FernUniversität
Gesamthochschule in Hagen

Universitätsrechenzentrum

Jens Vieler
Abt. Wiss. Anwendungen

Einführung in UNIX

A/037/9202

© FernUniversität, Februar 1992

Inhaltsverzeichnis

1 UNIX - Ein Betriebssystem 3

2 Etwas Historie 3

3 Die ersten Schritte 5
3.1 Voraussetzungen 5
3.2 Login 5
3.3 Kommandosyntax 6

4 Dateiorganisation 7
4.1 Dateien 7
4.2 Dateiverzeichnisse 7
4.3 Gerätedateien 9
4.4 Dateizugriffsrechte und -schutz 9

5 Etwas Hilfe gefällig ? 10
5.1 Manualpages 10
5.2 InfoExplorer 10

6 Erste Kommandos 12
6.1 ls – Inhaltsverzeichnis 12
6.2 cd – Arbeitsdirectory wechseln 13
6.3 pwd – Pfadname des Arbeitsdirectories 14
6.4 mkdir – Directory anlegen 14
6.5 rmdir – Directory löschen 15
6.6 rm – Dateien löschen 15
6.7 cp – Dateien kopieren 15
6.8 mv – Dateien umbenennen 16
6.9 chmod – Zugriffsrechte ändern 16
6.9.1 Rechte setzen (I) 16
6.9.2 Rechte setzen (II) 17
6.9.3 Gleichsetzung der Rechte 18
6.10 chown – Besitzer wechseln 18
6.11 find – Dateien suchen 18

7 Wildcards 19

8 Filter 20

9 Pipes 21

10 Noch mehr Kommandos 22
10.1 cat – Dateiinhalt anzeigen 22
10.2 sort – Datei sortieren 23
10.3 comm – Dateiinhalte vergleichen 24
10.4 date – Datum und Uhrzeit 24
10.5 time – Laufzeit 25
10.6 file – Dateiart ermitteln 25
10.7 grep – Textmustersuche 26
10.8 lp – Drucken 26
10.9 lpstat – Druckerwarteschlange 27
10.10pr – Druckeraufbereitung 27

10.11more – Bildschirmausgabe-Steuerung . . . 28
10.12passwd – Passwort ändern . . . 28
10.13ps – Prozessinformationen . . . 29
10.14tail – Dateiende ausgeben . . . 29
10.15tar – Disketten/Streamer/Bänder . . . 30
10.16wc – Elemente zählen . . . 31
10.17who – aktive Benutzer . . . 31

11 vi – der Editor **32**
11.1 Aufruf . . . 32
11.2 Verlassen . . . 32
11.3 Nur Lesen . . . 33
11.4 Texteingabe . . . 33
11.5 Cursor bewegen . . . 33
11.6 Textsuche . . . 34

12 Hintergrundprozesse **35**

13 Programmieren mit der Shell **37**
13.1 Programmieren – aber wo ? . . . 37
13.2 SHELL-Variablen . . . 37
13.3 Programmkonstrukte . . . 38
13.3.1 if – Verzweigung . . . 38
13.3.2 Bedingungen . . . 39
13.3.3 case – Verzweigung . . . 40
13.3.4 for – Schleife . . . 41
13.3.5 while – Schleife . . . 41
13.3.6 until – Schleife . . . 42
13.4 Substitution . . . 42
13.5 Prozessüberwachung . . . 43
13.6 Verschiedene SHELLs . . . 44

14 Hilfsmittel zur C-Programmierung **44**
14.1 Compilieren . . . 45
14.2 Makefiles . . . 46
14.3 C-beautifier . . . 49
14.4 C-semantic-checker . . . 50

15 Nachwort **51**

A INDEX **52**

Durch verschiedene UNIX- und SHELL-Versionen[1] (bsh,ksh,csh,..) können hier oder da Abweichungen zu den in diesem Script genannten Kommandos bestehen. Die Beispiele sind auf einer IBM RISC/6000 AIX getestet. Dabei wurde als SHELL die Korn-SHELL (ksh) eingesetzt.

1 UNIX - Ein Betriebssystem

Ein Betriebssystem wird wie folgt definiert (DIN 44300):

> Diejenigen Programme eines digitalen Rechnersystems, die zusammen mit den Eigenschaften der Rechenanlage die Basis der möglichen Betriebsarten des digitalen Rechensystems bilden und insbesondere die Abwicklung von Programmen steuern und überwachen.

Das war (zugegeben) sehr trocken. Als Faustregel behalten wir:

> Das Betriebssystem sorgt zum einen dafür, daß wir uns mit dem Rechner „unterhalten“ können (Befehl des Benutzers interpretieren und in die Tat umsetztén) und Programme kontrolliert ablaufen. Zum anderen verwaltet es die zu Verfügung stehenden Betriebsmittel des Rechners, also CPU-Zeit[2], Speicherplatz und Endgeräte (z.B. Drucker, Bildschirme, ...).

Das Betriebssystem UNIX hat sich für Anwender und Hersteller als zukunftsweisend herauskristallisiert. UNIX besteht zu ca. 90% aus der Programmiersprache C und zu 10% aus der jeweils maschinenabhängigen Assembler-Sprache. Dadurch kann UNIX durch die geringfügige Änderung des 10%-Anteils auf nahezu jeden Rechner portiert werden. Die Folge: Einmal erstellte Software kann auch bei Austausch der Hardware weiter genutzt werden; die Rechner- und Herstellerabhängigkeit ist aufgelöst.

Das mehrbenutzer- und mehrprogrammfähige System bietet dem Benutzer Dialogverarbeitung (Rechner$\Longleftrightarrow$Mensch), Stapelverarbeitung (Abarbeitung einer Folge von Befehlen) sowie Echtzeitverarbeitung (z.B. Meßwert-Analyse).

Neben einer Reihe von Kommandos soll dieses Script das Dateiensystem, den wichtigsten Editor, den vi, und das Pipes und Filter-Konzept vorstellen. Auf Vollständigkeit wird kein Anspruch erhoben; auf der Titelseite steht ja „Einführung in UNIX“.

2 Etwas Historie

Die „Geburt“ von UNIX wird auf das Jahr 1969 beziffert. An den Bell Laboratories, einer Tochter der amerikanischen Telefongesellschaft AT&T, entwickelte Kenneth Thompson ein in DEC PDP-7-Assembler geschriebenes Betriebssystem, das er UNIX taufte. In der ersten Stufe arbeitete UNIX noch als Ein-Platz-System; d.h. der Rechner kann nur einen Benutzer bedienen.

1971 erfolgte die UNIX-Implementierung auf Rechnern leistungsfähigerer Art. Um das Betriebssystem möglichst hardwareunabhängig zu gestalten, plante Thompson UNIX in einer Hochsprache zu schreiben. Hierzu entwickelte er deshalb 1970 aus BCPL die Sprache B, die sich jedoch als zu

[1]SHELL = Befehlsinterpreter, s. Kapitel 3.2

[2]Rechnerzeit, d.h. z.B. wenn mehrere Benutzer gleichzeitig mit dem System arbeiten muß das Betriebssystem entscheiden wer in welcher Reihenfolge bedient wird.

schwach erwies.

Nachdem Dennis Ritchie, ebenfalls bei Bell, die Sprache C kreiert hatte, wurde UNIX bis 1973 fast vollständig in diese Sprache umgeschrieben. UNIX wurde außerdem mehrbenutzer- und mehrprogrammfähig.

Mit der Zeit erkannten Firmen und Universitäten die Vorteile des rechnerunabhängigen Systems, kauften Quell-Lizenzen und entwickelten in verschiedene Bedarfsrichtungen weiter. Produkte wie das Siemens SINIX[3] oder Microsoft XENIX etc. entstanden.

1983 versucht AT&T diese verschiedenen Richtungen wieder zusammenzuführen; nach vielen Versionsnummern fällt die Bezeichnung „System V“, die dem Wildwuchs der UNIX-Derivate einen Riegel vorschieben soll.

Die X/OPEN-Group, ein Zusammenschluß div. Hersteller[4], wird gegründet um dieses Ziel zu verfolgen. Insbesondere:

- Die Anzahl und Qualität von Softwareprodukten zu vergrößern.
- Die Portabilität von Softwareprodukten auf Quellcode-Ebene sicherzustellen, um so die Investitionen von Software-Herstellern zu schützen und den Vertrieb von Software zu erleichtern.

Dieser neue de-facto-Standard schafft Einigung über Programmiersprachen, Daten-Management, Datenträger, Netzwerke, Oberflächen (X/WINDOWS), etc.

[3] verdrehen Sie mal die ersten beiden Buchstaben von SINIX; war nur'n Spaß..

[4] BULL, ERICSSON, ICL, SIEMENS/NIXDORF, OLIVETTI, PHILIPS, DIGITAL EQUIPMENT, UNISYS, HP, AT&T,etc. .

3 Die ersten Schritte

3.1 Voraussetzungen

Bevor wir loslegen bedarf es ein wenig an Vorarbeit. Eine UNIX-Maschine wird vom sog. Superuser verwaltet. Er ist neben der Betreuung des Rechners (Organisation, Softwaremanagement, etc.) für die Benutzerverwaltung zuständig. Er muß für Sie

- Benutzerkennung eintragen (Ihr „Name“ im System)
- Benutzergruppe eintragen (Ihre „Benutzergruppe“ im System[5])
- vorläufiges Passwort vergeben
- Home-Directory eintragen (Ihr „Stammplatz“ im UNIX-Dateienbaum, s. Kapitel 4.2 „Dateiverzeichnisse“ auf Seite 7)

3.2 Login

Ist der Rechner gestartet (darum muß sich der System Administrator/Operateur/Superuser kümmern) können Sie sich nun an ein Bildschirmterminal (Bildschirm mit Tastatur) des Rechners setzen; ggf. muß es eingeschaltet werden.

Nach Abarbeitung der Eingangsmaske, die die Benutzerkennung und das dazugehörige Passwort von Ihnen verlangt, meldet sich die sog. SHELL. Sie übernimmt die Funktion eines Mittlers zwischen Ihnen und dem UNIX-Kern; nimmt die Befehle entgegen und reagiert entsprechend. Man kann sie auf dem Bildschirm an einem $ (Dollarzeichen) oder sonstigen Promptzeichen erkennen (fragen Sie den Superuser).
Der Dialog sieht wie folgt aus:

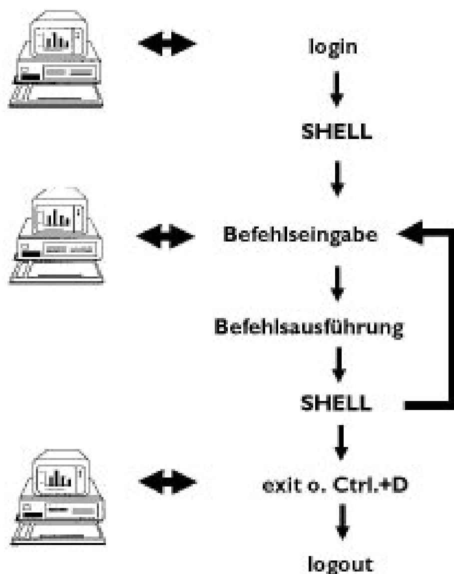

Abbildung 1: Dialog mit UNIX

[5]Benutzer können, ähnlich den Abteilungen in einer Firma, zu sog. Benutzergruppen zusammengefasst werden.

Dabei wird jede(r) Eingabe/Befehl mit der Tastatur eingegeben und mit `Enter` (oder `CRLF`) abgeschlossen. Beendet wird die Rechnersitzung mit der Eingabe *exit* und `Enter`, oder mit `Ctrl` + `D`[6].

3.3 Kommandosyntax

Kommandos werden in der Zeile nach dem Prompt eingegeben. Sie bestehen aus einem oder mehreren Worten (ein Wort = Zeichenkette bis zum Leerzeichen). Dabei ist das:

- erstes Wort = Kommandoname (=Programmname)
- zweites bis letzte Wort, Parameter, die dem Programm übergeben werden.

Es gibt 2 Arten von Parametern:

- Zusatzangaben (Optionen) die den Verlauf des Programms/Befehls beeinflussen. Dies kann z.B. bei Befehlen, die Informationen anlisten, eine Angabe zur Länge der Ausgabe sein. **Optionen werden mit einem vorangestellten "–" gekennzeichnet**.
- Argument, in der Regel die zu bearbeitende Datei (letztes Wort)

Also: Kommandoname `[` option `]` `[` argument `]`

Probieren Sie das Kommando:

```
$who
```

Tippen Sie *who* ein und drücken Sie die Taste `Enter`. Achten Sie dabei auf Groß-/Kleinschreibung; UNIX ist da sehr kleinlich!! Als Ergebnis sollten alle im System anwesenden/aktiven Benutzer auf Ihrem Bildschirm erscheinen. *who* wird hier ohne Argumente oder Parameter aufgerufen.

```
$wc /etc/passwd
```

wc (wordcount) zählt Zeilen, Worte und Zeichen in einer Datei. In diesem Fall mit dem Argument „/etc/passwd", die Passwortdatei des Systems (Passwörter stehen dort übrigens verschlüsselt, keine Chance).

```
$wc -l /etc/passwd
```

wc (wordcount) zählt nun nur die Zeilen in der Datei „/etc/passwd". Die Option `-l` sagt: *wc* bitte nur die Zeilen (Lines) ausgeben.

[6] die Angabe `x` + `y` bedeutet beide Tasten gleichzeitig drücken

4 Dateiorganisation

Grob gesagt ist eine Datei unter UNIX (wie auch unter anderen Betriebssystemen) eine Sammlung von Daten (Zeichen). Sie hat einen Namen und einen Typ; der Name kann sofern es eigene Dateien sind (fast) frei gewählt werden. Vom Datei-Typ gibt es 4 Arten:

- gewöhnliche Dateien (ordinary files)
- Dateiverzeichnisse (directories)
- Gerätedateien (special files, z.B. der Drucker)
- Pipes mit Namen (named pipes, s. Kapitel 9)

4.1 Dateien

Dateien können Programme, Datensätze oder Texte enthalten. (Wen es interessiert: UNIX kennt keine Strukturmerkmale wie Sätze, Blöcke, Sektoren, etc. sondern nur strukturlose Byteketten. Spezielle Formen der Abspeicherung (z.B. ISAM) sind Angelegenheit der Anwendungsprogramme.)

Eigene Dateinamen sind frei wählbar; erlaubt sind bis zu 14 Zeichen. Wählen sie nur alphanumerische Zeichen, sowie .(Punkt) und _(Unterstrich). Die meisten UNIX-Systeme verwenden Konventionen für Dateinamen. Z.B.:

- dateiname**.c** für C-Quellprogramme
- dateiname**.h** für C-Headerdateien
- dateiname**.a** für Objektbibliotheken

usw.

Groß-/Kleinschreibung wird bei Dateinamen unterschieden! Um Dateien anzusprechen benötigt man den sog. **Pfadnamen**. Er gibt Auskunft darüber, wo auf dem Rechner die Datei zu finden ist. Hierzu benötigen wir aber erst einmal die Erläuterung des Begriffes **Dateiverzeichnis**.

4.2 Dateiverzeichnisse

Dateiverzeichnisse (Directories) sind Sammelbecken für Dateien. Jeder Benutzer hat mindestens ein Directory: sein Home-Directory. Nach dem login befindet sich jeder Benutzer an dieser Stelle und kann hier seine Dateien abspeichern.

Es erscheint logisch, daß nach dem Erstellen des 432.ten Programmes und des 299.ten Dokumentationstextes das Dateisammelbecken (Directory) unübersichtlich wird. Der UNIX-Freund hat die Möglichkeit Unterdirectories in beliebiger Fülle und Schachtelungstiefe unter einem bestehenden Directory anzulegen. So ist es sinnvoll unter das Home-Directory z.B. ein Unterverzeichnis PASCAL, für alle Pascal-Programme, eine DOKU, für Dokumentationen, usw. anzulegen. Dabei ist bei solch selbst angelegten Directories die Namensvergabe auch hier völlig freigestellt. Eine durch solche Unterverzeichnisse geordnete Umgebung schafft Übersicht und sollte von Anfang an eingehalten werden.

Mit diesem Verfahren wird übrigens das gesammte Dateisystem des Rechners organisiert. Die folgende Abb. 2 soll verdeutlichen, wie dieses hierarchische Prinzip auf einem UNIX-Rechner realisiert ist. Den Ursprung des Dateisystems bildet das sog. *root*-Verzeichnis (dargestellt durch ein /). Wie bei der Wurzel (root) eines Baumes verzweigen von hier aus die einzelnen Äste (Directories), die dann die Früchte Ihrer Arbeit tragen sollen. Unter dem *root*-Verzeichnis befinden sich also Unterverzeichnisse in denen z.B.

- mögliche System-Befehle (*bin*),
- Gerätedateien (*dev*),
- Systemdateien (*etc*),
- Objektbibliotheken (*lib*),
- Temporäre Dateien (*tmp*),
- Benutzer- und anlagenspezifische Dateien (*usr*),

usw. stehen.

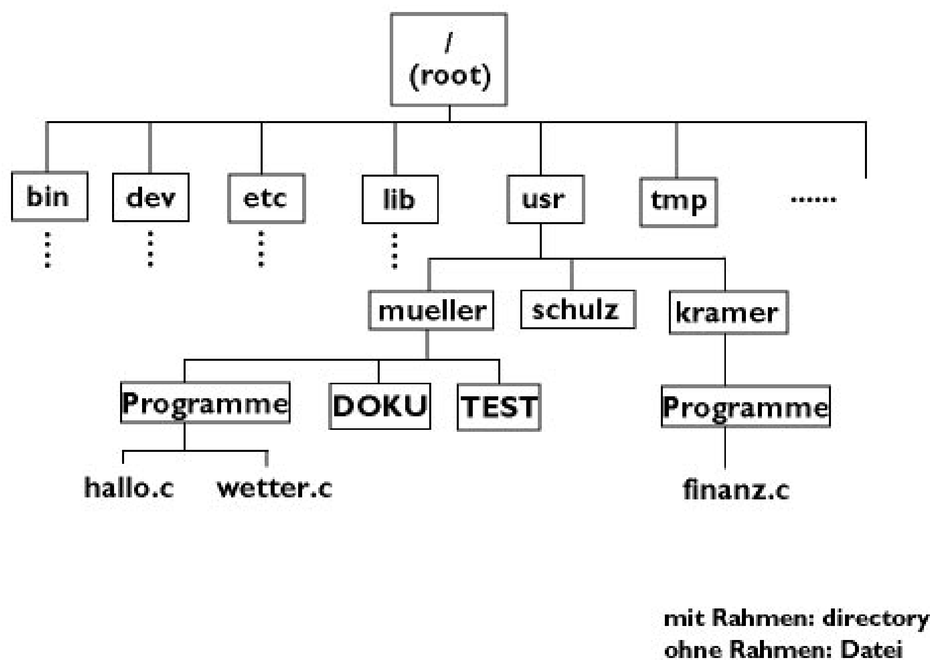

Abbildung 2: Dateien-Hierarchie

Um auf eine Datei zugreifen zu können, benötigt man den sog. **Pfadnamen**, der den Weg durch die verschiedenen Verzeichnisse angibt. Es zwei Möglichkeiten den Pfadnamen anzugeben:

1. **absoluter Pfadname**: alle Verzeichnisse, beginnend bei root, nacheinander aufzählen bis hinab zum Verzeichnis, in dem die Datei steht. Dabei werden die Verzeichnisse im Pfadnamen durch den / (Schrägstrich) voneinander getrennt.
   Im Beispiel wird mit */usr/mueller/Programme/hallo.c* der **absolute Pfadname** bis zur Datei hallo.c genannt; also: „hallo.c ist im Directory *Programme*, das unter dem Directory *mueller* ist, das unter dem Directory *usr* ist, das unter dem Directory *root(/)* im Rechner steht“.

2. **relativer Pfadname**: Der Benutzer befindet sich stets in einem sog. **working directory** (nach dem login ist es sein Home-Directory), in dem er arbeiten kann. Von dort kann in andere Directories gewechselt werden. Steht der Benutzer im o.g. Beispiel schon in */usr/mueller*, so kann er relativ zu seinem Standpunkt Angaben machen. Um hallo.c anzusprechen braucht er hier nur *Programme/hallo.c* anzugeben. Das System ergänzt den fehlenden Teil davor durch den Directory-„Standort“ des Benutzers.

Merke:

absoluter Pfadname	beginnt immer mit / (also ab root)
relativer Pfadname	beginnt immer ohne / (also ab Benutzer-Standort)

Die Befehle zum Anlegen, Löschen und Arbeiten mit solchen Verzeichnissen folgt im Teil „Erste Kommandos“ (Kapitel 6 auf Seite 12).

4.3 Gerätedateien

Gerätedateien sind Peripheriegeräten[7] zugeordnet. Dies vereinfacht das Handling mit dem Betriebssystem ungemein. Alle Befehle die für normale Dateien gelten, können also auch auf Gerätedateien und somit auf Geräte losgelassen werden (soweit die Geräte dies unterstützen).

So ist zum Beispiel der Befehl, eine Datei auf eine andere zu kopieren der gleiche, wie der Befehl eine Datei zu drucken. Der Kopier-Befehl wird nur auf die dem Drucker zugewiesene Gerätedatei angewandt.

4.4 Dateizugriffsrechte und -schutz

Für Dateien und Directories kann der Benutzer, der sie erzeugt hat (**owner**), die Zugriffsrechte festlegen oder ändern. UNIX kennt 3 Arten von Zugriffsrechten.

– r	Lesen (read)
– w	Schreiben (write)
– x	Ausführen (execute)

Diese Zugriffsrechte können, in Kombination, an 3 Benutzerklassen vergeben werden.

– u	Erzeuger der Datei (user = owner)
– g	Benutzergruppe des Eigentümers (group)
– o	alle anderen Benutzer im System (others)

Jede dieser 3 Benutzerklassen kann Zugriffsrechte-Kombinationen zu einer Datei oder einem Directory erhalten.

Beispiel:

Benutzerklasse	user	group	other
Datei1	rwx	rwx	rwx
Datei2	rwx	rw–	r––
Datei3	rwx	r–x	r–x
Datei4	rwx	––x	––x
Datei5	rw–	r––	–––

Zugriffe (lesen/schreiben/ausführen) beziehen sich bei Dateien ausschließlich auf **eine** Datei. Zugriffsrechte auf Directories gelten für alle darin befindlichen Dateien und Unterdirectories. So entsteht eine hierarchische Ordnung. Ein rasches Abschotten vieler Dateien kann durch die Zugriffs-

[7] alles was nicht im Rechnergehäuse Platz fand, z.B. Drucker, Bildschirme usw..

rechte übergeordneter Directories erfolgen. Die hierzu notwendigen Befehle folgen später[8].

5 Etwas Hilfe gefällig ?

UNIX bietet Ihnen ein umfangreiches Hilfesystem, das nicht nur Anfänger nutzen sollten. Zum einen verfügt das System über sog. *Manualpages*, ein Nachschlagewerk, das Hilfestellungen zu UNIX-Befehlen geben kann.

Sollten Sie ausgerechnet mit IBM AIX arbeiten (Sie Glückspilz), steht Ihnen der *InfoExplorer* zur Verfügung, der gleich vorgestellt wird.

5.1 Manualpages

Wenn Sie zu irgendeinem Zeitpunkt die genaue Sysntax eines Befehls mal nicht im Kopf haben, läßt UNIX Sie nicht im Stich. Geben Sie das Kommando

man gesuchter_Befehl

ein. Sie erhalten neben detaillierten Informationen zu Aufruf und möglichen Parameter evtl. auch Anwendungsbeispiele. Die *Manualpages* entsprechen quasi einem elektronischen Referenzhandbuch.

Die Ausgabe erfolgt seitenweise auf den Bildschirm. Mit der `Leertaste` wird um jeweils eine Seite weitergeblättert. Mit `Strg`+`C` oder `Q` wird die Ausgabe beendet. Weitere Möglichkeiten diese Ausgabe zu beeinflußen sehen Sie im Kapitel 10.11 auf der Seite 28 (*more* — bildschirmgerechte Ausgabe).

5.2 InfoExplorer

Der *InfoExplorer*, den die IBM auf ihren AIX-Maschinen zur Verfügung stellt, umfasst **alle** Handbücher, die auch in schriftlicher Form zum Rechner gehören. Dies betrifft über die Komandosyntax hinaus die gesamte hard- und softwaretechnische Systemdokumentation. Sie liegt zwecks schnelleren Zgriffs auf einer ROM-CD im Rechner.
Der *InfoExplorer* kann in zwei verschiedenen Umgebungen aufgerufen werden:

- Unter *X-Windows*[9] steht das *InfoExplorer*-Symbol zum anklicken/starten.
- Im normalen Dialog wird der Befehl **info** eingegeben.

Entsprechend ihrer Aufrufumgebung sehen die angezeigten Hilfefenster bzw. -anzeigen etwas unterschiedlich aus, und werden unter *X-Windows* mit der Maus, im normalen Dialog mit der Tastatur bearbeitet. Beide Versionen bieten sog. *Hypertext*-Funktionen, die innerhalb einer Informationsseite einen raschen Seitensprung zu Querverweisen erlauben. Hierzu werden die optisch besonders hervorgehobenen Begriffe angewählt/angeklickt.
Befehle in der Menueleiste werden entweder mit der Maus angeklickt (*X-Windows*) oder mit `Strg`+`O`, den Cursortasten, den Tabtasten, und der `Enter`-Taste ausgewählt (im normalen Dialog).

[8] Die Standardvoreinstellungen der Zugriffsrechte können auf jedem Rechner anders aussehen und sind vom jeweiligen Dateityp abhängig. So erhalten i. d. R. normale Textdateien read/write für den Besitzer und nur read für alle anderen Benutzer, während ausführbare Programme darüberhinaus die Execute-Rechte haben.

[9] *X-Windows steht nicht jedem Terminal zur Verfügung.*

Abbildung 3: InfoExplorer-Navigationsfenster unter X-Windows

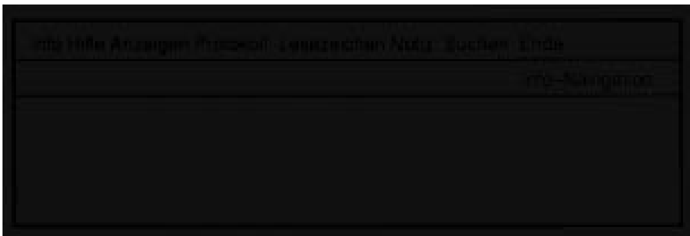

Abbildung 4: *InfoExplorer*-Navigationsanzeige im normalen Dialog

In beiden Fällen gibt es *Navigationsfenster* bzw. *-anzeigen*, die den Benutzer beim Auffinden von gewünschten Informationen unterstützen. Die Themen der Navigationshilfe können unter *X-Windows* direkt angeklickt werden, im normalen Dialog stehen sie unter dem Menueleistenpunkt `Anzeigen`:

- **Aufgabenindex:** enthält eine Reihe häufig durchzuführender Aufgaben (alltägliche Aufgaben, Routine- und Systemverwaltungsaufgaben, Systemfehlerbehebung, IBM RISC System/6000-Konzepte).
- **Befehlsindex:** mögliche Befehle.
- **Lesezeichen:** eine Liste von Artikeln, die bei Suchoperationen ermittelt wurden.
- **Bücher:** ein Inhaltsverzeichnisse der Handbücher.
- **Suchen:** Kann in Artikeln nach Wörtern oder Themen suchen.

Neben dem/der *Navigations*-Fenster/-Anzeige gibt es die sog. *Lese*-Fenster bzw. -Anzeigen. Sie enthalten ablaufbezogene und Referenzartikel.

Um den *InfoExplorer* näher kennenzulernen wird das Lernprogramm *InfoTrainer* angeboten. Es wird unter *X-Windows* im Navigationsfenster unter `Lernprogramm` angeklickt/gestartet, im Normaldialog muß der Benutzer in der Menueleiste (am oberen Anzeigerand) den Punkt `Anzeige` und danach `Lernprogramm` auswählen.

6 Erste Kommandos

UNIX unterscheidet nicht zwischen Kommandos und ausführbaren Programmen. Beide werden durch die Eingabe ihres Namens ausgeführt.

Hier nun eine Auswahl der wichtigsten Kommandos um Ihr Dateisystem zu gestalten und Informationen darüber einzuholen. Bitte beachten Sie, daß die [] (eckige Klammern) in der Komanndosyntax nicht mit eingetippt werden. Sie sollen zeigen, daß die eingeklammerten Befehlsteile optional sind. Denken Sie auch an das − (Minuszeichen), mit dem die Optionen eingeleitet werden.
Wie bereits im vorherigen Kapitel erwähnt: Sollten Sie zu irgendeinem Zeitpunkt die genaue Kommandosysntax mal nicht im Kopf haben, lassen Sie sich mittels *man gesuchter_Befehl* von UNIX unter die Arme greifen.

6.1 ls − Inhaltsverzeichnis

ls (list) liefert Angaben zu einer Datei oder einem Directory. Standard: Informationen zu Dateien/Directories im aktuellen Directory (working directory).
Syntax:

> **ls [Optionen]... [datei/directory]**

Optionen:

- − **l** wichtigste Option: Long listing, Dateityp, Zugriffsrechte, Größe in Byte, letzte Änderungszeit.
- − **x** Mehrspaltige Ausgabe, horizontal sortiert.
- − **C** Mehrspaltige Ausgabe, vertikal sortiert.
- − **a** Auflistung aller Einträge einschl. der mit Punkt beginnenden Dateien, die normalerweise nicht erscheinen. (z.B. .profile Kapitel 13.2)
- − **d** falls Argument directory ist, werden nur die Directory-Informationen, nicht die darin befindlichen Dateien angelistet.
- − **t** sortiert nach Änderungszeit, neueste zuerst.
- − **r** kehrt Sortierreihenfolge um, z.B. in Verbindung mit −t ergibt sich eine Sortierung nach Änderungszeit, älteste zuerst.
- − **R** listet rekursiv durch alle Unterverzeichnisse auf.

Die Kombination mehrerer Optionen wird durch Aneinanderreihung erreicht:

> z.B.: **ls -ltr /usr/mueller**

Beispiel:

```
ls -l
```

Für jede Datei und jedes Unterdirectory wird eine Zeile ausgegeben:

```
   $ls -l

-rwxr-xr-x  1  mueller sys  13  Feb 14  10:23  xx
-rwxr--r--  1  mueller sys 223  Jan 12  12:43  huhu.c
drwxr-xr-x  1  mueller sys 512  Jan 15  13:03  DOKU
-rwxr-xr-x  3  mueller sys 523  Jan 12  12:23  hallo.c

!---   ---  !     !     !   !     !        !     !
! ! --- !   !     !     !   !     !        !     --> Dateiname
! !  !  !   !     !     !   !     !        !
! !  !  !   !     !     !   !     -----------------> letzte Aenderung
! !  !  !   !     !     !   !
! !  !  !   !     !     !   ----------------------> Groesse in Byte
! !  !  !   !     !     !
! !  !  !   !     !     --------------------------> Besitzergruppe
! !  !  !   !     !
! !  !  !   !     --------------------------------> Besitzer
! !  !  !   !
! !  !  !   --------------------------------------> Anzahl Verweise auf
! !  !  !                                           diese Datei
! !  !  !
! !  !  ------------------------------------------> Zugriffsrechte others
! !  !
! !  ---------------------------------------------> Zugriffsrechte group
! !
! -------------------------------------------------> Zugriffsrechte user
!                                                    (owner/Besitzer)
!
---------------------------------------------------> Dateityp:
                                                       - : Datei
                                                       d : Directory
```

6.2 cd – Arbeitsdirectory wechseln

cd (change directory) erlaubt das Wechseln in andere Verzeichnisse. Nach Absetzen des Kommandos wird das angegebene Directory zum aktuellen Arbeitsdirectory (working directory).

Syntax:

cd directory

Beispiel:

cd DOKU	DOKU unter dem aktuellen wd wird neues wd (falls DOKU Unterdirectory des akt. Directory)
cd /usr/mueller/DOKU	/usr/mueller/DOKU wird neues wd
cd ..	wechselt vom aktuellen wd in das übergeordnete Directory
cd ../Programme	wechselt vom aktuellen wd in das übergeordnete Directory, und dann in das Directory Programme
cd	wechselt in das HOME-directory (wie beim login)

6.3 pwd – Pfadname des Arbeitsdirectories

pwd (print working directory) gibt den absoluten Pfadnamen des aktuellen Arbeitsverzeichnisses aus.

Syntax:

pwd

Beispiel:

```
$pwd
/usr/mueller/PROGRAMME
$
```

6.4 mkdir – Directory anlegen

mkdir (make directory) ermöglicht dem Benutzer ein neues Directory anzulegen. Der Directoryname kann mit Angabe eines Pfadnamens (relativ/absolut) erfolgen. Ohne Pfadname wird das neue Directory unter das aktuelle Arbeitsverzeichnis (*working directory*) gehängt.
Besitzer wird der Anleger; er muß die entsprechenden Zugriffsrechte (write) im übergeordneten Directory besitzen.

Syntax:

mkdir dir_1 [dir_2] [dir_3] .... [dir_n]

Beispiel:

```
$pwd
/usr/mueller/PROGRAMME
$mkdir TEST                   <--------- directory anlegen
$cd TEST
$pwd
/usr/mueller/PROGRAMME/TEST
```

6.5 rmdir – Directory löschen

rmdir (remove directory) löscht ein Verzeichnis. Der Befehlsaufbau ist wie bei *mkdir*. Der Benutzer muß auch hier Schreibrechte (write) im übergeordneten Directory besitzen.

Das zu löschende Directory muß zuvor leer sein; d.h. es dürfen keine Dateien darin stehen.

Syntax:

rmdir dir_1 [**dir_2**] [**dir_3**] **....** [**dir_n**]

Beispiel:

```
$pwd
/usr/mueller/PROGRAMME/TEST
$cd ..
$pwd
/usr/mueller/PROGRAMME
$rmdir TEST                    <--------- directory loeschen
$cd TEST
....Fehlermeldung...
```

6.6 rm – Dateien löschen

rm (remove) löscht Dateien und nicht leere Directories (s. Option −r).

Syntax:

rm [**Option**] **datei_1** [**datei_2**] **....** [**datei_n**]

Optionen:

- **f** löscht die angegebenen Dateien, auch wenn Schreibschutz besteht (wenn kein Schreibschutz im übergeordneten Directory), ohne Nachfrage.
- **i** interaktive Nachfrage, ob wirklich gelöscht werden soll
- **r** falls statt dateiname ein directoryname angegeben wird, kann dieses rekursiv mit allen Unterdirectories und darin befindlichen Dateien gelöscht werden.

Beispiel:

```
$rm -i x1 x2
rm: x1? n                     <------- y/n antworten
rm: x2? y
$
```

6.7 cp – Dateien kopieren

cp (copy) kopiert den Inhalt einer Datei in eine andere Datei. Ist die Zieldatei schon vorhanden, so wird ihr Inhalt überschrieben.
Ist das letzte Argument ein Directory werden alle angegebenen Dateien in dieses Verzeichnis kopiert.

Syntax:

cp datei1 datei2
cp datei1 datei2 .... directory

Beispiel:

```
$ cp hallo.c huhu.c
$ cp hallo.c huhu.c DOKU       <-- hallo.c und huhu.c werden in das
                                     Directory DOKU kopiert
```

6.8 mv – Dateien umbenennen

mv (move) wird in erster Linie zum Umbenennen einer Datei benutzt. Ist das letzte Argument ein Directory werden alle angegebenen Dateien in dieses Verzeichnis kopiert, und im aktuellen Verzeichnis gelöscht. Damit werden die Dateien im Dateiensystem umgehängt (gemoved).

Syntax:

mv dateinamealt dateinameneu
mv datei1 datei2 .... directory

Beispiel:

```
$ mv hallo.c huhu.c   <-- Namen aendern
$ mv huhu.c /tmp      <-- huhu.c in das Directory /tmp
                          umhaengen
```

6.9 chmod – Zugriffsrechte ändern

Mit *chmod* (change mode) werden die Zugriffsrechte für Dateien und Directories geändert. Zugriffsrechte können nur der Besitzer einer Datei und der Superuser ändern.
Es gibt 3 Arten mit chmod zu arbeiten:

6.9.1 Rechte setzen (I)

Syntax:

chmod XXX datei_1 [datei_2]... [datei_n]

Dabei steht XXX für 3 Oktalziffern, die sich aus der Addition der folgenden Einzelrechte errechnen:

400	Zugriff Lesen	für Besitzer (owner)
200	Zugriff Schreiben	für Besitzer
100	Zugriff Ausführen	für Besitzer
40	Zugriff Lesen	für Gruppe (group)
20	Zugriff Schreiben	für Gruppe
10	Zugriff Ausführen	für Gruppe
4	Zugriff Lesen	für Andere (others)
2	Zugriff Schreiben	für Andere
1	Zugriff Ausführen	für Andere

Beispiel:

```
  $ls -l hallo.c
-rw-r--r--   1 mueller  sys  92  Sep 04 09:49 hallo.c

  $chmod 751 hallo.c

  $ls -l hallo.c
-rwxr-x--x   1 mueller  sys  92  Sep 04 09:49 hallo.c
```

Damit darf die Datei hallo.c

- vom owner gelesen,beschrieben und ausgeführt
- von der group gelesen und ausgeführt
- von others nur ausgeführt

werden.

6.9.2 Rechte setzen (II)

Syntax:

chmod [**ugoa**] [+/−/=] [**rwx**] **datei_1 ...** [**datei_n**]

Die Zugriffsrechte können für

u	Besitzer (user)
g	Gruppe (group)
o	Andere (others)
a	Alle (all)

vergeben werden. Dabei werden

r	lesen
w	schreiben
x	ausführen

als Rechte durch die Operatoren

+	geben
−	entziehen
=	setzen

an die Dateien/Directories verteilt.

Beispiel:

```
  $chmod o+r hallo.c         <--- others sollen hallo.c zusaetzlich
                                  lesen koennen

  $chmod g=rx hallo.c        <--- group sollen hallo.c lesen und
                                  ausfuehren koennen

  $chmod u-w hallo.c         <--- user (Besitzer) soll keine
                                  Schreibrechte fuer hallo.c haben
```

6.9.3 Gleichsetzung der Rechte

Die Zugriffsrechte bestimmter Benutzerklassen können denen anderer gleichgesetzt werden.
Diese Form der Rechteerteilung wird von AIX leider nicht unterstützt !!

Syntax:

chmod [**ugoa**] = [**ugo**] **datei_1 ...** [**datei_n**]

Beispiel:

```
    $chmod o=g hallo.c           <--- others sollen die gleichen
                                      Zugriffsrechte haben wie group

    $chmod a=u hallo.c           <--- alle sollen die gleichen
                                      Zugriffsrechte haben wie user
```

6.10 chown – Besitzer wechseln

Mit *chown* (change owner) kann der Besitzer (und der Superuser) einen anderen Benutzer als Eigentümer einer Datei/Directory erklären. Der neue Eigentümer muß im System bereits als Benutzer bekannt sein.
(Dieser Befehl wird von AIX leider nur für den Superuser unterstützt!!)

Syntax:

chown benutzername datei_1 datei_2 ...

Beispiel:

```
   $ls -l hallo.c
-rwxr-xr-x  3  mueller 523  Jan 12   12:23  hallo.c

   $chown meyer hallo.c
   $ls -l hallo.c
-rwxr-xr-x  3  meyer   523  Jan 12   12:23  hallo.c
```

6.11 find – Dateien suchen

find sucht (rekursiv durch alle Unterdirectories) ab dem angegebenen Directory nach Dateien, die bestimmte Bedingungen erfüllen, und führt für diese bestimmte Aktionen durch.
Syntax:

find directory bedingung aktion_1 [**... aktion_n**]

Optionen:

- Für bedingung
  - **name "datei"** Dateien mit angegeben Dateinamen werden gesucht.
  - **perm XXX** Dateien mit oktalem Zugriffsrecht XXX werden gesucht.
  - **user name** Dateien mit Besitzer name werden gesucht.
  - **group name** Dateien mit Gruppe name werden gesucht.

- **size** [+/−]**n** Dateien mit der Größe n Blöcke a 512 Bytes werden gesucht. (+ größer als n, − kleiner als n)
- **atime** [+/−]**n** Dateien mit letztem Zugriff vor n Tagen werden gesucht. (+/− s.o.)
- **ctime** [+/−]**n** Dateien in den letzten n Tagen angelegt werden gesucht. (+/− s.o.)
- **mtime** [+/−]**n** Dateien in den letzten n Tagen modifiziert werden gesucht. (+/− s.o.)
- **newer datei** Dateien, deren Änderungsdatum jünger ist als das der angegebenen Datei werden gesucht.

- Für aktion
  - **print** Ausgabe des Pfadnamens der gefundenen Dateien
  - **space** wie −print, aber mit Größe der Datei
  - **exec ...** Ausführen eines SHELL-Kommandos; {} im Kommando werden durch den Pfadnamen ersetzt; Kommandoende mit der Zeichenfolge: Blank\; (s. Bsp.).

Beispiel:

```
$find /usr/meyer -name "*.c" -print

   ... finde alle Dateien unter /usr/meyer, die mit .c enden
       und zeige mir deren Pfadnamen. (* wird spaeter erklaert)


$find . -mtime +30 -exec rm {} \;        <--- . = im aktuellen directory

   ...finde alle Dateien in diesem Directory, die vor mehr als 30
      Tagen modifiziert wurden und loesche sie.
```

Außerdem können mehrere der Bedingungen verknüpft werden durch

- **-a** UND-Verknüpfung
- **-o** ODER-Verknüpfung
- **!** Negierung

7 Wildcards

Sicherlich haben Sie sich bei den bisher behandelten Befehlen schon gefragt (oder werden sich noch fragen), was tun, wenn viele Dateien mit dem gleichen Befehl bearbeitet werden müssen. Z.B. *rm pgm1.c pgm2.c ...pgm98.c* ist ja unendlich viel Tipparbeit !! Hierzu bietet UNIX, wie viele andere Betriebssysteme auch, das Wildcard-Konzept[10]. Unter Wildcards versteht man Sonderzeichen, die für eine beliebige Dateinamen-Zeichenkette in SHELL-Kommandos stehen können. Sie sind unter UNIX auf eine große Menge der Kommandos anwendbar.

Der Benutzer kann

*****	für beliebige, 0-14 Zeichen lange, Zeichenketten
?	für genau 1 Zeichen
[**xyz**]	für genau eines der Zeichen x,y oder z
[**a-f**]	für genau 1 Zeichen aus dem Bereich a-f (auch für Zahlen z.B. 1-9)

[10] manchmal auch Joker-Konzept gennant; abgeleitet vom Kartenspielen

als Wildcard benutzen.

Beispiel:

rm pgm*	löscht pgm, pgm1, pgma, pgmab, pgmab.c, ...
rm pgm*x.c	löscht pgmx.c, pgm1x.c, pgmax.c, pgmabcdx.c.......
rm ?	löscht a, b, c, 1, 3, z, ...
rm pgm?.c	löscht pgm1.c, pgma.c, pgmb.c, ...
rm pg??.c	löscht pgaa.c, pgab.c, pg34.c, ...
rm pgm[a-z].c	löscht pgma.c, pgmb.c, ... pgmz.c
rm pgm[123].c	löscht pgm1.c, pgm2.c, pgm3.c
rm [hmb]eute.c	löscht heute.c, meute.c, beute.c

Selbstverständlich können Wildcards auch auf *ls, chmod,....* angewandt werden (bei fast alle Kommandos, wo es einen Sinn ergeben würde). Beachten Sie aber bitte, daß *rm ** schnell zu fatalen Ergebnissen führen kann. Man sollte vorsichtshalber *rm -i ** (interaktive Abfrage) benutzen.

8 Filter

Unter UNIX existieren eine Reihe von Kommandos, die Filter genannt werden. Sie haben folgende Eigenschaften:

- Lesen von der Standardeingabe (**stdin**)
- Schreiben auf die Standardausgabe (**stdout**)
- Fehlermeldungen auf die Standardfehlerausgabe (**stderr**)

Ein Filter hat demnach einen Eingang, einen Ausgang und (für Notfälle) eine Fehlermeldungsausgabe. Im Grunde leuchtet dieses Verfahren ein: Ein Befehl soll eine Eingabe bearbeiten, das Ergebnis ausgeben und sich melden, falls etwas schief gegangen ist. Voreingestellt sind:

File	Gerät
stdin	Tastatur
stdout	Bildschirm
stderr	Bildschirm

Durch die Umlenkung dieser Standardeinstellungen können Dateien an die Befehle angeschloßen werden. Dies macht besonders bei großen Datenmengen (z.B. langen Ausgaben) einen Sinn.
Mit den folgenden Umlenkzeichen werden auf SHELL-Ebene Dateien mit den Befehlen verknüpft:

Umlenkzeichen	Resultat
>	Umlenkung der Ausgabe auf Datei (alter Inhalt wird überschrieben)
>>	Umlenkung der Ausgabe auf Datei (wird an alten Inhalt angehängt)
<	Umlenkung der Eingabe aus einer Datei
2>	Umlenkung der Fehlerausgabe auf Datei (z.B. Fehlerprotokoll)
2>>	wie >>, diesmal für Fehlerausgabe

Beispiel:

Ausgabeumlenkung:

```
$ls > otto        <--- die Ausgabe von ls in die Datei otto umlenken.
                       (otto enthaelt nun das Inhalts des aktuellen
                        Directories; mit cat koennte man sich den Inhalt
                        von otto nun anzeigen lassen)

$date >> otto     <--- das Tagesdatum wird an den Inhalt von otto
                       angehaengt.
```

Eingabeumlenkung:

```
$wc < otto        <--- wc (wordcount) soll Zeilen,Woerter und Zeichen in
                       otto zaehlen.
```

Umlenkung der Fehlermeldungen:

```
$cat fritz 2 > fehler     <--- wenn die Datei fritz nicht existiert,
                               wird die Fehlermeldung darueber in die
                               Datei fehler geschrieben.
```

Kombinationen:

```
$wc < otto > gezaehlte 2 > fehler       <--- der Inhalt von otto soll
                                        von wc verarbeitet werden,
                                        das Ergebnis in die Datei
                                        gezaehlte, evtl. auftretende
                                        Fehler in die Datei fehler
                                        geschrieben werden.
```

Beachten Sie, daß bei der Kombination mehrerer Umlenkzeichen von links nach rechts ausgewertet wird.

9 Pipes

Pipes sind Hilfsmittel um mehrere Filter miteinander zu verknüpfen. Der Begriff Pipe (Röhre) soll verdeutlichen, daß wie bei einer Öl-Pipeline Erzeuger und Abnehmer verbunden werden. Wie im Ölgeschäft arbeiten UNIX-Pipes auch nach dem fifo-Prinzip[11].

Der 1. Filter erzeugt ein Ergebnis (auf stdout) das durch eine Pipe an einen 2. Filter weitergereicht wird. Der 2. Filter stellt seine Eingabe (stdin) dabei an das andere Ende der Röhre und verarbeitet das Ergebnis des Ersten weiter. Dessen Ausgabe kann nun mittels einer weiteren Pipe wieder an den nächsten Filter gegeben werden usw.

[11] first in first out; die ersten Liter Öl beim Einpumpen kommen auch als erste am anderen Ende heraus. Die Reihenfolge der Daten bleibt gleich.

Die Verkettung zweier Filter miteinander geschieht durch einen senkrechten Strich in der Befehlseingabe.

Beispiel:

```
$ls | wc        <--- das Ergebnis von ls soll nicht ausgegeben
                     sondern von wc verarbeitet werden. Als Ergebnis
                     wird dann am Bildschirm die Ausgabe von wc stehen.
```

So können lästige Ergebnis-Zwischendateien eingespart werden. Bisher mussten wir dafür

```
$ls > otto      <--- das Ergebnis von ls in die Datei otto
$wc otto        <--- den Inhalt der Datei otto mit wc bearbeiten
```

schreiben. Beachten Sie daß,

```
$ls > wc
```

nicht dieselbe Wirkung hat. Mit > versucht UNIX das Ergebnis von *ls* in eine Datei Namens *wc* umzulenken. Nur | verknüpft stdout des 1. Filters mit stdin des 2. Filters.

Sie werden sich schnell mit dem Pipe-Mechanismus anfreunden, da viel Zeit und Tipparbeit eingespart wird. Noch ein einleuchtendes
Beispiel:

```
$ls -l | more
```

more (wird später ausführlicher behandelt) liest von der Standardeingabe und gibt diese Seitenweise wieder aus. Sollte der Bildschirm nicht zur Ausgabe von *ls* ausreichen, scrollt das Ergebnis an Ihnen vorbei. Wird das Inhaltsverzeichnis jedoch mit *more* weiterverarbeitet, kann in handlichen Seiten geblättert werden.
UNIX bietet außerdem die Verwendung sog. **named pipes** an. Hierbei handelt es sich um spezielle Dateien[12], die in der Lage sind Daten zwischen mehreren Prozessen auszutauschen. Natürlich müssen miteinander komunizierende Prozesse den gemeinsamen Pipenamen kennen.

10 Noch mehr Kommandos

Bisher haben wir Befehle zur Dateiorganisation kennengelernt. Wir wollen nun den Kommandoschatz unter UNIX um Befehle erweitern, die in der Praxis häufig benötigt werden. Dabei lassen wir es bei Syntax, Optionen und kleinen Beispielen bewenden. Eine Vertiefung kann nur am Objekt selbst stattfinden.

10.1 cat – Dateiinhalt anzeigen

cat schreibt Ihnen den Dateiinhalte auf den Bildschirm.
Syntax:

cat [Optionen] datei_1 [datei_2] ...... [datei_n]

Optionen:

- v macht nicht abdruckbare Zeichen sichtbar.
- div. weitere Optionen...

[12] Daher auch **named**, weil die Pipe einen Dateinamen erhält.

Beispiel:

```
$cat otto
 Citrone
 Apfel
 Birne
$cat -v hallo.prn
  ...
$cat otto fred hans > alle           <--- datei alle hat
  ...                                     anschliessend den Inhalt von
                                          otto, fred, hans
```

10.2 sort – Datei sortieren

Sortiert die Zeilen der angegebenen Datei lexigraphisch nach ASCII-Zeichensatz. *sort* wird häufig als Filter eingesetzt.
Syntax:

sort [**optionen**] **dateiname**

Optionen:

- – b Leer- und Tabulatorzeichen am Zeilenanfang ignorieren
- – d nur a-z,0-9,BLANK berücksichtigen
- – f Großbuchstaben wie Kleinbuchstaben behandeln
- – n numerische Zeichen nach ihrem Zahlenwert sortieren
- – r in umgekehrter Reihenfolge sortieren
- – m mischt nur, Eingabedateien sind schon sortiert
- – div. weitere Optionen (Spaltenposition, Sortierschlüssel.....)

Beispiel:

```
$cat otto                    <---- Inhalt von otto
 Citrone
 Apfel
 Birne
$sort otto                   <---- Inhalt von otto sortiert
 Apfel                             ausgegeben
 Birne
 Citrone
$cat otto                    <---- Inhalt bleibt jedoch unveraendert
 Citrone
 Apfel
 Birne
$sort otto > otto.sortiert   <---- Als Filter verwendet funktionierts
$cat otto.sortiert           <---- Inhalt von otto.sortiert
 Apfel                             ausgegeben
 Birne
 Citrone
$
$ls | sort -b > hans         <---- Anderes Beispiel
```

10.3 comm – Dateiinhalte vergleichen

Die Dateien werden zeilenweise miteinander verglichen.
Syntax:

comm [optionen] dat1 dat2

Es werden 3 Spalten ausgegeben:

1. Spalte: Zeilen, die nur in dat1 vorkommen
2. Spalte: Zeilen, die nur in dat2 vorkommen
3. Spalte: Zeilen, die in dat1 und dat2 vorkommen

Optionen:

- 1 — erste Spalte unterdrücken
- 2 — zweite Spalte unterdrücken
- 3 — dritte Spalte unterdrücken

Beispiel:

```
$cat otto1                    <---- Inhalt von otto1
 hallo
 huhu
$cat otto2                    <---- Inhalt von otto2
 hallo
 hehe
$comm otto1 otto2             <---- Vergleiche Inhalte

          hallo               <---- in beiden vorhanden
     hehe                     <---- nur in otto2 vorhanden
huhu                          <---- nur in otto1 vorhanden
```

10.4 date – Datum und Uhrzeit

Syntax:

date [+format]

format:

- + beliebiger Text mit
  - %d für Tag
  - %m für Monat
  - %y für Jahr
  - %H für Stunde
  - %M für Minute
  - %n für Zeilenumbruch

Beispiel:

```
$date                                <--- liefert Datum und Uhrzeit
Wed Sep 04 10:24:13  1991
$
$date +Datum:%d.%m.%y%nZeit%H:%M%n  <---- selbst formatiert
Datum:31.1.93
Zeit:11.54
```

10.5 time – Laufzeit

Irrtum: *time* liefert nicht, wie anzunehmen, die Uhrzeit (s. *date*). *time* als Vorsatz vor jedem UNIX-Befehl gibt die Laufzeit des Befehls (nach seiner Ausführung) aus. Diese wird angegeben in echter Laufzeit (real), Benutzermodus (user) und Systemmodus (sys).
Syntax:

time kommando

Beispiel:

```
$time who             <--- liefert Anwesende im System ( who ) und
                           anschliessend die Zeit, die der Rechner
                           zur Ausfuehrung von who benoetigte

mueller       hft/0     Sep 04 09:33      <---- who

real       0m0.06s                       <---- Zeiten
user       0m0.01s
sys        0m0.04s
```

10.6 file – Dateiart ermitteln

file untersucht eine Datei oder Dateien in einem Directory auf ihre Dateiart (Text, Directory, C-Programm, ...)
Syntax:

file [optionen] datei1 ......

Optionen:

- Maschinenabhängig (s. Manualeintrag)

Beispiel:

```
$file /usr/mueller/*    <---- Ergebnis Maschinenabhaengig

C:               directory
bin:             directory
bsp1:            ascii text
x1:              empty                <---- leere Datei
hallo.c:         c programm text
protok:          commands text
bsp2:            ascii text

$
```

10.7 grep – Textmustersuche

grep durchsucht die angegebene(n) Datei(en) nach dem angegebenen Textmuster, und gibt die Zeilen der Datei(en) aus, die das Muster enthalten.
Syntax:

grep [optionen] textmuster datei1 ......

Optionen:

- – **c** Anzahl der Trefferzeilen wird gezählt.
- – **f** dateiname — textmuster steht in datei, nicht im Argument.
- – **h** keine Dateinamen in der Ausgabe (header weglassen).
- – **n** vor jeder Trefferzeile die entsprechende Zeilennummer ausgeben.
- – **v** Invertierung des Befehls; alle nicht-Treffer ausgeben.
- – **y** Großbuchstaben wie Kleinbuchstaben behandeln.

Beispiel:

```
$grep -n "huhu" /usr/mueller/*

bsp1:54:huhu
bsp1:63:hallohuhuhallo
otto:103:huhuihrda
```

10.8 lp – Drucken

lp reiht den Druckauftrag in die Druck-Warteschlange des Systems. Vorsicht beim Löschen der Datei bevor noch nicht wirklich auf Papier ausgedruckt wurde.
Syntax:

lp [optionen] datei1 ......

Optionen:

- – **d***dest* Drucker oder Druckertyp wählen (falls mehr als einer vorhanden)
- – **m** Sendet Post, nachdem die Datei gedruckt wurde

- **n***number* Anzahl Kopien (Standard 1)
- **t***title* Kopfseite mit *title* vorweg

Beispiel:

```
$lp -n4 -m -tSuperprogramm hallo.c
```

10.9 lpstat – Druckerwarteschlange

lpstat gibt Informationen über die Druck-Warteschlange des Systems.
Syntax:

lpstat [optionen] [auftrag]

Optionen:

- **v** Namen der Drucker und deren Pfadnamen ausgeben
- **u***user* Status der Druckaufträge für Benutzer ausgeben
- **p** Druckerstatus ausgeben
- **a** Annahmestatus von Zielen für Druckaufträge ausgeben
- div. weitere Optionen

Auftrag:

- Infos nur über den Druckauftrag mit der Nummer *auftrag* (wurde vom System beim *lp* vergeben).

Beispiel:

```
$lpstat 3240            <--- Statusdaten fuer Druckauftrag 3240
                             (3240 aus Kommando ps s.u.)
$lpstat -umueller       <--- Statusdaten fuer Druckauftraege
                             des Benutzers mueller
```

10.10 pr – Druckeraufbereitung

pr bereitet Dateien druckergerecht auf und wird meist als Filter vor dem Drucken benutzt.
Syntax:

pr [optionen] dateien

Optionen:

- **h** "Text..." Text als Seitenüberschrift
- **l***n* Mit n Zeilen pro Druckseite (Standard 66; max 72)
- **n** n-spaltige Ausgabe
- **l***n* Mit n Zeichen Zwischenraum zum linken Druckrand
- **t** unterdrückt die von pr erzeugten Titel,Zeilennummern und Datum
- **w***n* Mit n Zeichen als Seitenbreite (Standard 72)
- div. weitere Optionen...

Beispiel:

```
$pr -h "Berguessungsprogramm !!" -l30 hallo.c | lp
```

10.11 more – Bildschirmausgabe-Steuerung

more wird meist als Filter benutzt. Der Befehl dient der bildschirmgerechten Aufarbeitung verschiedener Befehlsausgaben oder Dateiinhalte. Die Bildschirmausgabe hält nach jeder gezeigten Seite an, und zeigt am unteren Rand an wieviel noch an Ausgabe folgt.
Syntax:

more [optionen] datei
Befehl | more [optionen]

Optionen:

- **n** Ganzahlige Angabe für die Größe der Ausgabeseiten in Zeilen.
- **p** *zeichenkette* ersetzt das Promptzeichen „:„ gegen die angegebene Zeichenkette. Ein %d in Zeichenkette gibt die aktuelle Seitenzahl aus.
- +*zeilennummer* Start bei Zeilennummer
- +/*muster*/ Start bei der ersten Zeile, die Muster enthält

Benutzung:

- Leertaste: 1 Seite weiterblättern.
- Enter: 1 Zeile weiterblättern.
- `Ctrl`+`c`, oder `q`: *more* beenden.
- `h`: eine Übersicht aller verfügbaren *more*-Kommandos.

Beispiel:

```
$ls -l | more          <--- Seitenweise Ausgabe des Kommandos
$more hallo.c          <--- Seitenweise Ausgabe des Dateiinhalts
```

Die Manualpages des Systems werden automatisch mit *more* aufbereitet.

10.12 passwd – Passwort ändern

Der Benutzer wird nach dem alten Passwort gefragt. Danach muß er 2 mal das neue Passwort (min. 6 Zeichen lang) eingeben.
Syntax:

passwd

Beispiel:

```
$passwd
mueller‘s Old password:elfriede          <--- Passwoerter werden
mueller‘s New password:liesel                 am Bildschirm nicht
Enter the new password again:liesel           sichtbar; hier nur
Changing password for "mueller"               zur Verdeutlichung.
$
```

10.13 ps – Prozessinformationen

ps gibt die Statusinformationen zu aktiven Prozessen. Dies ist besonders im Zusammenhang mit selbstinitiierten Hintergrundprozessen wichtig (s. Kapitel 12 auf Seite 35). Es werden Prozeßnummer (**PID**), Dialogstation (**TTY**), verbrauchte Rechnerzeit (**TIME**) und Kommandoaufruf (**CMD**) angezeigt.
Syntax:

ps [optionen] [-prozessid...]

Optionen:

- **e** Infos über alle Prozesse aller Benutzer
- **f** vollständige Infoliste
- **l** long Infoliste
- **u** *userid* Infos zu allen Prozessen des Benutzers userid
- div. weiter Optionen...

Beispiel:

```
$ps
  PID    TTY      TIME  CMD
 5392    hft/0    0:00  ps               <--- ps-kommando selbst
 5759    hft/0    0:12  -ksh             <--- SHELL (ksh)

$ps -l
     F S  UID  PID   PPID   C PRI NI ADDR   SZ   TTY      TIME  CMD
200001 R  203  5392  5759  10  65 20  b29  164   hft/0    0:00  ps
240801 S  203  5759     1   1  60 20  c4d  148   hft/0    0:12  ksh

           ^    ^     ^                           ^        ^     ^
           !    !     !                           !        !     !
           !    !     !        Terminal-Adresse --!        !     !
           !    !     !-- Vater-Prozess-ID                 !     !
           !    !-- Prozess-ID                             !     !
           !                      verbrauchte Zeit  ---!     !
           !--- User-ID (Benutzernummer)                         !
                                        Laufender Prozess ---!
                                        (Befehl)
```

10.14 tail – Dateiende ausgeben

Mit dem Kommando *tail* wird, ab einer angegebenen Position bis zum Dateiende, der Inhalt einer Datei angezeigt. Die Ausgabe erfolgt auf *stdout* (Bildschirm).
Syntax:

tail [**−/+n**] [**position**] [**optionen**] **datei**

Position:

- **−n** Anzahl Zeilen relativ zum Dateiende
- **+n** Anzahl Zeilen relativ zum Dateianfang

Optionen:

- **c** keine Zeilen sondern Zeichen
- **b** keine Zeilen sondern Blöcke a 512 Bytes

Beispiel:

```
$tail -50 -c hallo.c
    .
    .
$tail -50 hallo.c > huhu.c
    .
    .
```

10.15 tar – Disketten/Streamer/Bänder

Der *tar*-Befehl dient zum Anlegen von Archiven. Er wird meist zum Ein- und Auslagern von Daten auf Disketten oder Streamer[13] benutzt.
Syntax:

tar [**optionen**] **datei(en)/directory**

Optionen:

- **c** : Sicherung anlegen (Export)
- **t** : Infos zu Dateien in der Sicherung (Info)
- **x** : Datei(en) aus der Sicherung zurückholen (Import)
- **C directory** : Pfadangabe
- **f device** : Device-Angabe, ob Diskette/Streamer-Laufwerk etc. Fragen Sie den Superuser nach dieser Adresse. (Standard- Streamerlaufwerk = /dev/rmt0)
- **v** : Liste aller betroffenen Dateien
- div. weitere Optionen s. Manualpage

[13]Streamer: sehr verbreitete und kapazitätsstarke Bandkassetten

Beispiel:

```
$tar -cvf /dev/rmt0 otto.c hugo.c Makefile
      ^      ^        ^     ^      ^
      !      !        !     !      !
      !      !        !-----!------!------- zu sichernde Dateien
      !      !---- Auf Standard-Bandlaufwerk
      !---- c:sichern
            v:Anzeige, welche Dateien betroffen
            f:fuer /dev/rmt0

$tar -tvf /dev/rmt0
      ^      ^
      !      !---- Vom Standard-Bandlaufwerk
      !---- t:Inhaltsverzeichnis
            v:Anzeige, welche Dateien betroffen
            f:fuer /dev/rmt0

$tar -xvf /dev/rmt0  /usr/mueller
      ^      ^           ^
      !      !           !---- In das Unterverzeichnis ... kopieren
      !      !---- Vom Standard-Bandlaufwerk
      !---- x:Einspielen aus der Sicherung
            v:Anzeige, welche Dateien betroffen
            f:fuer /dev/rmt0
```

10.16 wc – Elemente zählen

wc (wordcount) zählt Zeilen, Worte und Zeichen in einer Datei. Der Befehl wird häufig als Filter benutzt.
Syntax:

wc [optionen] datei

Optionen:

- **l** nur Zeilen zählen
- **w** nur Worte zählen
- **c** nur Zeichen zählen

Beispiel:

```
$wc hallo.c
$ls -l | wc -c
```

10.17 who – aktive Benutzer

Das Kommando *who* informiert über im System befindliche und aktive Benutzer.
Syntax:

who

11 vi – der Editor

Editoren sind Werkzeuge um neue Dateien anzulegen, bzw. den Inhalt einer schon bestehenden Datei zu verändern.
Die unter UNIX betriebenen 3 Editoren **ed**, **sed** und **vi** sind nicht gerade das, was man bedienerfreundlich nennt. Da **ed** und **sed** zwar relativ leicht erlernbar sind, zum anderen jedoch zeilenorientiert arbeiten, wird an dieser Stelle auf ihre Vorstellung verzichtet[14].

Wir wollen den **vi** (sprich: wie ei) kennenlernen, der einen mächtigen Kommandoumfang besitzt, und interaktiv sowie bildschirmorientiert (fullscreen) arbeitet.

11.1 Aufruf

Um eine Datei mit dem **vi** zu bearbeiten (anlegen, modifizieren), geben Sie den Befehl:

vi dateiname [**dateiname2 ....**]

ein.
Beispiel:

```
vi hallo.c
```

Der Bildschirminhalt wird gelöscht, der Cursor steht in der linken oberen Bildschirmecke; **vi** ist nun bereit den Inhalt der angegebenen Datei zu bearbeiten.

Der **vi** kennt zwei Betriebsmodi:

1. Kommandomodus – Befehle, die den Text oder die ganze Datei beeinflussen
2. Eingabemodus – Eingabe von Text

Nach dem Aufruf ist der Kommandomodus aktiv.

11.2 Verlassen

Zum Verlassen des Editors geben Sie:

:q CR	zum Verlassen ohne Änderungen gemacht zu haben
:q! CR	zum Verlassen ohne Änderungen abzuspeichern
:wq CR	zum Verlassen mit Änderungen abspeichern (auch **ZZ** ohne CR möglich)

ein.

[14] weitere, verbreitete Editoren: **ced** auf Siemens-SINIX und **e** auf Nixdorf-TARGON-Maschinen

11.3 Nur Lesen

Der **vi** kann auch mit

view dateiname [dateiname2 ....]

aufgerufen werden. Der Editor wird dann im read-only-Modus für die Datei aufgerufen. D.h. Änderungen sind zwar innerhalb des **vi** möglich, beim Verlassen oder Abspeichern erscheint die Fehlermeldung „`File is read only`“. Möchten sie dann trotzdem Änderungen mitabspeichern, geben sie

:w! `CR`

ein. Das Ausrufungszeichen (!) erzwingt also Befehle wie q und w ohne Rücksicht auf Verluste.

11.4 Texteingabe

Nach dem Aufruf des **vi** ist der Kommandomodus aktiv. Um Text einzugeben muß in den Texteingabe-Modus gewechselt werden. Dies geschieht durch

i (insert)	Text vor dem Cursor einfügen
I (insert)	Text vor dem Zeilenanfang einfügen
a (append)	Text hinter dem Cursor einfügen
A (append)	Text hinter dem Zeilenende einfügen
o (open)	neue Zeile, dann wie i

Der gewünschte Text wird eingegeben. Anschließend wird mit der `ESC` wieder in den Komandomodus zurückgeschaltet.

11.5 Cursor bewegen

Innerhalb einer Datei kann seitenweise und zeilenweise vorwärts/rückwärts geblättert werden. Darüberhinaus können Dateianfang und -ende direkt angesprungen werden, genauso wie Zeilen, die sich auch durch Suchstrings lokalisieren lassen.

Hier zur Vereinfachung eine Liste der “Bewegungsmöglichkeiten“ innerhalb einer Datei:

lokal bewegen – Cursor

w	zum Beginn des nächsten Wortes
e	zum Ende des nächsten Wortes
b	zum Beginn des vorherigen Wortes
`Backspace`	Cursor eine Position nach links
h	Cursor eine Position nach links
`LEER`	Cursor eine Position nach rechts
l	Cursor eine Position nach rechts
j	Cursor eine Position nach unten
k	Cursor eine Position nach oben
`RETURN`	Cursor an Anfang der nächsten Zeile
H	Cursor auf erste Zeile des Bildschirms
M	Cursor auf Mitte des Bildschirms
L	Cursor auf letzte Zeile des Bildschirms

Das sieht zwar unübersichtlich aus, schauen Sie aber mal auf die Tastatur. Sie werden z.B. das Bewegen des Cursors nach oben/unten/links/rechts durch handliches Zusammenstehen der entsprechenden Tasten als angenehm empfinden.

global bewegen – Bildschirmweise

CTRL + D	halbe Seite abwärts blättern (down)
CTRL + U	halbe Seite aufwärts blättern (up)
CTRL + F	ganze Seite abwärts blättern (forward)
CTRL + B	ganze Seite aufwärts blättern (backward)
CTRL + E	Bildschirm eine Zeile abwärts
CTRL + Y	Bildschirm eine Zeile aufwärts
:n + CR	auf n'te Zeile am Bildschirm springen
n + G	auf n'te Zeile in der Datei springen (Goto)

Wird der Cursor an den oberen oder untere Rand des Bildschirms geführt, so scrollt der Dateiinhalt um eine Zeile in die entsprechende Richtung.

11.6 Textsuche

Die Suche nach einem bestimmten Textmuster liefert zwei Ergebnisse:

1. Auskunft, ob der Textteil in der Datei vorhanden ist.
2. Positionierung des Cursors an diese Stelle, falls vorhanden.

Tippen Sie im Kommandomodus:

Syntax:

/suchmuster/ CR

Dabei wird ab der aktuellen Position des Cursors zyklisch vorwärts durch die Datei gesucht. Das Ganze zyklisch rückwärts mit:

Syntax:

?suchmuster? CR

Wiederholen des Suchvorgangs (vorwärts wie rückwärts):

Syntax:

n (Buchstabe „n“)

Auch der **vi** unterstützt bei der Textsuche das bereits vorgestellte Wildcard-Konzept.
Ein beliebiges Zeichen innerhalb eines Suchmusters wird mit **. (Punkt)** dargestellt.
Beispiel:

```
/M.yer/     <---- fuer Mayer, Meyer, Moyer, .....
```

Eine beliebige Anzahl sich wiederholender Zeichen innerhalb eines Suchmusters wird mit *** (Sternchen)** dargestellt.
Beispiel:

```
/hal*o/     <---- fuer halo, hallo, halllo, .....
```

Suchmusters, die am Ende einer Zeile gesucht werden sollen, werden mit **$ (Dollar)** beendet.
Beispiel:

```
/Fernuni$/     <---- fuer Fernuni als letzte Zeichenkette in einer
                        Zeile
```

Das Gegenstück zu **$ (Dollar)**, also Textmuster am Anfang einer Zeile stehend heißt ˆ **(Circumflex)**.
Beispiel:

```
/^Fernuni/     <---- fuer Fernuni als erste Zeichenkette in
                       einer Zeile
```

Weitere Tricks sind in der Zusammenfassung am Ende dieses Scriptes verewigt.

12 Hintergrundprozesse

Wie am Anfang der Broschüre (s. Graphik 1, Seite 29) erwähnt, findet zwischen dem Benutzer und dem Rechner stets ein Dialog statt. Dieser wiederholt sich immer in der Form:

1. SHELL wartet auf Eingabe
2. Benutzer startet Kommando (Prozess)
3. SHELL bearbeitet Kommando
4. wieder zu Punkt 1.

Der Benutzer muß warten, bis der Rechner mit der Abarbeitung seines Auftrages fertig ist.
UNIX kann jedoch mehr. Durch die Mehrprogrammfähigkeit ist der Benutzer in der Lage mehrere Programme/Kommandos/Prozesse gleichzeitig am Rechner arbeiten zu lassen. Dies geschieht durch sog. **Hintergrundprozesse**[15]. Von der SHELL aus kann der Benutzer sagen: „Diesen Befehl bitte ausführen, ohne daß ich bis zu seiner Beendigung warten muß“. Bei den Kommandos, die wir bisher kennengelernt haben ist dies wahrscheinlich überflüssig. Sie haben eine derart kurze Bearbeitungszeit, daß der Rechner schneller fertig ist, als wir neue Eingaben getätigt haben.
Da wir aber vielleicht zu einem späteren Zeitpunkt selbst Programme erstellen wollen, deren Laufzeit durchaus mehrere Minuten oder gar Stunden dauern kann, halten wir uns diese Tür mal offen.
Welche Art von Programmen kommen also für die Hintergrundverarbeitung in Frage?

- Programme, die eine lange Verarbeitungszeit benötigen
- Programme, die nicht interaktiv arbeiten. Wenn bei der Bearbeitung ständig Dialogabfragen auftauchen, kann ich mich sowieso keiner anderen Arbeit widmen.
- Programme, die möglichst keine Bildschirmausgaben erzeugen. Sie würden mich bei anderen Tätigkeiten ebenfalls stören.

[15]Mancherorts auch als Batch-Job bekannt

Ein solches Kommando, das im Hintergrund arbeitet haben wir (ohne es zu wissen) bereits kennengelernt. Der *lp*-Befehl meldet nach seinem Start nur, daß der Druck initiiert wurde, und der Benutzer kann in der SHELL fortfahren. Im Hintergrund arbeitet sich der Druckauftrag durch die Druckerwarteschlange (andere Benutzer Drucken evtl. zur gleichen Zeit) bis er tatsächlich die Daten auf das Papier geschoben hat.

Um eigene Hintergrundprozesse zu starten gibt der Benutzer hinter dem Kommando oder Programmaufruf ein **&** ein.

Beispiel:

```
 $find / -name "*.c" -print > gefundene.dat &
1241
  ^                                         ^
  !                                         |
  PID vom System vergeben                   |
                                            |
                        Ab root den System-Dateienbaum nach
                        C-Programmen durchsuchen kann lange dauern.
                        deshalb als Hintergrundprozess starten.
 $
```

Das System gibt die sog. **Prozeß-ID** auf den Bildschirm aus; die SHELL meldet sich wieder arbeitsbereit. Mit dem in Kapitel 10.13 (s.S. 29) vorgestellten Befehl **ps** kann jederzeit eine Statusabfrage zum Prozess gemacht werden.

Möchte der Benutzer einen Prozeß vorzeitig beenden, gibt er

kill -9 pid

mit pid als Prozeßnummer ein.

Mit Hilfe des *kill*-Kommandos, können sich Benutzer und Hintergrundprozeß miteinander „unterhalten“; je nach Optionen bringt *kill* den Prozeß nicht umbedingt um. Nur die Option **-9** bricht einen Auftrag ab.

Klar, daß Sie nur Ihre eigenen Prozesse stoppen können. Die Ausnahme bildet, wie immer, auch hier der Superuser.

13 Programmieren mit der Shell

Die SHELL war für uns bisher „Befehlsempfänger und -ausführung". Neben den bisher genannten Eigenschaften Befehlsinterpretation, Wildcardauflösung, Ein-Ausgabeumlenkung, Pipelines und Hintergrundverarbeitung kann mit der SHELL auch programmiert werden. Wie bei einer Programmiersprache verfügt die SHELL über:

- SHELL-Variablen
- Programmiersprachen-Konstrukte (Verzweigung, Schleifen,..)

Dabei kann eine Folge von SHELL-Befehlen in sog. SHELL-Scripten (ausführbare Dateien) hinterlegt werden[16]. Zusätzlich können Sie auch alle bisher kennengelernten SHELL-Befehle hinzufügen.

13.1 Programmieren – aber wo ?

Die Programmierung der SHELL kann auf zwei Arten erfolgen.

1. Durch direktes Eintippen der gleich folgenden Befehls-Sequenzen auf SHELL-Ebene. Dabei wird bei Schleifen und Verzweigungen, die sich über mehrere Zeilen erstrecken können, ab der zweiten Zeile das Promptzeichen geändert. Der Benutzer soll daran erkennen, daß die Eingabe noch nicht beendet ist.
2. In sog. SHELL-Scripten. Hierzu wird mit einem Editor (z.B. dem **vi**) eine Datei erstellt, deren Dateiname Sie frei vergeben können; eine Dateinamenserweiterung (Extension) ist nicht nötig. In die Datei wird pro Zeile jeweils eine Anweisung geschrieben. Als Anweisungen können alle bisher bekannten UNIX-Befehle sowie die gleich folgenden SHELL-Programmkonstrukte verwendet werden. Der Aufruf des SHELL-Scriptes erfolgt wie ein normales UNIX-Kommando; tippen Sie einfach den Dateinamen ein[17]. Diese Methode eignet sich insbesondere dann, wenn eine Folge von Anweisungen mehrmals benötigt wird.

13.2 SHELL-Variablen

Auch ohne Ihre Absicht existieren nach dem login eine Anzahl von SHELL-Variablen. Mit dem Befehl

set

werden sie angezeigt.

Die Wichtigsten:

Variable	Bedeutung
$$	PID (Prozess-ID)
$#	Anzahl SHELL-Parameter
$1 bis $n	SHELL-Parameter
$?	Exit-Status/Returncode, 0=Ok (true)
$HOME	Homedirectory
$PATH	Suchreihenfolge
$PS1	Erstes Promptzeichen
$PS2	Zweites Promptzeichen
$LOGNAME	Login Name

[16] für DOS-Kenner: ähnlich den .bat-Dateien; für VM-Kenner: ähnlich den REXX-Dateien

[17] Falls dies auf Anhieb nicht klappt müssen sie das SHELL-Script **ausführbar** machen. D.h. ändern Sie die Zugriffsrechte für die Datei mit: *chmod 751 dateiname*

SHELL-Variablen enthalten Zeichen oder Zeichenketten. Auch wenn man mit ihnen Rechenoperationen durchführen kann, gibt es keine Numerischen.

Für Variablennamen sind Zeichen und „_“ (Unterstrich) zulässig. Groß-Kleinschreibung wird unterschieden. Eine Zuweisung erfolgt durch das „=“ (Gleichheitszeichen). Angesprochen werden sie (nach eineer Zuweisung) durch ein „$“ (Dollar) vor ihrem Namen.

Beispiel:

```
Name=Hugo          <--- einfache Zuweisung
Name=Hugo Meyer    <--- nicht zulaessig, da pro Variable max.
                        1 Zeichenkette

Name="Hugo Meyer"  <--- so gehts  "" fasst mehere Zeichenketten zu
                        einer zusammen
echo $Name         <--- Bildschirmausgabe des Variableninhaltes

Name=12345         <--- einfache Zuweisung
```

Beachten Sie, daß bestimmte SHELL-Variablen nach dem login schon vom System besetzt wurden (s. **set**-Befehl). Diese können zwar auf andere Werte gesetzt werden, die Verantwortung tragen jedoch Sie selbst.

Selbstdefinierte SHELL-Variablen müssen vor der weiteren Nutzung durch andere Programme/Prozesse dem System bekannt gemacht (exportiert) werden:

export `varname1 varname2 ...`

Beispiel:

```
$NAME="Hugo Meyer"
$export NAME
$echo $NAME
```

Nach dem logout sind SHELL-Variablen verloren. Sollen sie von dauerhaftem Bestand sein, müssen sie in der `.profile`-Datei im Homedirectory eingetragen werden. Die `.profile` ist ein SHELL-Script, das beim login durchlaufen wird.

13.3 Programmkonstrukte

Wie in fast allen Programmiersprachen üblich, hat auch die SHELL Programmkonstrukte, die Verzweigungen und Schleifen ermöglichen.
Die Beispiele in den nun vorgestellten Punkten sind von der SHELL aus eingetippt.

13.3.1 if – Verzweigung

Der IF-Befehl kann in Abhängigkeit einer Bedingung zu verschiedenen Aktionen verzweigen. Ist die Bedingung erfüllt (0/TRUE) so werden die Kommandos des `then`-Zweiges ausgeführt. Andernfalls (nicht-0/FALSE) treten die Aktionen des `else`-Zweiges in Kraft. Der `else`-Fall kann auch entfallen; die Verarbeitung wird stets nach dem `fi` fortgeführt.

Syntax:

if BEDINGUNG
then KOMMANDOS
[**else KOMMANDOS**]
fi

Beispiel:

```
 $if [ $LOGNAME="meyer" ]                  <-- Abfrage bin ich meyer
 >then echo "Hallo Hugo !!"                <-- Anweisung falls ja (0)
 >else echo "Sie kenne ich gar nicht..."   <-- Anweisung falls nein (!=0)
 >fi                                       <-- Ende
Hallo Hugo !!                              <-- Ergebnis
 $                                         <-- SHELL meldet sich wieder
```

Unser Beispiel-user hatte also den LOGNAME=meyer. Sie sehen, daß das Promptzeichen bei der Eingabe auf SHELL-Ebene von $ nach > wechselt falls ein Befehl (hier die if-Abfrage) noch nicht beendet ist. Beachten Sie außerdem, daß die Vergleichsbedingung in eckigen Klammern steht, und die einzelnen Komponenten darin mit Blanks getrennt werden; UNIX ist hier eben sehr kleinlich.

Was für BEDINGUNG noch alles stehen kann zeigt der folgende Abstecher.

13.3.2 Bedingungen

Bedingungen sollten immer (wenn möglich) mit der praktischen Funktion `test` gestellt werden. Mit Test können Sie Eigenschaften von Dateien und Directories prüfen, sowie SHELL-Variablen auf Zeichenfolgen und numerische Werte hin untersuchen.

Syntax:

test [**optionen**] **datei/variable**

Optionen:

- Dateistatus Tests
  - `test -f datei` true, wenn Datei existiert
  - `test -r datei` true, wenn Datei lesbar ist
  - `test -w datei` true, wenn Datei beschreibbar ist
  - `test -x datei` true, wenn Datei ausführbar ist
  - `test -d datei` true, wenn Datei ein Directory ist
  - `test -s datei` true, wenn Datei nicht leer ist
- Numerische Tests
  - `test x1 -eq x2` true, wenn x1 gleich x2
  - `test x1 -ne x2` true, wenn x1 ungleich x2
  - `test x1 -gt x2` true, wenn x1 größer x2
  - `test x1 -lt x2` true, wenn x1 kleiner x2
  - `test x1 -le x2` true, wenn x1 kleiner gleich x2
  - `test x1 -ge x2` true, wenn x1 größer gleich x2

- Zeichenketten Tests
  - `test x1` true, wenn x1 nicht leer ist
  - `test -z x1` true, wenn x1 die Länge 0 hat
  - `test -n x1` true, wenn x1 nicht die Länge 0 hat
  - `test x1 = x2` true, wenn x1 gleich x2
  - `test x1 != x2` true, wenn x1 nicht gleich x2

Beispiel:

```
if test `who | wc -l` -eq 1
then
     echo Du bist allein !!
     echo Mach was draus...
fi                                <-- else kann ja auch wegfallen
```

Darüberhinaus lassen sich tests

- **!** verneinen
- **-a** UND-Verknüpfen
- **-o** ODER-Verknüpfen
- **()** Klammern

Anstatt `test...` können auch die eckigen Klammern `[ ]` um die Abfrage geschrieben werden; schnellere Variante.
Sie werden nun verstehen warum man

<u>In UNIX niemals ein Programm test nennen</u>

darf. Dieser Kommandoname ist halt schon für die SHELL-Programmierung vergeben worden.

13.3.3 case – Verzweigung

Syntax:

case VARIABLE in
liste1) KOMMANDOS;;
liste2) KOMMANDOS;;
.....
esac

Beispiel:

```
 $ESSEN=banane
 $export ESSEN
 $case $ESSEN in
 >apfel|pfirsich|banane|kirsche) echo "$ESSEN ist eine Frucht";;
 >rettich|kohl|erbse|moehre) echo "$ESSEN ist Gemuese";;
 >schwein|fisch|rind|hase) echo "$ESSEN ist eine Tier";;
 >*) echo "$ESSEN kenne ich nicht";;          <-- * fuer alles andere
 >esac
banane ist eine Frucht
 $
```

Sie sehen, so komplexe Anweisungen gibt man wegen der Tipparbeit besser in ein SHELL-Script.

13.3.4 for – Schleife

Mir der for-Schleife können für jeden Eintrag in einer Liste die gleichen Kommandos ausgeführt werden.

Syntax:

> **for VARIABLE in WORT1 WORT2.... WORTn**
> **do KOMMANDOS**
> **done**

Beispiel:

```
$for FRUCHT in apfel kirsche banane
> do
> echo $FRUCHT
> done
apfel
kirsche
banane
$                                         <-- SHELL meldet sich wieder
```

13.3.5 while – Schleife

Wiederholtes durchlaufen der Kommandos solange die Bedingung wahr (also TRUE oder 0) ist.
Syntax:

> **while BEDINGUNG**
> **do KOMMANDOS**
> **done**

Das Beispiel sei diesmal in einem SHELL-Script namens `schleife`.

Beispiel:

```
echo "Gib ein Wort ein:"
read a                                  <--- einlesen einer VARIABLEN
while test "$a" != "stop"
do
   echo $a
   echo "Gib noch ein Wort ein:"
   read a
done
```

Aufgerufen wird dieses SHELL-Script mit `schleife`. Der Befehl `read a` verlangt die Eingabe eines Wertes vom Benutzer. Dieser wird der SHELL-Variablen `a` zugewiesen.

Als Ergebnis werden die Eingaben einfach wieder auf dem Bildschirm ausgegeben bis ßtop äls Schlüßelwort die Schleife beendet.

13.3.6 until – Schleife

Until-Schleifen arbeiten wie While-Schleifen. Einziger Unterschied: Wiederholtes durchlaufen der Kommandos solange die Bedingung falsch ist (also FALSE oder ungleich 0).
Syntax:

until BEDINGUNG
do KOMMANDOS
done

While und Until-Schleifen können mit dem Befehl **break** irregulär beendet werden, und mit **continue** an den Schleifen-Anfang springen. Ist die Bedingung nicht mehr erfüllt (oder wurde mittels break abgebrochen), geht es nach dem `done` weiter.

13.4 Substitution

In SHELL-Anweisungen ist die Reihenfolge der Ersetzung von SHELL-Variablen sehr wichtig. Der SHELL muß mitgeteilt werden, welche der Sonderzeichen (z.B. Wildcards ∗, ?,...) vor der Ausführung des eigentlichen Befehls ausgewertet werden sollen. Dies geschieht durch sog. Metazeichen:

'string' (rechtsgerichtete Apostrophe): in string sollen keine Ersetzungen stattfinden.

"string" (doppelte Apostrophe): in string sollen nur SHELL Variablen ersetzt werden.

‘string‘ (linksgerichtete Apostrophe): string wird ausgewertet und dann erst dem eigentlichen Kommando zur Verfügung gestellt.

Beispiel:

```
XYZ = pwd
echo $XYZ                       -->  >pwd<
echo ‘$XYZ‘                     -->  >/usr/mueller<
echo 'Mein Zuhause ist $XYZ'    -->  >Mein Zuhause ist $XYZ<
echo "Mein Zuhause ist $XYZ"    -->  >Mein Zuhause ist pwd<
```

13.5 Prozessüberwachung

Etwas graue Theorie über die Prozesse unter UNIX:

Ein Prozess ist die Ausführung eines Programmes mit allen zum Ablauf benötigten Informationen (Programmcode, Daten, Pfade, Zähler,..).

Beim Einloggen in die Maschine wird z.B. ein solcher Prozeß gestartet: die SHELL. Wird nun ein Kommando unter ihr abgesetzt (z.B. *find*) so geschieht folgendes:

1. **fork()**: Die SHELL dupliziert sich selbst. Zwei Prozesse existieren: Vater (alter Prozeß) und Sohn[18] (Duplikat).
2. **exec**: Der Sohn-Prozeß überlagert sich mit der Ausführung des find-Kommandos; führt es aus.
3. **wait**: Gleichzeitig wartet der Vater-Prozeß auf die Beendigung seines Sohnes
4. **exit**: Der Sohn-Prozeß ist fertig und stirbt.
5. Vater-Prozeß SHELL wartet wieder auf Kommandos.

Dieses Vorgehen weicht bei der Hintergrundverarbeitung nur durch den Punkt 3. ab. Der Vater wartet nicht auf das Ende des Sohnes.

Sie können den Ablauf mit dem Kommando **ps** beobachten: Zu Ihrer Benutzernummer existiert stets der sh-Prozeß (SHELL). Setzen sie ein **find ... &** in den Hintergrund ab. Durch **ps -l** sieht man nun, daß ein find-Prozeß arbeitet, dessen PPID (parent-prozess-id) die PID (prozess-id) des sh-Prozesses ist — Vater *sh* hat einen Sohn *find*.

Soviel zur Theorie. Diese Vater-Sohn-Beziehung kann in der SHELL-Programmierung durch die o.g. 4 Befehle selbst erzeugt werden. Hier nochmal im Detail:

- **fork()** dupliziert den aktuellen Prozess mit Umgebung (SHELL-Variablen,...). Nach der Duplizierung befinden sich Vater und Sohn an der gleichen Stelle im SHELL-Script. Sie identifizieren sich anhand des Wertes, den fork als Status zurückliefert.

0	im Kindprozess
> 0	im Elternprozess (Status ist dann PID des Sohnes)
< 0	im Elternprozess, fork() lief auf Fehler

- **exec(file,argumente....)** überlagert/startet Programm, das in file steht. Dem Programmaufruf können Argumente übergeben werden.

- **wait(status)** veranlaßt den Vater auf die Beendigung des Sohnes zu warten. Status liefert Returncode des Kindes.

- **exit(status)** beendet einen Prozeß und aktiviert einen evtl. wartenden Elternprozeß. (exit ist meißt letztes Statement im Sohn)

[18]da haben sich schon ganz andere beschwert, von wegen Mutter und Tochter; ist in der Fachliteratur nicht mehr zu ändern !!

13.6 Verschiedene SHELLs

Wie im Kapitel zuvor beschrieben, wird für jedes Kommando ein Sohn-Prozess gestartet. Der Vater-Prozess ist dabei die SHELL.

Neben Kommandos können jedoch auch sog. Sub-SHELLs gestartet werden. Eine Sub-SHELL ist Sohn der „normalen“ SHELL und übernimmt alle SHELL-Variablen (falls zuvor mit `export` bekanntgemacht) und den derzeitigen Directorystandort.

Sie legt die Vater-SHELL auf Eis, friert also seine Einstellungen ein, und führt den Dialog mit dem Benutzer wie nach einem Login. Der Aufruf erfolgt durch das Kommando:

Syntax:

sh

Sinn und Zweck dieser Sub-SHELLs ist es, eine temporäre Veränderung der SHELL-Umgebung rasch wieder rückgängig machen zu können. Sollen z.B. SHELL-Variablen für die Dauer eines Verarbeitungsschrittes verändert, oder das Working-Directory anders plaziert werden, kann eine Sub-SHELL die Aktuelle überlagern. Nach den Umgebungsveränderungen und evtl. Programmabläufen kann die Sub-SHELL wieder verlassen werden; die Überwachung der Rechnersitzung obliegt wieder der Login-SHELL mit der gewohnten und „unveränderten“ SHELL-Umgebung. Selbstverständlich läßt sich diese Vorgehensweise beliebig schachteln (also Sub-Sub-SHELLs, Enkel der Login-SHELL usw.).

Eine Sub-SHELL wird wie die login-SHELL mit `exit` oder `CTRL`+`D` beendent; die jeweilige Vater-SHELL ist wieder aktiv.

Strafverschärfend kommt hinzu, daß verschiedene Typen von SHELLs existieren. Durch das Kommando *set* können Sie sich anschauen, von welchem Typ Ihre Login-SHELL ist; der Superuser hat dies beim Anlegen Ihrer Benutzernummer für Sie festgelegt. I. d. R. existieren die folgenden Typen:

- **bsh** (Bourne-SHELL)
- **csh** (C-SHELL)
- **ksh** (korn-SHELL)
- ....

Sie unterscheiden sich (mal mehr, mal weniger) durch ihre Kommandoaufrufe oder deren Parameter. Die in diesem Script vorgestellten Kommandos wurden unter einer **Korn-SHELL** abgesetzt[19].

Durch den o.g. Befehl *sh* wird eine Sub-SHELL des System-Standard-SHELL-Typs erzeugt. Wünscht der Benutzer jedoch eine andere SHELL-Typ-Umgebung (z.B. statt einer Bourne-SHELL eine C-SHELL) muß das Kommando *sh* durch einen in der Liste stehenden Typ ersetzt werden (z.B. *csh*).

14 Hilfsmittel zur C-Programmierung

Wie eingangs erwähnt, wurde UNIX zu einem nicht unerheblichen Teil in C verfasst. Klar, daß es sich bei C deshalb um **die** favorisierte Programmiersprache in diesem Betriebssystem handelt.
Eine Einführung in C würde an dieser Stelle den Rahmen sprengen. Schauen wir uns lieber die Möglichkeiten an, die uns UNIX zur Unterstützung der C-Programmierung bietet:

- die sog. Makefiles (make)
- den C-beautifier (cb)

[19] Abweichungen sind auszutesten oder der Fachliteratur zu entnehmen

- den C-semantic-checker (lint)

Verdeutlichen wir uns zunächst welche Schritte notwendig sind, um aus einem C-Source-Code (dateiname.c) ein lauffähiges File zu schaffen.

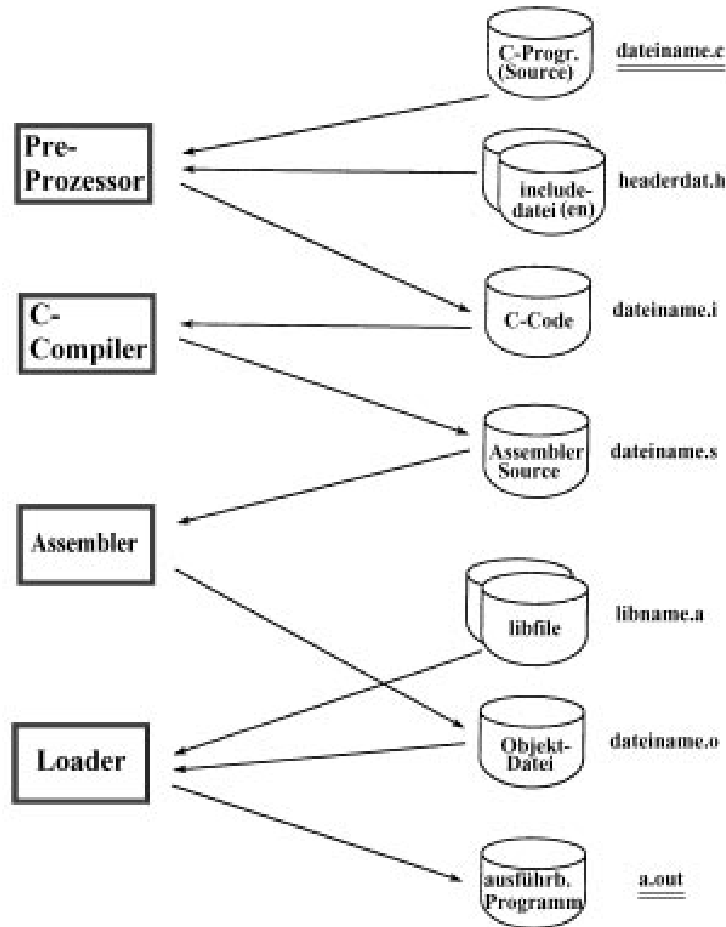

Abbildung 5: Übersetzung von C-Programmen

Der Source-Code wird vom Pre-Prozessor um die entsprechenden include, defines, etc... aufgebläht[20]. Danach erzeugt der Compiler ein Assembler-File, das wiederum in ein Objekt-File übersetzt wird. Letztendlich sorgt der Loader dafür, daß Objekt-File und evtl. benötigte Libraries (lib-Files) zu einem ausführbaren File zusammengebunden werden.

Das sieht zwar kompliziert aus (ist es auch), diese Schritte lassen sich jedoch durch den Einsatz von sog. Makefiles vereinfachen.

14.1 Compilieren

Nach der Erstellung eines C-Programmes (Source-Code) kann ein ausführbares File auf zwei Arten erzeugt werden:

[20] Alle Pre-Prozessor-Anweisungen werden ausgewertet; der C-Programmierer wird wissen um welche Statements es sich hier handelt.

Die einfachste Weg ist der Aufruf

```
$make otto
```

Die Datei *otto.c* wird bis zum ausführbaren File *otto* übersetzt; das Programm wird mit seinem Namen (*otto*) gestartet. Zur Übersetzung werden Standardvoreinstellungen benutzt.

Um bestimmte Wünsche bei der Übersetzung zu berücksichtigen, wird eine andere Form gewählt:

Syntax:

cc [optionen] dateiname [hinzuzuladende Dateien]

Optionen:

- **P** nur Pre-Prozessor
- **S** Übersetzung bis zur Assembler-Quelle
- **c** Übersetzung bis zum Objekt-Code
- **O** Objekt-Code-Optimizer aktivieren
- **o** *filename* erzeugt ein ausführbares File mit angegebenen Namen (sonst gilt *cc*-Standardeinstellung: *a.out*).

Alle unbekannten Optionen werden dem **loader** übergeben. Dies kann z.B. **-l***libname* sein, um eigene Libraries hinzuzuladen.

Beispiel:

```
$cc otto.c -o otto -lmeinelib
```

Das Programm *otto.c* wird bis zum ausführbaren File *otto* übersetzt. Dabei wird die Library *meinelib*, die selbsterstellte C-Funktionen enthalten kann, hinzugeladen.

14.2 Makefiles

Makefiles erleichtern insbesondere die modulare Erstellung von Programmpaketen. Einerseits werden in ihnen die Compilations-Befehle *cc dateiname....* etc. eingetragen (ähnlich den Shell-Scripten) um mehrfache Tipparbeit einzusparen. Andererseits können Makefiles die Übersetzungszeiten optimieren. Es handelt sich bei Ihnen um „intelligente“ Shell-Scripte; neu übersetzt werden nur die C-Sourcen, die nach der letzten Compilation verändert wurden.

Was kann in einem Makefile alles stehen? — Zunächst eine Ansammlung von *cc*-Anweisungen. Hinzu kommen Regeln, die die Abhängigkeitsbeziehungen meherer Dateien zueinander festlegen. Ein Beispiel soll helfen, die folgenden Zeilen müssen in einer Datei namens „Makefile“ stehen:

```
#    Makefile-Beispiel 1
#
#    >#< ist das Kommentarzeichen !!!
#
#    Anweisungen zur Erzeugung des ausfuehrbaren Files otto
otto:    otto.o
  ^      cc -o otto otto.o
  !             ^
  !             !----- cc-Anweisung: erstelle otto aus otto.o
  !
  !-------  Sprungmarke otto: otto ist abhaengig vom Objektcode otto.o

#    Anweisungen zur Erzeugung des Objektfiles otto.o
otto.o:  otto.c otto.h
  ^
  !
  !-------  Sprungmarke otto.o: otto.o ist abhaengig vom C-Sourcecode
                                otto.c und der Headerdatei otto.h .
            Zur Erzeugung von otto.o werden automatisch die
            Standardeinstellungen      cc -O -c dateiname.c
            ( -c : also nur bis zum Objektcode) benutzt.
```

Die Sprungmarken *dateiname:* werden angesteuert und ausgewertet. In diesem Beispiel wird die Sprungmarke *otto:* (als erste im Makefile) untersucht. Befehle, die der Sprungmarke folgen teilen sich in zwei Kategorien auf:

1. Abhängigkeiten: welche Dateien sind zur Ausführung der Befehle in dieser Sprungmarke von Bedeutung.

2. Befehl: z.B. Compilationsanweisungen

Ist innerhalb der Abhängigkeiten eine Datei aufgeführt, die seit der letzten Änderung von C-Source, Headerdatei oder Library aktualisiert werden muß, so wird vor der weiteren Verarbeitung eine dazu passende Sprungmarke abgearbeitet. Im Beispiel ist *otto* von *otto.o* abhängig. Also wird erst die Sprungmarke *otto.o:* ausgeführt. Anschließend wird in *otto:* fortgefahren.
Der Make-Aufruf erfolgt von der Shell-Ebene aus mit:
make otto oder einfach mit **make**

UNIX sucht nach einer Datei *Makefile* und meldet dann:

```
$make otto
     cc -O -c otto.c                <--- otto.o: Compilation mit
                                         Standardeinstellungen
     cc -o otto otto.o              <--- otto: wie angegeben
$
$make otto                          <--- nochmal aufgerufen
     Target otto is up to date.     <--- Meldung
```

Selbstverständlich können in einem Makefile auch die Compilations-Anweisungen für mehr als ein C-Programm stehen.
Die Notwendigkeit, den Übersetzungsvorgang in mehrere Schritte zu unterteilen, wird dann eigentlich erst deutlich; das ausführbare File kann aus meheren C-Sourcen (und dann Objektfiles) zusammengesetzt werden (s. nächstes Beispiel).

Makefiles können sich u.a. durch Shell-Variablen zu umfangreichen Gebilden mausern. Dies ist bei komplexen Programmprojekten, gerade wegen der vielen Abhängigkeiten verschiedener Sourcen zueinander, jedoch unumgänglich.

```
#    Makefile-Beispiel 2
#
#    Alle ausfuehrbaren Files sollen in das Unterdirectory /bin unter
#    dem USER-HOME Verzeichnis stehen:
BINDIR     = $(HOME)/bin

#    Alle Headerdateien stehen im Unterdirectory /include unter
#    dem USER-HOME Verzeichnis:
INC        = $(HOME)/include

#    Alle ausfuehrbaren Files die erstellt werden sollen:
BIN        = $(BINDIR)/otto   $(BINDIR)/finanz

#    Der Aufruf >make all< soll alle hier aufgefuehrten Programme
#    beruecksichtigen:
all:       $(BIN)

#    Sprungmarken koennen auch sonstige UNIX-Befehle beinhalten;
#    Aufruf: >make print< z.B.:
print:     pr -e4 -l72 otto.c otto.h funk1.c finanz.c finanz.h \
           | lpr

#    Anweisungen zur Erzeugung des ausfuehrbaren File otto
$(BINDIR)/otto:    otto.o \
                   funk1.o
                   cc -o $(BINDIR)/otto otto.o \
                              funk1.o

#    Anweisungen zur Erzeugung des ausfuehrbaren File finanz
$(BINDIR)/finanz:  finanz.o \
                   funk1.o
                   cc -o $(BINDIR)/finanz finanz.o \
                                          funk1.o

#    Objektfile otto.o ist abhaengig von C-Source otto.c und otto.h
otto.o:  otto.c $(INC)/otto.h

#    Objektfile funk1.o ist abhaengig von C-Source funk1.c
funk1.o: funk1.c

#    Objektfile finanz.o ist abhaengig von finanz.c und finanzen.h
finanz.o:  finanz.c $(INC)/finanzen.h
```

Beachten Sie, daß die Variablen in Makefiles mit $(NAME) angesprochen werden. Vielleicht haben Sie auch bemerkt: das \ ist das Zeilenfortsetzungszeichen ist.

Das Makefile läßt sich nun wie folgt aufrufen:

- `make all` : otto und finanz werden compiliert bis zum ausführbaren File.

- `make print` : alle C-Programme sowie Header-Dateien, die im Zusammenhang mit otto und finanz stehen werden (nach der Druckaufbereitung) gedruckt.
- `make otto` : otto wird bis zum ausführbaren File übersetzt.
- `make finanz` : finanz wird bis zum ausführbaren File übersetzt.
- `make` : wie `make all`, da `all:` die erste Sprungmarke ist.

UNIX meldet:

```
$make
    cc -O -c otto.c                        <--- otto.o: Compilation mit
                                                Standardeinstellungen
    cc -O -c funk1.c                       <--- funk1.o: Compilation mit
                                                Standardeinstellungen
    cc -o /u/mueller/bin/otto otto.o funk1.o   <--- otto: wie angegeben
    cc -O -c finanz.c                      <--- finanz.o: Compilation mit
                                                Standardeinstellungen
    cc -o /u/mueller/bin/finanz finanz.o funk1.o     <--- finanz: s.o.
```

Lassen Sie sich nicht durch den Aufwand beeindrucken, der hier getrieben wurde. Übung am Objekt verschafft Verständnis. Einmal erstellte Makefiles sparen Zeit und übersetzen in diesem Fall wirklich nur die Komponenten, die seit der letzten Compilation verändert wurden.

Ein letzter TIP: da UNIX bei Dateinamen zwischen Groß-/Kleinschreibung unterscheidet, sollten Sie in **einem** Directory immer nur **ein** Makefile haben. Verzichten Sie auf weitere, z.B. „makefile“ (klein geschrieben). Diese Form ist zwar zulässig, führt aber immer zu Verwirrungen, da nicht klar ist welcher (Makefile o. makefile) dann ausgeführt wird.

14.3 C-beautifier

Für Leute mit der schlechten Angewohnheit C-Sourcen unübersichtlich einzutippen stellt UNIX den C-beautifier zur Verfügung.

cb c-dateiname1 > c-dateiname2

Der Inhalt der c-datei1 wird optisch aufbereitet und in die c-datei2 umgelenkt.
Als Beispiel soll die Datei hallo.c herhalten:

```
#include <stdio.h>
main()
{
  int i;

  fprintf(stdout,"hier ist der anfang\n");for(i=0;i<10;i++){
  fprintf(stdout,"Zeile %d\n",i);}fprintf(stdout,"hier ist
das ende\n");
}
```

Der Aufruf

```
  $cb hallo.c > halloschoen.c
  $
```

führt zu folgendem Inhalt der Datei halloschoen.c:

```
#include <stdio.h>
main()
{
  int i;

  fprintf(stdout,"hier ist der anfang\n");
  for(i=0;i<10;i++){
      fprintf(stdout,"Zeile %d\n",i);
  }
  fprintf(stdout,"hier ist das ende\n");
}
```

Anmerkung: Wer so programmiert wie in der Datei hallo.c, hat ein solches Hilfsmittel eigentlich nicht verdient !!

14.4 C-semantic-checker

Der LINT überprüft C-Programme auf syntaktische und semantische Korrektheit. Betroffen sind:

- Typen
- Erreichbarkeit der Anweisungen
- Schleifen (Einsprungstellen)
- vereinbarte, aber nicht verwendete Variablen
- logische Ausdrücke (Konstante)
- Funktionsargumente

Der LINT schafft daher das, was der C-Compiler verschlampt: Detailfragen untersuchen, die u.a. für die Portierung auf andere Systeme unbedingt notwendig sind.

Aufruf:

lint c-dateiname

Die zusätzliche Angabe von Optionen entnehme man den Manualpages.

15 Nachwort

Wie eingangs erwähnt, ergeben sich durch die verschiedenen SHELL-Typen eine Reihe von Abweichungen und zusätzlichen Optionen bezüglich der Komandosyntax. Jeder wird die ihm vertraute SHELL-Umgebung als die Angenehmste empfinden. Generelle Aussagen können deshalb an dieser Stelle nicht getroffen werden.

Mein Dank gilt

- Einem gewissen Softwarehaus in Dortmund, bei dem ich UNIX zu lieben gelernt habe. Den Namen schreibe ich hier nicht hin, da Reklame bestimmt wieder verboten ist.
- Doris und einem Teil der Abt. Wiss. Anw. für's Fehlerlesen und Testen.
- ThF für ein bisschen Optik.

Endlich fertig ...

A INDEX

Index

Befehle, 5, 11, 21
Benutzerkennung, 4
Betriebssystem, 2
Bildschirmausgabe, 27

C, 2, 43, 48
C-beautifier, 48
C-Syntax, 49
CASE, 39
Compilieren, 44

Dateien, 6, 11, 14, 17, 24, 31
Dateinamen, 6, 18
Dateizugriffsrechte, 8, 15, 17
Directories, 6, 11–14
Druck, 25, 26

Editor, 31
EXEC, 42
EXIT, 42

Filter, 19, 20
FOR, 40
FORK(), 42

Gerätedateien, 8

Hilfe, 9, 11
Home-Directory, 4

IF, 37
Im-/Export, 29
InfoExplorer, 9

Kommandosyntax, 5

Laufzeit, 24
Lint, 49
Login, 4

Makefile, 44, 45
Manualpages, 9, 11

Optionen, 5

Parameter, 5
Passwort, 4, 27
Pfade, 7, 13
Pipes, 20
Prozeßid, 28, 42
Prozesse, 28, 34, 42

root, 6

Shell, 4, 34, 36
Shell-Programmierung, 37
Shell-Scripte, 36
Shell-Substitution, 41
Shell-Typen, 43
Shell-Variablen, 36
Standardausgabe, 19
Standardeingabe, 19
Sub-Shell, 43
Superuser, 4, 15

TEST, 38
Textmuster, 25, 33

UNTIL, 41

Voraussetzungen, 4

WAIT, 42
WHILE, 40
Wildcards, 18, 33