Unix Tutorial

Letzte Änderung war Dez. 2018. Tutorial entstand als Teil der Adminenfibel, by Princess.
Ich gehe hier vor "was brauche ich zuerst". Eine strukturierte Anleitung zur Bash findet sich z.B. unter https://www.linuxwiki.de/Bash

Arbeiten mit der Shell / wie bekomme ich Hilfe zu Kommandos

Für die remote administration eines Unix/Linux-Servers ist es unerläßlich, die Kommandozeile (Shell) zu beherrschen, weil eine grafische Nutzeroberfläche (GUI) meist nicht zur Verfügung steht.

Die **Standard-Shell unter Linux ist die bash** (Bourne-Again-Shell), die sich aus der Bourne Shell (sh) entwickelt hat.

Die Admine loggt per ssh auf many.haecksen.org mit ihrem Benutzeraccount (nicht gleich als root) ein. Idealerweise ist ein ssh-Schlüssel dort hinterlegt, so daß die Passworteingabe entfällt. Gibt es keinen ssh-Schlüssel, wird das Passwort abgefragt. Dieses Verhalten ist im sshd (dazu später) konfiguriert.

Einmal auf der Shell "gelandet", ist es gut, zu wissen, daß es drei Arten von Kommandos gibt. Gibt man einen Befehl ein, so wird in folgender Reihenfolge nach dem Kommando gesucht:

- ist es ein **Shell-Alias** (also ein meist selbst konfigurierte Befehlsabkürzung oder -Abfolge)
- ist das Kommando ein **Shell-Builtin**
- oder ist das **Kommando ein Programm "im Pfad"**.

Je nachdem unterscheidet sich, wie man sich Hilfe holt:

- Bei eigenen Aliasen gibt es keine Hilfe, es sei denn, diese wurde explizit implementiert (was sich bei eigendefinierten Abkürzungen meinst nicht lohnt)
- Bei Shell-Builtins bekommt man Hilfe mit **"help kommando"**
- Bei normalen Kommandos im Pfad gibt es in der Regel eine Manpage: **"man kommando"**

type / wie finde ich heraus, was ich denn für ein Kommando vor mir habe

Um herauszufinden, was wir vor uns haben, tippen wir:

```
type kommandoname
```

oder auch

```
type -a kommandoname
```

um alle Vorkommen eines Kommandos anzuzeigen.

Beispiel:

```
type cd
cd is a shell builtin
```

Hier bekommen wir also Hilfe mit

```
help cd
```

Beispiel 2:

```
type -a ls
```

ergibt bei mir (Princess):

```
ls is aliased to `ls -F'
ls is /bin/ls
```

Das erste ls ist also von mir selber definiert, keine Hilfe. Das zweite ist aber ein Kommando "im Pfad", also mit Angabe eines Verzeichnisses (/bin). Hier erhalte ich also Hilfe mit

```
man ls
```

Aufgabe: diese Seite anlesen. "durch" ist sportlich.

Der Pfad / die PATH Variable

Was aber hat es genau mit "dem Pfad" auf sich: **$PATH** ist eine **Shell-Variable**. Anzeigen läßt man sich den Inhalt mit:

```
echo $PATH
```

darauf folgt eine Angabe wie z.B. diese:

```
/home/princess/bin:/client/bin:/client/sbin:/usr/local/bin:/bin:/usr/bin:/sbin:/usr/sbin:/usr/X11R6/bin:.
```

Wir sehen hier eine **Abfolge von Verzeichnissen, getrennt mit dem Doppelpunkt**. In dieser Reihenfolge wird in den Verzeichnissen nach dem gerade eingegebenen Kommando gesucht, wenn es eben nicht von der Shell kommt. Wichtiger Hinweis: am Ende befindet sich der Punkt, also das aktuelle Verzeichnis. Dieses ist in einer root-Shell in der Regel NICHT gesetzt, weil es das Ausführen von untergeschobenen Kommandos erleichtern würde.

Festgelegt wird diese Variable entweder im systemweiten Konfigurationsfile /etc/bash.bashrc oder im Konfigurationsfile im Homeverzeichnis des Nutzers.

Das Homeverzeichnis des Nutzers liegt auch in einer Shell-Variable: $HOME.

Aufgabe: gib dieses aus.

In einem eigenen Verzeichnis /home/princess/bin kann ich eigene Kommandos und Skripte ablegen.

Die Konfigurationsfiles von Kommandos

Typischerweise gibt es zu jedem Unix Kommando ein Konfigurationsfile, das idR. auf "rc" für "resource config" endet. Das systemweite File der bash ist oben erwähnt. Die eigene Konfiguration im Home-Verzeichnis beginnt mit dem Punkt und endet auf rc, also in dem Fall: .bashrc . Der komplette Pfad der Datei lautet also:

```
/home/username/.bashrc
```

Ebenso gibt es z.B. für das Mailprogramm mutt das systemweite /etc/Muttrc und das .muttrc im Home, oder für den Editor vi das systemweite /etc/vim/vimrc und das .vimrc im Home.

Bewegen in Verzeichnissen: cd

Da wir ja schon wissen, daß dies ein Shell-Builtin ist, hole Dir Hilfe zu diesem Kommando.

cd ohne weitere Argumente wechselt in das Home-Verzeichnis des Nutzers. Ein Verzeichnis im Baum "höher" erreicht man mit **"cd .."**. Für das "Springen" in Verzeichnisse gibt es grundsätzlich zwei Wege:

- **absolute** Adressierung mit vollem Pfad
- **relative** Adressierung vom aktuellen Verzeichnis aus

Das Wurzelverzeichnis eines jeden Unix heißt "/" (vorwärtsgerichteter Schrägstrich, gesprochen "root"). Unterverzeichnisse des Systems liegen darunter:

```
bin   dev  home       lib    lost+found  mnt  proc  run   srv  tmp  var
boot  etc  initrd.img  lib64  media       opt  root  sbin  sys  usr  vmlinuz
```

Aufgabe: wie kommt man mit ls zu diesem listing?

Die absolute Adressierung ist unabhängig von dem Verzeichnis, in dem ich mich aktuell befinde. Nach dem Login bin ich in meinem Home, also /home/username. Möchte ich mir eine Datei anzeigen lassen, z.B. die Systemweite COnfig-Datei vom vim, geht das also mit

```
less /etc/vim/vimrc
```

Wenn ich relativ adressiere, ist mein Ausgangspunkt relevant. Das aktuelle Verzeichnis lasse ich mir ausgeben mit **"pwd"** für "print working directory", falls ich unsicher bin, wo ich mich befinde.

Möchte ich nun mein Verzeichnis bin wechseln, reicht:

```
cd bin
```

Möchte ich in das Verzeichnis eines anderen Users gehen (soweit Rechte gesetzt sind), ginge das mit:

```
cd  ../user2
```

also ein Verzeichnis hoch und dann der Name des anderen Users.

Aufgabe: wie hieße der absolute Pfad zum anderen user? Zum ersten Beispiel: wie hieße der relative Pfad zum vimrc vom Homeverzeichnis aus?

less / der Pager / Dateien ansehen

Bei den Konfigurationsfiles von Programmen handelt es sich um reine ASCII-(Text)Dateien, die man mit einem Pager ansehen kann. Das ursprüngliche Unix-Kommando dafür hieß "more". Unter Linux hat sich aber das in seinen Möglichkeiten umfangreichere **"less"** durchgesetzt.

Man kann nun

- in das Verzeichnis wechseln und mit **"less dateiname"** eine Datei anzeigen oder
- den vollen Pfad der Datei beim less angeben

Aufgabe: versuche beides für /etc/vim/vimrc .

Was aber passiert, wenn ich versehentlich eine Binärdatei ausgeben lassen will? **Aufgabe:** Teste es mit /bin/ls .

Natürlich kann ich vor dem Ansehen oder dem Versuch, eine Datei zu editieren (auch das geht nur mit Textdateien) herausfinden, was für eine Datei ich vor mir habe. Der Dateityp hängt unter Linux, anders als unter Windows NICHT an der Endung der Datei! Man findet den Dateityp heraus mit

```
file dateiname
```

Auch hier kann relativ oder absolut adressiert werden.

Aufgabe: führe das file-Kommando aus auf /etc/vim/vimrc, /bin/ls, eine Bilddatei, ein libreoffice-Dokument, ein PDF File....

---

mkdir: Verzeichnisse anlegen

Um sein Homeverzeichnis ein wenig zu strukturieren, braucht man **Verzeichnisse**.

Sinnvoll sind das bisher erwähnte Verzeichnis bin (Groß- und Kleinschreibung sind signifikant unter Unix!), sowie ein Verzeichnis tmp.

Diese kann man mit

```
mkdir name
```

anlegen.

Möchte man gleich mehrere Unterverzeichnisse mit anlegen, gibt es die Option -p:

```
mkdir -p Projekte/Adminen
```

Aufgabe: lege ein paar mögliche Projektverzeichnisse in Deinem $HOME an. Hier kann man Namen recht frei gestalten, bei den Systemverzeichnissen (/etc, /usr, /bin, /sbin...) sollte man tunlichst nichts an den Namen ändern.

Namenswahl: welche Zeichen in Namen sind erlaubt?

Im Prinzip: alle außer ASCII Null. Auch Leerzeichen in Dateinamen sind möglich, da man auf der Kommandozeile aber das Leerzeichen braucht, um bei Kommandos Optionen und Argumente zu trennen, **rate ich von Leerzeichen in Dateinamen ab**, außer man möchte sich viel und oft die Finger brechen. (Leer- und andere Sonderzeichen können aber durch \ (backslash) maskiert werden, das macht auch die bash bei der automatischen Dateinamenergänzung mit <tab><tab>).

Unkritische Zeichen sind: Buchstaben a-z-A-z, Umlaute sollten vermieden werden, Punkt, Bindestrich, Unterstrich. Nur als erstes Zeichen sollte der Bindestrich (Minus) vermieden werden, weil damit Optionen zu Kommandos eingeleitet werden und man dann Mühe hat, solche Dateien wieder zu löschen.

Als hilfreich erwiesen haben sich auch Datummstempel für eine Art "Versionierung für Arme": man schreibt das Datum statt in der Form **DD.MM.JJJJ** (was zwar menschenlesbar ist, aber für eine Sortierung mit ls nicht geeignet) in der Form **JJJJMMDD**, evtl. noch mit Zeit: JJJJMMDDSSMMSS, also z.B. 20181206 für Nikolausi 2018.

Bearbeitet man Dateien und möchte im Fehlerfall auf eine frühere Version zurückgreifen können, kopiert man die Datei vor dem Editieren auf eine Version mit Datumsstempel, z.B.

```
cp .bashrc .bashrc.20181206
```

Hat man mehrere Versionen, listet ls diese dann chronologisch, weil es Integer sortiert.

Übrigens kann man **lange, sprechende Namen** wählen: **die bash ergänzt** Namen bei Eindeutigkeit, wenn man die **Tabulator-Taste** verwendet. <tab><tab> listet alle Namen bei Nicht-Eindeutigkeit. Ohne zuvor eingegebenes Kommando listet <tab><tab> alle zur Verfügung

stehenden Befehle, fragt aber bei einer größeren Anzahl vorher nach, ob man wirklich alles gelistet haben möchte oder eher ein paar mehr Buchstaben zur Eindeutigkeit angibt).

---

rm: Dateien wieder löschen

Gelegentlich möchte man auch aufräumen und Dateien wieder löschen. Dazu gibt es

```
rm dateiname
```

In dieser Form wird die Datei dann ohne Nachfragen sofort gelöscht. Es empfiehlt sich, ein Alias (s.u.) rmi zu definieren für rm -i. Insbesondere, wenn man mehrere Dateien löschen möchte, wird dann bei jeder nachgefragt und man kann mit y oder n antworten.

Das Gegenteil bewirkt -f für force.

Leere Verzeichnisse löscht man mit

```
rmdir verzeichnisname
```

Sind die Verzeichnisse nicht leer, sollen aber gelöscht werden, verwendet man

```
rm -rf
```

VORSICHT ist geboten, hier sollte sicherheitshalber immer das Verzeichnis, in dem man sich befindet, überprüft werden (mit pwd) oder man läßt sich das aktuelle Verzeichnis immer im Prompt anzeigen (s.u., Einstellungen).

Der Editor: warum man vi(m) können muß

Eigentlich ist Unix/Linux ein Betriebssystem wo sich vieles nach gusto einstellen läßt und jeder sich die Umgebung so einrichten kann, wie es ihr beliebt.

Einige Systemkommandos jedoch rufen standardmäßig den vim als Editor auf. vim ist die neuere, besser zu bediendende Version des vi. Es lohnt sich also, diesen immer verfügbaren Editor als Standard zu erlernen.

Anleitungen finden sich zuhauf, z.B. unter https://www.selflinux.org/selflinux/html/vim.html

vi ist also idR. ein alias auf vim (**Aufgabe:** teste dies mit den bisher gelernten Kommandos!)

An dieser Stelle aber in aller Kürze: es gibt drei Modi des vi, zwischen denen man zu wechseln verstanden haben sollte.

Der vi wird gestartet mit

```
vi dateiname
```

sodann befindet man sich im **Bewegungsmodus**. Man KANN sich im vim zwar mit den Pfeiltasten im Text bewegen, jedoch empfiehlt es sich, die Tasten

```
hjkl
```

für

```
links, unten, oben, rechts
```

zu verwenden, weil diese, kombiniert mit einer vorangestellten Ziffer, das schnelle Springen im Text ermöglichen. Dieser Editor wurde für alte, langsame Terminals entwickelt und ist daher sehr effizient.

Ist man an die Stelle gesprungen, an der man etwas editieren will, kann man in den **Editiermodus** wechseln:

```
i   füge vor dem aktuellen Zeichen ein (insert)
I   füge am Anfang der Zeile ein
a   füge nach dem aktuellen Zeichen ein (append)
A   füge am Ende der Zeile ein
o   eröffne eine neue Zeile unterhalb der aktuellen
O   eröffne eine neue Zeile oberhalb der aktuellen
```

sodann kann Text eingefügt werden.

Löschen:

```
x     löscht das aktuelle Zeichen
dd    löscht die aktuelle Zeile
```

Aufgabe: lese in der Anleitung, die wortweise gesprungen bzw. gelöscht wird. Auch: wie löscht man mehrere Zeilen auf einmal?

Der **Editiermodus** wird beendet mit

```
ESC
```

und man ist wieder im **Bewegungsmodus**.

Zum Speichern (oder Einlesen von Dateien oder suchen und ersetzen) braucht man den Kommandomodus. In diesen wechselt man vom Bewegungsmodus mit : (Doppelpunkt). Darauf folgen Kommandos

```
:w        speichere die Datei ab
:w name   speichere die Datei unter einem neuen Namen ab
:r datei  lese eine Datei ein
:r !kommando lese den Output eines Kommandos ein, z.b. :r !date ist gut für Kommentare
:wq       speichern und verlassen (geht auch mit ZZ)
:q!       verlassen ohne Speichern
```

Gemütlich machen: was beim Login eingestellt werden kann und was das Arbeiten erleichtert

Jetzt, wo wir zumindest in groben Zügen einen Editor beherschen, können wir uns in den Konfigurationsfiles zur bash Einstellungen bei Variablen und Aliasen machen, die das Arbeiten erleichtern können.

Aufgabe: wie hieß nochmal das Konfiguratiosfile der bash?

Variablen setzen beim Login

Bei der bash gibt es eigentlich zwei Files, die ausgeführt werden: .bash_login für Loginshells und .bashrc für andere. Da diese Unterscheidung aber fürs tägliche Arbeiten irrelevant ist, machen wir da keinen Unterschied und schreiben alles, was wir brauchen ins .bashrc. Ins .bash_login schreiben wir nur:

```
source ~/.bashrc
```

damit das .bashrc beim login ausgeführt wird. Selbstverständlich wird vor den eigenen Files das systemweite Configfile ausgeführt, in diesem Fall ist es das /etc/profile (aus historischen Gründen) und /etc/bash.bashrc.

Wir wollen nun Variablen setzen (wie man sie abfragt s.o.!).

Eine Variable wird gesetzt mit

```
NAME=inhalt
```

Sie ist dann aber nur im aktuellen Prozeß (dazu später) verfügbar. Damit sie auch in allen Kindprozessen gilt, brauchen wir hier immer:

```
export NAME=inhalt
```

Dies gilt für die Kommandozeile wie für das .bashrc.

Wichtig: alles, was wir ins .bashrc schreiben, wird erst beim nächsten login ausgeführt. Um das **.bashrc neu einzulesen**, führt man auf der Kommandozeile

```
source .bashrc
```

aus.

Das TMP-Dir

Viele Programme brauchen ein temporäres Verzeichnis, um Dinge ab- oder zwischenspeichern zu können, daher setzen wir zwei Variablen:

```
export TMPDIR=/home/username/tmp
export TMP=/home/username/tmp
```

Stelle sicher, daß das Verzeichnis auch angelegt ist.

Der Prompt / die Eingabeaufforderung

Hier kann man durch geschickte Einstellung sich das Arbeiten sehr erleichtern. Der Prompt wird in der Variable PS1 festgelegt (ja es gibt auch PS2, dies kommt bei einfachen Shellscripten auf der Kommandozeile zum Tragen).

Aufgabe: gib den Inhalt von PS1 aus.

Im Prompt kann man username, hostname, aktuelles Verzeichnis und eine laufende Nummer in der bash-history angeben, daher ist eine sinnvolle Einstellung:

```
export PS1="\u @ \h:\w \!> "
```

Häufig will man sich aber anzeigen lassen, wenn man mit root-Rechten unterwegs ist, daher verwendet man lieber:

```
if [ -w /etc/passwd ]; then
  PS1="\h:\w \!# "
else
```

```
  PS1="\u @ \h:\w \!> "
fi
export PS1
```

dann wird als root das Doppelkreuz # statt > angezeigt.

Aufgabe: schreibe das erste Beispiel auf der Shell und sieh, was sich sofort ändert. Schreibe das zweite Beispiel ins .bashrc. Verwende gerne Kommentare zu Deinen Eintragungen: kennzeichne diese mit vorangestellten # (vor jeder Kommentarzeile).

Nimm gerne weitere Änderungen anhand der Anleitung zur bash vor.

Die Editor-Variable

Gibt den Standard-Editor an. Jede Userin kann diesen für ihre eigene Umgebung selber wählen, aber spätestens als root wird es immer der vi. Daher stellen wir ein:

```
EDITOR=/usr/bin/vi;          export EDITOR
VISUAL=/usr/bin/vi;          export VISUAL
```

Hier neu: mit dem Semikolon kann man mehrere Kommandos in einer Zeile schreiben und trennen.

Die Bash-History

Die bash merkt sich die letzten N Kommandos, wenn man es denn einstellt:

```
HISTSIZE=200;                export HISTSIZE
```

Nach Beenden der Shell werden die Kommandos aus dem Speicher in das File .bash_history geschrieben.

Möchte man nicht, daß ein Kommando in der History landet, schreibt man ein Leerzeichen vor das entsprechende Kommando.

Um Dubletten zu vermeiden, setzen wir außerdem:

```
history_control=ignoredups;              export history_control
set command_oriented_history;            export command_oriented_history;
```

Hacker setzen gern die HISTSIZE=0, damit nicht mitprotokolliert wird, was sie tun.

Aufgabe: da das History File erst beim Beenden der Shell geschrieben wird, was muss ein Hacker genau tun, um das zu verhindern oder seine Spuren zu verwischen?

Die Path-Variable

..hatten wir schon behandelt. Als root möchte man das aktuelle Verzeichnis (".") nicht dabeihaben, deswegen setzen wir:

```
if [ -w /etc/passwd ]; then
   PATH="$HOME/bin:/usr/local/bin:/bin:/usr/bin:/sbin:/usr/sbin:/usr/X11R6/bin"
 else
   PATH="$HOME/bin:/usr/local/bin:/bin:/usr/bin:/sbin:/usr/sbin:/usr/X11R6/bin:."
fi
```

Der Manpath

Ähnlich wie die Pfad-Variable legen wir noch die Pfade fest, wo nach manpages gesucht wird:

```
MANPATH=/usr/man:/usr/local/man:/usr/X11R6/man:/usr/openwin/man:/usr/share/man:/usr/share/catman:/usr/ca
tman;       export MANPATH
```

Aufpassen: alles muss in eine Zeile! Kein Umbruch!

Diverse kleine Dinge

```
export PAGER=less
```

(nur dass das klar ist :))

```
export LC_COLLATE=C
```

bestimmt die Sortierreihenfolge bei ls: zuerst "unsichtbare Dateien". Diese beginnen mit einem Punkt und werden nur bei ls -a angezeigt, sodann alphabetisch sortiert. Erst dann kommen die regulären Dateien, die mit Buchstaben oder Zahlen beginnen in der Reihenfolge: Ziffern, Großbuchstaben, Kleinbuchstaben (dies ist eine Empfehlung).

Sinnvolle Aliase

IdR. gilt: eigene Abkürzungen und Aliase NICHT so nennen wie das Ursprungskommando. Allerdings gibt es Ausnahmen, die auch nicht dramatisch sind: less ist z.B. nur gut nutzbar, wenn der Inhalt der ausgegebenen Datei nicht beim Beenden von less verschwindet.

Daher:

```
alias less='less -MeiQX'
alias les=less
```

ebenso:

```
alias df="df -h"
```

Aufgabe: finde heraus, was die Optionen im Einzelnen bedeuten

Als weiterhin sinnvoll haben sich ergeben:

```
alias 0="sudo $SHELL"
alias    ls='ls -F'
alias    ..='cd ..'
alias pstree='pstree -A -u'
alias rmi="rm -i"
```

Wieder zu allen Kommandos die **Aufgabe:** finde heraus, was sie tun und was die Optionen heißen.

Funktionen

Auch einfache Funktionen können definiert werden:

```
function ll     () { ls -alFh $* | less -MeiQ ;}
function l      () { echo $1* ;}
```

Über Prozesse sprechen wir noch, und auch über grep:

```
function psg    ()
{
  echo 'UID PID PPID C STIME TTY TIME CMD'
  ps -ef | grep $* | grep -v grep
}
```

Der Phantasie sind kaum Grenzen gesetzt und für komplizierte Kommandos kann man sich immer Abkürzungen bauen.

Das .vimrc

Auf älteren Unices benutzt man hier .exrc, also falls Ihr mal das .vimrc nicht vorfindet.

Auch hier sind mannigfaltige Einstellungen möglich! Startet den vi und ruft

```
:set
```

auf, da wird schon eine Menge angezeigt. Ich erachte als sinnvoll:

```
set autoindent       # Einrückungen in der nexten Zeile übernehmen, fürs Programmieren
set wrapmargin=8     # 8 Zeichen vor Zeilenende in die nächste gehen
set textwidth=72     # Text nicht breiter als 72 Zeichen machen, das ist der kleinste gem. Nenner
```

Braucht man einmal Zeilen, die länger sind und weil man Textpassagen mit Einrückungen hereinpasten möchte, setzt man im laufenden Editor:

```
:set paste
```

das überschreibt die Einstellungen mit Zeilenumbrüchen.

Zurückgesetzt wird es mit:

```
:set nopaste
```

Was aber, wenn ich einen Zeilenumbruch habe und den gar nicht will oder der da gar nicht sein darf (s.o.)? Ich gehe in den Bewegungsmodus (ESC) und gehe auf die obere Zeile. Mit J (groß J) füge ich die untere Zeile mit der oberen zusammen.

Weitere wichtige Kommandos und Abläufe

Die Ein- und Ausgabekanäle

Kommandos geben oft regulären Output aus, aber auch Fehlermeldungen. Beim Starten eines Kommandos in der Shell, werden diese angezeigt. Oft ist es aber gewünscht, jeglichen Output umzulenken, z.B. wenn ein Kommando per cron (zeitgesteuert) und ohne Terminal (!) aufgerufen wird. Auch jegliche daemons kommen ohne Terminal aus, müssen aber auch mit ihren Ausgaben irgendwohin.

Ebenso kann der Output eines Kommandos als Input des nächsten Kommandos verwendet werden, oder eine Tastatureingabe dient als Input.

Wir unterscheiden also drei Kanäle:

0	Die Standardeingabe	STDIN
1	Der Standardoutput	STDOUT
2	Der Standarderror	STDERR

Ausgaben umleiten

Sagen wir, ich setze den Befehl ab:

```
ls bahnf*
```

Und erhalte als regulären Output:

```
bahnfahrten2004  bahnfahrten2007  bahnfahrten2010  bahnfahrten2013  bahnfahrten2016
bahnfahrten2005  bahnfahrten2008  bahnfahrten2011  bahnfahrten2014  bahnfahrten2017
bahnfahrten2006  bahnfahrten2009  bahnfahrten2012  bahnfahrten2015  bahnfahrten2018
```

Nun möchte ich das Ergebnis aber nicht in der Shell angezeigt haben, sondern, daß es in eine Datei landet. Ich lenke also die Ausgabe um:

```
ls bahnf* > allemeinebahnfahrten
```

Auf dem Terminal erscheint nun nichts mehr, nur der Prompt.

Allerdings können ja auch Fehler entstehen, gebe ich ein:

```
ls bahnf* nase*
```

erhalte ich:

```
ls: cannot access 'nase*': No such file or directory
bahnfahrten2004  bahnfahrten2007  bahnfahrten2010  bahnfahrten2013  bahnfahrten2016
bahnfahrten2005  bahnfahrten2008  bahnfahrten2011  bahnfahrten2014  bahnfahrten2017
bahnfahrten2006  bahnfahrten2009  bahnfahrten2012  bahnfahrten2015  bahnfahrten2018
```

also zusätzlich eine Fehlermeldung.

Leite ich nun den Output um:

```
ls bahnf* nase* > blubb
```

kommt nur noch auf dem Terminal:

```
ls: cannot access 'nase*': No such file or directory
```

Hier sieht man sehr deutlich, daß Standardoutput und Standarderror verschiedene Kanäle sind.

Was aber, wenn ich den Error-Output auch nicht im Terminal brauchen kann? Ich leite erneut um:

```
ls bahnf* nase* > blubb 2> bla
```

Das bedeutet: lenke den Standarderror in die Datei bla.

Möchte beide Kanäle in der gleichen Dateie haben, verwenden wir:

```
ls bahnf* nase* > blubb 2>&1
```

Dies bedeutet: lenke den STDERR dahin, wo auch STDOUT hingeht.

> steht hier übrigens für "schreibe in Datei, wenn sie schon existiert, überschreibe den Inhalt".

>> steht für "hänge an eine bestehende Datei an, wenn sie nicht existiert, lege sie an".

Das Kommando

```
> bla
```

ist also eine einfache Art, Dateien zu leeren (z.B. Logfiles oder eben Ausgabefiles von Testkommandos etcpp.).

Andersherum kann man den Inhalt einer Datei als Eingabe eines Kommandos verwenden:

```
mail -s "Subject" foo@bar  < mail.txt
```

z.B. um eine oder mehrere Mails auf der Kommandozeile zu versenden.

Verkettung von Kommandos: die pipe

Der senkrechte Strich | wird als pipe bezeichnet. Er bedeutet:

"Nimm die Ausgabe von Kommando1 als Eingabe von Kommando2".

Dies kann z.B. nützlich sein, um aus dem Ergebnis von **ls -al** bestimmte Attribute zu filtern (z.B. mit grep s.u.) oder Spalten auszugeben (awk s.u.). Gern wird auch genommen: grep (nach Pattern in Zeilen suchen) und dann aber sortieren (sort), bzw. Dubletten aussortieren (sort -u).

In Vorgriff auf grep und Prozesse ein Beispiel:

```
ps -ef |grep princess
```

gibt alle Prozesse aus (ps -ef), gibt dann aber weiter und sucht nur nach dem pattern princess.

Weitere gute und nützliche Beispiele folgen! Auch und gerade hie ist es wichtig, eventuelle Fehlerausgaben NICHT in das nächste Kommando zu schicken!

Fileberechtigungen und wie man sie setzt (chmod)

Die lange Ausgabe von ls, ls -l gibt aus

- die Fileberechtigungen

- die Anzahl hardlinks
- Username
- Gruppenname
- Größe der Datei (mit -h in "human readable" statt in bytes ;-))
- Monat, Tag, Jahr oder Uhrzeit
- Filename

Die Fileberechtigungen zeigen an der ersten Stelle an, um was für eine Datei es sich handelt:

-	für eine normale Datei
d	für ein Directory
l	für einen symbolischen Link

(es gibt noch mehr, aber die meisten sieht man selten)

Sodann folgt je wein triplet aus

- Berechtigungen für den user
- Berechtigungen für die Gruppe
- Berechtigungen für alle anderen user auf dem Rechner

Bereits in den Rechten für das Home-Directory kann man viel falsch und richtigmachen. Das Home sollte nur für den user selbst lesbar sein. Gemeinsame Projekte sollten Gruppenrechte haben und in anderen Arbeitsverzeichnissen untergebracht sein (ist aber auch eine philosophische Frage).

Es gibt drei Rechte:

- r für lesen
- w für schreiben
- x für ausführen (execute). Bei Verzeichnissen heißt dieses Recht "hereinwechseln dürfen".

Beispiel:

```
drwx------
```

Heißt: in dieses Verzeichnis darf nur der Besitzer hereinwechseln und lesen und schreiben.

```
-rwxrwxrwx
```

heißt: diese Datei dürfen alle lesen, schreiben, ausführen.

```
-rwxr-xr-x
```

wird gern für ausführbare Kommandos genommen (**Aufgabe:** was kommt bei ls -l /bin/ls bei den Berechtigungen) und

```
-rw-r--r--
```

für Systemdateien (**Aufgabe:** ls -l /etc/passwd).

Geändert werden die Benachrichtigungen mit chmod.

```
chmod g+w datei
```

setzt Schreibrechte für die Gruppe. Man nennt also zuerst

- u für user, g für Group und o für others
- dann + für Recht hinzufügen und - für Recht entziehen
- und dann die Rechte r, w, x wie man es braucht.

Aufgabe: man kann statt den Buchstabenkombinationen auch Oktalzahlen verwenden, also 755 für rwxr-xr-x. Wie heißt die Zahl für rw-r--r--? (Ja das ist knifflig!)

Nutzer und Gruppe werden mit chown und chgroup geändert.

Aufgabe: lies beide manpages an.

Prozesse listen und killen

Wir haben schon erfahren, daß manchem Prozesse (daemons) ohne zugeordnetes Terminal auskommen.

Aber: alle Prozesse haben einen Elternprozess, von dem Eigenschaften "vererbt" werden und jeder Prozess hat eine eindeutige Nummer.

Der init-Prozess hat die 1 (und als Elternprozess die 0), alle weiteren Prozesse leiten sich von ihm ab.

Führt man in der Shell aus:

```
ps
```

so erhält man Auskunft, daß gerade eine bash läuft und ein ps:

```
 PID TTY          TIME CMD
49991 pts/0    00:00:00 bash
50183 pts/0    00:00:00 ps
```

Es wird die ProzessID ausgegeben (PID) das Terminal (tty), Zeit und das Kommando, also nur die gerade beteiligten Prozesse und Kindprozesse.

Es läuft auf einem Rechner natürlich viel mehr, diesen Output erhalten wir mit

```
ps -ef
```

und hier erhalten wir auch die PPID (Parent-Prozess-ID) und sehen, daß die meisten root-Prozesse bei TTY ein "?" stehenhaben und als PPIF 1 oder 2.

Aufgabe: leite den Output von ps -ef nach less und blättere seitenweise einmal durch. **Aufgabe 2:** leite den Output in eine Datei um und lies diese mit less.

Besonders deutlich wird das Eltern-Kind-Konzept, wenn man sich

```
pstree
```

ansieht.

kill

Manchmal ist es notwendig, ein Kommando gewaltsam zu beenden, weil es nichts mehr tut oder amok läuft oder ohnehin irgendetwas unerwünschtes tut. IdR. kann man nur seine eigenen Prozesse beenden, root (sudo) kann alle Prozesse abschießen.

Üblicherweise reicht

```
kill PID
```

Wir haben ja schon gesehen, wie wir uns die benötigte Prozeßnummer beschaffen können. Man führt das Kommando einfach zweimal aus, um zu sehen, ob es geklappt hat. Meckert die Shell beim zweiten Mal NICHT, daß es den Prozess nicht mehr gibt, muß man zu harten Mitteln greifen:

```
kill -9 PID
```

Das "-9" bitte sparsam verwenden und nicht "einfach immer", denn es gibt dem Prozess keine Chance mehr, hinter sich aufzuräumen, also z.B. temporäre Dateien zu schließen und wieder zu entfernen.

Klappt auch das nicht, ist der Prozess ein Geist und erscheint in der Prozessliste als "defunct". Oft gehen diese Prozesse erst beim reboot wieder weg. Meist stören sie aber auch nicht.

Eine weitere wichtige Option ist -1

```
kill -1 PID
```

wird oft bei Systemprozessen eingesetzt und bedeutet: "Beende dich nicht, aber lies mal Deine Configfiles neu ein". Dies ist meist sparsamer, als den Systemprozess mit stop und start zu traktieren und geht auch oft schneller.

Hintergrundprozesse

Befehle, die wir in die Shell eintippen, "laufen im Vordergrund". Manche Prozesse laufen aber sehr lange, geben ihren Output eh nicht auf der Shell aus (s.o.) oder sind einfach graphische Prozesse, die ein neues Fenster aufmachen und die Shell nicht mehr brauchen.

Man möchte sie starten, braucht dann aber wieder einen Shellprompt.

Wenn man in einem Fenster ein neues Fenster starten will, kann man tippen:

```
xterm
```

(oder andere terminalfenster wie eterm, gnome-terminal, lxterm, ... Finde notfalls mit ps -ef heraus, wie Dein Terminal genau heißt).

Macht man aber das, kommt der Shellprompt nicht wieder. Er ist quasi von der laufenden Applikation xterm "belegt". Er kommt wieder, wenn Du das neue Fenster beendest, aber das ist ja nicht Sinn der Sache? ;-)

Man startet also grafische Programme "in den Hintergrund":

```
xterm &
```

Was aber, wenn ich das "&" vergessen habe?

Ich schicke den Prozess kurz "ins aus" mit **Ctrl-z** und sodann mit **bg** (background) in den Hintergrund und habe meine Shell wieder.

Mit **Ctrl-z** kann man auch andere Programme unterbrechen, z.B. das Abspielen von mp3s:

```
mpg321 lied.jpg
```

und wenn es weitergehen kann, holt man das Programm zurück mit **fg** für foreground.

Vererbung bei Prozessen

Die Bezeichnung Eltern- und Kindprozess kommt nicht von ganz ungefähr, denn es werden Eigenschaften der Elternprozesse an das Kind vererbt, z.B. Variablen.

Setzt man eine Variable z.B.

```
NASE="Nasenbaer"
```

so kann man sie, wie gelernt, mit "echo $NASE" ausgeben. Macht man nun ein xterm auf mit **xterm &** und versucht, die Variable dort auszugeben, gibt es sie nicht.

Erst wenn man die Variable **exportiert**, wird sie an den Kindprozess weitergegeben:

```
NASE="Nasenbaer"
export NASE
```

oder gleich:

```
export NASE="Nasenbaer"
```

Aufgabe: setze eine Variable und vererbe sie an ein Kind-xterm.

---

grep: Pattern in Dateien suchen

Eine der wichtigsten Tätigkeiten als Admine ist das Logfile-lesen. Geht etwas schief, schaut man erstmal ins log, was denn so loswar. Die meisten logs finden sich unter **/var/log**.

Zuerst schaut man sicher mit less herein und sucht ectl. Begriffe mit **/suchbegriff** wie im vi. (oder mit ?suchbegriff für "suche rückwärts").

Manchmal braucht man aber nur die Zeilen, in der ein bestimmtes pattern vorkommt, z.B. eine Mailadresse oder ein Loginname oder ein Teil einer URL. Dafür gibt es dann **grep**.

grep zeigt alle Zeilen an, die einen bestimmten Suchbegriff enthalten:

```
grep suchbegriff datei
```

Es kann sinnvoll sein, den Output nach less zu lenken und -i zu verwenden für "ignore case". Gerade bei Mailadressen wird gern Groß- und Kleinschreibung verwendet, auch wenn es keinen Unterschied macht:

```
grep -i suchbegriff datei | less
```

Weitere Varianten sind **egrep** (hier kann eine regular expression als suchbegriff verwendet werden), oder auch **rgrep**, das nicht in einer Datei sucht, sondern rekursiv in den unterliegenden Verzeichnissen.

zgrep kann in gezipten Files suchen (Endung .gz), ohne daß man erst die Datei auspacken muss (s.a.: zless).

Eine gute Anwendung für grep ist das Filtern des Output von ps -ef:

```
ps -ef | grep kommandoname
```

oder auch:

```
ps -ef | grep username
```

Man sieht hier aber auch: der grep-Prozess selber wird mit angezeigt. Wenn man eine Negation des Suchpatterns möchte, gibt es die Option **-v** (klein v):

```
ps -ef | grep kommandoname | grep -v grep
```

Eine sinnvolle Funktion für das .bashrc könnte also sein:

```
psg ()
{
  echo 'UID PID PPID C STIME TTY TIME CMD';
  ps -ef | grep $* | grep -v grep
}
```

Der Aufruf lautet dann also

```
psg suchbegriff
```

Aufgabe: warum muss die erste Zeile des Output von ps hier extra mit echo mit ausgegeben werden?

Aufgabe 2: schreibe die Funktion in Dein .bashrc und aktiviere sie.

---

awk: Spalten aus Dateien ausgeben

Eigentlich alle Konfigurationsdateien folgen einem strikten Aufbau und dies ist auch in den dazugehörigen manpages (Sektion 5) aufgeführt.

Aufgabe: lies man 5 crontab und man 5 passwd.

Oft sind die einzelnen Schlüsselworte durch Whitespaces getrennt (also Leerzeichen oder Tabulatoren) oder durch Trennzeichen wie der Doppelpunkt im passwd. Trennzeichen kennen wir auch von Exporten aus der Tabellenkalkulation (excel, librecalc), hier kann man das Trennzeichen beim Speichern selber wählen (z.B. Semikolon, Komma).

Zuweilen braucht man aber (für Skripte etcpp.) nur eine Spalte aus einer Datei oder einem Output. Dafür gibt es **awk**. Der Name setzt sich aus den Anfangsbuchstaben der Entwickler zusammen.

Beispiel: ich möchte anzeigen, welche Mailaliase definiert sind, aber nur die Namen, nicht, wo sie hingehen. Die Mailaliase befinden sich in **/etc/aliases**. Das Format der Datei ist in man 5 aliases beschrieben, das Trennzeichen ist also der Doppelpunkt.

```
awk -F: '{print $1}' /etc/aliases
```

Lasse ich die Angabe des Trennzeichens -F weg, wird white space als Trennzeichen angenommen. Die jetzige Ausgabe aber enthält noch die Kommentarzeilen, die mit # beginnen.

Aufgabe: filtere mit grep die Zeilen mit # heraus. Beachte, daß # ein Sonderzeichen ist.

Aufgabe 2: gib die erste Spalte von /etc/passwd aus.

Aufgabe 3: verbinde grep und awk und gib nur Deinen Usernamen aus /etc/passwd aus.

Ein weiteres Anwendungsbeispiel brauche ich beim "Archivieren von Hand": ls -al gibt neben Berechtigungen und Dateinamen auch das Datum aus, wann eine Datei zuletzt geschrieben wurde. Möchte ich z.B. alle Dateien aus dem Jahr 2017 archivieren/verschrieben, kann ich filtern:

```
ls -al |grep 2017
```

Die Dateien stehen im Output ganz hinten, ich kann nun mit der Maus cut'n'paste machen, aber son von Hand machen wir das natürlich nicht! :)

Ich möchte also nur den Dateinamen ausgeben, der steht (mit whitespace getrennt) in Spalte 9:

```
ls -al |grep 2017 |awk '{print $9}'
```

Sodann möchte ich ja, daß etwas mit den Dateien passiert, z.B. cp. Wie füttere ich aber dem cp nun diese Fileliste ein? Ich schließe den Befehl in Backquotes ein: ` (rückwärtiges Hochkomma). So wird in den Backquotes der Befehl durch das Ergebnis ersetzt:

```
cp `ls -al |grep 2017 |awk '{print $9}'` neuesverzeichnis
```

(neuesverzeichnis sollte vorher angelegt werden.

Nun ist die Syntax von awk, wie wir gesehen haben, etwas gewöhnungsbedürftig. Menschen in meinem U,fed deklarieren a1 bis a9 (oder sogar weiter) wie folgt:

```
alias a1="awk '{print \$1}'"
```

Wenn Du das sinnvoll findest, übernimm es in Dein .bashrc.

Wir sehen: mit grep, awk und der pipe stehen uns mächtige Werkzeuge zum Ansehen und Bearbeiten von Files und Output zur Verfügung.

Aufgabe: was macht im Gegensatz zu awk das Kommando **cut**?

---

find: Dateien nach bestimmten Kriterien anzeigen

find kann viel mehr als "nur" nach Namen suchen, sondern auch nach Permissions, Dateien oder Verzeichnissen, nach Alter einer Datei und ähnlichem. Desweiteren braucht man find auch, wenn ein lausiger Programmierer in einem Verzeichnis soviele Dateien angelegt hat, daß ls diese nicht mehr verarbeiten kann und aussteigt. Alles schon dagewesen!

find gibt unterhalb eines Verzeichnisses alle Dateien aus, die er rekursiv in allen Verzeichnissen findet:

```
find . -name '*rc' -print
```

Man gibt also zuerst das Verzeichnis an (. für das aktuelle Verzeichnis ist gern genommen), sodann die Suchkriterien. Verwendet man bei -name Wildcards, sind einfache Quotes ratsam. Der obige Befehl gibt also alle Files aus (auch die, die mit . beginnen!), die auf rc enden. Was war nochmal rc? :)

find ohne weitere Argumente gibt unterhalb von . alle Files aus, die er findet. -print braucht man unter Linux idR. nicht, aber es ist guter Stil, das sicherheitshalber immer mitzuschreiben.

Gut verbinden läßt sich find natürlich auch mit anderen Kommandos wie grep. Wenn ich also mal nicht weiß, in welcher Datei ich was abgelegt habe, kann ich suchen:

```
find . -type f -print | xargs grep suchbegriff
```

-type f sucht nur Dateien (in Verzeichnissen kann ich nich greppen, versuch es mal) und xargs brauchen wir hier aus diversen komplexen Gründen.

Aufgabe: lies Dich in den manpage ein um ein Gefühl zu bekommen, was find alles kann. Alternativ: suche im Ubuntuusers Wiki, dort ist auch vieles gut und mit Beispielen erklärt!
