Workstation Linuxkurs

Paul Hänsch, Workstation Ideenwerkstatt Berlin e.V.

27. Februar 2010

Inhaltsverzeichnis

[1]*

1 Tag 1

1.1 Arbeit mit Dateien

Legen wir mal eine Datei an:
`touch datei1`

Der Befehl '`ls -l datei1`' zeigt uns detaillierte Informationen:
`-rw-rw---- 1 user user 0 Feb 21 16:49 datei1`

Was sagt uns das nun?

Zugriffsrechte	**Linkzahl**	**Eigentümer**	**Gruppe**	**Größe**	**Schreibzeit**	**Name**
`-rw-rw----`	`1`	`user`	`user`	`0`	`Feb 21 16:49`	`datei1`

Bevor wir uns das näher anschauen schreiben wir mal etwas in die Datei:
`echo "Hallo Welt" > datei1`

und nochmal '`ls -l datei1`':

Zugriffsrechte	**Linkzahl**	**Eigentümer**	**Gruppe**	**Größe**	**Schreibzeit**	**Name**
`-rw-rw----`	`1`	`user`	`user`	`11`	`Feb 21 17:02`	`datei1`

Wir sehen:

- Die Größe ist von 0 auf 11 gewachsen (11 stücke Kuchen?...)
- Die Zugriffszeit hat sich verändert (warum wohl?)

Die Größe wird natürlich in Byte angegeben, das kann einem bei größeren Dateien schon ziemlich auf den Senkel gehen. Geben wir z.B. mal ein:
`ls -l /boot/initrd.img-2.6.30-2-686` (vielleicht heißt die Datei auch ein bisschen anders)
⇒
`-rw-r--r-- 1 root root 7277035 Jan 17 13:00 /boot/initrd.img-2.6.30-2-686`

Soso... die Datei ist also 7277035 Byte groß (je nach System vielleicht auch etwas mehr oder weniger, machen wir uns darum gerade mal keinen Kopf) Die Einheit ist natürlich etwas unpraktisch. Fügen wir dem 'ls'-Befehl mal den Parameter 'h' für "human readable" (menschenlesbar) hinzu:
`ls -l -h /boot/initrd.img-2.6.30-2-686`
⇒
`-rw-r--r-- 1 root root 7.0M Jan 17 13:00 /boot/initrd.img-2.6.30-2-686`

Jetzt lautet die Ausgabe 7.0M, also 7 Megabyte, 'ls' wählt automatisch eine sinnvolle Einheit. Je nach Dateigröße könnte also z.B. auch der Buchstabe K (Kilobyte) oder G (Gigabyte) dort stehen.

Mit dem Befehl 'cat' können wir übrigens den Inhalt der Datei ausgeben:
`cat datei 1`
⇒
`Hallo Welt`

Wenn wir jetzt nochmal '`ls -l datei1`' eingeben sehen wir übrigens **keinen** Unterschied zu vorher:
`-rw-rw---- 1 user user 11 Feb 21 17:02 datei1`

Die Zugriffszeit ändert sich also nur beim **Schreiben**, wir haben aber gerade nur aus der Datei **gelesen**. Der Zeitstempel ändert sich dabei nicht. Nochwas fällt dem aufmerksamen Beobachter ins Auge: Die Datei ist angeblich 11 Byte groß, dabei hat "Hallo Welt" nur 10 Zeichen (mit Leerzeichen).
Die Antwort ist einfach: Am Ende der Datei steht noch ein Zeilenumbruch, der belegt ein Byte.

1.2 Zugriffsberechtigungen

Damit ist uns erst mal klar, wofür die felder "Größe", "Schreibzeit" und "Name" stehen. Wenn wir uns jetzt den Zugriffsrechten widmen, interessieren uns drei weitere Felder.

Zugriffsrechte	Linkzahl	**Eigentümer**	**Gruppe**	Größe	Schreibzeit	Name
`-rw-rw----`	`1`	`user`	`user`	`11`	`Feb 21 17:02`	`datei1`

- **Eigentümer:** ist der Benutzer, der die Datei angelegt hat. Der Eigentümer kann nachträglich nicht verändert werden (außer durch root).

- **Gruppe:** ist die Gruppe, der die Datei zugeordnet ist. Nur der Eigentümer (oder root) kann die Gruppenzugehörigkeit der Datei ändern. Der Eigentümer kann die Datei auch nur einer Gruppe zuordnen, in der er selbst Mitglied ist.
- **Zugriffsrechte:** hier wird es komplizierter, wir müssen uns das näher anschauen.

Das Feld Zugriffsrechte

Dieses erste Feld beschreibt in extrem kurz gefasster Weise, **wer** mit der datei **was** anfangen darf:

Typ	Eigentümer	Gruppe	Andere
-	rw-	rw-	---

Ignorieren wir für den Augenblick den Typ. Wir sehen unter Eigentümer die Buchstaben 'r' und 'w'. An dritter Stelle könnte ein 'x' stehen, dies ist hier jedoch **nicht** der Fall. Das 'r' steht für "read" (lesen) und das 'w' für "write" (schreiben). Das heißt der Eigentumer der Datei (in unserem Beispiel "user") darf die Datei lesen und darin schreiben. Die Datei ist in diesem Beispiel der Gruppe "user" zugeordnet. Die Gruppe "user" ist nicht zu verwechseln mit dem Benutzer "user"! (Das ist wirklich wichtig!) Mitglieder der Gruppe "user" haben, wie wir sehen ebenfalls die Berechtigungen 'lesen' (r) und 'schreiben' (w). Auch hier ist der dritte mogliche Buchstabe (x) nicht gesetzt. Unter "Andere" ist keiner der drei Buchstaben 'rwx' gesetzt. Andere Benutzer als der Benutzer "user" bzw. Benutzer, die nicht wenigstens in der Gruppe "user" sind dürfen mir der Datei folglich gar nichts anfangen.

Achja, wir reden die ganze Zeit von so einem 'x'. Das 'x' verleiht jemandem die Rechte eine Datei auszuführen ('execute'). Bei einer Datei in der nur "Hallo Welt" drin steht gibt es nun wohl kaum was auszuführen. Nur bei Programmen macht das Sinn (und bei Verzeichnissen, aber hier greife ich vor). Wenn wir uns also mal die Rechte von '/bin/cat' anschauen (ja, das ist das Programm 'cat', mit dem wir vorhin die Datei ausgegeben haben). Dann sehen wir:

```
-rwxr-xr-x 1 root root 46848 Jun  5  2009 /bin/cat
```

- Der Eigentümer, nämlich der Benutzer "root", ist der Einzige, der in die Datei schreiben darf.
- Die Datei gehort auch der Gruppe "root", da in dieser Gruppe niemand Mitglied ist außer der Benutzer "root" ist das uninteressant. Es ist übrigens sehr interessant, dass das uninteressant ist, dadurch wird der Gruppenzugehörigkeit nämlich bewusst die Bedeutung genommen. Die Datei ist quasi keiner besonderen Gruppe zugeordnet. Aus diesem Grund ist es üblich jedem Benutzer eine persönliche Gruppe zu verpassen.
- Jeder beliebige Benutzer darf das Programm lesen bzw. ausführen (dafür sorgt das 'r' und das 'x'). unter "Andere".

Jetzt wollen wir mal mit den Rechten unserer Experimentierdatei rumspielen. Zum Ändern der Rechte dient das Programm 'chmod' ("Change Mode" - Ändern des Zugriffsmodus). Wir können zum Beispiel uns, dem Besitzer der Datei die Leserechte entziehen:

```
chmod u-r datei1
```

Das 'u' zeigt an, dass wir die Rechte des Eigentümers (user, der Benutzer der Datei) ändern wollen. Der Besitzer hat hier nur zufällig den Namen "user", das Kürzel 'u' wird immer verwendet um den Besitzer anzusprechen. Wir benutzen das Minus-Zeichen (-) um Rechte zu entziehen. Das Recht, das wir entziehen ist das Leserecht (r).

```
ls -l datei1
⇒
--w-rw---- 1 user user 11 Feb 21 17:02 datei1
```

Wir sehen, der Besitzer der Datei hat kein Leserecht mehr. Probieren wir aus der Datei zu lesen:

```
cat datei 1
⇒
cat: datei1: Permission denied
```

Die Meldung ist "Permission denied" - Zugriff verweigert. Wir dürfen die Datei einfach nicht mehr lesen.

```
chmod u+r datei1
cat datei 1
⇒
Hallo Welt
```

Durch die Verwendung des Plus-Zeichens haben wir das Leserecht (r) für den Besitzer (u) wieder hinzugefügt.

Durch Eingabe des Kommandos '`groups`' finden wir heraus, in welchen Gruppen unser Benutzer Mitglied ist, z.B.:

```
groups
user fuse share
```

An erster Stelle wird die primäre Gruppe unseres Benutzers aufgeführt (im Beispiel ist das die Gruppe "user"). Die primäre Gruppe ist die Gruppe, der jede neu angelegte Datei standardmäßig zugeordnet wird. Mit dem Kommando 'chgrp' (Change Group - Gruppe Ändern) können wir die Datei einer anderen Gruppe zuordnen **in der wir Mitglied sind**.

```
chgrp share datei1
ls -l datei1
⇒
-rw-rw---- 1 user share 11 Feb 21 17:02 datei1
```

Alle Benutzer in der Gruppe "share" haben jetzt Lese- und Schreibzugriff (rw-) auf die Datei.
Mit '`chmod g-rw datei1`' entziehen wir Benutzern der Gruppe (g) "share" den Lese- (r) und Schreibzugriff (w):

```
ls -l datei1
⇒
-rw------- 1 user share 11 Feb 21 17:02 datei1
```

Mit '`chmod o+r datei1`' geben wir "anderen" ('o' wie others) Benutzern den Lesezugriff. "Andere" sind Benutzer, die weder Besitzer der Datei sind noch in der Gruppe der Datei Mitglied sind.

```
-rw----r-- 1 user share 11 Feb 21 17:02 datei1
```

1.2.1 Berechtigungen auf Verzeichnissen

Nehmen wir an unser Benutzer hat das Heimverzeichnis '/home/user', ein '`ls -l -d /home/user`' zeigt uns die Daten des Verzeichnisses an. Der Parameter 'd' sorgt dafür, dass wir das Verzeichnis selbst sehen, nicht dessen Inhalt.

```
drwx------ 120 user user 7160 Feb 22 13:28 /home/user
```

Wir sehen zunächst, dass das erste Zeichen in der Modusanzeige mit einem 'd' Belegt ist. Dies zeigt an, dass es sich nicht um eine normale Datei handelt, sondern um ein Verzeichnis (directory).
Wir sehen außerdem, dass der Besitzer von dem Verrzeichnis (user) dass Verzeichnis lesen (r), schreiben (w) und ausführen (x) kann.

- Ein verzeichnis **lesen** zu können bedeutet, sehen zu können welche Dateien darin enthalten sind (Mit den Leserechten auf den dateien selbst hat das nichts zu tun).
- In ein Verzeichnis **schreiben** zu können bedeutet Dateien darin anlegen oder löschen zu können, dies ist nämlich eine Änderung am Verzeichnis. Man braucht keine Schreibrechte auf einer Datei, um diese zu löschen.
- Ein Verzeichnis **ausführen** zu können bedeutet, das Verzeichnis betreten zu können und sich die Dateien darin näher anschauen zu können.

1.3 Zusammenfassung

Wir haben gelernt, wie man Dateien Anlegt, und wie man mit den Berechtigungen auf Dateien und Verzeichnissen umgeht. Die Folgenden Kommandos haben wir verwendet:

- **touch** - zum Anlegen einer Datei

  `touch foobar` legt eine Datei mit dem Namen 'foobar' an

- **ls** - zum Anzeigen von Dateien und Verzeichnissen

  `ls` zeigt Dateien im aktuellen Verzeichnis an

  `ls -l` zeigt Dateien im aktuellen Verzeichnis in detaillierter Form an

  `ls -l -h` oder `ls -lh` zeigt Dateien im aktuellen Verzeichnis in detaillierter Form mit menschenlesbaren Größenangaben an

  `ls -l foobar` zeigt detaillierte Informationen nur über die Datei 'foobar' (die sich im aktuellen Verzeichnis befindet) an

  `ls /home/` zeigt den Inhalt des Verzeichnisses '/home/' an

  `ls -d /home/` zeigt das verzeichnis '/home/' selbst an

  `ls -l /home/` zeigt den Inhalt des Verzeichnisses '/home' in detaillierter Form an

  `ls -l -d /home/` zeigt detaillierte Informationen über das Verzeichnis '/home/' selbst an

  Die Parameter 'l', 'h' und 'd' sind beliebig kombinierbar und können auch "zusammngezogen" werden (z.B. `ls -lh`)

- **echo** - zum Ausgeben von Text (den wir auch in eine Datei leiten können)

  `echo Hallo Welt` gibt "Hallo Welt" aus

  `echo "Hallo Welt"` gibt genau so "Hallo Welt" aus

  `echo -n Hallo Welt` gibt "Hallo Welt" ohne abschließenden Zeilenumbruch aus

  `echo Hallo Welt > quaquuz` leitet den Text "Hallo Welt" in die Datei 'quaquuz'

- **cat** - zum Ausgeben des Inhalts einer Datei

  `cat foo` gibt den Inhalt der Datei 'foo' aus

  `cat foo qua` gibt den Inhalt der Datei 'foo' und nahtlos anschließend den Inhalt der Datei 'qua' aus

- **chmod** - zum Ändern des Zugriffsmodus (Zugriffsrechte) einer Datei

  `chmod u-w bar` entzieht dem Besitzer der Datei 'bar' den Schreibzugriff auf die Datei

  `chmod o+rx /home/user/` gewährt "anderen" Benutzern, also in diesem Sinne Benutzern die weder Besitzer des Verzeichnisses '/home/user/' sind, noch der Gruppe des Verzeichnisses '/home/user/' angehören das Recht Dateien in diesem Verzeichnis zu sehen (r), Details über diese Dateien anzuzeigen (x) und das Verzeichnis zu betreten (x)

- **groups** - um anzuzeigen in welchen Gruppen ein Benutzer Mitglied ist

  `groups` zeigt die Gruppen des aktuellen Benutzers an

  `groups user` zeigt die Gruppen des Benutzers 'user' an

- **chgrp** - zum Ändern der Gruppenzuweisung einer Datei

  `chgrp share foobar` ordnet der Datei 'foobar' die Gruppe 'share' zu. Der Benutzer der den Befehl ausführt muss Mitglied von 'share' sein.

2 Tag 2

2.1 Navigation im Dateisystem

Wenn der Benutzer eine Kommndozeilenumgebung öffnet, so befindet er sich normalerweise in seinem Heimverzeichnis. In jedem Fall lässt sich das aktuelle Arbeitsverzeichnis feststellen durch Eingabe des Befehls '`pwd`' (print working directory - gib das Arbeitsverzeichnis aus).

```
pwd
⇒
/home/user
```

'/home/user' (in diesem Beispiel) ist das Verzeichnis, in dem wir gerade arbeiten. Mit dem Befehl '`cd`' (change directory - ändere das Verzeichnis) können wir ein anderes Verzeichnis (auf das wir Ausführungsrechte haben) betreten. Bevor wir das tun, wollen wir uns aber nochmal umschauen:

```
ls
⇒
...
```

Der Befehl '`ls`' wird alle Dateien im Heimverzeichnis des Benutzers auflisten. Auf den Abdruck der Befehlsausgabe soll hier verzichtet werden, da der Grundzustand des Heimverzeichnisses sich je nach Distribution und Benutzerumgebung stark unterscheiden kann.
Mit '`cd ./`' wechseln wir in das (nicht aufgelistete) "Unterverzeichnis" './'. Das schauen wir uns sogleich näher an:

```
cd ./
ls
⇒
...
```

Hoppla.. das sieht ziemlich genau so aus, wie vorher. Warum das ganze? Das Verzeichnis './' ist ein Sonderfall: './' ist immer das Verzeichnis, in dem wir uns gerade befinden. Dessen können wir uns auch noch einmal vergewissern:

```
pwd
⇒
/home/user ...das gleiche wie vorher.
```

Es gibt ein weiters "Sonderverzeichnis", das ist '../':

```
cd ../
pwd
⇒
/home
```

Wir sehen, dass wir nicht mehr im Verzeichnis '/home/user/' sind, sondern "nur" noch in '/home/'. Wir sind ein Verzeichnis **aufgestiegen**, denn '../' ist immer das **übergeordnete** Verzeichnis, des Verzeichnisses in dem wir uns gerade befinden.

```
ls
⇒
alice bob carl dave eve fiona user
```

Da wir uns in '/home' befinden, hat uns '`ls`' nun alle Dateien und Verzeichnisse unterhalb von '/home/' aufgelistet. wir sehen die Heimverzeichnisse aller Benutzer im System, darunter auch unser eigenes. Vielleicht ist unser Heimverzeichnis das einzige in der Liste, zum Beispiel an unserem Privatrechner, vielleicht existieren aber auch noch 20.000 weitere Benutzer im System, das könnte auf einem Universitätsrechner der Fall sein. Steigen wir noch ein Verzeichnis weiter auf:

```
cd ../
pwd
⇒
'/'
```

Der Pfad, in dem wir uns befinden wird jetzt nur noch durch einen Schrägstrich (oft englisch als "slash" bezeichnet) repräsentiert. Das heißt wir haben die Oberste Verzeichnisebene erreicht, das "Stamm-" oder "Wurzelverzeichnis" (englisch "root" - Wurzel). Wenn wir noch einmal '`cd ../`' eingeben, so wird dies keine Wirkung haben.

```
ls
⇒
bin    etc    media   proc   sys   var
```

```
boot   home   mnt     root   tmp
dev    lib    opt     sbin   usr
```

Was wir hier sehen ist eine Auflistung der Unterverzeichnisse (und -dateien) vom Stammverzeichnis. J nach Distribution wird sich die Liste etwas unterscheiden. Im wesentlichen ist sie aber nicht nur bei allen Linuxdistributionen sondern sogar bei allen Unix-Systemen gleich. Unabhangig davon ob wir z.B. vor einem BSD, Solaris oder MacOS X sitzen.

Mit '`ls`' können wir auch in die Verzeichnisse hinein gucken, ohne sie zu betreten:

`ls /bin/`

⇒

...

`ls /boot/`

⇒

...

`ls /dev/`

⇒

...

...

- **/bin** enthält ausführbare Programme, die für das System sehr grundlegend sind.
- **/boot** enthält den Systemkernel und einige Dateien, die damit unmittelbar in Zusammenhang stehen.
- **/dev** enthält Gerätedateien, das ist furchtbar interessant und wir werden in einem der nächsten Kapitel dazu kommen.
- **/etc** enthält systemweite Konfigurationsdateien.
- **/home** enthält die Heimverzeichnisse der Benutzer.
- **/lib** enthält sogenannte Softwarebibliotheken, die von Programmen eingebunden werden um auf bestimmte Funktionen zuzugreifen. Näheres hierzu gehört eher in einen Programmierkurs.
- **/media** ist in einigen Linuxdistributionen vorgesehen um die Verzeichnisbäume externer Laufwerke zu beherbergen.
- **/mnt** dient traditionell dem gleichen Zweck, ist allerdings älter.
- **/opt** wird in einigen Distributionen verwendet um Applikationen zu beherbergen die selbst eine Komplexe Verzeichnisstruktur mitbringen. Z.B. OpenOffice.org.
- **/proc** enthält laufende Prozesse (klingt komisch, ist aber so).
- **/root** ist das Heimverzeichnis des Benutzers root.
- **/sbin** enthält Programme zur Systemverwaltung.
- **/sys** enthält virtuelle Dateien, die den Status des Systems beschreiben.
- **/tmp** ist das systemweite Verzeichnis für temporäre Dateien.
- **/usr** enthält Daten für weniger grundlegende Programme wie z.B. die grafische Benutzeroberfläche.
- **/var** enthält Dateien, die sich im Normalbetrieb des Systems ohne Zutun des Benutzers verändern, z.B. die Systemlogbücher.

So wie wir in Verzeichnisse hinein schauen können, ohne sie zu betreten, können wir z.B. auch von unserem Heimverzeichnis aus in das Wurzelverzeichnis gucken:

`cd ~` (bringt uns immer in unser Heimverzeichnis)

`ls /`

⇒

...

Soso, der alleinstehende Schrägstrich bezeichnet also das Wurzelverzeichnis. Bleiben wir noch in unserem Heimverzeichnis, wir können alles von hier aus machen. Wir können wie gerade schon einmal getan wieder einen Blick ins Verzeichnis '/boot' werfen.

```
ls /boot/
```

⇒

...

Wie nicht anders zu erwarten sehen wir wieder den Inhalt vom '/boot'-Verzeichnis (mindestens einen Kernel und wahrscheinlich noch mehr). Die Verzeichnisangabe begann mit einem Schrägstrich ("slash"), das heißt, dass wir einen Verzeichnispfad ausgehend vom Stammverzeichnis angegeben haben. Solche Pfadangaben bezeichnen wir als **absolute Pfadangaben**, da sie immer vollkommen **eindeutig** sind.

Das Gegenteil von absoluten Pfadangaben sind **relative Pfadangaben**. Während aabsolute Pfadangaben immer vom Stammverzeichnis des Dateibaums ausgehen, geht eine relative Pfadangabe von dem Verzeichnis aus, in dem sich der Benutzer im Augenblick befindet. Wechseln wir zur Demonstaration in unser Heimverzeichnis:

```
cd ~
```

Hier können wir nun ein Unterverzeichnis anlegen, das geschieht mit dem Befehl '`mkdir`':

```
mkdir verz1
ls
```

⇒

... `verz1` ...

Wir sehen in der Verzeichnisausgabe des Heimverzeichnis unser frisch angelegtes unterverzeichnis 'verz1'. mit '`ls`' wollen wir uns dessen Inhalt ausgeben lassen:

```
ls verz1/
```

⇒

(keine Ausgabe)

Wir erhalten keine Ausgabe, das heißt das neu angelegte Verzeichnis ist leer (wie nicht anders zu erwarten). Wir haben uns nur den (nicht vorhandenen) Inhalt von 'verz1' anzeigen lassen, wir haben das Verzeichnis dabei **nicht** betreten.
Vorhin sind wir ein Verzeichnis aufgestiegen, wiederholen wir das einfach mal:

```
cd ../
```

Versuchen wir jetzt noch einmal den Inhalt vom gerade angelegten Verzeichnis anzeigen zu lassen, so wie wir es eben getan haben:

```
ls verz1/
```

⇒

```
ls: cannot access verz1: No such file or directory
```

"Kann auf verz1 nicht zugreifen: Keine solche Datei bzw. Verzeichnis". Das Verzeichnis 'verz1', existiert offenbar nicht. Wir haben es gerade angelegt und wir haben es seitdem nicht gelöscht, es gibt also keinen Grund, warum es nicht existieren sollte. Es existiert schlicht und ergreifend nicht an dem **Ort** zu dem wir uns hinbewegt haben.

Nochmal kurz zu absoluten und relativen Pfadangaben:

- **absolute Pfadangaben** beginnen **immer** mit einem Schrägstrich ('/'), dieser repräsentiert das Stammverzeichnis.
- **relative Pfadangaben** beginnen **niemals** mit einem Schrägstrich. Sie gehen immer von dem Verzeichnis aus, in dem sich der Benutzer im Augenblick befindet (siehe '`pwd`').

Immer wenn wir '`ls verz1/`' ausgeführt haben, haben wir uns also einer relativen Pfadangabe bedient. Wir können, von dem Ort aus, an dem wir uns befinden, das Verzeichnis über seinen absoluten Pfad ansprechen. Erinnern wir uns daran, dass wir im Heimverzeichnis waren, als wir 'verz1' angelegt haben. Dann muss der absolute Pfad von 'verz1' also (in diesem Beispiel) '/home/user/verz1/' sein:

```
ls /home/user/verz1/
```

⇒

(keine Ausgabe)

Es ging offenbar alles gut. Eine absolute Pfadangabe funktioniert unabhängig davon, wo wir uns befinden. Wir befinden uns (wie uns auch '`pwd`' verraten kann) im verzeichnis '/home/'. Von hier aus ist der **relative** Pfad zu 'verz1' folglich (in diesem Beispiel) 'user/verz1/'. `ls /home/user/verz1/`

⇒

(keine Ausgabe)

2.2 Zusammenfassung

Wir haben gelernt, wie wir uns im Dateisystem Bewegen, wie wir Ordner (gleichbedeutend Verzeichnisse) erstellen und wir haben einen flüchtigen Blick auf die in Unix üblichen Verzeichnisse (/bin, /boot, /etc ...) geworfen. Folgende Befehle haben wir verwendet:

- `pwd` - zeigt an, in welchem Verzeichnis wir uns aktuell befinden
- `cd` - bringt uns in ein angegebenes Verzeichnis

  `cd ./` bringt uns in das Verzeichnis, in dem wir uns gerade befinden (tut also genau nichts)

  `cd ../` bringt uns vom aktuellen Verzeichnis ein Verzeichnis nach oben (z.B. von '/home/user' nach '/home')

  `cd /` bringt uns ins Stammverzeichnis (egal von wo)

  `cd /var/spool/cron` bringt uns direkt ins Verzeichnis '/var/spool/cron'
- `mkdir` - erstellt ein Verzeichnis

  `mkdir foo` erstellt das Unterverzeichnis 'foo' im aktuellen Verzeichnis

  `mkdir /tmp/bar` erstellt das Unterverzeichnis 'bar' direkt unterhalb von '/tmp'

3 Tag 3

3.1 Arbeit mit Dateien, Verzeichnissen und Pfaden

An den vorherigen Tagen haben wir folgende Befehle Kennengelernt:

- **Tag 1**

  touch legt eine Datei an

  ls zeigt Dateien und Verzeichnissen an

  echo gibt einen Text aus

  cat gibt den Inhalt einer Datei aus

  chmod ändert den Zugriffsmodus (Zugriffsrechte) auf einer Datei

  groups zeigt an, in welchen Gruppen ein Benutzer Mitglied ist

  chgrp ändert die Gruppenzuweisung einer Datei

- **Tag 2**

  pwd um anzuzeigen, in welchem Verzeichnis wir aktuell arbeiten

  cd um uns in ein anderes Arbeitsverzeichnis zu bringen

  mkdir um ein Verzeichnis zu erstellen

Die Kommandozeile bietet noch sehr viel mehr Befehle, einige davon sollen hier vorgestellt werden.

- **rm** - "remove" (entfernen) löscht die angegebenen Dateien
- **rmdir** - "remove directory" (Verzeichnis entfernen) löscht die angegebenen (leeren) Verzeichnisse
- **cp** - "copy" (kopiere) kopiert die angegebenen Dateien in ein angegebenes Verzeichnis
  oder: kopiert eine angegebene Datei zu einer anderen Datei
- **mv** - "move" (verschiebe) verschiebt die angegebenen Dateien in ein angegebenes Verzeichnis
  oder: verschiebt eine angegebene Datei zu einer anderen Datei (d.h. **benennt die Datei um**)

Zur Übung legen wir ein paar Dateien an, und entfernen sie wieder:

```
touch foo bar qua
ls
⇒
bar foo qua ...

rm foo bar qua
ls
⇒
...
```

Das gleich jetzt mit Verzeichnissen:

```
mkdir quux baz
ls
⇒
baz quux ...

rmdir quux baz
ls
⇒
...
```

Das war einfach. Was passiert aber nun, wenn wir versuchen ein Verzeichnis zu löschen, in dem schon etwas drin ist?

```
mkdir test
touch test/datei
rmdir test
⇒
rmdir: failed to remove ‘test’: Directory not empty
```

"daran gescheitert 'test' zu entfernen: Verzeichnis nicht leer". Das zeigt uns, dass rmdir nur leere Verzeichnisse löschen kann. Was ist aber, wenn wir ein Verzeichnis, mit vielen Unterverzeichnissen und

-dateien entfernen wollen? Müssen wir dann jedes untergeordnete Objekt einzeln löschen? Naturlich nicht! Das Kommando '`rm`' bietet mit dem Parameter 'r' die Möglichkeit zum sogenannten "rekursiven" Löschen:

```
rm -r test
ls
⇒
...
```

Das Verzeichnis samt Inhalt wurde gelöscht und zwar unabhängig davon, wie komlex die untergeordnete Struktur war. Um das zu tun schaut `rm` in jedem Verzeichnis nach Unterverzeichnissen und dateien und in jedem Unterverzeichnis wiederum nach Unterverzeichnissen und Dateien um diese zu löschen. Ein solches Vorgehen bezeichnet man als "Rekursion".

Wir kenne bereits das Kommando '`ls`'. Dieses besitzt ebenfalls einen 'R'-Parameter zur "rekursiven Auflistung" von Verzeichnissen (hier ist es jedoch ein großes 'R', nicht klein 'r'):

```
ls -R /etc
⇒
...
```

Auf den Abdruck der Befehlsausgabe wird hier verständlicherweise wieder verzichtet. Der Befehl listet den gesamten Inhalt von '/etc' auf, danach den Inhalt aller Unterverzeichnisse und der Unterverzeichnisse davon, Eines nach dem Anderen.

Das kopieren von Dateien ist einfach:

```
mkdir Testverzeichnis
touch datei1 datei2 datei3
cp datei1 datei2 datei3 Testverzeichnis/
ls Testverzeichnis
⇒
datei1 datei2 datei3
```

An den Befehl `cp` wurden hier 4 Objekte übergeben. Das Programm hat einfach alle gelisteten Dateien in das Ziel kopiert. Als Kopierziel wurde das letzte ubergebene Objekt erkannt. Hätten wir als letztes eine Datei und kein Verzeichnis angegeben, so hätte '`cp`' sich darüber beschwert.

Der Befehl '`mv`' funktioniert genau so. Abgesehen davon natürlich, dass er Dateien verschiebt, statt sie zu kopieren.
→ selbst ausprobieren!

4 Tag 4

4.1 Programmaufrufe in der Shell

Wir haben bisher einige Befehle kennen gelernt, um mit Dateien umzugehen. Von unserer Kommandozeile aus haben wir externe Programme, wie z.B. '`touch`', '`ls`' und '`chmod`' aufgerufen. Das komplexeste Programm, das wir dabei benutzt haben, war allerdings die Kommandozeilenumgebung selbst.

Die Befehlszeile wird auch als "Shell" (Schale) bezeichnet, da sie gewissermaßen das Betriebssystem umschließt, und dem Benutzer präsentiert. Um die Arbeit der Shell zu verstehen müssen wir uns unsere bisherigen Befehlsaufrufe genauer angucken:

```
ls -l -h /home/user
```

Hier wird ein Programm aufgerufen, und es werden Parameter übergeben:

Programmname	**Parameter 1**	**Parameter 2**	**Parameter 3**
`ls`	`-l`	`-h`	`/home/user`

Die Shell ist dafür verantwortlich, das eingegebene Kommando anhand der Leerzeichen zu zerlegen, die Programmdatei aufzufinden, auszuführen und dabei die Parameter einzeln zu übergeben. Wie wir sehen handelt es sich bei Parameter 3 um eine Pfadangabe. Für die Shell ist dies zunächst egal (auch wenn sie uns z.T. besondere Unterstützung bei der Eingabe von Pfaden bietet). Die Shell übergibt 3 Parameter an das Programm, wie das Programm (in diesem fall '`ls`') diese Parameter interpretiert ist nicht mehr Sache der Shell.

Bei Parameter 1 ('-l') und Parameter 2 ('-h') handelt es sich offensichtlich nicht um Pfadangaben. Dies sind **Optionen** für das Programm. Auch hier ist die Interpretation der Optionen Sache des Programms. Die Shell übergibt die Parameter nur. Nehmen wir an, wir hätten beim Aufruf von '`ls`' die Kurzform für die Optionen verwendet:

```
ls -lh /home/user
```

Die Shell hätte das Programm auf folgende Weise aufgerufen:

Programmname	**Parameter 1**	**Parameter 2**
`ls`	`-lh`	`/home/user`

Wir sehen wieder, dass nun beide Optionen als ein Parameter übergeben werden: das Programm ist dafür verantwortlich die Bedeutung dieser verkürzten Optionskette zu erkennen. Nicht jedes Programm ist dazu in der Lage, Parameter in dieser Kurzzform zu erkennen. Generell ist das Identifizieren von Optionen (z.B. anhand eines vorangehenden Bindestrichs) Sache des Programms.

Ein noch komplexeres Beispiel für die Arbeit der Shell ist das am Tag 1 ausgeführte '`echo`':

```
echo Hallo Welt > datei1
```

Programmname	**Parameter 1**	**Parameter 2**	**Shellumleitung**	**Umleitungsziel**
`echo`	`Hallo`	`Welt`	`>`	`datei1`

Zwei Dinge sind hier interssant:

1. Die Shell betrachtet die Worte '`Hallo Welt`' als zwei unterschiedliche Parameter da sie ja durch ein Leerzeichen getrennt sind.
2. Die Umleitung der Ausgabe in die Datei 'datei1' wird von der **Shell** ausgeführt, **nicht vom Programm 'echo'**. Tatsächlich erfährt das Programm '`echo`' zu keinem Zeitpunkt, dass diese Umleitung überhaupt stattfindet. Der Umleitungspfeil und das Umleitungsziel werden **nicht** als Parameter an das Programm übergeben.

Diese zweite Aussage ist besonders wichtig und wir werden gleich nochmal darauf zurückkommen. Zunächst aber nochmal zur 1. Aussage: "Hallo" und "Welt" sind zwei verschiedene Parameter. Für die Shell ist es dabei egal, wie viele Leerzeichen vor, zwischen oder nach den Parametern stehen (sie muss die Worte nur unterscheiden können). Was passiert dann aber, wenn wir z.B. "`Hallo  Welt`" (mit 10 Leerzeichen) ausgeben wollen?

```
echo Hallo          Welt
```

Programmname	**Parameter 1**	**Parameter 2**
`echo`	`Hallo`	`Welt`

Ausgabe ⇒
```
Hallo Welt
```

Das Programm erfährt nie, dass diese Leerzeichen existieren. Bei beiden Aufrufvarianten wurde an '`echo`' nie ein Leerzeichen übergeben (aus Sicht von '`echo`' waren die beiden letzten Aufrufe sogar identisch). Das Programm hat nur deswegen ein Leerzeichen mit ausgegeben, weil es sich "denken" konnte (bzw. weil sich sein Entwickler denken konnte), dass zwischen den beiden Parametern bei der Eingabe ein Leerzeichen stehen musste. Wie übergeben wir die Leerzeichen mit?
```
echo 'Hallo          Welt'
```

Programmname	**Parameter 1**	
`echo`	`Hallo`	`Welt`

Ausgabe ⇒
```
Hallo          Welt
```

- Wir haben die Zeichenkette hier bei der Befehlseingabe in einfache Anführungsstriche eingeschlossen.
- Das Programm '`echo`' erhält nur einen einzigen Parameter
- Der Parameter enthält **genau das**, was in den Anführungszeichen eingeschlossen ist, dies kann sogar Zeilenumbrüche mit einschließen (→ Ausprobieren)
- Die Anführungszeichen selbst sind **nicht** Teil des übergeben Parameters!

Aus diesem letzten Punkt entsteht übrigens wieder ein Problem: was, wenn wir die Zeichenkette "`'Hallo Welt'`" (mit Anführungszeichen) ausgeben wollen?
```
echo \'Hallo Welt\'
```

Programmname	**Parameter 1**	**Parameter 2**
`echo`	`'Hallo`	`Welt'`

Ausgabe ⇒
```
'Hallo Welt'
```

Wir haben jedem Anführungszeichen einfach einen Rückschrägstrich (Backslash) vorangestellt. Ein Backslash macht der Shell klar, dass sie das jeweils nachfolgende Zeichen nicht selbst interpretieren, sondern einfach ans Programm "durchreichen" soll. Man bezeichnet den Backslash in dieser Funktion als "Escapecharacter" oder deutsch "Fluchtzeichen" (umgangssprachlich oft "Escapezeichen") da ein Zeichen damit der Interpretation durch die Shell "entflieht". das Fluchtzeichen selbst wirrd nicht ans Programm ubergeben. Wenn wir einen Backslash selbst übergeben wollen, "escapen" wir diesen übrigens einfach auch mit einem Backslash:
```
echo \\
⇒
\
```

5 Tag 5

5.1 Shellumleitungen

Wir erwähnten vorher die Shellumleitung in Form eines Pfeils. Für uns sind zunächst zwei Arten von Shellumleitungen interessant:

- die Umleitung von Programmausgaben in eine Datei (`>` und `>>`)
- die Umleitung von Programmausgaben an ein weiteres Programm (`|`)

5.1.1 Ausgabeumleitung in Dateien

Die Umleitung in eine Datei ist uns bereits bekannt. Wir haben sie wiederholt beim Aufruf von '`echo`' durchgeführt.

```
echo Hallo Welt > datei1
```

Da die Umleitung von der Shell ausgeführt wird, ist sie vollkommen Unabhängig vom Programm. Wir können damit also die Ausgabe von jedem beliebigen Programm in eine Datei leiten:

```
ls > datei1
cat datei1
⇒
... (Verzeichnisauflistung)
```

Angenommen, wir tun folgendes:

```
ls > datei1
echo Hallo Welt > datei1
cat datei1
⇒
Hallo Welt
```

Die Ausgabe von '`ls`' ist nicht mehr in "datei1" enthalten, denn die Datei wurde bei der zweiten Ausgabenumleitung (hinter '`echo`') überschrieben. Mit dem Doppelpfeil können wir eine umgeleitete Ausgabe an die Datei anhängen:

```
ls > datei1
echo Hallo Welt >> datei1
cat datei1
⇒
... (Verzeichnisauflistung)
Hallo Welt
```

5.1.2 Ausgabeumleitung an Programme

Sehen wir uns zunächst zwei weitere Unix-Befehle an, mit denen diese Art der Umleitung sinnvoll ist:

- **wc** - "word count" ("Wörterzahl") gibt die Anzahl von Zeilen, Wörtern und Zeichen in einer angegebenen Datei aus
- **tr** - "translate" (übersetze) ersetzt Zeichen in einem Datenstrom durch andere.

Das Kommando '`wc`' kann auf Dateien angesetzt werden, um darin enthaltene Zeilen, Wörter und Zeichen zu zählen:

```
echo Hallo Welt > datei1
echo Was geht ab? >> datei1
wc datei1
⇒
 2  5 24 datei1
```

Die Datei hat demnach 2 Zeilen, 5 Wörter und insgesamt 24 Zeichen. Was ist nun, wenn wir eine solche Statistik z.B. für die Ausgabe von '`ls`' haben wollen? Das Programm '`wc`' ist in der Lage, seinen Text nicht nur aus einer Datei, sondern auch aus einer Umleitung zu lesen:

```
ls |wc
⇒
     72      89     992
```

Die angezeigten Zahlenwerte hängen natürlich davon ab, welches Verzeichnis wir uns von '`ls`' ausgeben lassen. In diesem Fall haben wir 72 Zeilen ('`ls`' listet genau 1 Objekt pro Zeile), 89 Wörter (einige

Dateinamen hier enthalten also offenbar Leerzeichen) und insgesamt 992 Zeichen. Gucken wir uns die Zeile nochmal genauer an:

Programmname	Shellumleitung	Programmname
`ls`	`|`	`wc`

Wir haben also zwei Programme gleichzeitig ausgeführt und dabei die Programmausgabe des ersten Programms ('`ls`') zur Eingabe des zweiten Programms ('`wc`') gemacht.

Einige Programme sind sogar darauf ausgelegt, nur auf diese Weise zu arbeiten, und gar nicht mit Dateien umzugehen. ein solches Programm ist '`tr`':

```
echo Der Esel lauert im Teich. |tr s g
⇒
Der Egel lauert im Teich.
```

Das Programm '`tr`' hat aus unserem Esel einen Egel gemacht. Betrachten wir kurz den Informationsfluss, der dabei Stattgefunden hat. Jedes Programm hat beim Start mindestens 3 Kanäle zum Austausch von Informationen:

- **stdin** "Standard Input" ist der Kanal, auf dem Ein Programm Informationen liest.
- **stdout** "Standard Output" ist der Kanal, auf dem Ein Programm Informationen ausgibt.
- **stderr** "Standard Error" ist der Kanal, auf dem Ein Programm Fehlerausgaben schreibt.

Normalerweise ist 'stdin' eine Eingabe, die von der Tastatur gelesen wird, während 'stdout' und 'stderr' auf dem Terminal angezeigt werden. Die Shell ist in der Lage, diese Kanäle "umzubiegen". Betrachten wir den Befehl von eben noch einmal genau:

stdin von echo	Programm	Param1	Param2	Param3	Param4	Param5	stdout von echo
(nichts)	`echo`	`Der`	`Esel`	`lauert`	`im`	`Teich.`	`Der Esel lauert im Teich.`

Umleitung
`|`

stdin von tr	Programm	Param1	Param2	stdout von tr
`Der Esel lauert im Teich.`	`tr`	`s`	`g`	`Der Egel lauert im Teich.`

'`echo`' erhält keinerlei Daten auf seiner Standard**ein**gabe. Es generiert aber aufgrund seiner Parameter Daten auf seiner Standard**aus**gabe. Die Shell leitet diese ausgegebenen Daten in die Standard**ein**gabe von '`tr`'. Bei diesem Vorgang geschieht keine Aufbereitung der Daten. Die Shell versucht **nicht** diesen Datenstrom in irgend einer Weise zu interpretieren, wie sie es mit eingegebenen Kommandozeilen tut. '`tr`' erhält den Datenstrom auf seiner Standard**ein**gabe und führt eine Verarbeitung aus. '`tr` verarbeitet die Daten auf Basis seiner Aufrufparameter und gibt dann den verarbeiteten Datenstrom auf seiner Standard**aus**gabe aus. Da hier keine weitere Umleitung stattfindet, erscheint der Satz auf dem Terminal.

Ein Programm kann seine Standardkanäle nach dem Start schließen oder weitere Kanäle öffnen. Dateien sind z.B. solche Kanäle. Wenn wir beispielsweise an '`wc`' eine Datei übergeben, so wird das Programm nicht mehr von der Standardeingabe sondern aus dieser Datei lesen. Alle Daten, die es auf der Standardeingabe erhält gehen dabei gewissermaßen verloren:

```
echo Hallo Welt > datei1
ls | wc datei1
⇒
 1  2 11 datei1
```

Offensichtlich hat '`wc`' hier die Statistiken für 'datei1' ausgegeben. Die Standardeingabe, die Daten also, die '`ls`' generiert hat wurden verschluckt. Das liegt am Programmspezifischen verhalten von '`wc`'.