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!

Revision #1
Created 2022-03-10 15:16:23 UTC by merline
Updated 2022-06-13 11:49:22 UTC by merline