Unix

THE STORY OF CREATION
or
THE MYTH OF URK

In the beginning there was data. The data was without form and null, and darkness was upon the face of the console; and the Spirit of IBM was moving over the face of the market. And DEC said, “Let there be registers;” and there were registers. And DEC saw that they carried; and DEC separated the data from the instructions. DEC called the data Stack, and the instructions they called Code. And there was evening and there was morning, one interrupt ...

– Rico Tudor

Inhaltsverzeichnis

1 Unix-Grundlagen

1.1 Was ist "Unix"?

Unix ist alt. Uralt für ein Computersystem. Im Prinzip repräsentiert Unix Technologie der 60er Jahre des vorigen Jahrhunderts. Warum benutzt man es dann noch, wo es doch so viele andere, neuere Systeme gibt? Der wichtigste Grund: Es funktioniert. Die meisten neueren Systeme tun genau das nicht. Ein Beispiel: Die Voyager-Sonden sind seit den 70er Jahren unterwegs. Nach dem Start war keinem Systemadministrator mehr möglich, die Maschinen einfach neu zu installieren, geschweige denn das System komplett herunterzufahren und zu resetten. (Wer weiß, ob diese Sonde danach die Erde noch wiederfände?) Würde irgendjemand für ein derartiges Unterfangen ein "modernes" Betriebssystem ernsthaft auch nur in Erwägung ziehen? Fast 30 Jahre Uptime? Ohne Reboot? Welcher Bürocomputer schafft signifikant mehr als 8h?
Andere Eigenschaften:

- *Multiuser*: An einer Maschine kann gleichzeitig mehr als ein Benutzer arbeiten.
- *Multitasking*: Eine Maschine führt gleichzeitig Dutzende von Programmen aus, ohne daß der Benutzer dies merkt.
- Unix läuft auf verschiedenen Hardwareplattformen (z.B. PCs, Workstations, Mainframes).
- Es gibt verschiedene Implementationen (z.B. Linux, Tru64, HP-UX, Solaris), die sich aber alle für den Benutzer im Wesentlichen gleich bedienen lassen.
- Unix ist ein administriertes System, d.h. es gibt einen *Systemverwalter*, der alle Rechte hat und alle Dateien und Verzeichnisse lesen und schreiben kann. Ein Benutzer kann ihm das nicht verbieten. Gleichzeitig hat der Benutzer nur die Rechte, die ihm der Administrator explizit einräumt. Das hat zur Folge, daß ein Benutzer das System nur in einem eng begrenzten Bereich verändern kann. Auch sind diese Veränderungen nur für diesen Benutzer gültig und nicht systemweit. Das heißt, kein Benutzer kann irgendetwas kaputt machen. Was zunächst wie eine Beschränkung aussieht entpuppt sich in der Praxis als gewaltiger Vorteil.
- Unix ist dafür ausgelegt, daß viele Benutzer gleichzeitig auf der gleichen Maschine arbeiten. Das bringt es mit sich, daß man auf Unix-Maschinen

problemlos “remote” arbeteiten kann, daß man also nicht direkt an der Konsole sitzen muß. Es genügt eine Netzwerkverbindung (das Internet) um auf jedem angeschlossenen Rechner arbeiten zu können. Der Benutzer merkt i.d.R. keinen Unterschied ob er nun lokal oder auf einem fernen Host arbeitet.

1.2 Das Filesystem

Unix bildet fast alle Operationen in sein *Filesystem* ab. Daher ist es wichtig, das Filesystem zu verstehen. Das Unix-Filesystem ist ein eindrucksvolles Beispiel, wie man mit etwas einfachem, das aber wirklich durchdacht ist, komplexe Probleme löst.
Grundsätzliches:

- Dateien werden in Verzeichnissen organisiert.
- Jede *Datei* hat einen *Besitzer* und eine *Gruppe*, der sie angehört. So kann sichergestellt werden, daß jeder Benutzer nur die Dateien sehen und ändern kann, die ihm gehören.
- Für jede Datei gibt es bestimmte *Rechte*.
- Unix kennt keine Laufwerksbuchstaben; alle Verzeichnisse sind innerhalb eines großen Baumes angelegt. Wo diese Verzeichnisse herkommen (ob von verschiedenen Platten oder anderen Rechnern über das Netzwerk), ist für den Benutzer nicht ersichtlich und nicht wichtig.
- Der Anfang des Dateisystems ist das *Root-Verzeichnis* “/”; von dort aus werden alle anderen Objekte in Unterverzeichnissen abgelegt.

Ein typsiches Root-Filesystem:

```
drwxr-xr-x    2 root  root     4096 Sep 16 18:24 bin
drwxr-xr-x    3 root  root     1024 Oct 14 22:16 boot
drwxr-xr-x   14 root  root    81920 Oct 14 22:17 dev
drwxr-xr-x   58 root  root     4096 Oct 14 22:17 etc
drwxr-xr-x   13 root  users    4096 Oct 14 17:10 home
drwxr-xr-x    6 root  root     4096 Sep 16 18:24 lib
drwxrwxr-x    7 root  users    4096 Oct  3 11:55 local
drwxr-xr-x    2 root  root    16384 Apr 28 14:28 lost+found
drwxr-xr-x    2 root  root        0 Oct 14 22:17 misc
drwxr-xr-x    7 root  root     4096 Sep 15 19:26 mnt
drwxr-xr-x   26 root  root     4096 Oct 13 23:27 opt
```

```
dr-xr-xr-x   98 root  root        0 Oct 14 22:17 proc
drwxr-x---   29 root  root     4096 Oct 16 08:41 root
drwxr-xr-x    2 root  root     4096 Sep 16 18:25 sbin
drwxrwxrwt   26 root  root     4096 Oct 16 08:24 tmp
drwxr-xr-x   19 root  root     4096 Sep 16 09:48 usr
drwxr-xr-x   18 root  root     4096 Apr 28 13:22 var
```

Jedes angezeigte Objekt ist wiederum ein Verzeichnis, das dann entsprechende Unterverzeichnisse enthält.

1.3 Dateien und Verzeichnisse

Jeder Benutzer hat ein *Home-Verzeichnis*. Das ist dasjenige Verzeichnis, in dem er sozusagen wohnt. Innerhalb dieses Verzeichnisses darf er (fast) alles machen: weitere Verzeichnisse und Dateien anlegen, bearbeiten, löschen, Rechte für Dateien und Verzeichnisse vergeben usw.
Regeln:

1. Bei der Angabe von *Pfaden*, das heißt Wegen zu bestimmten Dateien und Verzeichnissen, werden Verzeichnisse und Dateien durch einen *Slash* "/" getrennt. Ein gültiger Pfad wäre z.B. `/usr/local/bin/vim`. Er würde die Datei "vim" aus dem Verzeichnis `/usr/local/bin` ansprechen.

2. *Absolute Pfade* beginnen immer im Root, d.h. mit "/", und zeigen den ganzen Pfad zu einer Datei durch das gesamte Filesystem an.

3. *Relative Pfade* beginnen im aktuellen Verzeichnis, sie beginnen entweder mit einem Datei- oder Verzeichnisnamen oder mit "./". Der Punkt steht dabei für das aktuelle Verzeichnis.

4. Für relative Pfade können die beiden Stellvertreter "." und ".." benutzt werden. "." meint das jetzt aktuelle Verzeichnis, ".." meint das übergeordnete Verzeichnis.

5. Für das eigene Heimatverzeichnis verwendet man die Tilde˜.

Beispiele für gültige Pfade:

- `/usr/local/bin/vim` - absoluter Pfad
- `./Documents/LaTeX/Unixkurs.tex` - relativer Pfad
- `Documents/LaTeX/Unixkurs.tex` - relativer Pfad

- `~/Documents/LaTeX/Unixkurs.tex` - absoluter Pfad. Die Tilde wird automatisch durch den absoluten Pfad zum jeweiligen Home-Verzeichnis ersetzt bevor der Pfad verarbeitet wird.

- `../../usr/local/bin/vim` - relativer Pfad

Regeln für Datei und Verzeichnisnamen:

- Unix ist *case-sensitive*, d.h. es unterscheidet, ob ein Name groß oder klein geschrieben wird. Die beiden Dateien `foo.bar` und `FOO.BAR` sind also verschiedene Objekte und können durchaus im gleichen Verzeichnis vorkommen. Eine dritte Datei mit Namen `fOo.BaR` wäre wieder eine andere Datei. (Zur Vermeidung von Chaos sollte man sowas nicht machen...) Das ist anders als z.B. unter OS/2 oder DOS und eine häufige Fehlerquelle für Neulinge.

- Dateinamen dürfen bis zu 255 Zeichen lang sein.

- Dateinamen setzen sich aus Buchstaben und Ziffern zusammen.

- Dateinamen können eine Erweiterung (z.B. `.tex`) haben, müssen das aber nicht. In gewissem Sinne ist Unix das von DOS und anderen Systemen bekannte Konzept der Erweiterung unbekannt. Es sind einfach nur Punkte in den Dateinamen erlaubt, und üblicherweise benutzt man eine kurze Kennung nach dem Dateinamen, um den Typ kenntlich zu machen.[1]

  Hinweis: Die verwendung bestimmter Erweiterungen ist eine Konvention. Man sollte sich an übliche Konventionen halten, muß das aber natürlich nicht. Das heißt umgekehrt, daß z.B. in einem File mit der Endung .png nicht notwendig auch ein "png drinnen ist". Um herauszufinden, was eine Datei wirklich enthält kann man das Kommando `file` benutzen.

- Dateienamen können Leerzeichen und Umlaute enthalten, man sollte dies aber vermeiden. An manchen Stellen verursacht es einfach nur unnötige Probleme. Leerzeichen sollte man durch "_" ersetzen, wenn man für Dateien englische Namen vergibt, stellt sich das Problem mit den Umlauten nicht. Ansonsten kann man ein "ä" z.B. als "ae" schreiben. Während in reinen Linux-Umgebungen Umlaute noch verhältnismäßig problemlos sind, stolpert man spätestens, wenn man mit verschiedenen *Unix-Flavours* arbeitet, über derartige *funny chars*. Und

[1] Selbst auf Systemen, die das überhaupt nicht brauchen, wie z.B. OS/2, hat es sich nach wie vor als sinnvoll erwiesen, so zu verfahren.

zwar gewaltig! (Man sollte außerdem nicht davon ausgehen, daß es überall deutsche Tastaturen gibt.)

- Dateinamen dürfen wirklich 255 Zeichen lang sein. Im Interesse des Benutzers hat es absolut keinen Sinn, eine Datei z.B. "a.tex" zu nennen (auch wenn eine große, aber unbedeutende Softwarefirma der Ansicht war, daß 8 Zeichen reichen), es sei denn, man benötigt sie nur temporär und löscht sie zwei Minuten später sowieso wieder. Andererseits macht es aber auch keinen Sinn, in den Dateinamen gleich den ganzen Brief an die Großmutter zu schreiben...

  Hinweis: In einigen alten Unix-Büchern wird darauf hingewiesen daß Dateinamen kürzer sein müssen. Das war früher in der Tat der Fall. Die Betonung liegt hier auf "früher".

- Dateien können eine oder mehrere durch einen "." abgetrennte Erweiterungen besitzen. Gültig wäre also z.B. "Unixkurs.tex", aber auch "LaTeX_Archiv.tar.gz". V.a. letzteres ist sogar extrem üblich. .tar.gz ist üblicherweise ein gzip-komprimiertes Tar-Archiv.

- Unixprogramme erkennt bestimmte *Dateitypen* normalerweise über ihre Erweiterung, man sollte also bestimmte Endungen nur für bestimmte Dokumenttypen verwenden. Manche Programme sind nicht besonders intelligent bei den Möglichkeiten Dateien auszuwählen und erlauben nur bestimmte Extensionen.

- *Versteckte Dateien* sind ein Spezialität unter Unix. Häufig benutzt man versteckte Dateien, um die Konfigurationen bestimmter Programme zu sichern. Versteckte Dateien erzeugt man unter Unix, indem man den Dateinamen mit einem Punkt beginnen läßt. Z.B. wäre `.tcshrc` eine versteckte Datei. Normale Unix-Kommandos zeigen diese Dateien nicht an; wenn man aber weiß, daß sie da sind, kann man mit ihnen arbeiten wie mit allen anderen Dateien auch und es gibt durchaus Mittel und Wege sie auch im Dateisystem zu sehen.

Alle diese Dinge gelten im Übrigen auch für Verzeichnisse. Unix macht da keine großen Unterschiede, d.h., kann man ein Kommando auf eine Datei anwenden, so gilt das i.d.R. auch für Verzeichnisse.
Einige gebräuchliche *Extensionen* und ihre Bedeutung:

- `tex`: LaTeX-Dokumente

- `dvi`: Kompilierte LaTeX-Dokumente

- `ps`: PostScript Dateien
- `pdf`: Adobe Acrobat (Portable Document Format)
- `c, cc, cpp`: C bzw. C++ Quelltexte
- `f, for, f90`: Fortran Quelltexte
- `o`: Objektdateien
- `a.out`: Standardausgabe viele Compiler, das ist ein ausführbares Programm
- `gpi`: GNU Plot Eingabedateien
- `tar`: Archive, die mit `tar` eingepackt wurden
- `gz`: Mit `gzip` gepackte Dateien
- `z`: Mit `compress` gepackte Dateien
- `zip`: Mit `zip` gepackte Archive
- `jpg, gif, pnm, png`: Bilddateien
- `mp3`: MP3 Musikdateien
- `m3u`: MP3 Musikindizes
- `bz2`: bzip2-komprimierte Dateien
- `tar.gz, tgz`: gzip-komprimierte Tar-Archive
- `tar.bz2`: bzip2-komprimierte Tar-Archive

core-Dateien sind eine Besonderheit unter Unix. Sie werden erzeugt, wenn ein Programm aus welchem Grund auch immer abstürzt. Ihr Sinn liegt darin, dem Programmierer eine Möglichkeit zu geben, den Fehler zu lokalisieren. Für den normalen Anwender sind sie ohne Nutzen. Da sie i.d.R. sehr groß sind, sollte man sie daher löschen, es sei denn, man will sie selbst verwenden... will man das, muß hier nicht erklärt werden, was sie sind.

1.4 Rechte

Die Rechte einer Datei gliedern sich folgendermaßen:

1. Besitzer (= "User")
2. Gruppe (= "Group")
3. Andere (= "Other")

Um mit einer Datei arbeiten zu können, muß man die dafür nötigen Rechte besizten. Diese Rechte können sein:

- r = *read*: lesen
- w = *write*: schreiben
- x = *execute*: ausführen (nur für Programme)

WICHTIG: Während unter DOS, OS/2 und ähnlichen Systemen ausführbare Dateien einen bestimmten Namen haben müssen (z. B. mit der Erweiterung .exe enden), gibt es das unter Unix nicht. Ein Programm unterscheidet sich nur durch das Recht "execute" von einer normalen Datei. Da dies potentiell ein Sicherheitsrisiko ist, steht in normalen Unix-Umgebungen der Punkt (also das aktuelle Verzeichnis) nicht im Suchpfad für ausführbare Programme. Will man also ein Programm im aktuellen Verzeichnis ausführen, so muß man i.d.R. folgendes eingeben: `./configure`. Dies startet das Programm `configure` im aktuellen Verzeichnis.
Beispiele:

- `-rwxr-xr-x`: Eine Datei, die für den Besitzer sowohl les- als auch schreibbar ist und die ein ausführbares Programm darstellt. Andere Benutzer können diese Datei lesen und auch ausführen. Standard für auf dem System installierte Programme. Besitzer ist dort i.d.R. `root`, der auch der Gruppe `root` angehört.
- `-rw-r--r--`: Diese Datei kann der Besitzer lesen und schreiben, die Gruppe und alle anderen können sie nur lesen.
- `-rw-rw----`: Hier kann sowohl der Besitzer als auch die Mitglieder der Gruppe die Datei lesen und schreiben, andere Benutzer haben keine Rechte.

- `drwx------`: Ein Verzeichnis (zu erkennen am `d`). Nur der Besitzer kann hineinwechseln, es lesen und schreiben. Das Verzeichnis in dem übliche Mailprogramme (`mutt` oder `pine` z.B.) ihre Post speichern hat i.d.R. diese Rechte. Post geht niemanden anderen etwas an. Der Administrator (root) kann solche Dateien und Verzeichnisse aber selbstverständlich trotzdem lesen!

Spezialitäten:

- Um in ein Verzeichnis wechseln zu können, muß man für dieses Verzeichnis "execute"-Rechte haben.
- Um den Verzeichnisinhalt lesen zu können, muß man Leserechte besitzen.
- Um ein Programm auszuführen, benötigt man sowohl Lese- als auch execute-Rechte.

Dies führt bisweilen zu einiger Verwirrung. Besitzt man für ein Verzeichnis z.B. keine Leserechte, aber durchaus execute-Rechte, so kann man dort hineinwechseln. Es ist aber eine Sackgasse. D.h., man kommt nicht in weitere Unterverzeichnisse und kann sich auch den Inhalt nicht anzeigen lassen. Ausführbare Dateien, für die man keine Leserechte hat, haben ebenfalls keine Nutzen. Man könnte sie zwar ausführen, aber dazu muß der Computer ja ihren Inhalt lesen...
Wenn man der Besitzer einer Datei oder eines Verzeichnisses ist, kann man die Rechte mit dem Befehl `chmod` ändern. Der Befehl nimmt Anweisungen in zwei Formaten entgegen: einerseits kann man einzelne Einträge ändern; z.B. gibt `chmod g+w test.txt` der Gruppe (`g`) Schreibrechte (`w`) für die Datei `test.txt`. Andererseits kann man alle Rechte gleichzeitig festlegen, indem man eine dreistellige Zahl angibt, in die Stellen für die Rechte von Besitzer, Gruppe bzw. anderen stehen. Die einzelnen Stellen sind Zahlen von 0 bis 7, die sich aus der Addition der einzelnen Rechte ergeben: ausführen gibt +1, schreiben +2 und lesen +4. `chmod 754 test.txt` würde die Rechte also auf `-rwxr-xr--` setzen.

2 Die Shell

Die Shell ist im Wesentlichen die Befehlszeile eines Unix-Systems. Es gibt verschiedene Shells, die sich alle in ihrer Bedienung und ihrem Funktionsumfang unterscheiden (z.B. `tcsh, bash, csh, ksh, zsh, ash`). Die grundlegenden Eigenschaften sind aber bei allen gleich. Als Beispiel wird hier die

tcsh verwendet, das sollte man aber eigentlich an fast keiner Stelle wirklich merken.
Wie bekommt man nun eine Shell? Loggt man sich an einem Unix-System ein, startet heute i.d.R. eine graphische Benutzeroberfläche. Je nach System gibt es dort irgendeine meist sehr einfache Art, eine Shell zu öffnen. Meist klickt man mit der Maus ein Symbol in irgendeiner Art Startleiste an. Üblicherweise stellt das Symbol in irgendeiner Ausprägung einen Monitor dar. Hat man ein Textmenü, so firmiert die Shell oft unter der Bezeichnung *Terminal* oder unter dem Namen der jeweiligen Shell. Ggf. muß man etwas suchen gehen. Gewonnen hat man, wenn man sie gefunden hat.
Loggt man sich an einer Unix-Maschine hingegen an der Konsole direkt ein, d.h. im "Textmodus" ohne graphische Benutzeroberfläche, so bekommt man die Shell automatisch. Unter Linux bekommt man eine Konsole mit den Tastenkombinationen `[CTRL]-[ALT]-[F1]` bis `[CTRL]-[ALT]-[F6]`. Mit `[CTRL]-[ALT]-[F7]` schaltet man üblicherweise wieder auf die graphische Oberfläche zurück. Bei anderen Unixtypen kann man i.d.R. bei einem graphischen Login den "Sitzungstyp" auswählen. "Text only" oder "Failsafe" führt dann i.d.R. zum gewünschten Ergebnis.

2.1 Das Prinzip

Unix bildet alle Operationen in seinem *Dateisystem* ab. D.h. alles, was man tut, hat auf die eine oder andere Art mit irgendeiner Datei zu tun. In der Tat verbringt man die meiste Zeit damit, Eingabedateien zu erstellen, aus denen dann Programme die gewünschten Ausgaben produzieren. Das ist auf allen Systemen so, nur sieht man dies bei Unix besonders deutlich. Das Arbeiten mit Eingabedateien für Programme, die dann Ausgabedateien erzeugen, wird bisweilen auch als *Batch* bezeichnet. Hat man sich erst einmal daran gewöhnt, ist diese Art der Verarbeitung den heute üblichen interaktiven Programmen (also Programmen, die direkt mit dem Benutzer kommunizieren wollen) in vielen Bereichen überlegen. V.a. eignet sie sich hervorragend, um bestimmte Dinge zu automatisieren. Und schließlich soll ja der Computer die Arbeit machen, nicht der Benutzer. Jedenfalls war das mal so gedacht...

2.2 Die wichtigsten Befehle

Ein grundlegender Unterschied zwischen Unix und anderen Betriebssystemen ist, daß sich Unix komplett (und im Wesentlichen sogar nur) durch eingegebene Kommandos bedienen läßt. Um das System zu benutzen, benötigt man prinzipiell weder eine graphische Benutzeroberfläche noch eine Maus. Die Tastatur reicht vollkommen aus. Der nächste Unterschied besteht darin, daß

es unter Unix sehr viele Kommandos gibt, die alle jeweils nur eine einzige kleine Aufgabe erfüllen können, diese aber perfekt. So kann es durchaus passieren, daß man z. B. für die Umwandlung eines Bildes in ein anderes Format 5 bis 10 einzelne Programme hintereinander bemühen muß von denen jedes einzelne einen kleinen Schritt vom Ausgangs- zum Endzustand vollzieht. Diese Eigenschaft trägt Unix im Allgemeinen den Ruf ein, nicht besonders benutzerfreundlich zu sein. Aber eigentlich bedeutet das nur, daß Unix etwas wählerisch bei der Auswahl seiner Freunde ist.
Hier nun die wichtigsten Befehle als Tabelle:

apropos		Listet alle vorhandenen Handbuchseiten zu einem Thema auf
chmod	change mode	Ändert die Rechte einer Datei
cp	copy	Kopiert Dateien
df	disk free	Listet den freien Platz auf allen Dateisystemen
diff	diffrence	Zeigt Unterschiede zwischen Dateien
du	disk usage	Listet den Platzverbrauch einzelner Verzeichnisse
dvips		Konvertiert `dvi`-Dateien in PostScript-Dateien
exit		Beendet die Shell
find		Findet Dateien und Verzeichnisse
g77		Startet den GNU Fortran77-Compiler
g++		Startet den GNU C++-Compiler
gcc		Startet den GNU C-Compiler
grep		Sucht nach dem Vorkommen eines Ausdrucks in Dateien
gv	Ghost View	Zeigt PostScript-Dokumente an
ggv	Ghost View	Zeigt PostScript-Dokumente an (GNOME)
gvim		Startet einen graphischen vim
gzip		Packt eine Datei zusammen
info		Zeigt Handbuchseiten an (ergänzend zu `man`)
kill		Tötet einen Prozess
latex		Startet den LaTeX-Compiler
less		Zeigt eine Datei an
ln	link	Legt Links (Verweise) im Dateisystem an
lp	line printer	Druckt eine PostScript-Datei aus
ls	list	Zeigt den Inhalt eines Verzeichnisses an
man	manual	Zeigt eine Handbuchseite an
mkdir	make directory	Erzeugt ein neues Unterverzeichnis
mv	move	Verschiebt Dateien und Verzeichnisse
ps		Zeigt die laufenden Prozesse an
psnup		Formatiert PostScript-Dokumente um
pstree		Zeigt die laufenden Prozesse als Baum
pwd	print working dir	Gibt das aktuelle Verzeichnis aus
rm	remove	Löscht Dateien
rmdir	remove dir	Löscht leere Verzeichnisse
rwho	remote who	Zeigt alle am Cluster eigeloggten User
tar		Archviert Dateien und Verzeichnisse
top		Zeigt die laufenden Prozesse an
vi		Startet den Editor `vi`
vim		Startet den Editor `vim`
who		Gibt eingeloggte Benutzer aus
zip		Packt Dateien zu einem Archiv

Alle diese Befehle sehen zunächst sehr harmlos aus. Allerdings kann man fast jedem dieser Befehle eine Unzahl sog. *Parameter* mitgeben, die sein Verhalten verändern. Parameter haben dabei nach dem *POSSIX*-Standard folgende Form:

- Einbuchstabige Parameter: Sie werden mit einem "-" abgetrennt. Mehrere einbuchstabige Parameter können zusammengefaßt werden.

  Beispiele:

  - `ls -l`: Long List
  - `ls -al`: Long List all Files

- Mehrbuchstabige Parameter werden mit "`--`" abgetrennt. Danach folgt der Parameter. Beispiel: `vim --cmd mycommand`

- Zeichenketten, die Leerzeichen enthalten, werden in Anführungszeichen gesetzt. Wichtig: Zwischen den durch "-" abgetrennten Prametern stehen ebenfalls Leerzeichen, hier werden aber keine Anführungszeichen gesetzt. Beispiel: `grep Mein TextMyfile.tex`

- Dateinamen werden dem Kommando einfach angehängt. Beispiel: `latex Myfile.tex`. Die meisten Kommandos operieren auf Dateien und erwarten deswegen einen Dateinamen als letztes Argument. Gibt man keinen Dateinamen an, so lesen die Kommandos statt dessen von der Standardeingabe, also der Tastatur. Dies ist immer dann der Fall wenn ein Kommando genau ein Eingabefile erwartet. Benötigt ein Kommando mehrere Eingabedateien (wie z.B. `diff`), so kann man das lesen von der Standardeingabe oft durch den Parameter "-" erreichen. Insbesondere im Zusammenhang mit der Eingabeumleitung ist dies oft interessant.

- *Wildcards* oder *Joker*: Sie ersetzen beliebige Teile von Dateinamen. Viele Unix-Kommandos erlauben hier sog. *Regular Expressions*, die hier nicht behandelt werden sollen (führt zu weit. Siehe. z.B. "Reguläre Ausdrücke" von Jeffrey Friedel, erschienen bei O'Reilley). Für den Hausgebrauch reicht es zu wissen, daß ein Sternchen "*" für eine beliebig lange Kette beliebiger Zeichen steht und ein Fragezeichen "?" einen beliebigen einzelnen Buchstaben repräsentiert.

  Weiterhin sollte man im Kopf behalten, daß bei einem Kommando wie z.B. `ls *` nicht ein "*" an den Befehl `ls` übergeben wird, sondern bereits die Shell den "*" interpretiert und durch alle Dateinamen ersetzt, die im aktuellen Verzeichnis vorhanden sind. D.h. hier ist die Shell dasjenige Unix-Kommando, das die regulären Ausdrücke und Wildcards

interpretiert! Aus diesem Grund führt z.B. `locate *.tex` i.d.R. zu keinem Erfolg. Die Shell wird "*.tex" bereits durch alle Dateien ersetzen die darauf passen, und das Ergebnis dann an `locate` übergeben. Gibt es im aktuellen Verzeichnis z.B. ein File "a.tex" so ist `locate *.tex` absolut identisch zu `locate a.tex` und wird folglich nur dieses File finden. In diesem Beispiel will man aber ja alle tex-Files auf dem System finden, d.h. dem `locate`-Befehl soll wirklich "*.tex" übergeben weren, die Shell soll dies nicht expandieren. Um diese zu verhindern kann man Jokerzeichen "escapen". Dies geschieht indem man ihnen einen Backslash
voranstellt. Das gewünsche Ergebnis erhlält man also mit `locate .tex`. Alternativ kann man den Ausdruck auch in Anführungszeichen setzen, auch dann wird die Zeichenkette literal d.h. ohne Ersetzungen ("genau so wie sie dasteht") an das Kommando übergeben.

Beispiele:

– `*.tex`: trifft alle Dateien, die die Erweiterung .tex tragen. Also z.B. MyFile.tex, bla.tex, blubb.ps.tex, nicht aber z.B. MyFile.TeX (Unix ist case-sensitive!!!).
– `MyFile?.tex`: Trifft alle Dateien, die `MyFile` heißen, dann einen beliebigen Buchstaben, eine Ziffer oder was auch immer enthalten und auf .tex enden. Z.B. also `MyFile1.tex, MyFile_.tex`, aber auch `MyFile.tex`.

Bis man sich ein daran gewöhnt hat, ist gerade das Matching auf diese Wildcards etwas seltsam. Es hilft hier, bevor man einen gefährlichen Befehl wie `rm` eingibt, einfach das *Pattern* mit einem ungefährlichen Befehl wie `ls` zu testen.

Es sei auch noch darauf hingewiesen daß Unix Wildcards auch für Verzeichnisnamen erlaubt. Eine Angabe wie `/home/*/bin/` ist also durchauch gültig und würde alle Verzeichnisse ansprechen die diesen Ausdruck erfüllen, d.h. in diesem Fall alle bin-Verzeichnisse die die jeweiligen User in ihrem Home angelegt haben.

Daran sieht man schon, daß man mit Parametern einiges erreichen kann, wobei hier noch kein einziger programmspezifischer Parameter vorgestellt wurde. Alleine der doch so simple Befehl `ls` zum Anzeigen des Verzeichnisinhalts hat über 50(!) Parameter. Da dieser Befehl relativ häufig verwendet, wird hier ein paar gebräuchliche und hilfreiche Ausführungen:

- `ls -al`: Liefert eine lange Liste aller Dateien, also auch der versteckten Dateien.

- `ls -d .[a-zA-Z]*`: Listet alle versteckten Dateien. Dies ist ein Beispiel für einen regulären Ausdruck. Die Eckigen Klammern bezeichnen dabei einen set, also eine Menge von Zeichen die vorkommen dürfen (hier alle Groß- und Kleinbuachstaben), das Sternchen am ende bedeutet "beliebig viele davon".
- `ls-F`: Eigentlich keine Abart von `ls`, sondern ein in der `tcsh` implementiertes Kommando. Es listet die Dateien farbig, für jeden Typ eine passende Farbe. Man kann so leicht ausführbare Dateien von Verzeichnissen unterscheiden. Da `ls-F` eine Eigenschaft der *tcsh* ist, funktioniert das auf jeder Plattform mit dieser Shell. Unter *Linux* erhält man ein äquivalentes Ergebnis mit `ls --color=tty`. Das setzt allerdings voraus, daß der `ls`-Befehl aus den GNU Fileutilities verwendet wird. Viele kommerzielle Unixe kommen mit ihrem eigenen Befehlssatz, der das nicht kann.

Alle Parameter hier zu erklären wäre sinnlos, und merken kann sie sich eh' niemand. Deswegen ist einer der wichtigsten Befehle `man`. Ihm gibt man i.d.R. nur einen einzigen Parameter mit: Den Befehl, zu dem man weitere Informationen wünscht (das $-Zeichen bedeutet dabei nur die Eingabeaufforderung der Shell):

```
$ man ls
```

daraufhin erscheint ein Dokument, das einem fast alles über den Befehl `ls` erzählt – meist mehr, als man wissen will. Üblicherweise funktioniert dieses Verfahren mit jedem Befehl, den Unix kennt. Hat man also erst einmal gefunden, welchen Befehl man benötigt, dann hat man bereits gewonnen: man fragt `man`, wie man ihn benutzt und dann tut man das einfach einmal. Ist man sich unsicher, und v.a. auch dann, wenn der Befehl ein mühsam erzeugtes Dokument verändert (wie z.B. `psnup`), dann probiert man den Befehl erst einmal in einem temporären Verzeichnis mit einer Kopie der Datei aus und sieht, welches Ergebnis man bekommt. Unix fördert hier durchaus den Spieltrieb. Da man unter Unix einfach nichts kaputt machen kann, was das System zu seinem Betrieb benötigt, ist das auch überhaupt kein Problem.

2.3 Ein- und Ausgabeumleitung

Was nun, wenn man nicht weiß, wie der Befehl heißt, den man benötigt? Wer kommt schon drauf, nach `psnup` zu suchen, wenn er zwei Seiten auf eine drucken möchte? Genau hier liegt das Hauptproblem des Unix-Hilfesystems. Diese Information ist nicht so einfach zu bekommen. Folgende Strategie bietet sich an:

```
$ apropos ps
```

fragt Unix, was ihm denn so zu “ps” einfällt. Das ist eine Menge, und es reicht von Referenzen in diversen Programmiersprachen bis zu `zipsplit`. Ein guter Ansatz, aber wie findet man in diesem Wust, was man braucht?

```
$ apropos ps | grep "multiple pages"
```

Ah! Zwei Kommandos auf einmal. grep sucht einen Ausdruck (hier “multiple pages”, ohne die Anführungszeichen), aber worin sucht er hier? In der Ausgabe von `apropos ps`! Warum macht er das? Wegen einer sogenannten Pfeife, so jedenfalls wenn man Pipe übersetzt. Die Pipe ist dabei der senkrechte Strich. Obiges Kommando bedeutet also: Erzähle mir, was Dir zu ps so einfällt, dann durchsuche das Ergebnis nach dem Begriff “multiple pages” und zeige mir an, was da dann übrig bleibt. Dies ist einer der Fälle in dem ein Kommando mangels angegebenem Dateinamen auf der Standardeingabe operiert, und diese wird mit dem Inhalt eines anderen Programms (hier `apropos ps`) gefüllt. Das ergbit dann z.B. folgendes:

```
 $ apropos ps | grep "multiple pages"
psnup               (1)  - multiple pages per sheet
psresize            (1)  - multiple pages per sheet
```

Und siehe da: die letzten zwei Einträge sind das Gesuchte. Jetzt kann man sich aussuchen, ob man nun `psnup` oder `psresize` benutzen möchte, und man konsultiert die zugehörigen Manpages.

2.4 Umleitungen

Eine Pipe ist eine der möglichen Umleitungen für Ein- und Ausgabe. Daneben kennt Unix noch die Operatoren > und <. Diese stellt man sich am einfachsten wie einen Pfeil vor. > schreibt die Ausgabe eines Programms in die danach angegebene Datei, logischerweise liest < aus einer Datei. Z.B. könnte man obige längliche Ausgabe des `apropos`-Befehls auch so behandeln:

```
$ apropos ps > temp.txt
```

Dies würde die Ausgabe in die Datei `temp.txt` schreiben, und in dieser könnte man dann mit seinem Lieblingseditor nach den gewünschten Begriffen suchen gehen.
Mit Hilfe dieser drei Operatoren ist es möglich, komplexe Kommandos aus einzelnen einfachen Befehlen zusammenzubauen. Und daraus resultieren die

beliebten Übungsaufgaben der Unix-Einführungsvorlesung etwa in dem Stile: "Schreiben Sie einen einzeiligen Unixbefehl, der die Welt erschafft und danach Kaffee kocht."
Ein besonderer Fan dieser Art der Kommunikation ist das Kommando `find`. Z. B. kann man mit folgendem, leicht zu merkendem Kommando alle Backupfiles (erkennbar an der Tilde nach dem Dateinamen) entfernen:

```
 find ~ \( -name '*~' -o -name '#*#' \) -print0 |
xargs --no-run-if-empty --null rm -vf
```

Wie kommt man auf sowas? Nun, entweder man liest die Manpages von `find`, `xargs` und `rm` und bastelt es sich dann zusammen oder man liest `info find` und findet es unter "Common Tasks". Selber draufkommen wäre auch möglich, aber das ist für die großen Leute ;-)

2.5 Nettigkeiten

- Startet man an der Shell ein Kommando, das seinerseits wieder ein Fenster produziert, so kann man an der Shell keine weiteren Befehle mehr eingeben, bis das Kommando beendet ist. Nun kann Unix aber mehrere Programme gleichzeitig ausführen. Wie erreicht man das also?

  ```
  $ gv &
  ```

  Das nachgestellte & (*Ampersand*) sorgt dafür, daß ein Programm in den *Hintergrund* geschickt wird. D.h. es wird ausgeführt, und die Shell steht für weitere Eingaben zur Verfügung.

- Was tun, wenn man das & vergessen hat? Ganz einfach: Man drücke [CTRL]-Z, das suspendiert ein Programm, sprich es hält es an und man kehrt auf die Shell zurück. Dort schickt man es mit dem Befehl `bg` in den Hintergrund.

- Letzteres funktioniert prinzipiell mit jedem Programm. Startet man aber z.B. einen Compiler wie `latex`, so produziert dieser seitenweise Bildschirmausgaben, und man wird an der so wiedergewonnenen Shell nicht viel Freude haben, da diese immer, wenn sie anfallen, trotzdem ausgegeben werden. Die Lösung ist allerdings sehr einfach: *Ausgabeumleitung*

  ```
  $ latex myfile.tex  > out.log &
  ```

Allerdings sollte ein so gestartetes Programm keine Benutzereingaben mehr erfordern. Tut es dies doch, so bleibt es hängen, bis man es in den Vordergrund holt respektive tötet. Gerade der LaTeX-Compiler, der, sobald er einen Fehler findet, mit dem Benutzer mal drüber sprechen möchte, eignet sich hierfür nicht so besonders. Allerdings hat LaTeXdruchaus einen Parameter, der ihm abgewöhnt, den Benutzer zu Fragen. (Übungsaufgabe: Wie findet man wohl diesen Parameter?)

- Hat man nun ein Programm, das sehr lange Zeit benötigt um seine Arbeit zu verrichten, so kann man dieses auch weiterlaufen lassen wenn man sich längst ausgeloggt hat. Hierfür dient das Kommando `nohup`. `nohup mycalc` würde also `mycalc` in den Hintergrund schicken und es weiterlaufen lassen wenn man sich danach ausloggt.

- Da Unix-Systeme von mehreren Leuten benutzt werden ist es `nice` wenn man Programmen im Hintergrund eine niedrigere Priorität zuordnet als den Programmen die vom Benutzer im Vordergrund benutzt werden. Das führt dazu, daß das eigene Programm zwar etwas langsamer abgearbeitet wird, da es weniger Rechenzeit bekommt, dafür antwortet das System für den Bentzer an der Konsole aber zügiger. Um das zu erreichen dient der Befehl `nice`. Diesem gibt man eine Zahl zwischen 0 und 19 mit die den Nice-Level festlegt. Das Vorzeichen für den Nicelevel ist allerdings leider implementationsabhängig. Man muß hier die Manpage konsultieren.

  Beachte: in der `tcsh` ist `nice` ein sog. "Shell Builtin", hier gilt also nicht die manpage von `nice` sondern die der `tcsh`, da diese das Kommando intern ausführt. Die Syntax ist hier `nice +[number] command`)

- *Tab Expand* beherrscht fast jede Shell mehr oder weniger ausgeprägt. Jeder Kommando-, Datei- oder Verzeichnisname kann hierbei von der Shell durch Drücken der [TAB]-Taste komplettiert werden. Allerdings gibt es leichte Unterschiede, wie das von Shell zu Shell funktioniert. Manche Shells geben nur ein Tonsignal, wenn der bisher eingegebene Name nicht eindeutig ist, andere listen alle auf diesen Namen passenden Einträge auf, manchen Shells kann man sagen, daß mit einem bestimmten Kommando nur einzelne Dateitypen expandiert werden (z.B. mit `gv` nur PostScript oder PDF-Dateien). Grundsätzlich kennen aber alle gebräuchlicheren Shells einen solchen Modus. Bisweilen ist es auch nicht die [TAB]-Taste die man drücken muß. Moderne gebräuchliche Shells (`tcsh, bash, zsh`) handhaben das allerdings so und den anderen kann man es beibringen.

In diesem Zusammenhang sei erwähnt, daß heute gebräuchliche Shells auch ein Durchsuchen der History erlauben. In der `bash` bekommt man diese Funktion mit [CTRL-R] automatisch. Man kann dann anfangen zu tippen und der jeweils erste passende Befehl wird in die Kommandozeile kopiert. Mit [ENTER] kann man diesen dann ausführen oder mit den Cursortasten editieren. Um in der `tcsh` eine analoge Funktion zu bekommen muß man folgende Zeilen zu seiner `.tcshrc` hinzufügen:

bindkey -b C-r i-search-back

bindkey -b C-R i-search-back

- Der *Kommandozeileneditor* erlaubt das Editieren vorhandener Kommandos. Mit den Cursortasten Up/Down kann man hierbei in der *History*, also der Liste eingegebener Kommandos, blättern. Paßt ein Befehl genau, kann man ihn einfach sobald, er angezeigt wird mit [ENTER] übernehmen. Paßt er so ungefähr, kann man ihn editieren. Die meisten Shells erlauben dieses Editieren in mindestens zwei Modi: *Emacs* oder *vi-Keybinding*. D.h. die Shell verhält sich so wie die unter Unix gebräuchlichsten Editoren. Kennt man sich mit einem der beiden gut aus, so ist dies ein sehr effizienter Weg, Kommandos erneut einzugeben.[2]

- *Aliase* sind nützliche Abkürzungen für komplizierte Kommandos, die man immer wieder braucht. Diese Aliase können relativ komplex sein, wie folgendes Beispiel zeigt:

```
alias r    "rwho|cut -c1-15 |uniq|grep ''\!*|
awk '{do{x = "'$1;y = $0;ch=getline;while($1==x && ch==1)
{y=y " " $2;ch=getline};print y} while(ch==1)}'"'"
```

(wobei das wohl der Grenzfall zum Einsatz eines kleinen Shellscripts ist. Anmerkung: Alles steht in einer Zeile!). Andere Aliase sind sehr einfach:

```
alias vi     "vim"
alias gvi    "gvim"
alias se     "vim -S Session.vim"
alias gse    "gvim -S Session.vim"
```

[2]Möchte man ein funktionierendes vi-Keybinding, sollte man sich vielleicht mit der `tcsh` vertraut machen. Die `bash` hat damit ihre Probleme, v.a. mit der History-Funktion.

```
alias di     "vimdiff"
alias gdi    "gvimdiff"
alias vr     "gvim --remote "
alias 2a4    "psnup -pA4 -2 "
alias 2p     "lp -od "
```

Wenn man sich selber Aliase definieren möchte, kann man sie in das Konfigurationsfile der benutzten Shell schreiben, was uns gleich zum nächsten Punkt bringt:

- Konfiguration: Jeder Benutzer bekommt beim Einrichten seinen Accounts eine Standardkonfiguration zugewiesen, die systemweit gilt. Diese Konfiguration kann aber selbstverständlich ergänzt oder sogar überschrieben werden. (Bei Letzterem sollte man wissen, was man tut...) All dies geschieht durch Einträge in den Konfigurationsdateien. Konfigurationsdateien sind unter Unix üblicherweise *versteckte Dateien*, beginnen also mit einem Punkt. Eine gebräuchliche Konvention für ihren Namen ist `.<programmname>rc`. Folglich heißt das Konfigurationsfile für die `tcsh` `.tcshrc`, dasjenige für die `bash` `.bashrc` usw. Daneben gibt es eine Reihe anderer Dateien, die beim Login oder beim Öffnen einer Shell ausgewertet werden. Welche dies im Details sind, steht in den jeweiligen Manpages. Die wichtigsten sind:

  Für die `bash`:

  - `.bash_logout`
  - `.bash_profile`
  - `.bashrc`

  Für die `tcsh`:

  - `.cshrc`
  - `.tcshrc`
  - `.history`
  - `.login`

  Sonstige:

  - `.signature`: Unterschrift unter eMails
  - `.inputrc`: Für die Tastaturbelegung
  - `.Xmodmap`: Tastaturbelegung in X

– `.vimrc`: Konfigurationsdatei für vim
– `.xsession`: Einstellungen zur X-Session, wird unter Linux aber i.d.R. nicht benutzt

2.6 Verbindungen zur großen weiten Welt

Unix ist von seiner Anlage her ein sehr kommunikatives System. Unix-Rechner reden ungemein gerne miteinander [3], und es ist problemlos möglich, sich von jeder Unix-Maschine aus auf jeder anderen einzuloggen und dort zu arbeiten, vorausgesetzt man hat dort eine gültige Benuzter-ID und ein Passwort. Mit

```
$ ssh joe@host1.unix.dom
```

würde man sich als Benutzer "joe" an der Maschine "host1.unix.dom" anmelden. ssh, die *Secure Shell*, sorgt dabei für die Verbindung und eine Verschlüsselung der übermittelten Daten, so daß niemand mitlesen kann. Hat man sich auf der fernen Maschine eingeloggt (i.d.R. wird man noch nach dem Passwort gefragt), kann man dort an der Shell genauso arbeiten wie auf der lokalen Maschine. Man kann jedes Programm starten und ausführen, seine Ausgaben werden automatisch lokal angezeigt.
Warum will man das? Z.B. wenn ein Programm, das man gerne verwenden würde, sehr groß ist, sehr viel Speicher oder Rechenleistung benötigt, die lokale Maschine dieses aber nicht zur Verfügung stellt, so kann man sich auf diese Weise auf einer schnelleren Maschine einloggen und diese benutzen. Oder wenn es ein Kommando vor Ort gar nicht gibt, das aber auf einer anderen Maschine installiert ist. Oder wenn man unterwegs ist und die Emails lesen will, die inzwischen daheim aufgelaufen sind.
Für die Verwendung von `ssh` gibt es einige gute Gründe und man sollte heute nicht mehr das früher verbreitete `telnet` benutzen. Zu den Details s. a. die Manpages von ssh. Hier findet man auch hinweise dazu wie man über ssh das ausführen von Kommandos auf anderen Hosts automatisieren kann, wie man die Eingabe von Passwörtern über Schlüssel vermeidet usw.

3 X-Window

X-Window, kurz X, (das "fehlende" s ist kein Druckfehler) ist die graphische Benutzeroberfläche von Unix. Eigentlich ist das nicht ganz korrekt: X dient dazu, daß Unix Graphiken anzeigen kann; die Benutzeroberfläche selbst wird in bester Unix-Manier von zig einzelnen Programmen zusammengebaut.

[3] ein Punkt, der Sicherheitsfragen unter Unix nicht gerade vereinfacht

Z. B. dient ein sog. Windowmanager dazu, daß angezeigte Fenster einen Rahmen bekommen, sich verschieben und schließen lassen usw. Diese technischen Details sollen hier nicht interessieren, all das wird normalerweise vom Systemadministrator so eingerichtet, daß es funktioniert, und man ändert es nur, wenn man bestimmte Funktionen haben möchte oder eine bestimmte Umgebung bevorzugt.
Die Bedienung von X selbst ist dann denkbar einfach. Sie funktioniert genau so, wie es die meisten Leute von Windows her kennen[4]:

- Fenster verschiebt man, indem man die Titelleiste mit der linken Maustaste anklickt und zieht.
- Vergrößern und Verkleinern funktioniert, indem man den Rahmen packt und zieht.
- Schießen? Nun ja, ein Doppelklick auf das Symbol oben links im Fensterrahmen.
- Vollbild und wieder zurück sowie Minimieren übernehmen die Symbole oben rechts.

Je nach Windowmanager gibt es dann noch weitere Schaltflächen in den Fensterrahmen. Üblich ist z.B. ein Schließen-Button. Am einfachsten lernt man den Umgang mit X, wenn man einfach ein Terminal-Fenster öffnet und ausprobiert, was passiert, wenn man hierhin und dorthin klickt.
Ein wichtiger Punkt ist die Zwischenablage. Mit der linken Maustaste kann man in jedem beliebigen Programm Textteile markieren. Es gibt keinen Befehl, um sie in die Zwischenablage zu kopieren, das geschieht automatisch. Auch zum Einfügen gibt es i.d.R. keinen Befehl. Man klickt einfach mit der mittleren Maustaste an die Stelle, an der der Text eingefügt werden soll. Hat eine Maus nur zwei Tasten, so klickt man mit beiden gleichzeitig. Leider kann man über die X-Zwischenablage nur Text kopieren, keine anderen Objekte wie z.B. Graphiken. Bei komplexeren graphischen Oberflächen wie z.B. GNOME wird sich dies aber in absehbarer Zeit ändern.
Was kann X noch? Die Konzeption von X ist in weiten Teilen dafür verantwortlich, daß man von jedem beliebigen Rechner aus auf jedem beliebigen anderen Rechner arbeiten kann. X implementiert hier ein sog. *Client-Server-Modell*. Das ist alles technisch ziemlich interessant, interessiert hier aber nicht.

[4]Wobei hier die Verhältnisse eigentlich umgekehrt liegen. X gab es schon länger, als es DOS gibt. Die aktuelle Version von X ist 11 Release 6, insofern haben sich andere Benutzeroberflächen wohl eher an X orientiert als X an anderen Oberflächen.

Was man noch wissen sollte ist, daß man das Aussehen der Fenster konfiguriert in dem man den "WindowManager" einstellt. Übliche Windowmanager sind `kwm` unter KDE, `sawfish` unter GNOME sowie der traditionelle fvwm2. Einen sehr genügsamen und doch sehr praktischen Windowmanager stellt auch `WindowMaker` dar, der sehr gut auch mit KDE oder GNOME zusammenarbeitet. In den Manpages oder auf den Webseiten der jeweiligen Programme findet man Details dazu wie man sie konfigurieren kann.
Die Kommerziellen Unixflavours (HP-UX, Tru64, Solaris) setzten oft noch CDE ein, das "Common Desktop Environment". Deshalb sehen sich diese Unix-Systeme extrem ähnlich. Allerdings muß man CDE schon mögen. Möchte man auf seinem heimischen Linux eine ähnliche Oberfläche haben, so sollte man sich xfce ansehen.

4 Der Editor vi(m)

vi: /V-I/, *not* /vi/ and *never* /siks/ n. [from 'Visual Interface']
A screen editor crufted together by Bill Joy for an early BSD release. Became the de facto standard UNIX editor and a nearly undisputed hacker favorite outside of MIT until the rise of EMACS after about 1984. Tends to frustrate new users no end, as it will neither take commands while expecting input text nor vice versa, and the default setup provides no indication of which mode the editor is in (one correspondent accordingly reports that he has often heard the editor's name pronounced /vi:l/). Nevertheless it is still widely used (about half the respondents in a 1991 Usenet poll preferred it), and even EMACS fans often resort to it as a mail editor and for small editing jobs (mainly because it starts up faster than the bulkier versions of EMACS). See "holy wars"

`vi` ist ein Editor, mit dem sich heute nur relativ wenige Leute noch anfreunden wollen. Er steht in dem Ruf, nicht besonders benutzerfreundlich zu sein.[5] Auf den ersten Blick ist das nicht ganz unrichtig. Allerdings gibt es ein paar gute Gründe, zumindest Grundzüge dieses Editors zu beherrschen:

- `vi` gibt es immer. Auf jedem Unixsystem der Welt ist immer ein vi installiert, er gehört zum Minimalumfang eines Unix-Systems. Bei anderen Editoren muß das nicht so sein.

- `vi` ist klein, d.h. er läuft auch auf einer langsamen Maschine mit wenig Speicher oder über eine langsame Wählverbindung ohne Probleme

[5] Freundet man sich mit ihm an, ist er aber ein sehr leistungsfähiges, Werkzeug und das Ergebnis ist i.d.R. eine innige Beziehung.

- `vi` benutzt normale Buchstaben für alle seine Funktionen. D.h. selbst durch das lausigste Windows-Terminal hindurch kann man damit auf einem Unix-System jederzeit arbeiten. Bei *Sondertasten* wie den Cursor-Tasten, CTRL- und ALT-Tastenkombinationen ist das nicht immer möglich, Editoren, die hiervon heftigsten Gebrauch machen (z.B. *emacs*) verbieten sich daher für diesen Zweck oft.

Weiterhin kann man mit vi(m) unglaublich schnell arbeiten, wenn man sich mal an ihn gewöhnt hat. Dies liegt hauptsächlich genau an der Tatsache, daß er für jeden Befehl eine äquivalente Buchstabenkombination besitzt. Gerade wenn man maschinenschreiben beherrscht gibt es kaum ein Programm mit dem man ähnlich schnell editieren kann wie mit vi.

4.1 Grundlagen

vi ist der erste "visuelle" Editor unter Unix. Sein Vorgeänger `ed` hatte diese Funktionalität nicht. Das bedeutet: mit `vi` sieht man das File, das man editiert! Betrachtet man die Geschichte von Unix, so ist das ein nicht zu verachtendes Feature. (Auch sind die meisten Leute der Ansicht, daß es die arbeit ungemein vereinfacht ;) Die ersten Computer hatten keinen Monitor, sondern haben alle Ausgaben auf Papier ausgedruckt. Hier ist natürlich dann ein visueller Editor etwas fehl am Platze. (Aus dieser Zeit resultiert auch die alte Unix-Regel: "No News is Good News", also sagt ein Programm nichts, hat es alles so gemacht wie aufgetragen. (`rm -rf *` löscht deshalb ohne jede Ausgabe alle Files in allen Unterverzeichnissen.)
Der nächste wichtige Unterschied zwischen vi und anderen Editoren ist, daß vi verschiedene Modi besitzt. Neulinge treibt gerade das bisweilen zur Weißglut. Dabei ist alles eigentlich total logisch:

- *Normal Mode*: In diesem Modus befindet sich vi, wenn man ihn startet. Jeder Tastendruck wird hier als Befehl interpretiert, d.h. startet man vi, muß man zunächst den Modus wechseln, um etwas schreiben zu können! Das stößt oft auf gewisses Unverständnis.

- *Insert Mode*: In diesem Modus schreibt man Text. Alles, was man tippt, wird hingeschrieben.

- *Command Line Mode*: Dieser öffnet eine Kommandozeile. Man benötigt sie für Befehle, die aus mehr als einem Buchstaben bestehen.

- *Visual Mode*: Eine Spezialität von vim, hiermit markiert man Blöcke. WICHTIG: ein "normaler" vi hat diesen Modus nicht! (Was nicht bedeutet, daß es dort keine Blockoperationen gäbe.)

Angemerkt sei, daß vi normalerweise den Modus nicht anzeigt. Drückt man ein paar mal ESC, kommt man aber immer zurück in den Normal Mode. Als Fehlermeldung piept er mehr oder weniger freundlich, so auch wenn man im Normal Mode ESC drückt. “Da bin ich doch schon, was willst Du denn?”
Hier nun eine Tabelle der wichtigsten `vim`-Befehle (manche davon sind im original `vi` nicht vorhanden):

A	Append	springt an das Ende der Zeile und schaltet in den Insert-Mode
I	Insert	springt an den Anfang der Zeile und schaltet in den Insert-Mode
i	Insert	schaltet in den Insert-Mode
ESC		schaltet in den Normal-Mode
J	Join	verbindet zwei Zeilen zu einer
x		löscht das Zeichen unter dem Cursor wie `[DEL]`
X		löscht “rückwärts” wie `[Backspace]`
dd	delete	löscht die aktuelle Zeile
D	delete	löscht bis zum Ende der Zeile
dw	delete word	löscht ein Wort
n G	Go	springt in Zeile n
G	Go	springt in die letzte Zeile
/*pattern*		sucht vorwärts nach *pattern*
?*pattern*		sucht rückwärts nach *pattern*
$		springt an das Ende der Zeile
h		links
j		nach unten
k		nach oben
l		nach rechts
w	word	wortweise vorwärts
b	back	wortweise rückwärts
}		zum Absatzende
{		zum Absatzanfang
.		wiederholt den letzten Befehl
v	visual	schaltet in den Visual Mode,
		markieren von Text mit durch bewegen des Cursors
y	yank	kopiert den markierten Text in die Zwischenablage
Y	Yank	kopiert die aktuelle Zeile in die Zwischenablage
p	paste	fügt die Zwischenablage nach der Cursorposition ein
P	Paste	fügt die Zwischenablage vor der Cursorposition ein
:x	Exit	speichert das Dokument, beendet den Editor
:q	Quit	verläßt den Editor wenn nichts geändert wurde
:w	Write	speichert das Dokument
:wq	Write Quit	speichert das Dokument, verläßt den Editor
:r *file*	Read	liest *file* ein
:cd *dir*		wechselt das aktuelle Verzeichnis
:pwd		gibt das aktuelle Verzeichnis aus
:! *befehl*		führ den Befehl *befehl* aus
:help *Thema*		listet die Hilfeseite zu einem Thema
		(Leider nur in vim, da aber sehr gut!)

Das ist nur eine winzig kleine Auswahl der vorhandenen Befehle. In der Tat

ist es (zumindest bei vim) so, daß alle Tasten der normalen Tastatur außer dem Backslash einen gültigen Befehl darstellen (jeweils einmal groß, einmal klein, und i.d.R. auch einmal mit CTRL.). Man kann also über Jahre hinaus viel Spaß damit haben, immer neue Befehle zu entdecken, die einem diese oder jene Aufgabe vereinfachen.
Glücklicherweise benötigt man für die tägliche Arbeit nur einen Bruchteil der vorhandenen Fuktionen. In der Tat ist es so, daß man, statt sich hinzusetzen und 250 Funktionen auswendig zu lernen, sich besser mit den paar grundlegenden vertraut macht. Wie Bram Moolenaar in seinem hervorragenden Aufsatz zu erfolgreichem Texteditieren schon darstellt, ist der Trick zum effektiven Einsatz der, daß man die paar Befehle, die man braucht, so gut kennt, daß man nicht drüber nachdenken muß, sondern es einfach tut. Beherrscht man dann einige so gut, so kann man daran gehen, neue zu lernen, bis man diese ebenfalls so beherrscht. Andernfalls verliert man einfach viel zu viel Zeit damit, über den nötigen Befehl nachzudenken. Z.B. wird jeder Anfänger es schätzen, daß vim die Cursortasten beherrscht (tut lange nicht jeder vi auf jedem Betriebssystem, der HP-UX-vi z.B. kann das standardmäßig nicht...). Genauso wird aber dieser Anfänger später feststellen, daß die Befehle hjkl eigenltich wesentlich praktischer sind (im Ernst!).

4.2 Anwendungen

Wozu braucht man nun einen Editor? Sehr einfach: Um die zig Eingabe- und Ausgabefiles, die unter Unix üblich sind, zu editieren und auch, um sie zu betrachten. Überhaupt ist es i.d.R. keine schlechte Idee, eine Datei von der man so garnicht weiß, was sie tut und soll, einfach mal in einen Editor zu werfen und zu gucken, was da wohl drinnen steht. Alternativ kann man sie natürlich mit `less` angucken. Mit der Zeit wird man relativ schnell herausfinden, daß ein Editor sowas wie ein Schweizer Taschenmesser darstellt, mit dem man alles das behandelt, zu dem einem sonst nicht so viel einfällt. Er eignet sich übrigens auch hervorragend um sich "mal eben" einen langen Befehl irgendwo zu speichern. Einfach den Editor aufmachen, Befehl hineinkopieren, fertig.

4.3 Konfiguration

Gerade vim hat eine Unzahl von Konfigurationsmöglichkeiten, und kein Mensch (außer vielleicht Bram und Benji) kann sich die alle merken. Für den Anfang klaut man sich am einfachsten die Konfigurationdateien anderer Benutzer. Die von mir verwendeten Dateien kann man in einer immer halbwegs aktuellen Fassung von `http://www.stellarcom.org` herunterladen. Diese Dateien sind für vim Version 6 gedacht und kümmern sich insbesondere um LaTeX, Fortran, Perl, html, vim, PGP/eMail und ein paar andere Einstellungen. Insbesondere binden sie die diversen verwendeten Compiler ein.

5 Literatur

Zu Unix selber gibt es einiges, z. B.

1. Welsch/Kaufmann: *Running Linux.* Eine gute Einführung in Linux die viele Dinge behandelt die man auf anderen Unixsystemen ebenfalls gut anwenden kann. Die Systemverwaltung ist dort natürlich etwas anders, aber auf der Benutzerseite sind sich die meisten Unix-Systeme sehr ähnlich

2. Kernigham/Pike: *The Unix Programming Environment.* Aus der Zeit als Turnschhe noch aus Holz waren, sozusagen "alterwürdig" aber sehr lesenswert. Auch wenn der Titel nicht so aussieht, es handelt sich nicht um ein Handbuch zu irgendeiner Programmiersprache.

`http://www.linux-mag.com/1999-11/joy_01.html`

`http://www.moolenaar.net/habits.html`

The number of UNIX installations has grown to 10, with more expected.

(The UNIX Programmers Manual, 2nd Edition, June, 1972)

Index