Linux Kurs - Command-Line

Franz Schäfer

2019-01-11

1 CLI basics (30 min)

1.1 Der BASH Prompt

```
karli@meinlaptop:~$
```

In einer Text-Console wird man/frau üblicherweise mit einem ''Prompt" begrüst. Der könnte z.b. So aussehen:

Mit der Eingabetaste bekommt man einen neuen Prompt. Der Prompt lässt sich konfigurieren. Üblicherweise ist er so eingestellt dass wir sehen unter welchem User wir arbeiten, auf welchem System wir arbeiten (könnte auch eine sein, dass am anderen Ende der Welt steht) und das aktuelle Verzeicnis. ein $ sagt uns dass wir "normaler User" sind. Ein # ist i.a. für den root User.

Es gibt verschiedene Shell Programme die uns die Kommandoeingabe ermöglichen. Default im Linux ist die **bash**. Alternativen sind **tcsh** (default bei Mac OS/X). Beliebt ist auch die **zsh**.

1.2 Bewegen im Verzeichnisbau

Mit den Befehlen `pwd, cd, df, ls ls -l, ls -ltr` (siehe voriges Kapital) können wir uns im Verzeichnisbaum bewegen und uns Datein und Verzeichnisse ansehen.

1.3 keyboard shortcuts für die BASH

ctrl-p ctrl-n cursor up/down

history der history befehl zeigt die zuletzt eingegen Befehle.

ctrl-c Abbruch des aktuellen befehls.

ctrl-a zum Zeilenanfang

ctrl-k löschen bis zum Ende der Zeile

ctrl-r "**r**everse incremental search" - suche in der history durch Eingabe von Buchstaben.

ctrl-s stopped die ausgabe am Terminel.

ctrl-q ausgabe wieder erlauben. (Continue).

ctrl-z stopped den gerade laufenden Befehl. Duch Eingabe von **bg** wird dieser in Hintergrund geschickt und läuft dort weiter. Damit ist die shell wieder frei zur Eingabe weiterer Befehle. Will man/frau einen Befehl gleich im Hintergrund starten geht das mit einem & am Ende des Befehls. z.B.: `sleep 30 &`. Mit **fg** kann ein im Hintergrund laufener Prozess wieder in den Vordergrund geholt werden. **jobs** zeigt alle im Hintergrund laufenden jobs der aktuellen shell. (Siehe weiter unten für das Verwalten aller jobs, nicht nur derer aus der aktuellebn shell).

ctrl-d ist das "Datei-Ende" und wird auch von vielen Programmen bentutzt die Eingabe lesen, um diese Eingabe zu Beenden. Mit dem Befehl **exit** oder mit einem **Ctrl-D** kann eine shell geschlossen werden.

TAB die Tabulator Taste dient zur Verfollständigung von Eingaben. Einmal gedrückt macht sie, falls möglich, eine Verfollständigung: Zweimal gedrückt macht sie Vorschläge. Normalerweise werden Datei- und Verzeichnisnamen und die Namen eigegebener Programme vervollständigt. Inzwischen sind aber auch Bash-Plugins gängig die, z.B. die gängigen Befehlsoptionen von Befehlen kennen und auch Vervollständigen können. Damit ist die Eingabe von Befehlen sehr komfortabel und schnell.

1.4 Wer ist eingelogt?

who und **w** zeigen dir wer gerade aller am selben Computer eingelogt ist. **last** zeigt dir wann du zuletzt eingeloggt warst.

1.5 Diverse kleine, nüztliche Tools

cal zeigt dir einen aktuellen Kalender. `cal 1 2025` zeigt dir den Kalender von Jänner 2025.

gcal ist ein erweiterter Kalender. `gcal -n -qAT` zeigt dir eine Liste aller österreichischen Feiertage im aktuellen Jahr.

date - zeigt Datum, Uhrzeit und Zeitzone an.

factor - Primfaktoren Zerlegung. Das geht erstaunlicherweise bis hin zu relativ großen Zahlen: `factor 380708481735525870694501390857711137 93`

ping schickt ein ping Packet zum angegeben Server.

echo gibt die angegeben Argumente wieder aus. z.B.: `echo hallo da`

wc zählt Zeichen, Wörter und Zeilen in einer Datei

seq macht eine Zahlensequenz: z.b: [seq 1 10] die Zahlen 1 bis 10.

md5sum rechnet eine Prüfsumme über den Inhalt der angegebenen Dateien. (praktisch zum sehen ob die gleich sind wie wo anders).

1.6 Ausgabeumleitung in und von Dateien

Das großer-Zeichen: > leitet die Ausgabe eines Befehls, die ansonsten am Bildschrim landen würde (auch STDOUT genannt) in ein File um.
z.B.:

```
date > heute.txt
```

Würde die Ausgabe vom `date` Befehl in die Datei namens “heute.txt” schreiben. **Achtung:** Falls heute.txt schon existiert wird es kommentarlos überschrieben!

Den Inhalt der neuerstellten Datei können wir uns mit einem Editor oder mit dem `cat` Befehl ansehen: z.B.: `cat heute.txt`

Cat gibt ein oder mehrere Datein am Bildschirm aus. Natürlich lässt sich auch die Ausgabe von cat wieder mit > umleiten. Das Minuszeichen − wird sehr oft an Stellen benutzt an denen ein Filename verwendet werdekn kann um statt dessen die Eingabe (also die Zeichen die wir gerade eintippen, auch STDIN genann) zu lesen.

`cat - > test.txt` liest die Eingaben die wir tippen und schreibt sie in eine Datei namens text.txt. (Beenden der Eingabe mit **Ctrl-D**).

Analog dazu macht das Kleiner-Zeichen < eine Umleitung einer Eingabe. Ein Programm das normalerweise eine Eingabe lesen würde bekommt nun die daten aus einer Datei. z.B:

```
seq 1 1000000 > mille.txt
wc < mille.txt
```

Die Zeile mit dem `seq 1 1000000 > mille.txt` erzeugt eine Datei namens mille.txt mit den Zahlen 1 bis 1000000 als Inhalt. (Eine Zahl pro Zeile). Diese Dateil leiten wir indas `wc` Tool um zu zählen wieviele Zeilen, Wörter und Zeichen dort drinnen sind.

Verwendet man statt einem > zwei >> so bedeutet das bei der Ausgabeumleitung, dass man eine eventuell vorhandene Datei nicht überschreiben will sondern den Inhalt am Ende **anfügen** will.

```
echo noch eine zeile >> mille.txt
```

macht unsere mille.txt Datei noch um eine Zeile länger.

Fehlermeldungen werden von den meisten Tools nicht über die STDOUT Ausgabe ausgegeben sondern über einen einen Kanal "gesndet". Der wird STDERR genannt und hat im unix die nummer 2. Will man dies Ausgabe auch umleiten dann Verwendet man `2 >`. Soll die STDERR Ausgabe wie eine normale Ausgabe behandelt werden so können wir das mit dem Kürzel `2 > &1` machen.

z.B.:

```
ls /etc /gibtsnicht  >bla.txt   2>&1
```

1.7 wildcards in der shell

* und ? haben in der shell eine spezielle Bedeuting. Sie sind "Wildcards" in Datei- und Verzeichnisnamen. Ein Stern * wird durch ein oder mehrere Beliebige Zeichen ersetzt. Ein Fragezeichen ? wird durch genau ein beliebiges Zeichen ersetzt:

cat *.txt gibt alle files aus dem aktuellen Verzeichnis, die auf .txt enden aus.

gimp bild?.jpg ruft das Programm **gimp** auf und übergibt alle Dateien die mit bild beginnen und dann genau ein zeichen haben. Also bild1.jpg und bildx.jpg aber **nicht** bild22.jpg.

1.8 Dateien kopieren, verschieben und verlinken

cp steht für copy. `cp quelle.txt ziel.txt` würde die Dateie quelle.txt auf die datei zeil.txt kopieren. (**Achtung:** falls die Datei ziel.txt schon existiert wird sie überschrieben). Ist das letzte Argument ein Verzeichnis so werden die Files (es können in diesem Falle mehrerer sein) mit dem selben Namen ins Zielverzeichnis kopiert. (Wenn mehr als 2 Argumente angegeben werden, so muss das letzte Argument ein Verzeichnis sein). cp hat viele Optionen u.a. die Option -r für "rekursiv", falls Verzeichnisse samt aller Unterordner kopiert werden sollen.

mv funktiniert ähnlich wie cp, aber es wird nicht kopiert sondern verschoben (move).

ln -s erstellt einen, so genannten "sym-link". Das schaut aus wie eine Datei ist aber nur ein Verweis auf eine Datei oder ein Verzeichnis. Das ist praktisch um sich z.b. "abkürzungen" zu erstellen. z.B.: `ln -s /usr/share/sounds Desktop/klaenge` würde einen Link namens "klaenge" im Verzeichnis Desktop erstellen das auf den Systemordner zeigt in dem Linux viele Audiofiles abgelegt hat.

1.9 Hilfe!

Neben Doktor google, gibt es auch direkt im Linux viel eingebaute Hilfe.

Viele Befehle haben direkt Hilfe eingebaut. Mit den Optionen **-h** –**help** oder **-help** (leider nicht einheitlich) wird eine kurze Hilfe angezeigt. Diese Hilfe ist meist nur kurz und listet meist nur die Optionen auf.

Der **help** Befehl zeigt Hilfe zur aktuellen shell (BASH) an. **help echo** würde z.B. zum (intern in der shell eingebauten) echo Befehl anzeigen. Das funktioniert aber nur bei Befehlen die direkt, intern in der shell eingebaut sind. Für die meisten Befehle brauchen wir daher eine andere Hilfe:

man ist der Befehl zum aufruf der **man**ual Seiten. Die meisten Befehle haben solche man-pages.

`man echo` würde z.b. die man page zum echo Befehl ausgeben (der neben dem eingeboten echo kommando ebenfalls existiert).

man verwendet zur Anzeige der Seiten einen so genannten "Pager". Da viele man-pages sehr lang sind kann man/frau damit in den Pages blättern.

Hier die wichtigsten Keyboard-Shortcuts für den standard Pager. Der standard-Pager im Linux ist "less" und ist eine weiterentwicklung von "more".

Leertaste zum weiterblättern.

q beenden

Pfeiltasten oder j k um jeweils eine Zeile nach unten oder oben zu verschieben.

/suchstring sucht nacht "suchstring". (Wobei "suchstring" eine so genannte "regular expression" ist - mehr dazu später).

n die vorgehende Suche wiederhohlen (weitersuchen -**n**ext).

N weitersuchen nach oben (also in die andere Richtung).

Auf einer grafischen oberfläche kann auch das Gnome-Help tool "yelp" verwendet werden: z.b.: `yelp man:ls`
Viele GNU Tools haben neben einer man-page auch eine "info" page. **info** ist etwas mühsam zu bediene, aber kann dafür Links zu anderen info Seiten enthalten. Eine frühe Form von "hypertext".[1]

1.10 Prozessmanagment

Um zu sehen was auf deinem Linux gerade so läuft:

```
ps uax
```

Das **ps** Komamndo zeigt dir die aktuell laufenden Prozesse im System. Die Optionen "uax" und "elf" werden oft verwendet um mehr info zu sehen als ps alleine annzeigt. Das **pstree** Tool zeigt eine Baumstruktur der Prozesse an (wer welchen gestartet hat).

Oft ist interessant zu sehen wer gerade die meiste CPU verbraucht. Dafür ist **top** nützlich. top ist ein interaktives tool und listet die Prozesse nach ihrer CPU Auslastung. (Alternativ wird nach Memory bedarf sortiert: nach dem Starten von top M (groß) drücken). Ein kleines q (quit) oder ein Ctrl-c beendet das Tool.

Alle Prozesse haben eine Nummer. Diese kann bentutz werden um die Prozesse zu steuern. z.B. um ihnen zu sagen sie sollen sich beenden:

`kill 1234` würde dem Prozess mit der nummer 1234 sagen, er soll sich bitte beenden. Notfalls muss das Betriebsystm mit dem Holzhammer nachhelfen: `kill -9 1234` hilft dann.

1.11 CLI für Vortgeschrittene

Die bisher gelernten Befehle reichen für einfache Aufgaben. Wer etwas mehr in der Shell machen will benötigt aber noch einige extrams. Hier die wichtisten davon:

1.12 Mit ssh auf einem anderen Rechner arbeiten

Mit

```
ssh username@hostname
```

kann eine verschlüsselte Konsole Verbindung zu einem anderen Linux Rechner aufgebaut werden. Dort kann man/frau dann so arbeiten wie am lokalen Rechner. Mehr dazu im eigenen SSH Kapitel.

1.13 Dateien suchen mit "find"

Find haben wir schon kurz erwähnt. Ohne Argumente listet es alle Datei und Verzeichnisnamen ab dem aktuellen Verzeichnis auf. Ein Argument wird als Verzeichnis gelesen, ab dem gesucht werden soll. z.B.:

```
find /etc
```

Die Liste ist dabei meist recht lang. Um spezifischer suchen zu können gibt es sehr sehr viele Argumente im find. Hier einige der wichtigsten um die Suche einzuschränken:

find -type f sucht nur Files aber keine Verzeichnisse.

find -type d sucht nur Verzeichnisse (directories).

find -name bla* sucht nur files die mit bla beginnen. (der Backslash \wird benötigt damit die shell nicht versucht den * durch eine liste aller Datein zu ersezten).

find -mtime -3 listet alle Files die in den letzten 3 Tagen (eigentlich 72 Stunden) verändert wurden. (Wichtig ist das Minus es sagt: bis zu 3 Tagen. Ansonsten würden nur Files gelistet die genau 3 Tage (gerundet) alt sind.

[1] Der KDE Eigene Webbrowser konqueror erlaut die Eingabe von `man:` und `info:` URLs im Browser die direkt zu info und man Seiten führen. z.B.: `info:seq`

1.14 In Dateien Suchen mit “grep”

Grep erlaubt das Suchen in Files nach so genannten “regular expressions”. Die werden wir nicht im Detail behandeln sondern nur die wichtisten Fälle auflisten. Zum Üben benötigen wir zuerst einmal eine Datei mit viel Inhalt: z.B. `find /etc > uebung.txt` uebung.txt enthält jetzt eine Liste aller Files unterhalb von /etc. Die können wir jetzt durchsuchen:

```
grep ost uebung.txt
```

Damit durchsucen wir die datein uebung.txt und listen alle Zeilen die die Zeichenfolgen “ost” enthalten. Wir können natürlich auch jede andere Zeichenfolge versuchen. Hier die Wichtigsten Optionen für grep, die die Suche und die Ausgabe der Suche beeinflussen:

-i Case**i**ndependent - also Groß/Kleinschreibung ignorieren.

-v re**v**erse - die Suche umkerhren. Es werden alle Zeilen ausgegeben die den text **nicht** enhalten.

-c count - es wird ausgegeben wie oft die Zeichenfolge vorkommt

-n li**n**e number - es wird die Zeilennummer mit ausgegeben

-2 es werden 2 Zeilen vor und nach dem Treffer mitangezeigt.

Die Optionen lassen sich natürlich auch kombinieren. z.B.:

```
grep -3ni ost uebung.txt
```

Zeigt alle Files die die Zeichenfolge “ost” enthalten, ignoriert dabei Groß/Kleinschreibung, zeigt 3 Zeilen vor und nach jedem Treffer und gibt bei der Ausgabe die Zeilennummer mit an.

Hier noch einige Beispiele für “Regular Expressions”:

blabla Sucht nach genau dem Text “blabla”.

bl.bla Würde auch blibla und bl7bla finden. Der Punkt ist Joker für ein beliebiges zeichen.

bl[0-9]bla findet nur Zeichenketten die an der dritten Stelle eine Ziffer haben.

bla*bla findet blbla blabla blaaaaaaabla. Der * bedeutet: eine beliebige (inkl. 0) Anzahl des vorigen Zeichens.

^bla Nur Zeilen die mit bla **beginnen**

bla$ Nur Zeilen die mit bla **enden**.

1.15 Mit einer “Pipe” - Ausgabe von einem Programm zum nächsten weiterleiten

Wir haben bereits gelernt wie wir die Ausgabe eines Programmes in ein File umleiten (>) und wie wir die Eingabe zu einem Programm aus einem File lesen (<). Oft wollen wir uns den Zwischenschritt ersparen und einfach die Ausgabe eines Programmes an ein weiteres Programm leiten. Dazu dient der Pipestrich (|). z.B.:

```
find . | grep Dok
```

Hier verwenden wir das vorher glernte grep Tool um die lange Ausgabe von find etwas zu filtern. Natürlich können zusätzliche Pipes benutzt werden um die Ausgabe wieder Weiter zu leiten. z.B.:

```
find . | grep Dok | grep -v [0-9]$
```

Würde aus der vorigen Ausgabe noch alle Zeilen entfernen (-v) die mindestens mit einer Ziffer enden.

1.16 Nützliche Tools für die Pipe

Die meisten der Tools hier können auch direkt mit einem oder mehreren Files als Argument benutzt werden. Sehr praktisch sind sie aber auch innerhalb einer Pipe:

less ist ein "Pager" (Siehe das Kapitel zu "man"). Less ist die weiterentwicklung des alten Unix Tools "more". Mit dem Pager kann in der Ausgabe geblättert werden.

tr Zeichen ersetzen: z.b: `tr A-Z a-z` alle Großbuchstaben durch Kleinbuchstaben ersetzen.

sed der "stream-editor". Ersetzt Zeichenfolgen. z.B.: `sed -es/Kapitalismus/Kommunismus/g`

sort sortiert die Zeilen alphabetisch.

sort -n sortiert die Zeilen numerisch nach der Zahl am Zeilenanfang.

sort -u unterdrückt gleiche Zeilen (**u**nique)

uniq -c zählt wie oft eine Zeile vorkommt (muss zuerst mit sort gefiltert sein).

uniq -u gibt nur Zeilen aus die nicht doppelt vor kommen (welche die doppelt vorkommen werden, werden im Gegensatz zu `sort -u` gar nicht ausgegeben).

Hier ein Beispiel: Angenommen wir haben eine Liste an Namen die sich für eine Veranstaltung angemeldet haben und wir haben eine Zweite Liste von Namen die sich abgemeldet haben. Wir wollen nun eine alphabetisch geordnete Liste aller Namen die nicht agemeledet sind.

```
cat anmeld.txt  abmeld.txt abmeld.txt | sort | uniq -u
```

Wie funktioniert das? Das cat gibt die anmeld.txt und abmeld.txt aus. Die abmeld.txt aber doppelt. D.h.: Jeder Name der in der abmeld.txt ist kommt mindestens 2 mal in der ausgabe vor. (Falls er auch in der anmeld.txt vor kommt dann sogar 3 mal). `sort` sortiert die liste und `uniq -u` gibt nur noch die Zeilen aus die nur einmal vor kommen. Damit sind alle Enthalten die Angemeldet waren oder nicht in der Abemeldung aufscheinen. (Zur Sicherheit sollte zuvor überprüft werden ob in der anmeld.txt keine Doppelten vorkommen: `cat anmeld.txt | sort | uniq -d`)

Ein realistischers Beispiel: Wir haben in einem Verzeichnisbaum 2 Millionen Files und davon eine Kopie in der einige Files fehelen oder hinzu gebekommen sind und wir wollen sehen welche das sind.

1.17 Environment Variablen

Neben de Argumenten und Optionen die einem Programm beim Starten übergeben werden gibt es noch eine Zusätzliche möglichkeit den Programmen etwas Mitzugeben. Die so genannten "Environment Variablen".

Die Variablen werden in der shell verwaltet. Variablen die zum "export" bestimmt sind, werden dabei auch an alle gestarteten Programme weiter gegeben. Alle nicht exportierten sind nur inten für die shell selbst sichtbar. Viele der Variablen werden auch benutzt um bash-funktionen zu Speichern.

Wichtige Befehle zur Verwaltung dieser Variablen in der Bash sind:

set zeigt eine Liste aller Variablen

export zeigt eine Liste aller exportierten Variablen

BLA=sonstwas setzt den Inhalt der Variable $BLA auf "sonstwas".

echo $BLA zeigt den Inhalt der Variable bla. (genau genommen wird der Inhalt der Variablen auf der shell eingesetzt. echo gibt disen Inhalt dann 1-zu-1 wieder aus.)

export BLI=hammer setzt den Inhalt der Variable BLI auf "hammer" und exportiert diese Variable. Nachfolgend aufgerufen Programme in dieser shell können diese Varialbe sehen.

let wochenstunden=7*24 setzt die Variable wochenstunden auf 168 (let führt einfache (integer) Berechnungen aus)

echo X${BLA}Y zeigt den Inhalt von $BLA aber zuvor ein X und danach ein Y. Die Klammern dienen daher dazu um die Benennung der Variable eindeutig zu machen.

Manche Variablen sind in der shell schon eingebaut und zeigen Bestimmte shell-interne Zustände an. Andere werden per konvention von Verschiedenen Programmen verwendet. Hier eine Liste der wichtigsten Variablen:

$HOME das Home-Verzeichnis der gerade eingelogten BenutzerIn.

$? der Status-Code des Zuletzt beendeten Befehls (0 = OK).

$PATH Eine mit Doppelpunkten : getrennte Liste aller Pfade in denen Programme gesucht werden ohne Angabe des absoluten Pfadnamens gestartet werden sollen. Wenn wir ein Programm starten und nicht wissen woher es ausgeführt wird (In welchen der vielen Verzeichnisse von $PATH es liegt) können wir dem Befehl ein **which** voranstellen.

$HOSTNAME Der Name des Computers. Der kann auch mit dem **hostname** Befehl abgefragt werden.

$EDITOR der default Editor der von vielen Programmen aufgerufen wird.

1.18 Profile

Die Variablen die man/frau setzt sind nur in der aktuellen shell und, falls sie exportiert wurden, auch in den davon aufgerufenen Programmen gültig. Wollen wir bestimmte Einstellungen permanent machen, so müssen wir die Variablen immer beim einloggen automatisch setzen. Es gibt folgende Möglichkeiten das zu tun:

.profile Die Datei wird beim einloggen von der Bash - und (falls andere shells verwendet werden i.a. auch von diese) geladen.

.bash_profile Falls diese Datei existiert wird die für die BASH verwendet. Die .profile wird in diesem Falle **nicht** gelesen.

.bashrc Diese Datei wird für **jede** neue BASH shell gelesen. Man/Frau sollte daher sehr vorsichtig sein was dort hinein geschrieben wird, da intern oft sehr viele shells aufgerufen werden.

/etc/profile Diese Datei wirkt wie die .profile aber systemweit für alle User. Man sollte daher damit ebenfalls Vorsichtig sein.

Sehr beliebt in den profil-scripts sind **alias** Definitionen. Damit lassen sich Abkürzungen für Befehle definieren. Mit dem Befehl `alias` ohne Argumente können wir uns die momenatanen alias Definitionen ansehen.

1.19 Command Substitution

Wie wir mit dem Pipe (|) Stirch die Ausgabe von einem Befehl in den nächsten umleiten haben wir bereits durch gemacht. Was aber wenn wir die Ausgabe eines Befehls in die Kommandozeile für einen anderen Befehl einsetzen wollen? Das geht in der Bash mit $(`befehlszeile`). Eine Variante die auch in andern Shells funktioniert sind die Backticks. Die einfachen Anführungszeichen nach hinten: `` `befehlszeile` ``.

Einige Beispiele:

```
echo Wir schreiben das Jahr $(date +%Y). Heute ist $(date +%A).
cp $(find Dokumente/ -name \*.jpg -type f) archiv/
```

Der erste Befehl gibt das Jahr und den Wochentag aus. (Es werden dabei spezielle Optionen des date Befehls benutzt.

Der zweite Befehl sucht alle Datein (keine Verzeichnisse), die mit auf .jpg enden und kopiert sie in ein Verzeichnis namens archiv. Dabei gibt es einen möglichen Stolperstein: Da hier die Namen der Dateien mit Leerzeichen getrennt sind würde der Befehl fehlschlagen falls auch Dateiene mit Leerzeichen im Namen vorhanden sind. Einer der Gründe um Leerzeichen in Dateinamen zu vermeiden. Ein weiterer: Die länge der Befehlszeile ist beschränkt, aber meist kein Problem (Üblich auf einem modernen Linux 2 Millionen Zeichen).

Es gibt jedoch bessere Methoden um Befehle auf sehr viele Files anzuwenden:

1.20 Den selben Befehl auf viele Files anwenden

Viele Befehle erlauben ohnehin mehre Files als Argument. Falls das nicht geht, oder falls die Liste der Files auf die ein Kommando angewendet werden soll erst durch ein Tool erstellt wird, gibt es mehrere Möglichkeiten.

1.20.1 find mit der -exec option

```
find Meinefiles/ -type f -name \*.pdf -exec pdftotext {} \;
```

Obiges würde den befehl pdftotext füre alle pdf files aufrufen die unterhalb von Meinefiles/ liegen. Das {} wird durch den aktuellen Filenamen ersetzt.

1.20.2 xargs

Das xargs Tool liest Zeilen von STDIN ein und ruft für jede Zeile den Befehl auf der als erstes Argument angegeben wurde. Dabei kann angegeben werden wieviele Argumente maximal übergeben werden. Per default nimmt aber auch xargs Leerzeichen als Trennzeichen. Um das zu Vermeiden kann das Trennzeichen mit der option -d explizit angegeben werden. Das Zeilenende kann mit \n angegeben werden.

```
seq 1 20 | xargs -n 6 echo
```

der Output würde dann so aussehen:

```
1 2 3 4 5 6
7 8 9 10 11 12
13 14 15 16 17 18
19 20
```

`find` kann mit der option `-print0` die Ausgabe mit Ascii-0 zeichen getrennt ausgeben. Diese können in Filenamen nicht vorkommen. `xargs` kann mit der option `-0` dieses Format lesen. Eine Leerzeichensichere version der PDF in text Konvertierung könnte also so aussehn:

```
find Meinefiles/ -type f -name \*.pdf -print0 | xargs -0 -n 1 pdftotext
```

1.21 Ein einfaches shell Script

Obwohl sich sehr viel in "Einzeilern" erledigen lässt ist es oft übersichtlicher und einfacher für komplexere Aufgaben ein so genanntes "shell script" zu schreiben. Ein script kann in verschiedenen Programmiersprachen geschrieben werden aber natürlich auch direkt mit der bash. Hier ein Beispiel für ein sehr einfaches script:

```
#!/bin/bash
# mein erstes shell script
echo heute ist der $(datum).
echo dein homeverzeichnis ist $HOME
```

Was tut das script? Das Kanalgitter # beginnt in der shell einen Kommentar. D.h. was nach dem # kommt wird ignoriert. Die erste Zeile ist somit aus sicht der shell auch ein Kommentar: `#!/bin/bash`

Die Kombination `#!` sagt allerdings dem Linux dass dieses script mit diesem Programm (eben der bash shell) ausgeführt werden soll.

Die nächste Zeile ist dann ein echter Kommentar und in den letzten Beiden Zeilen wird echo benutzt um eine Ausgabe am Bildschrim zu machen.

Wie können wir das script jetzt starten? Zuerst müssen wir es in ein File schreiben. z.b. `meinscript.sh` und danach müssen wir dem Linux sagen, dass dies ein ausführbares Script oder Programm ist. Dazu der Befehl:

```
chmod ugo+x meinscript.sh
```

Um das script dann wirklich auszuführen müssen wir den namen des scripts eintippen. Da üblicherweise das aktuelle Verzeichnis nicht im $PATH Pfad ist müssen wir das explizit angeben. Also z.B.:

```
./meinscript.sh
```

Hier noch ein Beispiel für ein etwas komplexeres script das files in thumbnails mit der breite 120 pixel umwandelt.

```
#!/bin/bash
#
# wir wollen im script zaehlen. als zaehler benutzen wir die variable cnt
# die setzen wir am anfang auf 0
cnt=0
```

```
#

# mit einer for schleife koennen wir ueber all angegeben argumente dieses
# scripts durchlaufen. der shell variable hier k wird dabei nach der reihe
# eines der argumte zugweisen.

for k ; do
  echo ich arbeite jetzt mit $k
  let cnt=cnt+1
  # hier geben wir die variable $k aus und erhoehen unseren cnt zaehler
  convert -resize 120  $k .thumbnail.$k
  # das programm convert gehoert zum imagemagik packet und kann bilder
  # in verschiedene formate konvertieren und skalieren.
  # obiger befehl nimmt das originalfile mit namen $k und
  # skaliert es auf 120 pixel breite und nennt das ausgabefile .thumbnail.$k
  # files die mit einem . beginnen sind hidden files.
done
echo wir haben $cnt files konvertiertet
```

Um das script laufen zu lassen müssen wir es wieder ausführbar machen und danach mit den passenden Argumente aufrufen:

```
chmod ugo+x meinthumbnail.sh
./meinthumbnail.sh *.jpg *.png
```

1.22 User, Gruppen und Zugriffsrechte

Im Unix haben alle User einen Usernamen und eine numerische ID. Zusätzlich ist jede/r User Mitglied ein oder mehrere Gruppen. Um zu sehen welchen Gruppen wir angehören können wir den Befehl **id** verwenden:

Die Ausgabe könnte etwa so aussehen:

```
uid=1000(karli) gid=1000(karli) groups=1000(karli),24(cdrom),29(audio),119(bluetooth),23000(staff)
```

Wir sind also Benutzer “karli” mit ID 1000 und der primären Gruppe des selben Namens und der selben ID (1000). Weiters sind wir Mitglieder der Gruppen cdrom,audio,bluetooth und staff. Damit dürfen wir auf die entsprechende Hardware zugreifen und können auf Files zugreifen die der Gruppe “staff” zugeordnet sind.

Eine spezielle Rolle hat der user root (ID 0). Der/die darf alles. Dazu mehr später mehr.

Jedes File gehört immer eine/r BenutzerIn und einer Gruppe.

Die normalen Zugrifssrechte im Unix sind immer Dreigeteilt: Es gibt Rechte die auf den User selbst beschränkt sind dem/der die Datei gehört. Es gibt Rechte für die Gruppe der die Datei gehört und dann noch: Rechte für alle Anderen (“Other” manchmal auch “World”).

Die elementaren Rechte sind **r**: Lesen, **w**: Schreiben und **x**: Ausführen (execute). Der **ls -l** Befehl zeigt uns an wem eine Datei gehört und wie die Zugriffsrechte aussehen. Die Ausgabe ist dabei in drei Dreierblocks `rwx` jeweils für User Group und Other. Davor gibt es noch einen Buchstaben der Zeigt welcher Art das File ist. - zeigt normales File: z.B:

```
-rw-r--r-- 1 karli staff 31 Dec 28 15:35 bla.txt
-rw-r----- 1 karli cdrom 31 Dec  8 15:35 bli.txt
-rwxrwx--- 1 karli staff 31 Dec 12 15:35 blo.sh
drwxr-x--- 5 anna  staff 31 Dec 20 17:10 Dok/
```

Die bla.txt darf nur von karli geschrieben werden (w) aber von allen anderen (Group und Other) geschrieben werden.

Die bli.txt darf neben karli auch von allen Mitgliedern der gruppe “cdrom” gelesen werden.

Die blo.sh sowohl von karl als auch allen Mitgliedern der “staff” Gruppe gelesen, geschrieben und ausgeführt werden, aber von sonst niemanden.

Der letzte Eintrag ist ein Verzeichnis (zu erkennen am d am Anfang) namens Dok. Das Verzeichnis gehört anna die dort auch lesen und schreiben darf. Schreiben in einem Verzeichnis bedeutet hier: Files anlegen. Alle Mitgliede der staff Gruppe dürfen dort auch lesen (d.h. Files auflisten). Das **x** hat für Verzeichnisse eine spezielle bedeutung. Ein fehlendes x verbietet jeglichen Zugriff auf alles was unterhalb dieses Verzeichnisses

liegt, insbesondere auch auf Files auf die man/frau sonst Zugriff hätte. Niemand ausser anna und die Mitglieder der staff Gruppe können also sehen was unterhalb von Dok liegt und können dort auch keine Files lesen, selbst wenn diese ansonsten von “Other” lesbar wären.

1.22.1 Wie verändern wir jetzt die Zugriffsrechte?

```
chown karli:staff bli.txt
```

Andern BenutzerInnen ein File mit chown zu geben darf nur der/die root-UserIn. Die Gruppe darf man/frau auch nur auf Gruppen ändern denen man/frau selbst angehört. Obiger Befehl würde die Datei bli.txt der Gruppe staff zuordnen.

```
chmod go-w bli.txt
chmod ugo+r bli.txt
```

Obiger Befehl würde für Group und Others die Schreibrechte entfernen (falls vorhanden) und der zweite Befehl würde User, Group und Others Leserechte geben.

Wichtig für scripts (siehe oben) ist die e**x**ecute Berechtigung für scripts:

```
chmod ugo+rx meinscript.sh
```

- alias
- builtin vs external
- which
- su -, sudo, id
- passwd

* useful CLI tools (15 min)
* tar

c,v,x,p

* gzip
* locate
* pdftk
* cal gcal -n
* w3m, curl, wget
* mutt
* host, ping, whois
* minicom
* factor
* bc
* dict, fortune
* irc
* figlet
* dd
* rsync