Unix Shell und einige Tools

Johann Mitlöhner

16. November 2006

1 Shell

Beim Einloggen in das Linuxsystem am eigenen PC bekommt der Benutzer typischerweise eine grafische Schnittstelle, z.B den Gnome **Desktop**. Daneben gibt es aber auch die **Shell**, eine Text-basierte Schnittstelle, die auf Eingaben über die Tastatur reagiert. Diese Schnittstelle hat gegenüber der grafischen eine Reihe von Vorteilen, insbesondere dann, wenn immer wieder gleiche Aufgaben zu erledigen sind, oder nicht alltägliche, sodaß in den Desktopmenüs nichts brauchbares vorgesehen ist. Es lohnt sich daher, diese Shell kennenzulernen.

Mit einem **Terminal** (in Gnome unter *Applications/Accessories*) starten Sie eine Shell. Dort blinkt nun ein Cursor, links daneben sehen Sie ein **Prompt**. Typischerweise zeigt es das aktuelle Verzeichnis, kann aber auch anders konfiguriert werden. Die Shell wartet auf Kommandos von Ihnen, die Sie über die Tastatur eingeben und mit der **Return** Taste (Neue Zeile) abschließen. Die Maus hat hier keine Funktionen außer Copy/Paste. Die wichtigsten Kommandos sind:

`ls`	list
`ls -la`	long list
`mkdir tmp`	make new directory “tmp”
`cd x`	change working directory to “x”
`cd`	change to home directory
`pwd`	print working directory
`cp x y`	copy file x to y
`mv x y`	move (rename) file “x” to “y”
`rm x`	remove file x
`rm -rf x`	remove directory x and all its contents

Mit dem Editor **pico** können Sie Textdateien erstellen, indem Sie den Namen der neuen Datei angeben:

```
pico newfile.txt
```

Schreiben Sie einige Zeilen Text und speichern Sie mit Strg-O gefolgt von Return, verlassen Sie mit Strg-X. Wenn Sie nun wieder “ls” eingeben, sehen Sie die neue Datei in der Liste.

Unter Linux verwenden wir die Shell **bash**, eine neuere Version der traditionellen Unix Shell **sh**. Die bash bietet viele Features, von denen wohl die history und die file name completion die wichtigsten sind.

- Mit der Pfeil-Aufwärts-Taste bekommen Sie die letzten Befehle, die Sie eingegeben haben, und können diese nochmal ausführen, indem Sie wieder auf Return drücken; Sie können den alten Befehl vorher auch ändern, was viel Schreibarbeit sparen kann.

- Oft möchte man einen der letzten Befehle aus der history wiederholen, der mit den Anfangsbuchstaben einfacher bezeichnet werden kann als mit einigen Pfeil-Aufwärts, z.B. “pico MyClass.java”. Das geht mit dem Rufzeichen:

  ```
  !pi
  ```

- Datei- und Programmnamen können mit der Tabulatortaste vervollständigt werden, sobald sie eindeutig sind. Wenn Sie z.B. die Datei file1.txt mit ls -l auflisten wollen, können Sie nach “ls -l f” den Tabulator drücken. Falls Sie keine andere Datei im aktuellen Verzeichnis haben, die mit “f” beginnt, wird der restliche Dateiname vervollständigt. Falls es mehrere Dateien gibt, dann können Sie nochmals auf Tabulator drücken und sehen eine Liste der in Frage kommenden Dateien. Dann können Sie noch weitere Buchstaben eingeben, bis der Name eindeutig ist und mit Tabulator vervollständigt werden kann. Auch damit kann man viel Schreibarbeit sparen.

Eine Liste Ihrer früheren Kommandos sehen Sie mit dem Befehl

```
history
```

2 Permissions

Geben Sie zum Vergleich

```
ls -la
```

ein, um alle Dateien und Verzeichnisse zu sehen (auch versteckte), sowie die **Permissions**:

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

Das erste Zeichen gibt den Typ an, hier “-” für Datei, z.B. “d” für Verzeichnis, “l” für Link. Danach folgen in drei Bereichen zu je drei Zeichen die Berechtigungen für Sie selbst, Ihre Gruppe (jeder Unixbenutzer gehört einer oder mehreren Gruppen an), und für alle anderen. Dabei steht in jedem Bereich das erste Zeichen für Lesen, das zweite für Schreiben, und das dritte für Ausführen.

Die Datei newfile.txt können Sie selbst lesen und schreiben, andere können sie nur lesen, ausführen kann sie niemand. Bei Verzeichnissen bedeutet Ausführen soviel wie Wechseln in das Verzeichnis.

Beachten Sie den Eintrag “.” in dem Ergebnis von “ls -la”. Der Ausdruck “.” steht für das aktuelle Verzeichnis, in diesem Fall Ihr home directory, das normalerweise die permissions “drwxr-xr-x hat, d.h. andere dürfen in Ihrem Verzeichnis zwar lesen, aber nichts ändern.

Zum Ändern der permissions gibt es das Kommando “chmod”:

```
chmod go-r newfile.txt
```

Nun darf niemand außer Ihnen die Datei newfile.txt lesen oder schreiben. Dabei beinhaltet Schreiben auch Ändern und Löschen.

```
chmod go+r *.txt
```

Das Zeichen "*" steht für "alle". Alle Dateien im aktuellen Verzeichnis können nun von allen Benutzern ("g" für group, also alle aus Ihrer Gruppe, "o" für others, also alle anderen) gelesen werden. Benutzer bezieht sich dabei nicht nur auf Personen, die mit dem Rechner arbeiten, sondern auch auf Programm, die auf dem Rechner laufen, denn jedes Programm läuft unter Unix unter einem Benutzer und daher auch mit bestimmten Berechtigungen. Interessant ist das z.B. für Webseiten, die für alle lesbar sein müssen, denn der Webserver läuft typischerweise unter dem Benutzer "apache" und kann daher Ihre Webseiten nur lesen, wenn sie die richtigen permissions haben, und in einem für alle lesbaren Verzeichnis stehen; ein Verzeichnis ist aber nur lesbar, wenn auch die übergeordneten Verzeichnisse lesbar sind. Sowohl Ihr "www" Verzeichnis (oft auch "public_html" genannt) als auch Ihr home directory müssen daher die permissions drwxr-xr-x haben.

3 Man

Alle installierten Programme sollten im Online Manuel dokumentiert sein. Dieses steht Ihnen jederzeit mit dem Befehl "man" zur Verfügung:

```
man chmod
man ls
man cp
```

In der Beschreibung können Sie mit den Bildaufwärts/abwärts Tasten blättern und mit "q" verlassen.

4 Environment variables

In der Shell haben Sie eine Reihe von Werten als **environment variables**, die verschiedene Programme verwenden, um Einstellungen abzufragen. Mit

```
env
```

sehen Sie eine Liste der Variablennamen mit den Werten. Interessant ist z.B. die Java-Version, die Sie verwenden:

```
echo $JAVA_HOME
```

Ist eine Variable nicht gesetzt, kommt eine Leerzeile zurück. Soll eine Variable gesetzt werden, und zwar so, daß sie auch innerhalb von anderen shell scripts (nicht nur innerhalb Ihrer Shell) zur Verfügung steht, dann verwenden Sie den Befehl "export":

```
export JAVA_HOME=$HOME/jdk1.6.0
```

Die Variable HOME enthählt natürlich den Pfad auf Ihr home directory. Wenn Sie nun nochmals

```
echo $JAVA_HOME
```

eingeben, sehen Sie den gesamten Pfad. Environment variables werden typischerweise in .bashrc gesetzt, eine Datei, die beim Start jeder Shell ausgeführt wird. Sie können sich diese Datei mit dem Editor ansehen; ändern sollten Sie zunächst besser nichts, bevor Sie sich nicht mit shell scripts auskennen.

5 Shell scripts

Wenn der Inhalt von Variablen abgefragt werden soll, dann muß das Zeichen $ verwendet werden. Das ist oft umständlich, funktioniert aber auch innerhalb von Texten:

```
echo "Im Verzeichnis $HOME sind meine Dateien."
```

Der Befehl “echo” wertet den Ausdruck aus und gibt das Ergebnis zurück. Damit sind wir auch schon mitten im Shell-Programmieren, das nicht anders ist als das interaktive Arbeiten. Legen Sie mit dem Pico-Editor eine Datei “doit” an und schreiben Sie folgendes hinein:

```
#!/bin/sh

rm -rf tmp.dir
mkdir tmp.dir
cd tmp.dir
date > file1.txt
pwd >> file1.txt
```

1. Zeile: dieses Skript wird mit /bin/sh als Shell ausgeführt. Auch andere Programme können Textdateien als Code auffassen und ausführen, z.B. Perl, Python.

2. Zeile: bleibt per Konvention leer. Hier könnten Sie Ihren Namen und Datum hinterlassen, falls es sich um etwas größeres handelt.

3. Wir löschen das Verzeichnis “tmp.dir” (und hoffentlich hatten Sie dort nichts wichtiges gespeichert).

4. Wir legen ein Verzeichnis mit gleichem Namen neu an, das jetzt auf jeden Fall existiert und leer ist.

5. Wir wechseln in das neue Verzeichnis. Da dieses Skript mit einer eigenen Shell ausgeführt wird, hat sich damit unser aktuelles Verzeichnis in der interaktiven Shell nicht geändert.

6. Der Befehl “date” gibt Datum, Uhrzeit und Zeitzone aus. Das Ergebnis leiten wir in eine neue Datei mit dem Namen “file1.txt” um.

7. Den Namen des aktuellen Verzeichnisses hängen wir an das Ende der Dateil file1.txt an.

Speichern Sie, verlassen Sie den Editor und setzen Sie die permissions auf Ausführen:

```
chmod +x doit
```

Nur Sie selbst sollen dieses Script ausführen, daher diesmal kein “go”. Geben Sie “doit” ein und danach “ls”. Nun haben Sie ein Verzeichnis “tmp.dir”.

Falls Sie eine Fehlermeldung “not found” bekommen, dann ist das aktuelle Verzeichnis nicht in Ihrem Pfad:

```
echo $PATH
```

Der Pfad ist eine Liste von Verzeichnissen, in denen ausführbare Dateien gesucht werden. Es gibt es zwei Möglichkeiten, Ihr Shell script dennoch auszuführen: mit

```
./doit
```

stellen Sie klar, daß die Datei “doit” im aktuellen Verzeichnis gemeint ist; oder Sie fügen das aktuelle Verzeichnis zum Pfad hinzu:

```
export PATH=$PATH:.
```

Eine weitere, bessere Variante ist, sich ein eigenes bin Verzeichnis anzulegen (cd; mkdir bin) und dort alle Shell scripts zu speichern. Dann können Sie mit

```
export PATH=$PATH:$HOME/bin
```

dieses Verzeichnis zum Suchpfad hinzufügen, vermeiden einige Sicherheitsprobleme, haben einen guten Überblick über Ihre Shell scripts, und können Sie in jedem anderen Verzeichnis verwenden!

Werfen Sie nun einen Blick ins Verzeichnis tmp.dir und kontrollieren Sie den Inhalt von file1.txt mit

```
cat file1.txt
```

Wir müssen also nicht jedesmal einen Editor starten; den Inhalt einer kurzen Textdatei können wir uns mit "cat" auch im Terminal ausgeben lassen.

6 Pipe

Oft haben wir die Situation, daß wir das Ergebnis des einen Programms als Eingabe für das nächste verwenden wollen. Natürlich könnten wir eine temporäre Datei verwenden, die wir nachher wieder löschen, eleganter ist aber eine **pipe**:

```
ls *.txt | wc
```

Das Ergebnis von ls ist eine Liste, die an das Programm **wc** (für word count) weitergegeben wird. Ein kurzer Blick ins online manual (man wc) sagt uns, daß der erste Wert die Anzahl der Zeilen ist, daher wissen wir nun, wieviele Dateien mit der Endung ".txt" wir im aktuellen Verzeichnis haben.

Ein Unixsystem besteht aus einer großen Zahl von scheinbar unzusammenhängenden Programmen wie wc, die verschiedenste Aufgaben erfüllen. Einige dieser Werkzeuge sollte man kennen, und eines der wichtigsten ist wohl grep.

7 Grep

Mit dem Programm **grep** können wir Zeilen aus Dateien ausgeben, die bestimmte Texte enthalten:

```
grep section *.tex
```

Alle Zeilen mit "section", "section" usw. in allen Dateien mit der Endung ".tex" werden ausgegeben.

```
grep printf *.java | wc -l
```

Wieviele Printf-Befehle haben wir in allen unseren Java-Programmen?

```
grep -l Wurlitzer *.java | sort -u
```

Nicht die Zeilen, sondern die Namen der Dateien werden ausgegeben, uns zwar dank "sort -u" (für unique) nur einmal: nun wissen wir, wo unsere Klasse Wurlitzer verwendet wird.

8 Find

Oft stehen wir vor Aufgaben, die sich nur lösen lassen, wenn mehrere Verzeichnisse durchsucht oder sonstwie abgearbeitet werden. Mit dem Programm "find" ist das einfach:

```
find . -name '*.java' -exec grep -l Wurlitzer {} \; | sort -u
```

Im aktuellen Verzeichnis und allen Unterverzeichnissen werden alle Java-Dateien nach dem Text "Wurlitzer" durchsucht. Wie vorher wird nur der Name der Datei ausgegeben, und das Ergebnis wird mit "sort -u" sortiert, sodaß mehrfache Vorkommen entfernt werden.

```
find . -name '*.zip' -size +1000k
```

Alle ZIP-Dateien, die größer als 1 MB sind.

```
find . -name '*.zip' -size +1000k -exec ls -s {} \;
```

Die Größe in KB dazu.

```
find . -name '*.zip' -size +1000k -exec ls -l {} \; | sort -n
```

Das Ergebnis sortiert, und zwar numerisch.

Wie man sieht, ist find vor allem zusammen mit anderen Werkzeugen wir grep und sort sehr nützlich. Es lohnt sich daher, die man pages und die dortigen Beispiele zu studieren!

9 Stdin, stdout, stderr

Unix-Programme haben eine Standard-Eingabe, das ist ein Strom von Bytes, die numerisch mit 0 und alphabetisch mit "stdin" bezeichnet werden. Entprechend gibt es auch den "stdout" mit der Nummer 1, sowie den "stderr" mit der Nummer 2. Die Standard-Eingabe kann eine Datei sein, aber auch die Tastatur:

```
wc
```

Nun passiert zunächst nichts; geben Sie einige kurze Zeilen Text ein und danach Strg-D, das bedeutet Ende der Eingabe. Nun arbeitet das Programm wc (word count) mit Ihrem Text und zählt Zeilen, Wörter und Bytes.

Wenn auf der Kommandozeile nichts angegeben wird, dann wird als stdin von der Tastatur gelesen und als stdout am Terminal ausgegeben; die beiden können aber auch in Dateien umgeleitet werden:

```
wc < file1.txt > cnt.txt
```

Dabei sind die spitzen Klammern wie Pfeile zu verstehen, d.h. der Inhalt der Dateil file1.txt "fließt" hinein ins word count, und das Ergebnis fließt in die Datei cnt.txt.

Manchmal produzieren Programme eine große Zahl von Fehlermeldungen, die ebenfalls am Terminal ausgegeben werden. Diese kann man in eine Datei umleiten, um später (z.B. mit grep) nach bestimmten Fehlern zu suchen:

```
bigStatistics < largeInput.dat > largeOutput.dat 2>error.log
```

Wenn die Fehlermeldungen ignoriert werden können, müssen sie nicht gespeichert werden. Unix stellt dafür das null device zur Verfügung:

```
prog <input >output 2>/dev/null
```

Damit werden alle Ausgaben nach stderr auf das null device umgeleitet d.h. weggeworfen.

10 .bashrc

Da wir nun einige shell script Grundlagen kennen, werfen wir einen Blick auf die Datei .bashrc in Ihrem home directory. Dort werden einige Einstellungen gemacht, die sich auf Ihre Arbeit auswirken. Sie können diese verändern und z.B. weitere Umgebungsvariablen setzen, wie die schon bekannte Java-Version, indem Sie am Ende der Datei folgenden Befehl hinzufügen:

```
export JAVA_HOME=$HOME/jdk1.6.0
```

Natürlich hat das nur Sinn, wenn sich in diesem Verzeichnis auch wirklich eine Java-Installation befindet. Sonst suchen Sie lieber die systemweite Version, z.B. mit

```
find / -type d -name '*/bin/javac'
```

oder (schneller)

```
locate javac
```

Die Datei .bashrc wird jedesmal ausgeführt, wenn eine neue Shell gestartet wird, also z.B. auch, wenn Sie ein weiteres Terminalfenster öffnen. Wenn Sie die .bashrc geändert haben und die Änderungen in Ihrer aktuellen Shell merken wollen, müssen Sie

```
source ~/.bashrc
```

eingeben, wobei die Tilde "~" für Ihr home directory steht. Stattdessen können Sie natürlich auch

```
source $HOME/.bashrc
```

verwenden. Damit ist das shell script .bashrc neuerlich ausgeführt worden.

Im .bashrc können Sie auch Ihren Pfad ändern, wenn Sie z.B. ein bin/ Verzeichnis in Ihrem home directory anlegen und alle shell scripts dort speichern. Dann lassen Sie den Suchpfad mit diesem Verzeichnis beginnen, damit bei Namensgleichheit Ihre Version verwendet wird:

```
export PATH=$HOME/bin:$PATH
```

11 Alias

Viele Aufgaben sind so einfach, daß nicht jedesmal ein shell script geschrieben werden muß. So könnte es z.B. ein Program geben, das Sie immer mit bestimmten Optionen verwenden. Dann könnten Sie natürlich ein shell script gleichen Namens schreiben:

```
#!/bin/bash

someprog -b $*
```

Nun bekommt Ihr someprog immer die Option -b sowie eventuelle restliche Parameter mit dem Ausdruck `$*`. Einfacher ist dieser Effekt aber mit einem **alias** zu erreichen. Schreiben Sie in Ihr .bashrc folgendes:

```
alias someprog='/usr/bin/someprog -b'
```

Dazu müssen Sie zunächst feststellen, in welchem Verzeichnis sich Ihr someprog befindet:

```
which someprog
```

Der Befehl **which** gibt Ihnen den kompletten Pfad, den Sie im alias verwenden. Mit

```
alias
```

bekommen Sie eine Liste der gesetzten alias Einträge.

12 Top

Einen Überblick über die laufenden Prozesse auf Ihren Rechner erhalten Sie mit

```
top
```

Dort werden alle Prozesse aufgelistet und nach definierbaren Kriterien sortiert. Alle fünf Sekunden wird die Liste erneuert. Es gibt Parameter wie z.B.

```
top -d 1
```

um jede Sekunde eine neue Liste zu erhalten, aber auch top versteht auch einige interaktive Kommandos, z.B. "1" zum Einblenden der Auslastung der einzelnen Prozessoren (falls es mehrere gibt), sowie "k" (für kill) zum Beenden von Prozessen; danach werden Sie zur Eingabe der prozess id und des Interrupts aufgefordert, dabei ist 9 der sicherste kill.

Manchmal brauchen einzelne Prozesse sehr viel Rechenzeit, was den interaktiven Betrieb mühsam macht. Dann können Sie versuchen, durch geringere Priorität die Situation zu retten, ohne den Prozess zu killen, indem Sie ihn renicen, d.h. Sie geben "R" gefolgt vom PID und priority ein. Die normale Priorität ist 10, höhrere Werte bedeuten weniger Rechenzeit und damit weniger Belastung das Systems.

Die meisten Prozesse sind die meiste Zeit im Modus S für sleep, d.h. sie verbrauchen fast keine Resourcen. In der top Anzeige sind die Prozesse nach der verbrauchten CPU Zeit gereiht, und der oberste kann im Zustand R für running sein. Gelegentlich sieht man auch D für dead und sogar Z für Zombie. Solche Prozesse sind meist die Folge von Programmabstürzen und können gekillt werden.

Im Unterschied zu anderen Betriebssystemen wird Unix nicht durch einzelne Prozesse zum Stillstand gebracht. Die Prozesse sind hier sauber voneinander und vom Betriebssystemkern getrennt. Auch vieles, was als Betriebssystemaufgabe verstanden wird läuft über Prozesse. Typischerweise bringen nur schwere Hardwareprobleme das ganze System zum Stillstand, wenn z.B. nicht mehr auf die Festplatte zugegriffen werden kann. Es ist nicht ungewöhnlich, daß Unixrechner wochenlang oder monatelang ohne Unterbrechung und

ohne Probleme laufen, und zwar nicht nur Server ohne direkte Benutzer, sondern auch solche, mit denen als PC interaktiv am Desktop gearbeitet wird.

Sie können die Liste der Prozesse auch auf den stdout leiten, indem Sie das Kommando

```
ps -ef
```

verwenden. Nun können Sie die Liste in Ruhe studieren, oder auch als inputs in ihren Shell scripts verwenden.

13 bg, fg, nohup

Wenn Sie Programme verwenden, die sehr lange laufen, weil z.B. Simulationen gerechnet werden oder aufwendige Auswertungen vorgenommen werden, dann können Sie mit

```
bigjob -some -options < input.txt > output.txt &
```

ein solches Programm in den "Hintergrund" schicken und in der Shell weiterarbeiten. Mit

```
jobs
```

erhalten Sie eine Liste der aktuell im Hintergrund arbeitenden Prozesse. Falls ein Prozess doch wieder in den Vordergrung soll, dann geben Sie ein:

```
fg
```

oder, falls es mehrere gibt:

```
fg %1
```

für den ersten Job in der Liste.

Hintergrundprozesse laufen weiter, solange Sie eingeloggt sind; wenn Sie allerdings ausloggen, werden alle jobs gekillt, ausser jene, die Sie mit **nohup** gestartet haben:

```
nohup bigjob > output.txt &
```

Nun können Sie nach Hause gehen und morgen wieder einloggen, und der Job läuft immer noch, außer der Rechner wurde zwischendurch neugestartet, oder einer der Sysadmins hat Ihren job gekillt, weil er zuviel Systemresourcen verbraucht hat. Wie sehr Ihr Job das System belastet, können Sie selbst ja mit top feststellen.

xargs

Oft möchte man ein bestimmtes Kommando auf eine Liste von Dateien anwenden, die als Resultat eines vorherigen Kommandos in einer Pipe zur Verfügung stehen; dazu kann **xargs** verwendet werden:

```
find . -exec grep -l println {} \; | xargs wc -l
```

Der Befehl `wc` wird nun auf jede einzelne Datei angewendet, die aus dem `find` kommt, also den Text "println" enthält.

Würden wir `xargs` aus der obigen Zeile weglassen, dann würden wir die Anzahl der Zeilen zählen, die `find` ausgegeben hat.

Rsync

Zur Synchronisierung von Dateien, oft genutzt für backup, weil es nur kopiert soweit nötig:

```
#!/bin/sh

# removable media
dest=/media/usbdisk/bak/

# rsync options
OPTS="-av --max-size=1m"

# check media is plugged in
if [ ! -x $dest ]; then
  echo "Backup media not mounted"
  exit 1
fi

# local host
cd
rsync $OPTS  vw publ projects "$dest/pc"

# remote host
rsync $OPTS  balrog:www/le  "$dest/balrog/www"
```

Hier wird auf eine mobile USB disk gesichert, daher prüfen wir zunächst, ob diese Platte auch wirklich angeschlossen ist.

Mit rsync können wir nun Verzeichnisse auf dem lokalen Rechner sichern, und auch Verzeichnisse von anderen Rechnern (remote hosts), wenn wir dort mit ssh ohne Passwortangabe einloggen können.[1]

Kurzer Vergleich bash, Perl, Python, C

- bash: kurze Scripts mit einfachen Operationen auf Strings. Performance extrem schlecht.
- perl: viel genutzt für verschiedenste Aufgaben in Unix, aber nicht numerische. Performance mäßig.
- python: immer beliebter wegen einfacher und übersichtlicher Syntax – executable pseudo code. Performance gut.
- C: für sehr rechenaufwendige Aufgaben, klassische Unix Programmiersprache. ISO Standard! Performance maximal.
- C++, Java: durch Objektorientierung für Kooperation in Programmierteams besonders gut geeignet. Performance maximal, bei Java Einschränkung durch startup der runtime.

Zur Messung der Performance:

```
time myprog someargs
```

[1] D.h. wenn unser id_dsa.pub aus dem .ssh Verzeichnis im authorized_keys file am remote host enthalten ist, und wir uns lokal mit ssh-add authentisiert haben.