echo $TEST
A B
$ echo "$TEST"
A     B
```

Backticks

- Kommandos innerhalb von Backticks werden ausgeführt
- Standardausgabe wird als Zeichenkette zurückgegeben
- Alternative: `$( ...)`

Beispiele

```
# Variable TEST enthält Ausgabe von ls -la
$ TEST=`ls -la`
$ TEST=$( ls -la )
# Inhalt von Variable TEST wird ausgegeben
$ echo $TEST
... Ausgabe von ls -la ...
```

Arithmetik

- Operatoren +,-,*,/,**,%,
- Einfache Befehle ausführen:

```
:~$ expr 1 + 1
2
```

- Ergebnis eines arithmetischen Ausdrucks in eine Variable schreiben:

```
$ VAR1=$((1+1))
$ ((VAR2 = 1 + 1)) # Spaces optional
$ let "VAR3=1+1" # Spaces optional
$ VAR4=$((VAR3**2))
```

- In (( )) müssen Variablen nicht mit $ referenziert werden.
- Floating Point Arithmetik ist auf der Shell nicht möglich.

Umgebungsvariablen

- Werden vom Betriebssystem beim Einloggen belegt
- Können auch geändert werden,
  allerdings sollte man wissen, was man tut

`$HOME`	Der Pfad zum Homeverzeichnis
`$PWD`	Working Directory
`$OLDPWD`	Previous Working Directory
`$_`	Letztes Argument des letzten Kommandos
`$PATH`	Der Suchpfad für ausführbare Programme
`$TERM`	Angaben über das Terminal
`$OSTYPE`	Angaben über das Betriebssystem
`$BASH_VERSION`	Version der Shell
`$USER`	Der Name des Users
...	...

Exkurs: Suchpfad

- Der Suchpfad (`$PATH`) gibt an, in welchen Verzeichnissen ausführbare Dateien liegen
- Programme werden ohne genaue Pfadangabe gefunden
- Verzeichnisse werden durch Doppelpunkt getrennt
- Viele Programme werden auf diese Weise gefunden: `ls` , `grep` , ...
- Mit `export` kann der Suchpfad geändert werden, so dass auch in anderen Verzeichnissen gesucht wird

```
:~$ export PATH=$PATH:/the/new/path
```

Umgebungsvariablen deklarieren

- Eine Variable wird zur Umgebungsvariable, indem das Schlüsselwort **`export`** vor die Deklaration gestellt wird

Beispiel

```
# VARNAME1 ist eine Umgebungsvariable
$ export VARNAME1=Hallo

# VARNAME2 ist eine normale Variable
$ VARNAME2=Welt
```

- Wozu sollte man Umgebungsvariablen selber deklarieren?

Umgebungsvariablen deklarieren

- Eine Variable wird zur Umgebungsvariable, indem das Schlüsselwort **`export`** vor die Deklaration gestellt wird

Beispiel

```
# VARNAME1 ist eine Umgebungsvariable
$ export VARNAME1=Hallo

# VARNAME2 ist eine normale Variable
$ VARNAME2=Welt
```

- Wozu sollte man Umgebungsvariablen selber deklarieren?

Subprozesse (Wiederholung)

- Alle Prozesse auf einem Computer sind hierarchisch angeordnet.
- Wird ein neuer Prozess gestartet, läuft er als Subprozess des Prozesses, von dem er gestartet wurde.
- Der Kontext des Elternprozesses wird übernommen.

Variablen in Subprozessen

- (Normale) Variablen sind in Subprozessen nicht (mehr) verfügbar

Beispiel

```
# Deklaration einer Variable
$ VAR=hallo

# Start eines Subprozesses
$ bash

# Ausgabe des Inhaltes der Variable VAR
$ echo "VAR=$VAR"
VAR=
```

Umgebungsvariablen in Subprozessen

- Umgebungsvariablen sind in Subprozessen verfügbar.

Beispiel

```
# Deklaration der Umgebungsvariable
$ export VAR=hallo

# Start eines Subprozesses
$ bash

# Ausgabe des Inhalts der Variablen
$ echo "VAR=$VAR"
VAR=hallo
```

Linux II

Shell-Skripte

- Anstatt Befehle auf der Shell per Hand einzugeben kann man sie auch automatisch sequenziell verarbeiten lassen.
- Befehle werden in Dateien geschrieben.
- Die Dateien (sog. Shell-Skripte) werden ausgeführt wie ein Programm.
- Einrückung spielt grundsätzlich *keine* Rolle.
- Alle Konstruktionen, die wir bisher kennengelernt haben, sind in Skripten verwendbar:
  - Befehle, Programme
  - Ein/Ausgabeumleitungen, Pipes
  - Variablen und Umgebungsvariablen
  - ...

Shell-Skripte

- Anstatt Befehle auf der Shell per Hand einzugeben kann man sie auch automatisch sequenziell verarbeiten lassen.
- Befehle werden in Dateien geschrieben.
- Die Dateien (sog. Shell-Skripte) werden ausgeführt wie ein Programm.
- Einrückung spielt grundsätzlich *keine* Rolle.
- Alle Konstruktionen, die wir bisher kennengelernt haben, sind in Skripten verwendbar:
  - Befehle, Programme
  - Ein/Ausgabeumleitungen, Pipes
  - Variablen und Umgebungsvariablen
  - ...

Ausführen von Shell-Skripten

- Es gibt verschiedene Möglichkeiten, ein Shell-Skript auszuführen
  - Interpreter, Datei als Argument

    ```
    :~$ bash script.sh
    ```
  - Ausführbare Datei, script.sh in `$PATH`

    ```
    :~$ script.sh
    ```
  - Ausführbare Datei, script.sh nicht in `$PATH`

    ```
    :~$ /path/to/script.sh
    ```
- Dateirechte steuern, ob eine Datei ausführbar ist oder nicht

  `:~$ chmod ugo+x script.sh` macht die Datei ausführbar.

Ausführen von Shell-Skripten

- Es gibt verschiedene Möglichkeiten, ein Shell-Skript auszuführen
  - Interpreter, Datei als Argument

```
:~$ bash script.sh
```

  - Ausführbare Datei, script.sh in `$PATH`

```
:~$ script.sh
```

  - Ausführbare Datei, script.sh nicht in `$PATH`

```
:~$ /path/to/script.sh
```

- Dateirechte steuern, ob eine Datei ausführbar ist oder nicht

`:~$ chmod ugo+x script.sh` macht die Datei ausführbar.

Ausführen von Shell-Skripten

- Es gibt verschiedene Möglichkeiten, ein Shell-Skript auszuführen
  - Interpreter, Datei als Argument

    ```
    :~$ bash script.sh
    ```

  - Ausführbare Datei, script.sh in `$PATH`

    ```
    :~$ script.sh
    ```

  - Ausführbare Datei, script.sh nicht in `$PATH`

    ```
    :~$ /path/to/script.sh
    ```

- Dateirechte steuern, ob eine Datei ausführbar ist oder nicht

  `:~$ chmod ugo+x script.sh` macht die Datei ausführbar.

Ausführen von Shell-Skripten

- Es gibt verschiedene Möglichkeiten, ein Shell-Skript auszuführen
  - Interpreter, Datei als Argument

    ```
    :~$ bash script.sh
    ```
  - Ausführbare Datei, script.sh in `$PATH`

    ```
    :~$ script.sh
    ```
  - Ausführbare Datei, script.sh nicht in `$PATH`

    ```
    :~$ /path/to/script.sh
    ```
- Dateirechte steuern, ob eine Datei ausführbar ist oder nicht

  `:~$ chmod ugo+x script.sh` macht die Datei ausführbar.

Ausführen von Shell-Skripten

- Es gibt verschiedene Möglichkeiten, ein Shell-Skript auszuführen
  - Interpreter, Datei als Argument

    ```
    :~$ bash script.sh
    ```
  - Ausführbare Datei, script.sh in `$PATH`

    ```
    :~$ script.sh
    ```
  - Ausführbare Datei, script.sh nicht in `$PATH`

    ```
    :~$ /path/to/script.sh
    ```
- Dateirechte steuern, ob eine Datei ausführbar ist oder nicht

  `:~$ chmod ugo+x script.sh` macht die Datei ausführbar.

Ein einfaches Shell-Skript I

- Am Anfang jedes Shell-Skripts steht die sog. Shebang: `#!/bin/bash`
- Danach folgen beliebige Kommandos: Shell-Builtins sowie Programme des Systems

Ein einfaches Shell-Skript II

Beispiel (Backup-Szenario)

- Sicherungskopie von Verzeichnis **`Documents`**
- Ziel: Verzeichnis **`/mnt/backup`**

Beispiel (backup1.sh)

```
#!/bin/bash

cp -r /home/nils/Documents /mnt/backup
```

Ein Shell-Skript mit Variablen

- Variablen und Umgebungsvariablen lassen sich einfach einbauen
- Sie werden dann *zur Laufzeit* ersetzt

Beispiel (backup2.sh)

```
#!/bin/bash

cp -r $HOME/Documents /mnt/backup/$USER
```

Positionsvariablen

- Einige Variablen ergeben nur in Verbindung mit Skripten Sinn
- Sie erlauben Zugriff auf den Orts des Skripts und die Argumente

`$0`	Dateiname des Skripts
`$1, $2, …`	Parameter, die dem Skript auf der Kommandozeile mitgegeben wurden
`$#`	Anzahl der Parameter
`$@`	Die Parameter als Liste

Ein Shell-Skript mit Positionsvariablen

Beispiel (backup3.sh)

```
#!/bin/bash

# Kopiert das als Argument angegebene Verzeichnis
# aus dem Homeverzeichnis des angemeldeten Users
cp -r $HOME/$1 /mnt/backup/$USER
```

Beispiel

```
$ backup3.sh Documents
```

Ein Shell-Skript mit Positionsvariablen

Beispiel (backup3.sh)

```
#!/bin/bash

# Kopiert das als Argument angegebene Verzeichnis
# aus dem Homeverzeichnis des angemeldeten Users
cp -r $HOME/$1 /mnt/backup/$USER
```

Beispiel

```
$ backup3.sh Documents
```

Linux II

Kontrollstrukturen

- Die Bash bietet auch Kontrollstrukturen
  - `if`
  - `while` , `until`
  - `case`
  - `for`
  - `select`
- Damit lassen sich vollwertige Programme in der Shell realisieren

Kontrollstrukturen: if

- `if` führt einen angegebenen Test aus
- Wenn der Test 0 zurückgibt, wird der Code ausgeführt
- Optional: Sonst wird anderer Code ausgeführt

Warum Null?

- Traditionellerweise ist 0 in der Unix-Welt das Zeichen für Erfolg
- Als Merkhilfe: die Zahl repräsentiert die Zahl der Fehler
- Wenn 0 Fehler aufgetreten sind, ist alles gutgegangen

Kontrollstrukturen: if

- `if` führt einen angegebenen Test aus
- Wenn der Test 0 zurückgibt, wird der Code ausgeführt
- Optional: Sonst wird anderer Code ausgeführt

Warum Null?

- Traditionellerweise ist 0 in der Unix-Welt das Zeichen für Erfolg
- Als Merkhilfe: die Zahl repräsentiert die Zahl der Fehler
- Wenn 0 Fehler aufgetreten sind, ist alles gutgegangen

Kontrollstrukturen: if und Tests

Syntax

```
if TEST
then
    COMMANDS
else
    COMMANDS
fi
```

`COMMANDS`

Beliebige Kommandos

`TEST`

- `[[ … ]]`
  Extended Test Command
- Vergleiche
  - Zeichenketten
  - Zahlen
  - Daten
  - …
- Dateitests
  - Existenz
  - Letzte Änderung
  - Datei/Verzeichnis
  - …

Kontrollstrukturen: if und Tests

Syntax

```
if TEST
then
    COMMANDS
else
    COMMANDS
fi
```

COMMANDS

Beliebige Kommandos

TEST

- `[[ … ]]`
  Extended Test Command
- Vergleiche
  - Zeichenketten
  - Zahlen
  - Daten
  - …
- Dateitests
  - Existenz
  - Letzte Änderung
  - Datei/Verzeichnis
  - …

Kontrollstrukturen: if und Tests

Syntax

```
if TEST
then
    COMMANDS
else
    COMMANDS
fi
```

COMMANDS

Beliebige Kommandos

TEST

- `[[ … ]]`
  Extended Test Command
- Vergleiche
  - Zeichenketten
  - Zahlen
  - Daten
  - …
- Dateitests
  - Existenz
  - Letzte Änderung
  - Datei/Verzeichnis
  - …

Ein Skript mit Test

Beispiel (backup4.sh)

```
#!/bin/bash

# Kopiert das als Argument angegebene Verzeichnis
# aus dem Homeverzeichnis des angemeldeten Users

# Testet, ob es existiert
if [[ -e $HOME/$1 ]]
then
        cp -r $HOME/$1 /mnt/backup/$USER
else
        # Gibt Fehlermeldung aus, wenn nicht
        echo "$HOME/$1 does not exist."
fi
```

Ein Skript mit verschalteten Tests

Beispiel (backup4b.sh)

```
#!/bin/bash

# Testet, ob Verzeichnis existiert
if [[ -e $HOME/$1 ]]
then
        # Testet, ob es ein Verzeichnis ist
        if [[ -d $HOME/$1 ]]
        then
                cp -r $HOME/$1 /mnt/backup/$USER
        else
                echo "$HOME/$1 is not a directory."
        fi
else
        # Gibt Fehlermeldung aus, wenn nicht
        echo "$HOME/$1 does not exist."
fi
```

Vergleichsoperatoren

Integer

`-eq`	is equal to
`-ne`	is not equal to
`-gt`	is greater than
`-ge`	is greater than or equal to
…	…

String

`=`	is equal to
`<`	is less than (ASCII order)
`-z`	is null (has zero length)
`-n`	is not null
…	…

Achtung!

- Single und double quotes und die Art der Klammerung des Testausdrucks ändern das Verhalten des Tests!
- `[[ … ]]`, `[ … ]`, `(( … ))`

Vergleichsoperatoren

Integer

`-eq`	is equal to
`-ne`	is not equal to
`-gt`	is greater than
`-ge`	is greater than or equal to
…	…

String

`=`	is equal to
`<`	is less than (ASCII order)
`-z`	is null (has zero length)
`-n`	is not null
…	…

Achtung!

- Single und double quotes und die Art der Klammerung des Testausdrucks ändern das Verhalten des Tests!
- `[[ … ]]`, `[ … ]`, `(( … ))`

Dateitests

`-e`	Datei existiert
`-f`	Reguläre Datei
`-s`	Dateigröße ungleich Null
`-d`	Datei ist ein Verzeichnis
`-x`	Datei ist ausführbar (für den User, der den Test ausführt)
`f1 -nt f2`	f1 ist neuer als f2
…	…

Referenz

Wo schlägt man das nach?

- Auf der man-page der Bash (`:~$ man bash`)
- Advanced Bash Scripting Guide
  `tldp.org/LDP/abs/html/index.html`
- Man-page als Webseite
  `www.gnu.org/software/bash/manual/bashref.html`

Kontrollstrukturen: for

- Eine **`for`** -Schleife wiederholt eine Reihe von Anweisungen
- Anzahl der Iterationen vorher festgelegt

Syntax

```
for ARG in [LIST]
do
      COMMANDS
done
```

LIST

- **`LIST`** ist eine Liste
- Variable **`ARG`** wird der Reihe nach auf die Elemente der Liste gesetzt

Kontrollstrukturen: for – Beispiel

Beispiel (backup5.sh)

```
#!/bin/bash

# Iteriere ueber alle Argumente der Kommandozeile
for arg in "$@"
do
  if [[ -e $HOME/$arg ]]
  then
    if [[ -d $HOME/$arg ]]
    then
      cp -r $HOME/$arg /mnt/backup/$USER
    else
      echo "$HOME/$arg is not a directory."
    fi
  else
    echo "$HOME/$arg does not exist."
  fi
done
```

Kontrollstrukturen: for – Sequenzen

- Schleifen mit Zählervariable

```
for i in 1 2 3 4 5
do
      COMMANDS
done
```

- Lange Sequenzen: {1..1000}

```
for i in {1..1000}
do
      COMMANDS
done
```

Kontrollstrukturen: for – Sequenzen

- Schleifen mit Zählervariable

```
for i in 1 2 3 4 5
do
      COMMANDS
done
```

- Lange Sequenzen: `{1..1000}`

```
for i in {1..1000}
do
      COMMANDS
done
```

Kontrollstrukturen: for – Dateien

- Automatisch über Dateien in einem Verzeichnis iterieren
- `ls` liefert eine Liste von Dateien in einem Verzeichnis
- `$( … )` wird benutzt, um die Ausgabe von einem Kommando als Wert zuzuweisen
- `$( ls )` liefert eine Liste von Dateien in einem Verzeichnis als Wert für eine Variable

Beispiel

```
for file in $( ls directory )
do
  echo $file
done
```

Iterieren über Dateien – Beispiel

Beispiel (backup6.sh)

```
#!/bin/bash

for arg in "$@"
do
  # Iteriere ueber alle Dateien in $arg
  for directory in $( ls $arg )
  do
    if [[ -d $arg/$directory ]]
    then
      cp -r $arg/$directory /mnt/backup/$USER
    fi
  done
done
```

Übung 2