Berufsoberschule München
Fachrichtung Wirtschaft

# Begleitschrift zum Unix-Systemverwalter Kurs

(Yet Another Useless Unix Book)

| | | |
|---|---|---|
| Autor: | Stefan Zehl | Oliver Paulzen |
| | sec@42.Org | opp@MuFFiN.Org |
| | http://sec.42.org/ | http://www.muffin.org/~opp/ |

# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

## 1.1 Was soll das ganze Papier?

Dies ist nun der Unix-Systemverwalter-Kurs, besser gesagt, die schriftlichen Aufzeichnungen dazu. Das Hauptaugenmerk liegt unsererseits auf den regelmässig stattfindenden, nachmittäglichen Kursen an der BOS-Wirtschaft, da nur dort genauer auf Fragen eingegangen werden kann und nur dort das kontrollierte Arbeiten am Rechner möglich ist. Und genau das ist es was in unseren Augen einzig zum Erfolg führen kann: Das Arbeiten am Rechner. Dies kann sich natürlich nicht nur auf die kurze Zeit im Kurs vor Ort beschränken, vielmehr sollte man versuchen so viel Zeit wie möglich am heimischen oder schulischen (Unix-) Rechner zu verbringen um Neugelerntes aber auch und vor allem eigene Ideen auszuprobieren. Oben Genanntes soll natürlich keine Aufforderung sein eine eventuelle Familie und/oder Arbeit zu vernachlässigen, das "isses nich'" wert, vielmehr eine Anregung zum Weiterdenken und -handeln. Und genau hierbei sollten diese Unterlagen unterstützend helfen.

## 1.2 Themenauswahl

Der Bereich, der hier so salopp *Unix-Systemverwaltung* genannt wird, umfaßt eine Vielzahl von Tätigkeiten, ein großes Wissen im Bereich der verschiedensten Unix-Varianten[1] und Netzwerktechniken sowie -diensten. Wir werden versuchen einen allgemeinen Überblick über die Herkunft von Unix, die Unterschiede zu gängigen PC-Betriebssytemen und die Möglichkeiten, die uns dieses System bietet, zu geben und weiterhin auf die Einrichtung und Pflege

[1] und davon gibt es unwahrscheinlich viele wie wir sehen werden.

einer 'schultypischen' Installation eingehen - letzteres am Beispiel der an der BOS-Wirtschaft und -Technik verwendeten Hard- und Software.
Wie so vieles im Leben hat auch Unix in fast jedem Bereich verschiedene Lösungsansätze für einzelne Probleme. So gibt es allein z.B. mind. sechs verschiedene *Shells*[2] und eine Vielzahl von Unix-Herstellern und somit -Varianten. In der Regel beziehen wir uns auf die an der BOS verwendeten Utilities, versuchen aber auch, soweit nötig und sinnvoll, die "Konkurenten" dazu anzusprechen. Prinzipiell behandeln wir die *BSD*-Form von Unix (im Gegensatz zu SystemV), da wir diese an der BOS und andernorts erfolgreich einsetzen und mit dem Betriebsystem *FreeBSD* ein umfangreiches, kostengünstiges und leistungsfähiges Unix-Derivat für Intel-basierte Rechner zur Verfügung steht. Wer noch ein paar MegaBytes (ab 200, ohne Grenze nach oben) auf seiner Festplatte frei hat und auch sonst einen zeitgemäßen Rechner besitzt, sollte sich deshalb o.g. Betriebssystem installieren, um auch daheim, in Ruhe, eventuell in Zusammenhang mit einer schönen Tasse Kaffee oder Tee, die Vorzüge und vielleicht auch die Faszination dieses Systems kennen und schätzen zu lernen.

[2]Kommandointerpreter, vergleichbar mit command.com bei DOS, siehe 3.6

# Kapitel 2

# Geschichtliches

Im Jahre 1969 entwickelte Ken Thompson an den Bell Laboratories die erste Unix-Version auf einer PDP-7, noch vollkommen in Assembler geschrieben. Zu dieser Zeit schrieb Dennis Ritchie die Programmiersprache C, die auf Thompsons' B basierte. Bis 1971 wurde Unix dann nahezu komplett neu in C programmiert und auf die PDP-11 portiert, da diese Sprache Unix eine hohe Portablitiltät garantierte. Somit gelten beide Entwickler als die Väter dieses Betriebsystems. Die grundlegenden Ideen dazu holten sie sich aus dem, zusammen mit weiteren Entwicklern an den Bell Laboratories konzipierten Multics, sowie CTSS vom MIT und GENIE von Berkeley. Diese Ideen waren, im Gegensatz zu denen der zu dieser Zeit bekannten Systemen, die Verfügbarkeit im Quellcode, die Implementierung in einer Hochsprache und die Möglichkeit für jeden Benutzer mehrere Prozesse zur gleichen Zeit laufen zu lassen. Die erste offizielle Version, die auch an Interessierte weitergegeben wurde, war Version 6. Version 7 wurde möglichst portabel gehalten und lief auch auf Interdate 8/32 und der damals aufkommenden VAX-Architektur.
Um diesen Zeitpunkt entwickelten verschiedene Gruppen an den vorhandenen Quellen weiter:

- Bell Unix Time-Sharing System 8th bis 10th Edition (1989), danach Plan 9 (von Unix eher unabhängig)
- AT&T Unix Support Group (USG, System III), danach System V (1983) welches AT&T nach einem Gerichtsbeschluß auch kommerziell vermarkten durfte. System V Release 2 wurde dann 1984 von dem USDL (Unix System Development Laboratory) entwickelt und weiterhin von AT&T vertrieben. System V.3 (1987) sowie V.4 (1989) kamen von ATTIS (AT&T Information Systems) und wurden durch die AT&T-Tochter USL (Unix System Laboratories) übernommen, welche 1993 an Novel verkauft wurde. Der Markenname "Unix" wurde an

das X/Open Konsortium übergeben, welche damit das alleinige Recht besaß, Standards festzulegen, die von Unix-Systemen erfüllt werden müssen, um den Namen "Unix" tragen zu dürfen. 1995 wurde dieses Recht an SCO (Santa Cruz Operating Systems) verkauft, welche das PC-Unix Xenix zusammen mit Microsoft entwickelten und es nach deren Ausstieg als SCO-Unix für Intel-basierte Rechner verkauften.

- Der University of California in Berkeley entsprang Ende der 70er Jahre die Berkeley Software Distributions, welche als BSD-Unix bekannt sind. BSD war lange Zeit das einzig lauffähige Betriebssystem auf VAX-Rechnern und erfreute sich bis 1983 auch auf anderen Plattformen großer Beliebtheit. Ab der Version 4.2BSD, die zeitgleich mit dem Vermarkten von System V zur freien Verfügbar stand, kam es zu Lizenzstreitigkeiten zwischen Berkeley und den System V Besitzern, da beide Systeme auf den Version 7 Quellen beruhten, aber nur letzteren das Recht zur Weitergabe und -vermarktung zugesprochen wurde. Die Entwicklung von 4BSD wurde durch die Defence Advanced Research Projects Agency (DARPA) gefördert, um Unix um Netzwerkunterstützung zu erweitern, die im ARPA-Net (Grundlage des Internets) verwendet wurde und auch heute noch Grundlage für jedes Unix-Netzwerk bildet. Da die BSD-Quellen nur an Besitzer von "Unix Source Licence" weitergegeben werden durfte, entschied man sich bei Berkeley alle lizenzbehafteten Quellen von grundauf neu zu schreiben. Bei 4.3BSD gab es die lizenzfreien NET-Releases, bestehend aus der aktuellen 4.3BSD-Release ohne die jeweiligen lizenzbehafteten Teile. Erstmals 4.4BSD-Lite (1994) stellt eine nahezu komplette und vollkommen lizenzfreie Unix-Version dar, welche als Quellcode und Binary an jedermann weitergegeben werden durfte. Auf 4.4BSD-Lite basieren die momentan verfügbaren, freien Betriebssysteme FreeBSD, NetBSD und OpenBSD sowie das kommerzielle BSDI. Teile aus der neuen Lite2-Version gelangen noch heute in diese BSD-Varianten.

Technische Entwicklungen flossen quer durch die verschiedenen Zweige, wodurch jedes Unix-Derivat von Gedanken der anderen profitierte. Workstation-Hersteller und Software-Häuser verkauften ihre jeweilige Version von Unix, die auf den (lizensierten) Quellen der o.g. Zweige beruhten unter eigenem Namen (HP/UX, Unixware, AIX, A/UX, IRIX, SunOS, Solaris, Domain/OS, Interactive, BSDI, OSF/1, Ultrix, DEC Unix, ...). Bekannte, nicht-Unix-basierende Systeme (OS/2, Windows 95, Windows/NT), wurden als Unix-killer gehandelt, aber keins davon hat es bisher geschafft Unix aus den Marksegmenten zu verdrängen, für die es von Anfang an gedacht war und für die es heute noch keine wirklich brauchbaren Alternativen gibt.

# Kapitel 3

# Grundlagen

## 3.1 Multitasking/Multiuser

Eine grundlegende Eigenschaft von Unix ist dessen Multitaskingfähigkeit. Unix lebt von der Möglichkeit, mehrere Prozesse quasi-gleichzeitig (im Zeitscheibenverfahren) ablaufen zu lassen. Dies geschieht nach dem Prinzip des *präemptiven Multitaskings* bei dem einzig das Betriebssystem für die Vergabe von Rechenzeit an einzelne Programme zuständig ist und nicht wie beim *cooperativen Multitasking* die Programme selber entscheiden ob und wieviel an Prozessorkapazität sie beanspruchen, bzw. an andere Programme abgeben (z.B. Windows 3.x). Praktisch jede Aufgabe, jeder Dienst und was sonst noch alles zu einem Betriebsystem gehört, wird jeweils durch einen eigenen Prozeß gelöst. Jeder Prozeß bekommt vom Unix-Kernel[1] einen Adressbereich[2] zugewiesen, auf den nur er Zugriff hat. Benötigt der Prozeß mehr Speicherplatz, so muß er ihn beim Kernel "beantragen" (*malloc(3)*). Zur Vergabe und Verwaltung der einzelnen Prozeß-Adressbereiche ist einzig der Kernel zuständig. Dies hat den Vorteil, daß kein Prozeß (z.B. ein Anwenderprogramm), ohne den Weg über den kontrollierenden Kernel zu nehmen, Zugriff auf den Speicherbereich eines anderen Prozesses hat. Gelöst wird dies durch die Verwendung verschiedener Speichersegmente: Das Program selber, das *Executable*, wird im Codesegment abgelegt, welches für den Prozeß selber nur lesbar, nicht aber schreibbar ist. Dadurch wird vermieden, daß sich Programme während ihrer Laufzeit modifizieren können, welches in Bezug auf Systemsicherheit sehr wichtig ist. Im Datasegment des

[1] Der Kernel ist jenes Programm, welches beim Booten von Unix geladen wird. Seine Aufgabe ist es, alle Prozesse, den Speicher, angeschlossene Geräte und Ein/Ausgabe-Operationen, gleich welcher Art, zu verwalten.

[2] Der gesamte, zur Verfügung stehende Adressraum berechnet sich normalerweise aus physikalischem RAM + Swap-Bereich auf der Festplatte (oder im Netz).

Prozesses werden alle anfallenden Daten des Programms verwaltet und nur der Prozeß selber hat darauf Lese- und Schreibrechte. Beantragt der Prozeß allerdings beim Kernel einen sog. *Shared-Memory* Bereich, so kann aus diesem auch von anderen Prozessen gelesen werden. Im Stacksegment eines Prozesses werden Daten wie Aufrufparameter, Rückgabewerte, Register und ähnliches zwischengelagert. Auch dieser Speicherbereich ist nur Les- und Schreibbar für den Prozeß selber, andere besitzen keine Rechte darauf. Programme, die durch Speicheroperationen andere Programme oder gar das komplette Betriebsystem (den Kernel) zum Absturz bringen, können durch dieses Speicherschutz-Konzept nicht vorkommen. Dies gewährt natürlich ein hohes Maß an Ausfallsicherheit und begründet die teilweise immens hohen Laufzeiten (Uptimes) von Unix-Systemen; 500 Tage oder mehr sind keine Seltenheit.

Dadurch, daß immer mehrere Prozesse "laufen", also einen Bereich im Speicher belegen und vom Kernel verwaltet werden, müssen gleichzeitige Zugriffe auf ein und dasselbe Gerät (Bildschrim, Schnittstellen, Netzwerkkarte, Festplatte, Tastatur, Maus, usw.) vermieden werden. Auch das ist Aufgabe des Kernels. Indem kein anderer Prozeß außer ihm Zugriff auf die Hardware-Resourcen hat und jeder Hardware-Zugriff eines Prozesses, z.B. das Schreiben auf die Festplatte oder das Lesen einer CD, letzlich vom Kernel getätigt wird, kann dieser auch konkurrierende Zugriffe vermeiden. Zu diesem Zweck ist jedes Gerät, selbst die Grafikkarte eines Rechners oder der "Speicher", als Datei im Verzeichnis /dev (zum Thema Filesystem siehe 3.5) abgebildet. Benötigt ein Prozeß nun Zugriff auf ein Stück der Hardware, so liest oder schreibt er einfach in die stellvertretende Datei (z.B. /dev/lpt1 für eine parallele Schnittstelle oder /dev/cd0a für ein CD-ROM[3]). Die eigentliche Ein/Ausgabe-Operation wird dann vom Kernel durchgeführt. Dies bedingt natürlich, daß der Kernel über die Funktionsweise der Gerätschaften informiert ist. Das was unter DOS oder Windows als "Treiber" bekannt ist, findet man unter Unix direkt im Kernel, da ja nur hier alle hardwarenahen Operationen durchgeführt werden dürfen. Ein Nachteil dieser Funktionsweise ist, daß bei jedem neuangeschafften Gerät (z.B. neue Ethernetkarte) oder neugewünschten Funktion die der Kernel behandeln soll, dieser neu übersetzt werden muß. Da dies mitunter ziemlich zeitaufwendig sein kann, ist man dazu übergegangen einige Treiber-Funktionen in sogenannte *loadable kernel modules* auszulagern, welche dann bei Bedarf zur Laufzeit des Systems zum Kernel "hinzugelinkt" werden können. Um bei Änderungen von Hardware-Informationen, wie IRQ oder Base-Adress, einer PC-Steckkarte, nicht jedesmal einen neuen Kernel compilieren zu

[3] Die Dateinamen für Device-Files unterscheiden sich je nach Unix-Derivat.

müssen, versuchen einige PC-Unixe diese zu "proben" (auf Verdacht alle Adressen durchgehen und schauen ob sich eine Karte meldet), z.B. Linux, oder einen Hardware-Konfigurations-Modus zwischen Laden und Ausführen des Kernels zu starten, wie z.B. FreeBSD, in dem man die gewünschten Einstellungen dann von Hand durchführen kann.

Mit der o.g. Fähigkeit des Multitaskings geht die Multiuserfähigkeit einher. Jeder Benutzer eines Unix-Systems muß diesem bekannt sein. Der Account (Zugangsberechtigung) wird durch den Loginnamen repräsentiert und besitzt ein Passwort. Durch die Möglichkeit von Unix, mehrere Prozesse gleichzeitig laufen zu lassen und die Unterscheidung zwischen verschiedenen Benutzern ist es möglich, daß mehrer Benutzer auch gleichzeitig mit dem System arbeiten und ihrerseits mehrere Prozesse laufen lassen können. Die Ein- und Ausgabe zu den Benutzer-Prozessen (z.B. dessen Shell, s. 3.6) kann z.B. auf der Console[4] (angeschlossener Monitor und Tastatur), seriellen Terminals (ASCII, vt100) oder Netzwerkverbindungen (Telnet, Rlogin) geschehen, aber auch aus einer Datei gelesen, bzw. in ein geschrieben werden, oder ganz unterdrückt werden. Das Betriebssystem muß gewährleisten, daß die Daten des Benutzers (seine Dateien, seine laufenden Prozesse) vor unberechtigten Zugriffen anderer Benutzer geschützt sind, bzw. nur soweit veröffentlicht werden, wie es der Benutzer möchte.

Zusammenfassend kann man daher drei wichtige Unterschiede eines Mehrbenutzerbetriebsystems gegenüber einem Einzelplatzsystems benennen:

- Die Anwenderprogramme können sich gegenseitig nicht stören
- Es müssen konkurierende Hardwarezugriffe verhindert beziehungsweise verwaltet werden.
- Es müssen die privaten Daten der Benutzer geschützt werden.

## 3.2 Benutzerverwaltung

Da jeder Benutzer, im Folgenden auch User genannt, dem System bekannt sein muß, werden in einer Datei mit dem Namen *passwd(5)* im Verzeichnis */etc* alle benötigten Benutzer-Daten abgelegt. Am Beispiel eines Eintrages in dieser Datei möchte ich diese Daten erläutern. */etc/passwd* ist Zeilenweise

[4] FreeBSD verwaltet mehrere virtuelle Consolen. Mit der Tastenkombination Alt + F1-F9 schaltet man zwischen den verschiedenen Consolen hin und her.

aufgebaut, d.h. für jeden Benutzer gibt es genau eine Zeile. Die einzelnen Datenfelder sind durch Doppelpunkte getrennt:

```
orpaulze:Hg98bga1Kq7cgE:2004:2000:Oliver Paulzen:/home/gast/orpaulze:/bin/tcsh
```

- Das erste Feld gibt den Usernamen an, in diesem Fall "orpaulze"[5] , welcher i.d.R. nicht länger als acht Zeichen sein darf.

- Das zweite Feld enthält eine verschlüsselte[6] Form des Passworts für diesen Benutzer, welches er mit dem Befehl *passwd(1)* jeder Zeit ändern kann. In letzter Zeit trifft man aber auch immer mehr *Shadow-Password* Systeme an, bei denen der Passwort-String in einer eigenen, nur für den Systemverwalter lesbaren Datei (z.B. */etc/master.passwd*) gespeichert wird und in der */etc/passwd* nur ein Platzhalter im zweiten Feld steht. Dadurch wird es Benutzern des Systems unmöglich gemacht den verschlüsselten Passwort-Eintrag zu lesen, um mit diesem dann z.B. einen Crack-Versuch[7] zu unternehmen.

- Im dritten Feld wird die UserID verwaltet. Diese ist fest mit dem Benutzernamen verknüpft und dient der eindeutigen Zuordnung[8] von Dateien und Prozessen zum Benutzer auf diesem System oder im Netzwerk (NFS, s. 3.5).

- Das vierte Feld stellt die GroupID dar. Dies ist die Gruppe, die der User nach dem Einloggen angehört.

- Im fünften, dem sogenannten GECOS-Feld, können prinzipiell beliebige Daten zum Benutzer gespeichert werden, allerdings gilt es als Standard dort den vollen Vor- und Zunamen und bei Interesse noch Adresse und Telefonnummer abzulegen. Diese Informationen werden durch Komma getrennt und z.B. von Mailprogrammen ausgewertet. Der Benutzer kann diese Daten mit dem Befehl *chfn(1)* modifizieren.

[5] An der BOS-Wirtschaft ergeben sich die Usernamen (so weit möglich) aus dem ersten und dem letzten Buchstaben des Vornamens plus den ersten sechs Zeichen des Nachnamens.

[6] Meist nach dem DES-Algorithmus, eine Art "Falltür"-Algorithmus. Alternativen sind z.B. MD5 oder Blowfish.

[7] Mit Hilfe von Crack-Programmen, Wörterbuch-Dateien und genügend Rechenleistung lassen sich einfach gewählte Passwörter entschlüsseln.

[8] Zur Verwaltung von Zugehörigkeiten einer Datei oder eines Prozesses wird einzig die UID bzw. GID verwendet. Ein Auflösung zum Namen geschieht über die *passwd*-Datei. z.B. wird bei NFS nur die UID/GID übertragen.

- Das sechste Feld gibt das Home-Verzeichnis des Benutzers an. In diesem Verzeichnis befindet sich der User direkt nach dem Einloggen (Bekanntmachen mit dem System) und besitzt normalerweise Schreibrechte dafür. Im Home-Verzeichnis eines Benutzers werden z.B. die meisten Konfigurations-Dateien für die von ihm verwendeten Programme abgelegt und nur hier hat er i.d.R. uneingeschränkte Schreibrechte und kann sich seine eigene, private Verzeichnisstruktur zur Ablage eigener Dateien schaffen.
- Das letzte Feld bezeichnet das Programm, welches direkt nach dem Einloggen des Users gestartet wird, im Normalfall ist dies eine Shell, dann Loginshell genannt. Ist dieses Feld leer, so geht das System von dem Programm */bin/sh* aus. Geändert werden kann dieses Feld durch den Befehl *chsh(1)*, dafür muß das Programm aber vorher in der Datei */etc/shells* (*shells(4)*) vom Systemverwalter als mögliche Loginshell berechtigt werden.

In neueren BSD-Arten werden noch drei weitere Felder (ein meist noch unbenutztes namens "class", eins zur Angabe wie lang das Passwort gültig sein wird und wann es geändert werden muß und eins zur Angabe wann der Account ungültig wird) verwendet. Diese befinden sich zwischen dem GID- und dem GECOS-Feld und sind vom Benutzer nicht einsehbar (in master.passwd).

Der erste Eintrag in der *passwd*-Datei bezeichnet den sogenannten Superuser, Accountname "root". Seine UID und GID ist 0 und er besitzt uneingeschränkte Lese- und Schreibrechte auf jede lokale Datei und das Recht Prozesse anderer Benutzer zu manipulieren (z.B. beenden; s. 3.6). Sein Passwort ist sozusagen der "Schlüssel" des Unix-Rechners, mit ihm kann alles am System modifiziert werden.

Unix kann mehrere Benutzer zu Benutzergruppen zusammenfassen und Zugriffsrechte für Mitglieder einer solchen Gruppe vergeben. Die Gruppen werden in der Datei */etc/group* (*group(5)*) verwaltet. Der Aufbau ist ähnlich der der *passwd*, z.B.:

```
infosys:*:30:meguro,sabartes
```

Das erste Feld bezeichnet den Gruppennamen, fest verknüpft wieder mit dem dritten Feld, dem der GroupID. Dazwischen kann ein (verschlüsseltes) Passwort stehen, wird aber i.d.R. nicht verwendet und deshalb ein "*" als "kein mögliches Passwort" verwendet. Im letzten Feld werden, durch Komma getrennt, alle Mitglieder der Gruppe aufgezählt. Im Gegensatz zu System V Varianten von Unix, bei denen Benutzer immer nur einer Gruppe angehören und diese bei Bedarf und Berechtigung wechseln können, gehört ein BSD-User zu jedem Zeitpunkt jeder Gruppe an in die er eingetragen ist, mindestens aber seiner Login-Group (s. GID in der *passwd*). Mit dem Befehl *id(1)* kann man überprüfen in welchen Gruppen man Mitglied ist:

```
orpaulze@suninfo1:~> id
uid=2004(orpaulze) gid=2000(admin) groups=2000(admin),0(wheel),30(infosys)
```

In diesem Fall ist die UID 2004, aufgelöst als "orpaulze", die Login-Group ist 2000, aufgelöst als "admin". Desweiteren gehört der User noch der Gruppe 0 ("wheel"[9]) und der Infosys-Gruppe (GID=30) an.

In Netzwerken, in denen sich jeder Benutzer auf jedem Rechner einloggen können soll, empfiehlt es sich die *passwd*-Datei netzweit zu verteilen. Dies kann z.B. durch regelmässiges kopieren (z.B. mit dem *rdist(1)* Kommando) der Datei auf alle Rechner geschehen oder, wie im Fall des Rechnerpools an der BOS-Wirtschaft, durch den Einsatz eines Network-Information-Systems (NIS). Bei diesem wird die *passwd*- und ebenso die *group*-Datei (oder auch weitere) von einem zentralen NIS-Server verwaltet. Auf ihm läuft ein Prozeß (*ypserv(8)*), welcher auf Anfragen von Prozessen auf den NIS-Clients (*ypbind(8)*) hin, die gewünschten Dateien zur Verfügung stellt. Auf den Clients wird am Ende der *passwd*- bzw. group-Datei ein Eintrag mit "+" beginnend angefügt, welcher darauf hinweist, daß hier die Datei vom NIS-Server "angedacht" wird. Dem NIS-Client Prozeß *ypbind* muß dann nur noch mitgeteilt werden, welcher Rechner im Netz für ihn als Server zuständig ist und welcher NIS-Domain[10] er angehört.

## 3.3 Dateirechte

Um festzulegen wer eine Datei lesen, schreiben oder ausführen darf, besitzen auch Dateien User- und Gruppenzugehörigkeiten. Wird eine angelegt, so geschieht dies unter der UserID und (normalerweise) Login-Group des Benutzers der dies tut. Darüber hinaus werden zu jeder Datei die Dateirechte mit im Filesystem abgelegt, sowie eine Art Typ-Bezeichnung. Die kompletten Zugriffsinformationen zu einer Datei erhält man mit dem Befehl *ls -l*[11] (*ls(1)* ist vergleichbar mit dem `DIR`-Befehl von DOS). Eine typische Ausgabe kann z.B. so aussehen:

```
orpaulze@pcinfo7:~> ls -l einkaufszettel
-rw-r--r--  1 orpaulze admin         665 Mar 11 22:58 einkaufszettel
```

Der erste Bereich (-rw-r--r--) gibt die Zugriffsrechte an, das zweite Feld die Anzahl der Hardlinks welche diese Datei hat (zum Thema Links mehr in

[9] Nur Mitgliedern dieser Gruppe ist es gestattet mit dem Befehl *su(1)* Root-Rechte zu erhalten.

[10] Zur Verwaltung verschiedener Netzbereiche in denen verschiedene Dateien gleichen Typs verteilt werden, trennt NIS zwischen den NIS-Domains.

[11] -l für "long". Bei einigen Systemen benötigt es noch ein "-g" zum Anzeigen der GroupID.

3.5), gefolgt von der User- und Gruppenzugehörigkeit, der Größe der Datei in Bytes, dem Datum und der Uhrzeit des letzten Schreibzugriffes und dem Dateinamen. Das Feld der Zugriffsrechte setzt sich wie folgt zusammen:

- Das erste Zeichen (hier "-") gibt den Typ dieses Files an:
  - Eine normale Datei, also ein Programm oder Daten in jedlicher Form, werden eben mit jenem "-" dargestellt.
  - Verzeichnisse sind im Sinne des Unix-Dateisystems auch nur spezielle Dateien und werden in diesem Feld mit "d" gekennzeichnet.
  - Dateien im /dev-Verzeichnis, also die Hardware-Repräsentanten, die sogenannten Devices-Files werden mit einem "b" für blockorientierte oder einem "c" für zeichenorientierte Geräte wiedergegeben.
  - Ein symbolischer Link wird hier mit einem "l" gekennzeichnet.
- Die darauffolgenden 9 Zeichen geben die eigentlichen Zugriffsrechte wieder und zwar in drei 3er Gruppen geteilt:
  - Die ersten 3 Zeichen bezeichnen die Zugriffsliste für den Inhaber (User, UID) der Datei
  - Die zweiten 3 Zeichen die Rechte für Mitglieder in der Gruppe die dieser Datei zugeordnet wurde.
  - Die letzten 3 Zeichen bezeichnen die Rechte für alle anderen Benutzer (Others), die nicht den ersten beiden zuzuordnen sind.

  Innerhalb eines 3er-Blocks gilt ein

  - "r" als das Recht zum Lesen der Datei oder im Falle eines Verzeichnisses, um die Dateien in einem Verzeichnis anzeigen zu können.
  - "w" als das Recht in diese Datei oder das Verzeichnis zu schreiben oder zu löschen.
  - "x" als Ausführungsrecht. Dieses wird z.B. von allen Programmen zum ausführen benötigt, aber auch für Verzeichnisse um diese mit *cd(1)* zu betreten.

Bei der Datei im oberen Beispiel handelt es sich also um ein "normales" File mit der Größe 665 Bytes. Besitzer dieser Datei ist "orpaulze" und ihre GID ist "admin". Der Besitzer darf diese Datei beschreiben, Mitglieder der Admin-Gruppe, aber auch alle anderen Benutzer dieses Systems, daraus lesen.

Im User- und Group-Feld für das Ausführungsrecht ("x") kann als Sonderfall auch noch ein "s" oder "S", das sogenannte Set-User/Group-ID-Flag gesetzt sein. Ist ein "s" im User-Feld einer normalen Datei gesetzt, so bedeutet dies, daß im Fall einer Ausführung der Datei als Programm, dieses unter der UID des Datei-Besitzers abläuft. Folgender Fall ist denkbar:

```
orpaulze@pcinfo7:~> ls -l /usr/bin/passwd
-r-sr-xr-x  2 root  bin  20480 Nov 16  1995 /usr/bin/passwd
```

Dies bedeutet, daß jeder Benutzer des Systems dieses Programm lesen[12] und starten kann, es aber sobald es läuft die effektive UID (EUID) "root", also "0" annimt[13] und nicht wie normalerweise unter den Rechten des Aufrufers läuft.
Ist das s-Flag im Group-Feld eines Verzeichnisses gesetzt, so werden Dateien unterhalb dieses Verzeichnisses unter der GroupID dieses Verzeichnisses abgelegt, falls man Mitglied dieser Gruppe ist. Ein populäres Beispiel wäre:

```
orpaulze@suninfo1:/usr/local/infosystems/www/home> ls -lg
total 2
drwxrwsr-x  3 infosys  infosys       512 Oct 12 20:28 bos
drwxrwsr-x 17 infosys  infosys       512 Mar 10 15:35 wirtschaft
```

Falls man Mitglied in der Gruppe "infosys" ist und Dateien in einem der beiden aufgeführten Verzeichnisse erzeugt (darf man ja dank dem "w" für die Gruppe), so werden diese unter der GID "infosys" abgelegt, was das Bearbeiten von Dateisystem-Bereichen (hier z.B. die WWW-Dateien) durch mehrere Benutzer u.U. vereinfachen kann.
Ein "S" zählt ähnlich einem "s", nur daß keine Ausführungsrechte für User, bzw. Group bestehen.

Um die User- und Gruppenzugehörigkeit einer Datei nachträglich zu bestimmen, kann man diese mit dem Befehl *chown(8)* ändern.
"*chown meguro:infosys test.gif*"[14] ändert (entsprechende Rechte vorrausgesetzt) die Zugehörigkeit der Datei "test.gif" zum User "meguro" und der Gruppe "infosys". Um nur die Gruppenzugehörigkeit zu ändern, genügt der Befehl *chgrp(1) <Gruppe> <Datei>*. Die Zugriffsrechte lassen sich nachträglich mit dem Befehl *chmod(1) <Mode> <Datei>* modifizieren. Als Mode kann man die Zugriffsrechte im Zahlencode angeben. Diese sind in einer vierstelligen Zahl codiert:

- Eine "2" in der ersten Ziffer setzt das SUID-Flag, eine "4" das SGID-Flag.

[12] Leserechte benötigt man u.a. zum Kopieren.
[13] Nur so ist das Ändern von */etc/passwd*, die nur für Root schreibbar ist, möglich.
[14] Oft kann man statt des ":" auch ein "." verwenden.

- Die weiteren drei Ziffern sind analog der ls-Ausgabe für User-, Group- und Others zu setzen, wobei

  - eine "1" für ausführbar
  - eine "2" für schreibbar
  - eine "4" für lesbar

  steht.

Möchte man z.B. eine Datei für sich selbst und Mitglieder der gleichen Gruppe ausführ- und lesbar machen, für sich selbst schreibbar lassen und allen anderen nur Leserechte auf diese Datei geben, so setzt man *chmod 0754 <Datei>* was in der ls-Ausgabe "-rwxr-xr--" endet.
Eine weitere Möglichkeit Zugriffsrechte mit dem Befehl *chmod* zu setzen, besteht darin die bestehenden Rechte einzeln abzuändern. Dies geschieht mit *chmod <Wer><+/-><Was> <Datei>*:

- <Wer> kann aus einem oder einer Kombination von "u" für User, "g" für Group, "o" für Others und "a" für alle bestehen.
- "+" oder "-" sagen aus, ob das Recht hinzukommen oder weggenommen werden soll.
- <Was> kann wiederrum aus einem oder einer Kombination aus "r", "w", "x" und "s" bestehen.

So entzieht *chmod go-rwx <Datei>* z.B. allen Nicht-Eigentümern der Datei alle Rechte.

Um die Zugriffsrechte einer neuangelegten Datei zu beeinflussen, gibt es eine Art "default[15]" für Zugriffsrechte, die mit der *umask(2)* Funktion der Shell gesetzt werden. Mit dieser Funktion wird eine Maske gesetzt, welche Rechte ausblendet. Ausgehend von "0777" des o.g. Zahlenmodells (also Schreib- und Leserechte für jedermann; Bei Erzeugung von Executables, z.B. durch einen Compiler, auch noch Ausführungsrechte), werden mit *umask <Maske>* Rechte ausgeblendet. z.B. setzt *umask 22*[16] die Default-Rechte auf "-rw-r--r--" oder *umask 0007* auf "-rw-rw----".

[15] Grundeinstellung
[16] Führende Nullen werden ignoriert.

## 3.4 Prozeßrechte

Obengenannte Rechte für Dateien, gelten bedingt genauso für laufende Prozesse. Wird ein Prozeß gestartet, so läuft dieser prinzipiell mit der UID des Aufrufers. Im Falle des s-Bits (s. 3.3) trennt sich die Identität des Prozesses in die UID des Aufrufers, also dessen gesamten Umfeld, wie Umgebungsvariablen o.ä., und der effektiven UID (EUID) auf. Die EUID bestimmt die Rechte die dieser Prozess bekommt, nämlich jene des Besitzer der Datei. Ein Prozeß kann dann wiederrum lesend oder schreibend auf Dateien zugreifen, so wie dies der jeweilige User tun dürfte. Prozeßsignale, also Signale die ein Benutzer an einen laufenden Prozeß senden kann (z.B. um den Prozeß zu beenden oder ein gewisses Verhalten zu erzwingen), können an einen Prozeß gesendet werden, wenn der Prozeß dem Benutzer gehört. Prozesse, die von einem (Parent-) Prozeß abstammen, sog. Childs die durch den *fork(2)* Vorgang entstehen, können auf gleiche Weise beeinflusst werden.

Mit der "-u" Option des *ps(1)* Kommandos läßt sich die UID eines Prozesses anzeigen, z.B.:

```
orpaulze@pcinfo7:~> ps -u
USER       PID %CPU %MEM   VSZ  RSS  TT  STAT STARTED       TIME COMMAND
orpaulze   565  5.0  3.2   448  192  p1  R+    12:53AM   0:00.06 ps -u
orpaulze   341  0.1  9.6   656  600  p1  Ss     8:15PM   0:01.68 -tcsh (tcsh)
```

Hier gehören beide Prozesse (PID 565 und 341) dem User "orpaulze".
Wieviel Prozesse unter der UserID eines Benutzers laufen dürfen, kann in den meisten Unix-Systemen vom Systemverwalter begrenzt werden. Mehr zum Thema Prozesse und deren Verwaltung im Kapitel 3.6

## 3.5 Filesystem

Das Filesystem-Konzept von Unix unterscheidet sich prinzipiell von dem der Microsoft Produkte DOS und Windows3.x/95/NT. Es macht, aus der Sicht des Benutzers, keine Unterscheidung zwischen den physikalischen Laufwerken anhand von Laufwerksbuchstaben oder Ähnlichem. Vielmehr "verbaut" es alle zur Verfügung stehenden Massenspeicher-Devices, wie Festplatten-Partitionen, CD-ROM's , Disketten, Netzwerkverzeichnisse, usw., im Dateisystem-Baum des Systems. Für den Benutzer ist es deshalb auf den ersten Blick nicht ersichtlich, auf welchem Stück der Hardware er sich beim Betreten eines Verzeichnisses mit *cd(1)* befindet, bzw. woher eine Datei wirklich stammt. Welche Devices nun welchen Teil im Dateisystem repräsentieren, kann der Systemverwalter jederzeit festlegen, der entsprechende Befehl zum "anmontieren" eines Massenspeichers an ein Verzeichnis ist *mount(8)*.

Ihm übergibt man das entsprechende Device-File (z.B. /dev/wd0a), das Verzeichnis unter dem das Device "hängen" soll, den Filesystem-Typ und eventuelle Optionen, die z.B. anzeigen das das Device nur read-only gemountet werden soll, also Schreibzugriffe prinzipiell nicht möglich sind. Gängige Filesystem-Typen für FreeBSD sind z.B.:

- `ufs` Das lokale Unix-Filesystem für Festplatten oder Disketten. Auf ihm lassen sich Dateien mit einer Dateinamenlänge von bis zu 254 Zeichen speichern und Rechte, wie in 3.3 beschrieben, vergeben.

- `msdos` Eine Möglichkeit auch MS-DOS Filesysteme ("FAT") unter Unix[17] verwenden zu können. Dieses Filesystem unterliegt in der Dateinamen-Vergabe natürlich der 8.3-Einschränkung von DOS und besitzt keine Art von Benutzerreche, bzw. Datei-Eigentümer.

- `iso9660` Das ISO-kompatible Filesystem für die meisten CD-ROMs. Dateinamen dürfen hier bis zu 254 Zeichen lang sein. Eine Vergabe von Rechten oder Datei-Eigentümern gibt es nicht.

- `nfs` Das Network-Filesystem für den Zugriff auf Verzeichnisse anderer Rechner im Netzwerk. Prinzipiell kann jeder Unix-Rechner Verzeichnisse auf diese Weise für andere zur Verfügung stellen, Clients existieren aber auch für fast alle gängigen Betriebssysteme. Dateinamenlängen und Zugriffsrechte sind vom verwendeten Filesystem auf dem NFS-Server abhängig.

- `procfs` Dies ist ein Filesystem, welches ähnlich wie NFS nicht auf lokal vorhandene Hardware zugreift. Es repräsentiert die momentan auf dem Unix-System laufenden Prozesse. Jeder Prozess erhält ein Verzeichnis mit der PID als Namen, in dem verschiedene Daten über den Status dieses Prozesses gespeichert werden. Programmme zur Prozeß-Überwachung können aus diesen Dateien lesen und so die Prozeß-Informationen anzeigen.

- `swap` Auf Devices diesen Typs werden Inhalte von Speicherbereich ausgelagert, die im RAM keinen Platz mehr haben und momentan nicht von Prozessen benutzt werden. Im Gegensatz zu den "Auslagerungsdateien" von Windows wird unter Unix prinzipiell nur in komplette Festplatten-Partitionen ge-"swappt" und nicht in eine Datei im Dateisystem. Mehrere Swap-Partitions sind möglich.

[17] Meist nur auf PC-Unixarten zu finden

Darüber hinaus gibt es noch eine Vielzahl weiterer Filesystem-Typen, die man für spezielle Anwendungen, aufgrund ihrer Robustheit, besonderer Funktionen oder Geschwindigkeit, einsetzen kann.

Welche Devices beim Booten des Systems wohin gemountet werden, wird in der Datei */etc/fstab* (*fstab(5)*) festgelegt. Eine Typische fstab kann z.B. so aussehen:

```
/dev/wd0a                  /             ufs     rw 1 1
/dev/wd0s1                 /dos          msdos   ro 0 0
/dev/wd0s2e                /usr          ufs     rw 1 1
proc                       /proc         procfs  rw 0 0
suninfo1:/home             /home         nfs     rw 0 0
suninfo1:/usr/spool.mail   /var/mail     nfs     rw 0 0
suninfo1:/soft/FreeBSD     /usr/local    nfs     rw 0 0
```

In diesem Beispiel werden 3 lokale Festplatten-Partitionen beim Booten gemountet. Zwei davon mit dem Unix-Filesystem (ufs) für Lese- und Schreibzugriffe (rw), eine mit einem MSDOS-Filesystem für Nur-Lese-Zugriff (ro). Desweiteren wird das Prozeß-Filesystem unter */proc* gemountet und drei Netzwerk-Verzeichnisse eines anderen Rechners (suninfo1) per NFS eingehängt. Das fünfte Feld gibt an, ob das Filesystem mit *dump(8)* gesichert werden soll. Im letzten Feld läßt sich eine Reihenfolge für den Filesystem-Check[18] beim Booten festlegen.
Benutzer können jederzeit mit dem Kommando *mount* (ohne Parameter aufgerufen) feststellen welche Devices oder Netz-Verzeichnisse im System verwendet werden, aber auch die Ausgabe von *df(1)* zeigt dies, zusammen mit Informationen über den belegten Platz auf dem Filesystem, an.

```
orpaulze@pcinfo7:~> df
Filesystem              1K-blocks    Used    Avail Capacity  Mounted on
/dev/wd0a                   19966   14458     3910    79%    /
/dev/wd0s1                  30676   14016    16660    46%    /dos
/dev/wd0s2e                 89102   71702    10270    87%    /usr
procfs                          4       4        0   100%    /proc
suninfo1:/home             673043  461387   144352    76%    /home
suninfo1:/usr/spool.mail   411022  156740   213180    42%    /var/mail
suninfo1:/soft/FreeBSD    1476207  761703   640694    54%    /usr/local
```

Das die Prozentangaben für die Kapazität manchmal auf den ersten Blick nicht mit dem belegten Platz in Einklang stehen, rührt daher, daß beim Anlegen eines Filesystems (mit *newfs(8)*) vom Systemverwalter ein gewisser Prozentsatz an Speicherplatz reserviert werden kann, auf den nur “root”, bzw. Prozesse mit der (E)UID 0, schreiben darf. Dies verhindert das Massenspeicher durch Benutzer soweit belegt werden, daß der normale

[18] Damit keine “unsauberen” Filesysteme, z.B. nach einem unsachgemässen Reboot oder einem Absturz des Systems, verwendet werden, wird während des Bootens eine Überprüfung und, soweit möglich, Reperatur des (lokalen) Filesystems mit *fsck(8)* vorgenommen.

System-Betrieb aufgrund von Speicherplatzmangel nicht mehr gewährleistet ist.

Selbst bei einem rein lokalen Unix-System empfiehlt es sich den Dateisystem-Baum auf verschiedene Filesysteme aufzuteilen. Meist wählt man hier eine Aufteilung in mindestens "/" (die Root-Partition, immer die erste im System) und "/usr", oft auch noch "/var". Im ersten Bereich liegen die wichtigsten Programme zum Betrieb des Systems, zumindest soweit das es bootet und andere Filesysteme mounten, checken oder anlegen kann, sowie der Kernel selber. In "/usr" befinden sich alle weiteren, oft auch nachträglich installierten Programme, sowie Manual-Pages, Programm-Bibliotheken und ähnliches. Unter "/var" sollten sich ausschließlich schnell-verändernde Daten wie Logfiles oder Statusinformationen befinden. Vorteil dieser Trennung im Filesystem ist, daß wenn eine Partition defekt ist, nicht gleich alles betroffen ist, sondern nur jener Teil auf eben dieser Partition. Zumindest Bootet das System (solang nich "/" defekt ist) und man kann dann mit den Utilities auf der Root-Partition weitere Maßnahmen ergreifen. Wenn die Root-Partition defekt sein sollte, reicht es in den meisten Fällen diese neu zu erstellen[19] und hoffentlich vorher gesicherte Konfigurations-Dateien aus */etc* wieder einzuspielen. Auch ein gewisser Geschwindigkeitsgewinn, vorallem bei Aufteilung auf verschiedene Festplatten oder der Verwendung von speziellen Filesystem-Typen für spezielle Anwendungen (z.B. News-Server oder */tmp*), ist zu beobachten.

Soll ein Rechner für einen oder mehrer andere einen Teil des Filesystems zugänglich machen, so muß dies in der Datei */etc/exports* (*exports(5)*) eingetragen werden. Als mögliche "Nutzer" der in dieser Datei angegebenen Verzeichnisse können Rechner über Hostname oder IP-Adresse, Rechnergruppen (über NIS-netgroups) oder (in neueren BSD-Varianten) IP-Subnetze eingetragen werden. Dort kann auch festgelegt werden, ob die NFS-Clients in diesen Bereich schreiben dürfen und ob dortige User mit der UID 0 (also root) gleichwertig wie dem root auf dem NFS-Server behandelt werden sollen[20]. Zur Identifikation aller anderen User werden einzig die UID und die GID verwendet, d.h. gehört eine Datei auf einem Rechner einer gewissen UID, so wird diese Zugehörigkeit auch auf allen anderen Rechnern, die dieses NFS-Verzeichnis verwenden, erstellt, gleich o.b. diese UID/GID auch dem "Namen nach" gleichen Benutzer gehöert.
Eine exports-Datei kann z.B. so aussehen:

```
/proj/archiv          -ro,access=pcinfo:pclehr:eddie.technik.bos-muenchen.de
/var/spool/mail       -access=pcinfo:pclehr
/usr/spool.mail       -access=pcinfo:pclehr
```

[19]z.B. mit der Fixit-Floppy von FreeBSD.

[20]Normalerweise wird UID 0 auf NFS-Clients zu "nobody" und GID 0 zu "nogroup" gemappt, da man nicht davon ausgehen kann, daß ein root auf dem NFS-Server der gleiche ist wie auf dem NFS-Server. Mappt man diese UID/GID allerdings auch auf dem Server als 0, so hat ein root auf einem NFS-Client die gleichen Rechte wie der Server-root in diesem Bereich des Dateisystems.

```
/home                   -access=pcinfo:pclehr:kiste.muffin.org
/soft/local             -ro,access=pcinfo
/mnt                    -access=pcinfo,root=pcinfo7
/cdrom                  -ro,access=pcinfo
```

## 3.6 Shell

Die Shell ist der Prozess welcher die Benutzereingaben entgegennimmt und die vom Benutzer gewünschten Befehle ausführt, ähnlich dem "`COMMAND.COM`" unter DOS. Sie wird üblicherweise direkt nach dem Einloggen[21] gestartet. Die typischen Shells beinhalten in der Regel eigene Porgrammiersprachen die Schleifenkonstrukte und Funktionen beherrschen. Man unterscheidet im groben drei Klassen von Shells, die Bourne-Shells, die C-Shells und alle übrigen. Die Gruppe der Bourne-Shells umfasst die Bourne-Shell (sh), die Korn-Shell (ksh), die GNU Bourne-Again Shell (bash) und die Z-Shell (zsh). Diese Gruppe versteht (mehr oder weniger) die gleiche Syntax an Befehlen deshalb ist es besonders leicht zwischen ihnen hin und her zu wechseln. Die zweite bekannte Gruppe der Shells umfasst die TC-Shell (tcsh) und die C-Shell (csh). Sie unterscheiden sich von den Bourne-shells durch eine andere, mehr an die Programmiersprache C angelehnte, Syntax, welche sie inkompatibel zu den Bourne-Shells macht. Zuletzt gibt es auch noch jede Menge Shells, die nicht in die o.g. Gruppen passen, z.b. die Plan9-Shell (rc), die Key-Shell (keysh) und viele andere, die aber alle meistens für irgend einen speziellen Zweck und nicht als allgemeine Shell zu empfehlen sind. Um die Wahl der richtigen Shell zu erleichtern hier ein kleiner Überblick:

- `sh` Die sh ist sozusagen die Ur-Shell, man wird sie auf jedem Unix-System als /bin/sh finden. Sie ist als User-Shell nicht zu empfehlen, da sie kaum Editiermöglichkeiten der Eingabezeile bietet. Allerdings wird sie aufgrund ihrer hohen Verfügbarkeit gerne für Shell-skripten verwendet.

- `csh` Die csh ist die "erste" benutzerfreundlichere Shell gewesen, und deshalb auch auf vielen Systemen unter /bin/csh zu finden, allerdings hat sie viele BUG's[22] und ist wie alle C-Shells nicht kompatibel zur sh. Darum kann ich sie nicht empfehlen. [23]

- `tcsh` Die tcsh ist die Weiterentwicklung der csh, und deshalb zu dieser kompatibel. Sie ist schon in der Grundeinstellung eine der benut-

[21] siehe auch 3.2
[22] Bug's sind Fehler in einem Programm
[23] http://late5.e-technik.uni-erlangen.de/Software-Doku/csh-harmful.html

zerfreundlichsten Shells, und wird deshalb auch von den mit Abstand meisten Benutern bevorzugt. Allerdings ist die mangelnde Bourne-Shell Kompatibilität ein Manko wenn man ab und zu auch Shell-Skripten schreiben will, denn wer lernt schon gerne doppelt so viel ?

- `ksh` Wieder eine Shell aus der Bourne-Shell-Reihe, die auch recht benutzerfreundlich ist und einem die Wahl zwischen einem Emacs- und einem vi-kompatiblen Editiermodus bietet. Sie ist eine der wenigen Shells, die mehrzeilige Kommandos editieren und in der History verarbeiten kann.
- `bash` Die Neuimplementation der Bourne-Shell von GNU. Sie ist auch in der Grund-Einstellung recht benutzerfreundlich, bietet aber nicht so viele Konfigurationsmöglichkeiten. Sie ist mit der tcsh eine der beliebtesten Shells.
- `zsh` Die zsh bietet alles was die vorhergehenden Shells auch bieten (insbesondere ein vernünftiges multi-line editing (ähnlich der ksh) und eine programmierebare Completion (ähnlich der tcsh)) allerdings ist es ein höherer Aufwand die Features zu lernen und zu konfigurieren, so daß sie den eigenen Wünschen ensprechen.

Nun etwas über die Benutzung einer typischen Bourne-Shell:

kommando

Falls "kommando" ein Alias oder eine Shell-Funktion ist, wird dieses intern ausgeführt. Ansonsten wird kommando im Pfad (`$PATH`)[24] gesucht und ausgeführt, (Ausführbare Programme werden unter UNIX mit einem x-bit[25] gekennzeichnet und nicht mit einer Endung (.com,.exe)). Die Trennung zwischen einem Befehl und dem nächsten erfolgt entweder durch den Beginn einer neuen Zeile oder durch ein ;.
Beispiel: `ls -l ; who`

Normalerweise liest ein Befehl von der Tastatur und gibt die Daten auf dem Bildschirm aus. Dies kann man natürlich auch ändern:

kommando < datei1 >datei2 2>datei3

Liest die Eingabe für den Prozeß "kommando" aus datei1 und schreibt das Ergebnis in datei2. Etwaige Fehlermeldungen landen in datei3[26]
Beispiel:
`grep Milch <einkaufszettel`; `uptime > 8.April`; `make 2>errors`

[24] Im Gegensatz zu DOS wird . nicht implitzit durchsucht

[25] siehe 3.3

[26] In Wirklichkeit wird lediglich Filedescriptor Nr. 2 in die Datei umgelenkt, per Konvention sollte ein Programm Fehlermeldungen aber dort ausgeben, was dann zum gewünschten

kommando<<Marke

Liest die Eingabe für "kommando" aus der selben Quelle aus der die Befehle gelesen werden, bis zur Endemarkierung **Marke**:

```
mail sec@42.org <<ende
Hi,
dies ist meine testmail
ende
```

Man kann natürlich auch die Ausgabe eines Prozesses direkt als Eingabe für den nächsten Prozeß nehmen. Dies ist eine sogenante "Pipeline":

kommando1 | kommando2 | kommando3

Verwendet die Ausgabe von kommando1 direkt als Eingabe für kommando2 u.s.w.
Beispiel: `ls -l | sort -n +4 | more`

'kommando'

Die sogenannten "Backticks" ersetzen den Platz in der Befehlszeile durch die Ausgabe des Befehls.
Beispiel: `finger 'whoami'`

Filename globbing

Die Shell verwendet gewisse Zeichen um Filenamen zu "erzeugen"[27]
`*` steht für beliebig viele (auch garkeine) Zeichen, **nicht** jedoch für einen Punkt am Anfang eines Filenamens.
`?` steht für genau ein beliebiges Zeichen, wiederum jedoch **nicht** für einen Punkt, wenn es am Anfang eines Filenamens steht.
`[abc]` steht für genau eins der Zeichen zwischen den eckigen Klammern.
`[a-q]` für genau ein Zeichen aus dem Bereich.
`[!...]` für genau ein Zeichen das **nicht** aus ... ist
Beispiel: `ls einkauf*` oder `lp [1-3].April`

Variablen

Die Shell verwaltet sogenannte "Variablen" die beliebige Werte (Zahlen, Strings) annehmen können. Man unterscheidet 2 Arten von

Effekt führt. Man kann Filedescriptoren aber auch auf andere Filedescriptoren umlenken z.B.: `traceroute pcinfo7>/dev/null 2>1` wird die Fehlerausgabe zusammen mit der Standardausgabe ins nichts umgelenkt.

[27] Im Gegensatz zu DOS, wo jede Anwendung selbst dafür verantwortlich ist, diese Zeichen auszuwerten.

Variablen, normale und sog. Environment-Variablen. Erstere sind nur der Shell selbst bekannt, wobei letzere auch an von der Shell gestartete Prozesse weitergegeben werden. Eine normale Variable wird zu einer Environment-Variablen in dem sie mit dem `export`-Befehl als solche markiert wird.
Beispiel: `DISPLAY=pcinfo7:0;export DISPLAY;xterm`

Die Shell selber expandiert Variablen zu ihren werten, wenn man ihnen ein $ voranstellt: `eins=zwei; echo eins: $eins`
Man kann allerdings eine Variable gezielt an genau ein Programm übergeben, indem man die Definition durch ein Leerzeichen getrennt vor den Programmaufruf stellt
Beispiel: `NNTPSERVER=news.lrz-muenchen.de tin`

Die Shell stellt uns einige vordefinierte Variablen zu Verfügung:

| | |
|---|---|
| `$SHELL` | Pfad zur Loginshell |
| `$HOME` | Home-Directory |
| `$USER` | Username |
| `$?` | Exit-Code des zuletzt ausgeführten Prozesses |
| `$$` | Prozeß-Id der Shell |
| `$!` | Prozeß-Id des zuletzt im Hintergrund ausgeführten Prozesses |

Quoting

Es gibt verschiedene Möglichkeiten die Expansion von Variablen und das Filename-globbing zu verhindern, dies nennt man Quoting. Wenn ein Text von "single quotes" `'` begrenzt wird, findet darin keinerlei Umwandlung durch die Shell statt, selbst Zeilenumbrüche können auf diese Weise eingegeben werden: `echo 'eink*' '$eins'`
Verwendet man zur Begrenzung "double quotes" `"`, so wird das Filename-globbing unterbunden, alles andere (wie Variablenexpansion, Auswerten von Backticks ...) wird weiterhin durchgeführt: ``echo "*" "`id -u`"``
Der Backslash `\` verhindert lediglich die Sonderbehandlung des nachfolgenden Zeichens: `echo \'eink*\'`

Job control

Natürlich kann man auch von einer Shell aus mehrere Prozsse starten, und sogar verwalten. Um einen Befehl im "Hintergrund" auszuführen schliesst man ihn einfach mit einem `&` statt mit einem `;` ab: `xterm &`
Um einen bereits normal im Vordergrund laufenden Prozeß in den Hintergrund zu befördern muß man ihn zuerst "suspenden" (anhalten)

um wieder zur Shell zurückzukehren. Dies erreicht man in der Regel mit `^Z`[28] in der Shell in der Prozeß gestartet wurde. Man kann allerdings auch aus einer anderen Shell heraus dem Prozeß ein STOP-Signal schicken: `kill -STOP <pid>`
Die Shell selber bietet einem dann die Möglichkeit mit `jobs` die laufenden Prozesse einzusehen, und sie dann mit `fg` oder `bg` in der Vorder- bzw. Hintergrund zu verschieben. Zu diesem Zweck nummeriert die Shell die Prozesse durch, diese Zahlen können an stelle der PID's fuer fg,bg und kill verwendet werden, wenn man ihnen ein `%` voran stellt. die Sobald ein im Hintergrund laufender Prozeß etwas von der Tastatur lesen[29] will wird er jedoch angehalten (um Konflikte zu vermeiden) was die Shell mit einem `<pid> Stopped (input)` anmerkt und wartet bis man ihn wieder in den Vordergrund holt.

Funktionen

Man kann beliebige Shell-stückchen als Funktion deklarieren, indem man das Stück in `{}` fasst und ein `funktionsname()` voranstellt. Man kann innerhalb der Funktion auf die Aufrufparameter mittels `$1 $2 ...` oder mit `$@` und `$*` zugreifen: (die beiden unterscheiden sich nur durch ihre Expansion innerhalb von double quotes: `"$@"` → `"$1" "$2"` und `"$*"` → `"$1 $2"` )

```
hello(){
  echo Hello $*, this is a nice new World.
}

hello you there
```

Kontrollstrukturen

Nun fehlen uns noch die bedingten Befehle:

`befehl1 && befehl2` : befehl2 wird nur ausgeführt, wenn befehl1 wahr zurückgeliefert[30] hat.
`befehl1 || befehl2` : befehl2 wird nur ausgeführt, wenn befehl1 false zurückgeliefert hat.

```
if bedbefehl ; then
```

[28] Dieses Kontroll-Zeichen kann man mit dem Befehl `stty(1)` beliebig umstellen

[29] Manchmal auch schon bei der Ausgabe auf den Bildschirm, dieses Verhalten lässt sich jedoch wiederum mit `stty tostop` einstellen

[30] wahr entspricht einem Rückgabewert (`$?`) von 0, false allen anderen.

```
  wbefehl
else
  fbefehl
fi
```

Wenn der bedbefehl wahr zurückliefert, wird wbefehl ausgeführt, ansonsten fbefehl. Der else-Fall kann natürlich auch komplett weggelassen werden. Für die Verwendung von if ist die Kenntniss des "test(1)" - Befehls wichtig:
`test ausdruck` liefert true zurück wenn "ausdruck" wahr ist, und false sonst. Die wichtigsten Elemente für "ausdruck" sind:

| | |
|---|---|
| `-d file` | wahr wenn file ein Directory ist |
| `-w file` | wahr wenn file schreibbar ist |
| `-e file` | wahr wenn file existiert |
| `n1 -lt n2` | wahr wenn n1<=n2 |
| `n1 -eq n2` | wahr wenn n1=n2 (Zahlen) |
| `s1 = s2` | wahr wenn sq=s1 (Strings) |
| `! ausdruck` | wahr wenn ausdruck falsch ist |
| `a1 -a a2` | wahr wenn a1 und a2 wahr sind |
| `a1 -o a2` | wahr wenn a1 oder a2 wahr ist |

test(1) kann auch als `[` aufgerufen werden, wenn man der Argumentliste eine abschließende `]` hinzufügt.

```
case a in
match) befehl ;;
...
esac
```

wenn der Inhalt von Variable a auf einen der "match[31]" passt, dann wird der zugehörige Befehl ausgeführt.

Schleifen

Um die Shell zu einer richtigen Programmiersprache zu, machen fehlt natürlich noch etwas. Richtig – Schleifen:

```
for variable in liste ; do
  befehl
done
```

[31] verwendet die selben Sonderzeichen wie beim Filename globbing

“liste” sind durch Spaces[32] getrennte Werte. Die Schleife wird für jeden Wert aus liste einmal durchlaufen

```
while befehl; do
  befehl2
done
```

solange die Ausführung von “befehl” nicht fehlschlägt, wird befehl2 ausgeführt.

[32] In Wirklichkeit alle in `$IFS` aufgezählten Zeichen, normalerweise Space, Tab und Newline. Als Benutzer kann ich das aber jederzeit ändern, ist aber normalerweise nur in Shellskripten sinnvoll

**Beispiel:**

```
crsr(){
  BKSP=`tput kb`
  BKSP="$BKSP$BKSP"
  while true ; do
    echo -n " |$BKSP" ; sleep 1
    echo -n " /$BKSP" ; sleep 1
    echo -n " -$BKSP" ; sleep 1
    echo -n " \\$BKSP"  ; sleep 1
  done
}

checkfiles() {
  for file in * ; do
    if [ -d $file ] ; then
      cd $file && checkfiles && cd ..
    fi
    case file in
      *.bak) echo $file sollte gel"oscht werden;;
    esac
  done
}

crsr &
prozess=$!
checkfiles
kill $prozess
echo Fertig.
```

# Kapitel 4

# Das FreeBSD-Ports-Konzept

## 4.1 Software-Installation

Software auf einem Unix-System zu installieren gestaltet sich, will man es intelligent, homogen und auf benutzerfreundliche Weise machen, doch oft schwieriger als es auf den ersten Blick scheint. Schnell verliert man den Überblick über installierte Programme und deren Versionen, erst recht wenn man diese in einem ganzen Netz installieren muß. Spätestens wenn sich die Kapazität der Festplatte dem Ende zuneigt und man versucht Nichtbenötigtes zu löschen oder temporär "auszulagern", merkt man in welchen Ecken des Filesystems sich ein Programm überall hininstalliert hat: Konfigurationsdateien, Manualpages, weiterführende Dokumentation, evtl. benötigte Bibliotheken und nicht zuletzt das ausführbare Programm (oder mehrere) liegen über die Festplatte verteilt und lassen sich oft nur schwer als zusammengehörend erkennen.
Eine Möglichkeit zumindest den gröbsten Wildwuchs zu vermeiden, ist das gewissenhafte Editieren des zu den Programm-Sourcen mitgelieferten *Makefile* um dort auf die Gegebenheiten des verwendeten Systems einzugehen und so die einzelnen Programmbestandteile an möglichst zentralen Stellen zu installieren (*make install*). Dies verhindert zwar das sich jedes Programm an anderer Stelle im System installiert, eine Kontrolle was alles und wohin installiert wurde hat man damit aber noch nicht.

## 4.2 Package-Systeme

Viele moderne Unix-Systeme (u.a. Solaris, Linux und FreeBSD) gehen deshalb einen für den Anwender sehr einfachen und komfortablen Weg: Programme samt allem benötigten Zubehör werden für das System fertig compiliert

in einem “Packet”, dem sog. Package, geliefert. dieses muß der Anwender, die nötigen Rechte vorrausgesetzt, nur noch mit einem Befehl (z.B. *pkg_add(1)*) installieren. Bei diesem Vorgang wird das Package ausgepackt, alle benötigten Dateien an sinnvolle Stellen kopiert und das ganze in einer Datenbank vermerkt. Anhand dieser Datenbank ist es jederzeit möglich, nachzuvollziehen was alles für dieses Programm installiert wurde. Dadurch gestaltet sich auch das Entfernen (z.B. mit *pkg_delete(1)*) denkbar einfach, da das System nur noch in der Datenbank nach der Aufzeichung suchen muß, die dort vermerken Datein löschen und den Eintrag aus der Datenbank entfernen. Unter FreeBSD befindet sich diese Datenbank im Verzeichnis */var/db/pkg/*. Darunter ist für jedes installierte Package ein Verzeichnis mit dem Programmnamen als Verzeichnisnamen zu finden, welches ein File mit den für das Programm installierten Dateien enthält (*+CONTENTS*). Zusätzlich finden soch dort noch zwei Dateien, welche Beschreibungen für diese Software (eine Einzeilen-Version, eine ausführlichere Variante) enthalten. Diese werden z.B. von Package-Verwaltungsprogramme wie *pkg_info(1)* angezeigt. Mit diesem Kommando läßt sich (mit der Option *-aI*) auch anzeigen welche Packages sich auf dem System befinden.
Das unter Linux sehr verbreitete RPM-Format und das Solaris-Packagesystem arbeiten ähnlich wie oben beschrieben. Allen gemeinsam ist der Nachteil das das Programm schon fertig compiliert, und somit evtl. nicht an die eigenen Bedürfnisse angepaßt, geliefert wird. Programm-Sourcen um eigene Einstellungen, die nur vor dem Compilieren möglich sind, vorzunehmen, sind nicht in solchen Packages enthalten. Zudem ist man vom Willen der System-Hersteller oder Programmautoren, auch solche Pakete für das entsprechende System anzufertigen und zu verteilen (FTP, CDROMs), abhängig. Auch Abhängigkeiten zwischen verschiedenen Packeten, wenn also der Betrieb eines bestimmten Programms das Vorhandensein eines anderen bedingt, sind meißt nur in Form einer kurzen Warnung zu finden.

## 4.3 Die Kombination: Ports

Die Vorteile des Package-Systems zu genießen und trotzdem möglichst unabhängig von vor-compilierten Programmen und dem Goodwill der Autoren oder freiwilliger Package-Maintainer zu sein, veranlaßte die FreeBSD-Entwickler das sog. Ports-System zu erstellen. Bei dieser Art der Software-Verwaltung wird sich ausschließlich auf die Original-Quellen des jeweiligen Programmautors gestützt und mit Hilfe der für jeden Port verfügbaren Patches[1] das Programm compiliert und, basierend auf dem

[1]Änderungen im Quellcode von Programmen um sie möglichst optimal an das vorhandene System anzupassen, bekannte Bugs zu entfernen oder spezielle Features zu imple-

Package-System, verwaltet. Dies setzt natürlich vorraus das sich jemand für das Erstellen und die Aktualität eines solchen "Ports" zuständig fühlt. Erledigt wird dies durch die Port-Maintainer, welche sich jeweils für ein oder mehrere Ports verantwortlich zeichnen. Die für den Port benötigten Quellen, sucht das System zuerst im Verzeichnis */usr/ports/distfiles/* und versucht sie per FTP oder HTTP von der Home-Site oder einem anderen Server zu bekommen, wenn sie sich nicht in diesem Verzeichnis befinden. Nachdem sie über das Netz geholt wurden, werden sie dann in */usr/ports/distfiles/* abgelegt. Im Verzeichnis */usr/ports/* befinden sich mehrere Unterverzeichnisse, welche eine grobe Gliederung in Kategorien (System-Utilities, X11-Programme, Netzwerk-Programme, News- und Mail-Systeme und -Clients, usw.) darstellen, darunter jeweils ein Verzeichnis pro Port, also jedes Programmpacket welches von FreeBSD bzw. einem Maintainer als Port unterstützt wird. Als Beispiel sei hier der Port für die "pdksh", der Public-Domain Korn-Shell, näher erklärt. Alle anderen Ports halten sich aber streng an die beschriebenen Mechanismen.

Das Verzeichnis */usr/ports/shells/pdksh/* sieht in etwa so aus:

```
drwxrwxr-x  2 root  wheel  512 Apr  6 18:18 CVS
-rw-rw-r--  1 root  wheel  374 Nov 18 12:39 Makefile
drwxrwxr-x  3 root  wheel  512 Apr  6 18:18 files
drwxrwxr-x  3 root  wheel  512 Apr  6 18:18 patches
drwxrwxr-x  3 root  wheel  512 Apr  6 18:18 pkg
```

Im Folgenden seien die wichtichsten Dateien erklärt:

```
opp@semmel2:/usr/ports/shells/pdksh% cat Makefile
# New ports collection makefile for:    pdksh
# Version required:     5.2.12
# Date created:         23 May 1995
# Whom:                 thomas@ghpc8.ihf.rwth-aachen.de
#
# $Id: Makefile,v 1.16 1996/11/18 11:39:51 asami Exp $
#

DISTNAME=       pdksh-5.2.12
CATEGORIES=     shells
MASTER_SITES=   ftp://ftp.cs.mun.ca/pub/pdksh/

MAINTAINER=     tg@freebsd.org

GNU_CONFIGURE=  yes
MAN1=           ksh.1

.include <bsd.port.mk>

opp@semmel2:/usr/ports/shells/pdksh% cat files/md5
MD5 (pdksh-5.2.12.tar.gz) = b622d7b961dbb6dc9cccd237a68ff151
```

mentieren.

```
opp@semmel2:/usr/ports/shells/pdksh% ls -l patches/
total 2
drwxrwxr-x  2 root  wheel  512 Apr  6 18:18 CVS
-rw-rw-r--  1 root  wheel  313 Nov  5  1996 patch-aa

opp@semmel2:/usr/ports/shells/pdksh% cat pkg/COMMENT
The Public Domain Korn Shell.

opp@semmel2:/usr/ports/shells/pdksh% cat pkg/DESCR
PDKSH is the Public Domain Korn Shell. Its command language is a
superset of the sh(1) shell language.

opp@semmel2:/usr/ports/shells/pdksh% cat pkg/PLIST
bin/ksh
@exec echo "updating /etc/shells"; cp /etc/shells /etc/shells.bak;
(grep -v %D/%F /etc/shells.bak; echo %D/%F) >/etc/shells
@unexec echo "updating /etc/shells"; cp /etc/shells /etc/shells.bak;
(grep -v %D/%F /etc/shells.bak) >/etc/shells
man/man1/ksh.1.gz
```

Zum Compilieren des Ports genügt ein einfaches *make* in diesem Verzeichnis:

```
opp@semmel2:/usr/ports/shells/pdksh% make
>> Checksum OK for pdksh-5.2.12.tar.gz.
===>  Extracting for pdksh-5.2.12
===>  Patching for pdksh-5.2.12
===>  Applying FreeBSD patches for pdksh-5.2.12
===>  Configuring for pdksh-5.2.12
creating cache ./config.cache
checking for gcc... cc
[....]
creating ./config.status
creating Makefile
creating config.h
===>  Building for pdksh-5.2.12
CONFIG_FILES="" CONFIG_HEADERS=config.h ./config.status
creating config.h
config.h is unchanged
date > stamp-h
cc -c  -DHAVE_CONFIG_H -I. -I. -O alloc.c
[...]
```

Dieses “make” entpackt im Unterverzeichnis *work/* die Sourcen, bereitet sie anhand der Patches optimal für FreeBSD vor und compiliert das Programm, so wie es der Autor vorgesehen hat. Es kann jederzeit in diesen Vorgang eingegriffen werden um z.B. Änderungen der Konfiguration vorzunehmen.
Ein “make install” installiert das Programmm im System und registriert es im Package-System:

```
opp@semmel2:/usr/ports/shells/pdksh% sudo make install
>> Checksum OK for pdksh-5.2.12.tar.gz.
===>  Installing for pdksh-5.2.12
./mkinstalldirs /usr/local/bin /usr/local/man/man1
install -c -s -o bin -g bin -m 555 ksh /usr/local/bin/`echo ksh|sed 's,x,x,'`
/usr/bin/install -c -o bin -g bin -m 644 ksh.1
/usr/local/man/man1/`echo ksh|sed 's,x,x,'`.1
NOTE: /etc/shells does not contain /usr/local/bin/ksh  you should add
it if you want to set your shell to ksh
===>   Compressing manual pages for pdksh-5.2.12
===>  Registering installation for pdksh-5.2.12
```

Mit einem "make clean" wird das Verzeichnis *work/* gelöscht und somit alles nichtmehr benötigte entfernt.