Linux ohne Maus

Andreas Dunker

Vorwort

Selbstverständlich gibt es eine schicke graphische Oberfläche für Linux-Rechner - Server, Arbeitsplatzrechner oder sogar Raspberry Pi. Aber gerade, wenn der Linux-Rechner als Server eingesetzt wird, benötigt diese nur unnötig Ressourcen wie Speicher, Plattenplatz, CPU-Zeit, ... Und Server stehen oft in einer Ecke rum oder in Server-Räumen, die normalerweise abgeschlossen sind.
Jedoch kann der Anwender (z.B. ein Maker am Raspberry Pi) ausschließlich ohne graphische Oberfläche und nur mit der Tastatur bedienen. (Was nicht ausschließt, dass sich auch in der graphischen Oberfläche ein sog. Terminal öffnen läßt, mit die folgenden Aktionen durchgeführt werden können.)
Steht die Maschine irgendwo in der Ecke rum, so kann der Benutzer sich über eine sog. Remote-Verbindung mit dem Server verbinden und so den Server administrieren, Programme schreiben und sogar einige Spiele spielen.
Die folgenden Erläuterungen gelten für einen Raspberry Pi mit einem Raspbian-Wheezy- bzw. Raspbian-Jessie-Betriebssystem. Das Raspbian-Linux findet in fast gleicher Form auf vielen größeren Linux-Maschinen als Debian wieder. Und dieses ist im wesentlichen das gleiche wie ein RedHat-Betriebssystem (in einer älteren Version). Auch andere Linux-Distrubutionen wie SuSe oder Ubuntu verwenden die selben Befehle; sie unterscheiden sich nur "ganz unten" in den Tiefen des Betriebssystems, wo kaum ein üblicher Anwender oder Maker hinzuschauen braucht.
Wer sich also problemlos auf dem Raspberry Pi bewegen kann, kann sein Wissen auch auf großen Linux-Servern verwenden - ein Systemadministrator bei T-Systems, der Deutschen Bahn oder Google, der hunderte großer Server mit mehreren Dutzend Gigabyte und vielen Terabyte Festplatten verwaltet, benutzt die gleichen Befehle wie ein Maker, der seinen Raspberry Pi verwendet.
Wenn der Maker sich von seinem Klapprechner oder PC auf dem Raspberry Pi einloggen möchte so benutzt beim MacOS ein Terminal und den Befehl *ssh*, bei Windows das Programm *putty* oder *kitty*.
Schlüpfen wir also in die Rolle eines Makers, der lernen möchte, wie der Raspberry Pi in der hintersten Ecke des Arbeitszimmers bedient wird.

Einleitung

Hat sich der Maker auf dem Raspberry angemeldet, so wartet der Rechner mit einem sog. *Prompt* auf Befehle, z.B.:

```
maker@maker-pi-01:~$
```

In diesem Fall steht vor dem @ der Benutzername, zwischen @ und : der Rechnername und zwischen : und $ der aktuelle Pfad. Da sich der Prompt nach eigenem Gefallen anpassen läßt, verwenden wir die folgende Konvention:

1. Befehle, die als normaler Benutzer ausgeführt werden können, werden mit einem $ markiert (dies braucht nicht eingegeben zu werden!):

```
$ ls -l
```

2. Befehle, die als privilegierter Systembenutzer *root* eingeben werden müssen, werden mit einem Lattenkreuz # eingeleitet.

```
# nano /etc/motd
```

Solche Befehle dienen z.B. zur Einrichtung des Linux-Systems. Als Benutzer *root* hat man keine Einschränkungen, so dass der unaufmerksame Maker so sein System völlig vergurken kann. In der Makerszene gibt es heftige Diskussion, ob man dauerhaft als Benutzer *root* arbeiten darf oder nicht. Im professionellen Umfeld ist dies keine Frage: nur ausgewählte Mitarbeiter dürfen das *root*-Passwort kennen, und in vielen Firmen existiert die Richtlinie, dass nach der Administrator Abschluss *root*-Arbeiten den Account sofort wieder verlassen muss. Zu Hause kann jeder machen, was er oder sie will; trotzdem empfehle ich den *root*-Account nur zu benutzen, wenn es unbedingt notwendig ist.
Nun wartet also der Prompt darauf, dass der Maker Befehle eingibt. Als Schnittstelle zwischen Benutzer und Betriebssystem ist die sog. *Shell*. So wie eine Muschelschale die Muschel vor den dem Unbill des Meeres schützt, schützt die Linux-Shell das System vor den wirren Anwendern. Es gibt einige Shells. Die älteste nennt sich *sh* (*Bourne Shell*, benannt nach dem Entwickler). Hierher stammt auch die oben erwähnte Konvention. Dann gibt es noch *zsh*, *fish*, *ksh*, *csh*, ... Die standardmäßig eingerichtete Shell auf dem Raspberry Pi (und den meisten anderen Linuxen) ist die *bash* - die wiedergeborene Shell (*Bourne again shell*). Diese werden wir im folgenden auch verwenden.

KAPITEL EINS

bash

Programe ausführen

Der Maker fragt sich nun, wieso Programme ausgeführt werden, wenn er auf der Kommandozeile einen Befehl eingibt.
Zum einen gibt es Funktionen, die in die Shell eingebaut sind (*cd*, *pushd*, *history*, ...). Doch diese sollen uns hier nicht interessieren.
Die anderen (die meisten) Programme sind Dateien irgendwo im Dateisystem. Gibt der Maker nun einen Befehle ein (z.B. *ls*), so sucht die Shell die Verzeichnisse, die in der Umgebungsvariablen *PATH* angegeben sind, ob eine Datei dieses Namens vorhanden und ausführbar ist. Diesen langen Satz müssen wir ein wenig aufdröseln:
Die Umgebungsvariable (genaueres in einem späterem Beitrag) *PATH* lässt sich anzeigen mit

```
$ echo $PATH
```

Die einzelnen Verzeichnisse sind mit einem Doppelpunkt getrennt. Möchte der Maker ein Verzeichnis hinzufügen, so ist dies kein Problem:

```
$ export PATH=$PATH:/home/maker/sensoren
```

Wenn die Shell eine Datei mit dem geforderten Namen in einem Verzeichnis findet (es gilt der erste Treffer in der Liste), so prüft sie, ob diese Datei ausführbar ist. Der Maker kann dies selbst prüfen:

```
$ ls -l /bin/ls
```

In der ersten Spalte werde die Dateiattribute (auch hierzu später mehr) angezeigt; ist dort *x* vorhanden, so ist die Datei ausführbar, und das Programm wird ausgeführt.
Eine weitere Möglichkeit - z.B. für Dateien, die nicht im Pfad vorhanden sind - ist es einen absoluten oder relativen Pfad anzugeben:

```
$ /home/maker/tmp/gethumidity
$ ../tmp/gethumidity
```

Ist die Datei im aktuellen Verzeichnis, so verwendet man

```
$ ./gettemperature
```

Ganz wichtig: das aktuelle Verzeichnis (.) sollte nie im Suchpfad vorhanden sein! Dies kann zu sehr merkwürdigen Effekten führen, wenn dort zufällig ein Programm mit den Namen eines anderen Befehls (oder eines Tippfehlers) liegt.

Umgebungsvariablen

In der Shell gibt es einige Umgebungsvariablen, die das Verhalten der Shell oder einiger Befehle beeinflussen. *PATH* haben wir schon kennen gelernt (Beitrag *Suchpfad*). Schauen wir uns an, was so gibt:

```
$ env
```

Einzelne Variablen lassen sich (wie schon mit PATH erläutert) mit (z.B.)

```
$ echo $LANG
```

ausgeben. Schauen wir uns ein paar Variablen an:

SSH_CLIENT: wenn sich der Maker über *ssh* (z.B. *putty*) einloggt, steht die IP-Adresse des Rechners, von dem er sich eingeloggt hat, in dieser Variablen.

USER: Der Name des Benutzers.

PAGER: welches Programm (inkl. Parameter) von *man* benutzt wird, um blättern zu können.

PWD: das aktuelle Verzeichnis.

EDITOR: welchen EDITOR diverse Programme verwenden (z.B. *crontab -e*).

_: der zuletzt verwendet Befehl

LANG: die eingestellt Sprache und der verwendete Zeichensatz.

Umgebungsvariablen lassen sich auch selbst anlegen.

Geschichte

Die *bash* führt eine Geschichte der eingegebenen Befehle - zwar nicht bis zum Anfang der der Zeit, aber (standardmäßig) 500 Befehle. Die Anzahl der Befehle, die aufgehoben werden, wird mit der Umgebungsvariablen HISTSIZE geändert.
In der *history* kann sich der Maker nach oben, unten, links und rechts bewegen. Hierzu gibt es zwei Modi: zum einen der *vi mode*, der allerdings einige Kenntnisse des Editors vi verlangt. Zum anderen (und standardmäßig eingestellt) der *emacs mode* (der wiederum am besten zu benutzen ist, wenn man einige emacs-Befehle beherrscht). Der *emacs mode* soll hier ein wenig erläutert werden.
Zuerst kann sich der Maker mit den Cursor-Tasten in der *history* bewegen.

- Cursor hoch: den vorigen Befehl anzeigen;
- Cursor runter: den nächsten Befehl anzeigen (ziemlich sinnlos, wenn der zuletzt eingegebene Befehl angezeigt ist);
- Cursor links und Cursor rechts: Bewegung innerhalb einer Zeile;
- Strg-A: zum Anfang der Zeile;
- Strg-E: zum Ende der Zeile

Ein angezeigter Befehl kann dann bearbeitet werden: *Entf* zum Löschen, zum Einfügen einfach tippen. Darüberhinaus:

- Strg-U: Löschen von aktueller Position bis zum Zeilenanfang;
- Strg-K: Löschen von aktueller Position bis zum Zeilenende.

Um zu sehen, was überhaupt in der *history* steht, gibt es (Überraschung!) den Befehl

```
$ history
```

Suchen in der *history* funktioniert mit "inkrementeller Suche" Strg-R. Nach Strg-R fängt man an, den Suchbegriff zu tippen. Mit jedem Buchstaben wird die Suche genauer. Hat man genug getippt, ist aber noch nicht am gewünschten Befehl angekommen, so kann man weiter Strg-R drücken. Die Suche wird dann mit den bisher eingegebenen Zeichen fortgesetzt.
Eine andere Möglichkeit: in der Liste, die nach Eingabe des Kommandos *history* angezeigt wird, steht vor jedem Kommando die laufende Nummer des Befehls. Stellt der Maker dieser Nummer ein Ausrufezeichen voran, so wird der Befehl mit dieser Nummer ausgeführt. Hat z.B. das Kommando *ls -l* in der Liste die Nummer 472, so wird mit

```
$ !472
```

dieser Befehl noch einmal ausgeführt.

Eingabeumleitung

Nachdem Ihr wisst, wie die Ausgabe umgeleitet werden kann, stellt sich die Frage, ob das auch mit der Eingabe geschehen kann. Antwort: ja (und damit könnte der Beitrag fertig sein).

Ein (sinnloses) Beispiel:

```
$ grep suchstring < datei
```

Hier wird der Inhalt der Datei "datei" zur Standardeingabe (oder auch *stdin*) geschickt. (Fast) jedes Programm, dass Daten von der Konsole lesen will, kann so mit Daten aus einer Datei gefüttert werden.
Wenn ich die Ausgabe umleiten kann und die Eingabe auch, ist es dann möglich die Ausgabe eines Programms als Eingabe eines anderen Programms zu verwenden? Auch hier die Antwort: ja. Dazu dient die "pipe" (|). Als Beispiel suchen wir in einer Verzeichnisliste nach Verzeichnissen:

```
$ ls -l | grep '^d'
```

Die Ausgabe des ls-Befehls wir als Eingabe des grep-Befehls verwendet (der grep-Befehl sucht hier nach Zeilen, die mit einem "d" beginnen (das "^" ist die *grep*-Syntax für "Zeilenbeginn")). Es ist nicht ungewöhnlich solche *pipes* mit mehreren Kommandos zu bauen (mein persönlicher Rekord liegt bei 18). Z.B.: liegt eine Telephonliste (tel.txt) in der Form

```
Name, Vorname, Nummer
```

vor, so zählt die folgende Befehlskette wie häufig jeder Vorname vertreten ist und sortiert die Liste nach Häufigkeit:

```
$ awk -F',' '{print $2}' tel.txt | sort | uniq -c | sort -n
```

Zu *awk* gibt's später vielleicht mal einen Beitrag, *sort* sortiert einfach, *uniq* entfernt doppelte Zeilen (und gibt mit dem *-c* die Häufigkeit aus, und das letzte *sort* sortiert numerisch (*-n*).

Kleines Bonbon zum Schluss: Verzeichnisliste mit farblich markierten Unterverzeichnissen:

```
$ ls -l | grep -E --color '^d.*|$'
```

Shell-Funktionen

Nun arbeitet der Maker lange mit seinem Raspberry Pi und stellt fest, dass er die immer wieder die selben Befehle ausführt. Lassen sich Befehle vielleicht zusammenfassen? Ja das geht. Als erstes werden wir Shell-Funktionen kenne lernen.
Die allgemeine Syntax ist:

```
name ()
{
    befehl1;
    befehl2;
    ...
}
```

Ein Beispiel mag dies verdeutlichen: regelmäßig legt der Maker ein Verzeichnis mit aktuellen Datum an und wechselt dort hinein:

```
$ mkdir $(date '+%F')
```

(*$(...)* bedeutet: Führe das Kommando zwischen den Klammern aus und ersetze die Ausgabe anstelle dieser Konstruktion)
Auf die Dauer ist dies dem Maker zu viel Tipparbeit - deshalb definiert er sich eine Funktion *mkdd* (*make date directory*):

```
mkdd ()
{
    D=$(date '+%F');
    mkdir $D;
    cd $D
}
```

Nun kann der Maker einfach *mkdd* aufrufen:

```
$ pwd
/home/maker/tmp
$ mkdd
$ pwd
/home/maker/tmp/2016-01-29
```

(*pwd* (*print working directory*) gibt das aktuelle Verzeichnis aus.)
Funktionen können auch Parameter übergeben werden. Eine häufige Aufgabe ist, ein Verzeichnis anzulegen und gleich hinein zu wechseln:

```
$ mkdir version1
$ cd version1
```

Dies ist immer viel Tipparbeit - also legen wir uns eine Funktion an:

```
mcd ()
{
     mkdir -p $1
     cd $1
 }
```

Durch Aufruf der Funktion mit einem Parameter wird also das Verzeichnis angelegt und gleich hinein gewechselt:

```
$ pwd
/home/maker/tmp
$ mcd version1
$ pwd
/home/maker/tmp/version1
```

Nun beobachtet Euch selbst beim Tippen und überlegt, wo Ihr am besten Tastendrücke spart.
Muss der Maker nun die Funktionen nach jedem Einloggen neu eingeben? Natürlich nicht. Er trägt die Funktionen in seine *.bashrc* ein, und nach jedem Einloggen stehen sie zur Verfügung.

Shell-Skripte

Die Arbeit im Hintergrund

Wenn der Maker ein lang laufendes Programm startet, kann er auf dieser Verbindung während des Programmlaufs keine weiteren Programme aufrufen kann. Wenn vorher klar ist, kann das Programm im Hintergrund ausgeführt werden:

```
$ programm &
```

Die Shell gibt dann die Jobnummer (in eckigen Klammern) sowie die Prozess-ID (*pid*) aus.

Doch wenn der Maker erst nach dem Start bemerkt, dass das Programm zu lange dauert? Dann drückt er die Tastenkombination Strg-Z. Die Shell meldet, dass das Programm jetzt angehalten ist. Das ist noch nicht ganz das, was wir wollten. Aber mit

```
$ bg
```

wird die Ausführung im Hintergrund fortgesetzt. Wenn das Programm wieder im Vordergrund laufen soll, dann hilft der Befehl

```
$ fg
```

Sinnvoll ist bei der Hintergrundausführung die Ausgaben in eine Datei umzuleiten:

```
$ programm > programm.out 2> programm.err &
```

KAPITEL ZWEI

Linux- & bash-Befehle

pushd/popd

Szenario: ich befinden mich tief in der Verzeichnisstruktur eines Projekts. Nun muss ich kurz in ein anderes Projekt eintauchen. Normalerweise verwendet man hierfür endlich viele *cd*-Kommandos und muss anschließend mühsam sein Ausgangsverzeichnis wieder finden.

Jedoch: *pushd* und *popd* sind Deine Freunde.

pushd legt das aktuelle Verzeichnis auf einen Stack und wechselt anschließend in das Verzeichnis, dass als Parameter angegeben wird:

```
$ pushd ~/projekt2/source
```

Anschließend kann man sich mit *cd* weiter in der Verzeichnisstruktur bewegen. Ein

```
$ popd
```

wechselt dann wieder in das Verzeichnis zurück, in dem *pushd* aufgerufen wurde.

Selbstverständlich ist der Linux-Guru nicht auf ein *pushd* beschränkt. Jedes *pushd* legt das aktuelle Verzeichnis auf dem Stack ab, und mit sukzessiven *popd* wird der Stack wieder abgearbeitet.

Einfach am Wochenende mal ausprobieren

```
$ pushd /etc
$ ls
$ cd /tmp
$ ls
$ pushd /usr/local
$ cd bin
$ ls
$ popd
$ pwd
$ popd
$ pwd
```

wc

Nein - kein Klo. *wc* bedeutet *word count*. Der Befehl gibt die Anzahl der Zeilen, Wörter und Zeichen einer (oder mehrerer) Dateien aus:

```
$ wc *.log
```

Ein Wort ist hier eine Zeichenkette, die von Leerzeichen, Tabulatoren oder Zeilenenden begrenzt wird. Interessieren den Maker nicht alle Werte, so kann er (oder sie) die Ausgabe mit einem entsprechenden Schalter eingrenzen:

Anzahl der Zeichen:

```
$ wc -c temperatur.log
```

Anzahl der Wörter:

```
$ wc -w humidity.log
```

Anzahl der Zeilen:

```
$ wc -l users.log
```

Was läuft denn so?

Was für Prozesse laufen eigentlich gerade? Die Antwort hierauf liefert *ps* (*process summary*).

```
$ ps
```

Nun - dies Ausgabe hier ist nicht sonderlich informativ. Sie gibt nur an, welche Prozesse gerade laufen, die von diesem Terminal aus gestartet wurden. I.A. sind dies nur die aktuelle Shell und *ps* selber:

```
  PID TTY          TIME CMD
12111 pts/1    00:00:00 bash
12320 pts/1    00:00:00 ps
```

Aber schon hier kann der Maker einige Informationen sehen:

-
- die Prozess-ID (*PID*)
- die Bezeichnung des Terminals (*TTY*)
- die CPU-Zeit, die der Prozess bisher verwendet hat (*TIME*)
- das Kommando ohne Parameter (*CMD*)

Bevor wir uns ein paar Parameter ansehen, muss ich paar Worte darüber verlieren. *ps* versteht drei Arten von Parametern: UNIX-Parameter, BSD-Parameter und GNU-Parameter. Ich werde mich auf die UNIX-Parameter beschränken. Die anderen Arten können das Selbe, haben aber andere Bezeichnungen. Also: UNIX-Parameter werden mit einem Bindestrich begonnen.
Der Parameter *-f* (*full*) zeigt ein paar mehr Werte an; die wichtigstem sind die Startzeit des Prozesses und die Parameter der Kommandos.

```
$ ps –f
```

Der Parameter *-e* zeigt auch die Prozesse der anderen Benutzer:

```
$ ps –e
```

Und die Kombination liefert die erweiterte Prozessliste aller Benutzer.

```
$ ps –ef
```

Sucht man nun alle Prozesse, die einem Benutzer zugeordnet sind, so kann der Parameter *-u* (*user*) mit dem Benutzernamen kombiniert werden.

```
$ ps –u pi
$ ps –u root
```

Anmerkung: die Kombination *-eu* ist hier ziemlich sinnlos. Warum wohl? Aber der Parameter *-f* lässt sich gut kombinieren:

```
$ ps –fu pi
```

Mit dem Schalter *-o* (*output*) kann der Maker die Felder angeben; z.B.:

```
$ ps –o pid,%cpu,vsz,cmd –u pi
```

gibt die Prozess-ID, wieviel Prozent der CPU der Prozess belegt, Die Größe des Prozesses in kB und das Kommando inkl. Parameter. Für die anderen etwa hundert Ausgabeoptionen sowie die vielen Parameter verweise ich wieder auf die *man page*:

```
$ man ps
```

Und möchte der Maker wissen, ob noch weitere Benutzer eingeloggt sind, so verwendet er den Befehl

```
$ who
```

Wird dieser Befehl mit zwei (beliebigen) Parametern aufgerufen, so wird der aktuelle Benutzername ausgegeben:

```
$ who am I
```

Da die Parameter beliebig sein können, kann der Maker auch eingeben:

```
$ who mom loves
```

erste und letzte Zeilen von Dateien

Da liegt nun eine ellenlange Datei, aber interessant sind nur die ersten Zeilen. Natürlich kann der Maker *less* verwenden. Aber nur die ersten Zeilen kann man sich mit *head* anzeigen lassen:

```
$ head temperatur.log
```

Standardmäßig zeigt *head* die ersten zehn Zeilen an. Braucht man mehr oder weniger, so gibt man dies mit dem Parameter *-n* an:

```
$ head -n 3 temperatur.log
```

```
$ head -n 100 temperatur.log
```

Und wenn der Maker das Ende einer Datei ansehen will?

```
$ tac index.php | head | tac
```

OK - das war ein Scherz, um mal ein paar Befehle zu wiederholen. Der bessere Weg ist *tail*:

```
$ tail temperatur.log
```

Auch hier gibt es den Parameter *-n*:

```
$ tail -n 25 temperatur.log
```

tail hat aber noch ein schönes Feature. Angenommen ein Programm schreibt Sensordaten in eine Datei. Oder der Webserver schreibt seine Logdatei. Nun möchte der Maker in Echtzeit die Logdatei verfolgen. Dafür ist der Schalter *-f*:

```
$ tail -f temperatur.log
```

Nun zeigt tail die neuen Zeilen, sobald sie in die Datei geschrieben wird. Und noch ein kleines Detail: es gibt Programme, die nur temporäre Logdaten schreiben, oder die Logdateien in regelmäßigen Abständen zur Seite schieben und neu anlegen. Hier hilft der Schalter *-f* nicht mehr, aber dafür gibt es den Schalter *-F*. Dann öffnet tail die Datei neu oder wartet bis die Datei erzeugt wird.

```
$ tail -F neuedatei.txt
```

Dateien anzeigen

Nun hat der Maker viele Textdateien rumliegen: Logdateien, Programmcode, Notizen, ... Wie lassen sich diese Dateien anzeigen? Nur mal durchscrollen lassen? Oder die Datei ist kurz? *cat* benutzen:

```
$ cat datei
```

Eigentlich hat *cat* einen anderen Zweck. *cat* ist die Kurzform von *concatenate* (verbinden, aneinanderhängen). Es dient dazu Dateien aneinanderzuhängen:

```
$ cat temp1.log temp2.log temp3.log
```

Oder (wir erinnern uns) daraus eine neue Datei erzeugen:

```
$ cat temp1.log temp2.log temp3.log > temp_gesamt.log
```

Ist die Datei länger, verwendet man *more* (oder eigentlich nicht; dazu gleich). Damit wird ein Dateiausschnitt angezeigt, der so lang ist wie der Bildschirm hoch. Zur nächsten Seite gelangt man durch Drücken der Leertaste.

```
$ more temp_gesamt.log
```

Aber das war nur ein kurzer Ausflug in die Geschichte - heutzutage benutzt kaum noch jemand *more*. Denn weniger ist mehr. Oder besser: *less* is *more*.

```
$ less temp_gesamt.log
```

Auch kann man mit der Leertaste (oder mit Strg-F - *Forward*) seitenweise weiter blättern. Aber man kommt auch wieder zurück mit Strg-B (*Backward*). Auch zeilenweises Scrollen ist möglich: mit Cursor-hoch bzw. Cursor-runter; oder auch j bzw. k. (Wieso j und k? Das sind die Standardtasten für den Standard-Unix-Editor *vi*. Und nein: nano ist nicht der Unix-Standard-Editor, sondern nur das Dreirad unter den Editoren.)

Mit *less* lässt sich auch suchen. Nach Eingabe von / wird der Suchbegriff eingegeben und Enter gedrückt. Mit n wird der letzte Suchbegriff erneut gesucht. Zu weit gesucht? Dann N drücken und es wird in die andere Richtung gesucht. (Warum /? Auch dies ist ein *vi*-Kommando.) Und von Anfang an rückwärts suchen? ? Das ist kein Tippfehler: ? ist das Kommando. Und auch hier kann man n und N verwenden.

Und ein besonders nettes Feature: Die Taste F (großes F). Nun folgt *less* dem Ende Datei. Während eine Datei geschrieben wird, kann man so die neuen Einträge verfolgen. (Schöner lässt sich das allerdings mit *tail* erledigen - doch dazu später mehr.)

Und hier der mittlerweile traditionelle Hinweis

```
$ man less
```

Dateiinhalt finden

"Ich hatte doch mal 'ne Funktion geschrieben", denkt der Maker. "Wo war sie denn gleich? Sie hieß doch .... äh .... irgendwas mit ... äh ... Entfernung ... äh ... ach ja! get_distance." Jetzt könnte der Maker jede Datei anzeigen lassen (z.B. mit *less*). Wie immer geht es auch einfacher. *grep* ist hier der Freund des Makers. In der einfachsten Form wird als erster Parameter ein Suchbegriff und dann eine Liste von Dateien angegeben; also z.B.:

```
$ grep get_distance *.ino
```

Und schon wird eine Liste der Dateien ausgegeben, die den Suchbegriff enthalten, zusammen mit den entsprechenden kompletten Zeilen.

Doch wenn der Maker nicht weiß, ob da vielleicht Großbuchstaben im Text sind? ("Habe ich vielleicht *get_Distance* verwendet?") Hier hilft der Schalter *-i* (*ignore case*).

```
$ grep –i get_distance *.ino
```

Manchmal ist die Umgebung der Treffer nützlich. Dem Parameter *-C* (*context*) wird die Anzahl der vorher und nachher anzuzeigenden Zeilen angegeben:

```
$ grep –C 3 get_distance *.ino
```

Aber in welcher Zeile in den gefunden Dateien erscheint der Suchbegriff?

```
$ grep –n get_distance *.ino
```

Wenn kein Dateiname angegeben wird, dann liest *grep* von Standardeingabe (*stdin*). Dies kann man ausnutzen, wenn nur gewisse Zeilen ausgegeben werden sollen, falls eine Logausgabe mitgelesen wird (s. auch Ein- & Ausgabeumleitung; *tail* kommt in einer der nächsten Wochen):

```
$ tail –f program.log | grep ERROR
```

Wenn die gefunden Suchbegriffe hervorgehoben werden sollen, dann kann man *grep* anweisen die Begriffe einzufärben:

```
$ grep --color get_distance *.ino
```

Und wie immer gibt auch für *grep* noch viel mehr Parameter. Und wie immer hilft:

```
$ man grep
```

Nachtrag: das mächtigste Werkzeug bei *grep* sind reguläre Ausdrücke. Aber das wäre

ein eigenes Buch. Z.B. bei der fortlaufenden Anzeige mehrerer Logdateien:

```
$ tail -f *.log | grep -E --color '|==>.+<=='
```

Dateien finden

Manchmal sucht man eine Datei. "Wo habe ich denn jetzt diese Text-Datei gespeichert?". Hier hilft der Befehl *find*. Im einfachsten Fall gibt man das Startverzeichnis, den Parameter *-name* und einen Dateinamen an:

```
$ find . –name meinedatei.txt
```

'.' ist das aktuelle Verzeichnis. Hier kann auch ein relativer oder absoluter Pfad angegeben werden. Beim Dateinamen können auch wildcards angegeben werden. Allerdings werden diese von Shell vor der Programmausführung erweitert, was zu unerwünschten Effekten führen kann. Deshalb ist das Einschließen in (einfache oder doppelte) Anführungszeichen Pflicht:

```
$ find /home/maker/projekte –name '*.c'
```

(Die Anführungszeichen sind nötig, damit die die Shell den Wildcard * nicht erweitert, wenn im aktuellen Verzeichnis C-Dateien vorhanden sind.)
Braucht der Maker noch die Dateiattribute, so gibt er den Parameter *-ls* an:

```
$ find /home/maker/doku –name '*.txt' –ls
```

Vielleicht möchte der Maker auch Dateien finden, die seit mehr als zwei Wochen nicht mehr geändert wurden:

```
$ find /home/maker/backup –name '*.tgz' –mtime +14
```

Oder alle Dateien, die seit gestern geändert wurden:

```
$ find /home/maker/projekte –mtime –1
```

Beachte hier das + bzw. - vor der Zahl!

Dateien können auch gleich gelöscht werden, z.B. die alten Backups aus dem Beispiel oben:

```
$ find /home/maker/backup –name '*.tgz' –mtime + 14 –delete
```

Doch Vorsicht: weg ist weg! Ein Tippfehler und wichtige Dateien können für immer verschwunden sein (dies ist eine sehr lange Zeit!). Wenn man wenigstens wissen möchte, welche wichtige Datei weg ist:

```
$ find /home/maker/backup –name '*.tgz' –mtime + 14 –delete –print
```

find hat noch viele weitere Parameter. Einfach mal die *man page* aufrufen:

```
$ man find
```

Übungsaufgabe:

```
$ find / -name '*.py'
```

Wie bekommt man die lästigen Fehlerausgaben weg? Die Antwort kennt Ihr schon ;-)

lustige Kommandos

rev: gibt die Zeilen von hinten nach vorn aus.

```
$ echo "testing" | rev
$ rev .bashrc
```

":": (nur in der bash) liest die Standardeingabe und gibt als Rückgabewert 0 zurück.

```
$ ls -l | :
```

tac: gibt Dateien von der letzten zur ersten Zeile aus (Umkehrung con "cat")

```
$ tac .bashrc
```

(besonders schön:
`$ tac datei | rev`
)

sleep: schläft

```
$ sleep 10
```

schläft 10 Sekunden. Wird verwendet, um einem Shellskript eine Pause zu gönnen (z.B. um einem Programm Zeit zu geben vollständig zu starten bis das Skript weitermacht).

nützliche ls-Parameter

Der Befehl *ls* zeigt die Namen der Dateien und Verzeichnisse an.

-l: (long format) Zeigt nicht nur den Dateinamen, sondern auch die Zugriffsrechte, Anzahl der Links (das wäre ein eigenes Thema), Benutzer, Gruppe, Größe in Byte und letztes Änderungsdatum an.

-h: (human readable) Gerade bei großen Dateien ist häufig nicht auf den ersten Blick sichtbar, wie groß die Datei nun ist (2GB oder doch nur 200MB?). Der Schalter -h zeigt die Größe in Byte, kB, MB, GB, TB,... an.

-t: (time) Die Liste der Dateien wird nach dem letzten Änderungsdatum sortiert (von jung bis alt).

-S: (Size) Sortiert die Liste nach Dateigröße (von groß nach klein)

-r: (reverse) Dreht die Sortierreihenfolge um. Dies funktioniert z.B. bei *-l*, *-t*, *-S*

-u: (last accessed - keine Ahnung, warum das *u* heißt; vermutlich war kein anderer Buchstabe frei) im Zusammenhang mit *-l* zeigt dieser Schalter nicht die Zeit der letzten Änderung, sondern die Zeit des letzten Zugriffs.

-a: (all) Zeigt auch die Dateien und Verzeichnisse, die mit einen Punkt beginnen - inklusive . (aktuelles Verzeichnis) und .. (übergeordnetes Verzeichnis).

Dies sind längst noch nicht alle Optionen - spielt am besten lest die *man page* und selbst mal mit herum.

mkdir -p

Heute lernen wir ein wenig über das Anlegen von Verzeichnissen. Es kommt ab und zu vor, das der Maker eine komplette Verzeichnisstruktur braucht - z.B. für Projekte. Nehmen wir als Beispiel ein Projekt „Temperaturmessung“; wir benötigen dann ein Basisverzeichnis, darin Verzeichnisse für den Quellcode, die Dokumentation und Log-Dateien. Die Struktur ist also:

```
HOME/projekte
`— temperatur
     +— doc
     +— log
     `— src
```

Eigentlich müsste man hier fünf Verzeichnisse anlegen - also fünfmal den Befehle *mkdir* … eingeben. Doch es geht auch einfacher: *mkdir -p* legt einen kompletten Verzeichnispfad an:

```
$ mkdir -p projekte/temperatur/doc
```

erzeugt den kompletten Pfad, so dass nur noch die Unterverzeichnisse *log* und *src* erzeugt werden müssen.

Doch es geht mit noch weniger Befehlen. Dazu einen Ausflug in die bash. Das Stichwort ist hier „brace expansion“. Einfache mal ausprobieren:

```
$ echo bla{eins,zwei,drei}fasel
blaeinsfasel blazweifasel bladreifasel
```

Was ist hier passiert? Die bash nimmt die Werte, die innerhalb der geschweiften Klammern stehen und setzt sie (einen nach dem anderen) anstelle der Zeichen der Zeichenkette innerhalb der geschweiften Klammern.
So können wir alle Verzeichnisse in einem Befehl anlegen:

```
$ mkdir -p projekte/temperatur/{doc,log,src}
```

PS1

Den Prompt auf der Shell einstellen.

Den Prompt (das Teil, dass anzeigt, dass die Shell einen Befehl erwartet) kann man sich nach eigenen Bedürfnissen zurecht schneidern. Ganz ganz früher (lange vor Linux) gab es nur einen einfachen Prompt:
$
für normale Benutzer,
#
für *root*

Eingestellt wird der Prompt in der Umgebungsvariablen *PS1*:

```
$ export PS1='$ '
```

Der erste Schritt zum Ändern:

```
$ export PS1='\$ '
```

Hier wird automatisch das Zeichen *$* für normale Benutzer und # für root verwendet.

In der *man page* für die *bash* stehen viele Platzhalter für diverse Informationen (im Abschnitt PROMPTING). Ein sinnvoller Prompt ist z.B.:

```
export PS1='\u@\h:\w \$ '
```

Er zeigt Benutzer, Maschinenname und aktuelles Verzeichnis.

Eingetragen wird der Befehl in der Datei *.bashrc* im Heimatverzeichnis des jeweiligen Benutzers. So wird der Prompt bei jedem Login gesetzt.

Übungsaufgabe: verstehe

```
PS1='\[\e]0;\u@\H:\w\a\e[0;32m\]\u@\h:\W (\#)\[\e[m\] \$ '
```

KAPITEL DREI

Linux ganz unten

Wieviel Platz habe ich noch? Was sind die fetten Brocken?

Nun hat der Maker Software für viele Projekte geschrieben, Unmengen von Photos für die Dokumentation abgelegt und ellenlange Log-Dateien erzeugt. Schweißgebadet wacht er nachts auf mit der Frage "Wieviel kann ich noch speichern?"
Auch hier helfen Unix-Befehle - in diesem Fall *df* und *du* (*disk free* und *disk usage*). Fangen wir an mit

```
$ df
```

Der Befehl gibt den belegten und den freien Speicherplatz aller Dateisysteme an. Die Ausgabe ist erstmal nicht leicht zu interpretieren, da die Größen in Vielfache von Blöcken ausgegeben wird. Dies können 512-Byte- oder auch 1kB-Blöcke sein.
Einfacher zu lesen ist die Ausgabe in vertrauten Einheiten:

```
$ df -h
```

-h für *human readable*.

Doch welches sind die größten Verzeichnisse? Hier hilft *du* (*disk usage*).

```
$ du
```

Standardmäßig gibt der Befehl die Größen aller Unterverzeichnisse in kB aus. Mit dem Schalter *-h* (*human readable*) werden die Größen in lesbarere Einheiten umgerechnet, sofern dies nötig ist (MB, GB,...).

```
$ du -h
```

Der Schalter *-s* gibt nur die Summe des aktuellen Verzeichnisses aus.

```
$ du -s
```

Selbstverständlich lassen sich die beiden Schalter kombinieren:

```
$ du -sh
```

Auch einzelne Verzeichnisse (oder auch Dateien) lassen sich angeben. Häufig verwendet wird

```
$ du -sh *
```

für die Größen der Unterverzeichnisse des aktuellen Verzeichnisses, aber ohne tiefere Unterverzeichnisse.
Auch diese beiden Befehle haben noch viel mehr Parameter. Wie immer:

```
$ man du
```

und

```
$ man df
```

crontab

Manchmal muss ein Maker ein Programm in regelmäßigen Abständen ausführen:

- einmal in fünf Minuten den Temperatur- und Luftfeuchtigkeitssensor auslesen und den Wert speichern
- stündlich ein Photo aufnehmen
- jeden Mittag die Umgebungshelligkeit twittern
- einmal im Monat das HOME-Verzeichnis auf die externe Platte kopieren.
- Jeden Freitag eine Wochenendmail senden

Erste Möglichkeit: Wecker und Kurzzeituhr stellen - das führt aber zu Schlafdefizit und Vereinsamung.
Zum Glück hilft hier Unix (so auch Linux) weiter. *cron* heißt das Zauberwort.
Der Zeitplan der Jobs wird in die *crontab* eingetragen. Dies geschieht mit dem Befehl

```
crontab -e
```

Dies öffnet den Standardeditor (der in der Umgebungsvariablen EDITOR gespeichert ist).

Eine Zeile besteht aus sechs Feldern:

```
m h dom mo dow  command
```

m: Minute
h: Stunde
dom: Tag des Monats
mo: Monat
dow: Wochentag (0 = Sonntag)
cmd: Kommando

Als Platzhalter kann * verwendet werden. Jeder Wert kann auch als Bereich eingegeben werden, z.B. bei Wochentagen die Werktage: 1-5. Auch eine Liste von Werte kann eingegeben werden. Dies Werte werden durch ein Komma getrennt. Beispielsweise für gewisse Stunden: 6,13,20,23. Eine weitere Möglichkeit ist ein regelmäßiges Intervall. Z.B. alle fünf Minuten: */5

Eine *crontab* mit den Beispielen von oben:

```
*/5 * * * * /home/maker/bin/temp > /home/maker/logs/temp.log 2>&1
17 * * * * /home/maker/bin/take_photo > /home/maker/log/take_photo.log 2>&1
3 12 * * * /home/maker/bin/tweet_light > /home/maker/log/tweet_light.log 2>&1
6 15 1 * * /home/bin/make/backup_files > /home/maker/log/backup_files.log 2>&1
4 10 * * 5 echo "Wochenende!!" | mail -s "Wochenendmail" maker@arduinohannover.de
```

Die Jobs werden hier nicht alle zu gleichen Minute ausgeführt. Werden alle Jobs zur Minute 0 ausgeführt kann das (je nach Anzahl der Benutzer, die *cron jobs* einrichten, und Anzahl der *cron jobs*) dazu führen, dass zu dieser Minute der Rechner annähernd unbrauchbar wird. (Ich habe schon erlebt, dass ein Server zu jeder vollen Stunde für etwa 10 Minuten nicht mehr zu benutzen war, weil dann etwa siebzig Prozesse los gelaufen sind.)

Von Zombies, Töten und Kindermördern

Der Maker war sehr produktiv und hat viele Programm laufen - im Hintergrund, von *cron* aus, ... Doch ein Prozess läuft Amok und belastet das System zu sehr, oder ein Programm soll verbessert werden und darf dabei nicht laufen. Also muss dieser Prozess angehalten werden. Der Befehl dazu heißt *kill*. (Häufiger Spruch bei Systemadministratoren: "Dann werde ich mal den Server killen.")
Um eine Prozess abzuschießen (ein weiteres Synonym), benötigt der Maker die Prozess-Id. Wir kennen schon den Befehl *ps*; z.B.:

```
$ ps –fu maker
```

In der zweiten Spalte der Ausgabe steht die Prozess-Id, in der letzten der Prozessname. Die Prozess-Id wird nun zum abschießen verwendet:

```
$ kill 18365
```

mit *ps* kann der Maker dann überprüfen, ob der *kill*-Befehl erfolgreich war. Unter Umständen läuft der Prozess aber weiterhin. Dies kann passieren, wenn der Prozess gerade heftige Ein- oder Ausgaben (auf der Platte oder im Netz) ausführt. Dafür wird ein weiterer Parameter eingegeben - die sog. Signalnummer - der ein "bedingungslosen Töten" auslöst:

```
$ kill –9 18365
```

Geht auch das nicht, dann hilft nur warten oder Neustart des Rechners; aber dies kommt ziemlich selten vor.

Ab und zu (OK, OK: ziemlich selten) sieht man in der Prozessliste Prozesse, die mit "[zombie]" gekennzeichnet sind. Dies sind Prozesse, die nicht mehr laufen, aber noch in der Prozessliste stehen. Sie belegen (außer dem Eintrag in der Prozessliste) keine Ressourcen und können ignoriert werden. Außer dieses Phänomen tritt häufig auf. Dann ist evtl. der Rechner zu klein dimensioniert für die Last, die er stemmen muss.

Was ist der Unterschied zwischen *kill* und *kill -9*?

Es gibt Prozesse, die weitere Prozesse starten - sog. Kind-Prozesse (*child processes*). (Der Webserver *apache* ist ein Beispiel für so eine Konstruktion. Hier werden mehrere Kind-Prozesse gestartet, damit mehrere Anfragen an den Webserver gleichzeitig behandelt werden können.)
Ein normales *kill* wird an diese Kind-Prozesse durchgereicht, und der Eltern-Prozess wartet, bis alle Kinder verschwunden sind. Ein *kill -9* tötet den Eltern-Prozess bedingungslos. Dadurch können aber die Kind-Prozesse überleben und müssen dann einzeln abgeschossen werden.
Lustiges am Rande: Es gab mal eine Administrationsoberfläche, die auf dem Ego-

Shooter Quake basiert. Hier konnte der Administrator die Prozesse wirklich abschießen. Der Nachteil war allerdings, dass die Prozesse auch zurück schießen.

Und zum Schluß eine Warnung:
"Don't tell your neighbors, that you sometimes kill child processes!"