Freie Universität Berlin

Fachbereich Mathematik und Informatik

Systemverwaltung (19615)

Automatisierung

Skript zum Blockkurs im Sommersemester 2007

Daniel Bößwetter

<*boesswet@inf.fu-berlin.de*>

Stand 31.07.07

Inhaltsverzeichnis

Einleitung

Dieses Dokument entstand als Begleitskript zur Lehrveranstaltung „Systemverwaltung“ (19615) am Fachbereich Mathematik und Informatik der Freien Universität Berlin im Sommersemester 2007. Es umfasst nicht den kompletten Kurs-Umfang, sondern nur den Teil „Automatisierung“, welcher sich mit dem Aspekt der Automatisierung des Betriebs von Unix- und Windows-Systemen und -Netzen, d.h. überwiegend mit dem Scripting von Verwaltungsaufgaben beschäftigt. Da die Veranstaltung in dieser Form zum ersten Mal stattfindet, ist der Inhalt naturgemäß noch Änderungen unterworfen und Fehler sind nicht ganz auszuschließen, aber wegen des sehr genau definierten inhaltlichen Schwerpunktes (Automatisierung im Gegensatz zu allgemeinen Systemverwaltung), wurde davon abgesehen ein Skript von einer anderen Lehrveranstaltung (einer anderen Hochschule) oder ein fertiges Buch zu verwenden.

Das Skript teilt sich in zwei Teile: einen zu Unix und einen zu Windows. Beide Teile erklären die notwendigen theoretischen Grundkenntnisse, die für die Automatisierung notwendig sind und gehen dann auf die jeweiligen Möglichkeiten zur Script-Steuerung ein. Es wird im allgemeinen sehr beispielhaft vorgegangen, da in der verfügbaren Zeit (ca. 3 Tage) natürlich nicht alle Feinheiten und Kniffe vermittelt werden können. Die theoretischen Erläuterungen sollten aber in Kombination mit den (meistens aus der Praxis gegriffenen) Beispielen ein Gefühl für die Vorgehensweise vermitteln und Anhaltspunkte für weitere Recherchen liefern (Hilfe zur Selbsthilfe). So kann es oft vorkommen, dass als Beispiel für ein Konzept ein Kommando verwendet wird, dessen Wirkung zuvor nicht erläutert wurde – wenn dies nicht sofort im Anschluss im Text geschieht, sollte spätestens im entsprechenden Anhang eine Erklärung zu finden sein. Notfalls sollte man nach der Lektüre dieses Skripts in der Lage sein, die jeweilige Dokumentation zu finden und zu verstehen.

Durch den gesamten Text hindurch werden Markennamen verwendet, die Eigentum ihrer jeweiligen Inhaber sind. Es wird manchmal, aber nicht immer explizit darauf hingewiesen.

Nomenklatur

Zeilen mit einem Dollarzeichen ($) am Anfang sind exemplarische Unix-Kommandozeilen, wobei das $ für die Eingabeaufforderung (Prompt) steht und nicht eingegeben werden muss. Die Eingabeaufforderung sieht auf echten Systemen meistens anders aus, wird aber in der Literatur oft aus Platzgründen als $ dargestellt.

Im Unix-Teil werden häufig Begriffe mit einer Ziffer in Klammern dahinter erwähnt, z.B. *ls(1)*. Dies sind Hinweise auf Unix Manualseiten, wobei die Ziffer die Sektion des Manuals und der Teil davor der Name des dokumentierten Kommandos oder Systemaufrufs bzw. der dokumentierten Funktion oder Datei ist. Manualseiten können unter allen Unix-Systemen mit dem Kommando

```
$ man -s 1 ls
```

gelesen werden. Kennt man die Sektionsnummer nicht, reicht auch

```
$ man ls
```

welches dann die Sektionen der Reihe nach durchsucht und den ersten Treffer ausgibt. Eine Manualseite verlässt man durch drücken der Taste „q“ wieder.

Teil 1 – Unix

„Unix *is* user friendly. It's just selective about who its friends are.“
(Verfasser unbekannt)

Einführung in Unix

Das erste Unix wurde 1969 bei AT&T als Alternative zum kommerziellen Multics entwickelt[1] und hat sich im Laufe der Jahre in Forschung und Wirtschaft verbreitet und kontinuierlich weiterentwickelt, was zu einer Vielzahl kommerzieller und quelloffener, kostenfreier Versionen geführt hat, die auf einer großen Anzahl von Rechner-Plattformen vom Mobiltelefon bis zum Großrechner laufen. Genau genommen dürfen sich nur Systeme, die gemäß der „Single Unix Specification“ der Open Group zertifiziert sind, auch mit dem Markennamen Unix® schmücken, aber im allgemeinen Sprachgebrauch (sowie in diesem Skript) wird „Unix“ oftmals auch synonym für den exakteren Begriff der Unix-ähnlichen Systeme verwendet.

Unterschiedliche Versionen

Im Laufe der Jahrzehnte haben sich verschiedene Entwicklungsstränge aus der ursprünglichen Version herausgebildet (siehe Abbildung 1). Allen gemeinsam ist die „Unix-Philosophie“, welche grob gesagt bedeutet, dass es sich um offene Systeme handelt: oftmals wird damit die freie Verfügbarkeit von Quelltexten gemeint, durch die ein System auf beliebige Rechnerplattformen portierbar ist, manchmal aber auch der Einsatz von standardisierten Formaten für den Datenaustausch oder der Einsatz von offenen Standards bei den Netzwerk-Protokollen. Über die Offenheit hinaus gehört zur Unix-Philosophie, dass das System aus einer Sammlung von hochspezialisierten Werkzeugen besteht, die durch Script-Sprachen (Shells) zu mächtigeren Konstrukten kombiniert werden können. Der Austausch und die Speicherung von Daten erfolgt immer über *plain text*, also einfache, unstrukturierte oder tabellarische Textdateien (oder -ströme). Ebenfalls kann die große Auswahl an Möglichkeiten (unter Perl-Programmieren auch als TIMTOWTDI[2] bezeichnet) als Teil der Unix-Philosophie betrachtet werden – es gibt mehrere Systeme, mehrere Shells, mehrere grafische Oberflächen und für jede Aufgabe eine Vielzahl an Möglichkeiten, diese zu bewältigen. In dieser Einführung wird meistens der populärste Ansatz (z.B. die Bash-Shell) verwendet um den Teilnehmern des Kurses möglichst praktisch verwendbare Informationen an die Hand zu geben, aber dies soll natürlich nicht als der Weisheit letzter Schluss dargestellt werden.

1 Genaueres zur Historie findet sich auf Wikipedia unter http://en.wikipedia.org/wiki/Unix
2 „There is more than one way to do it.“

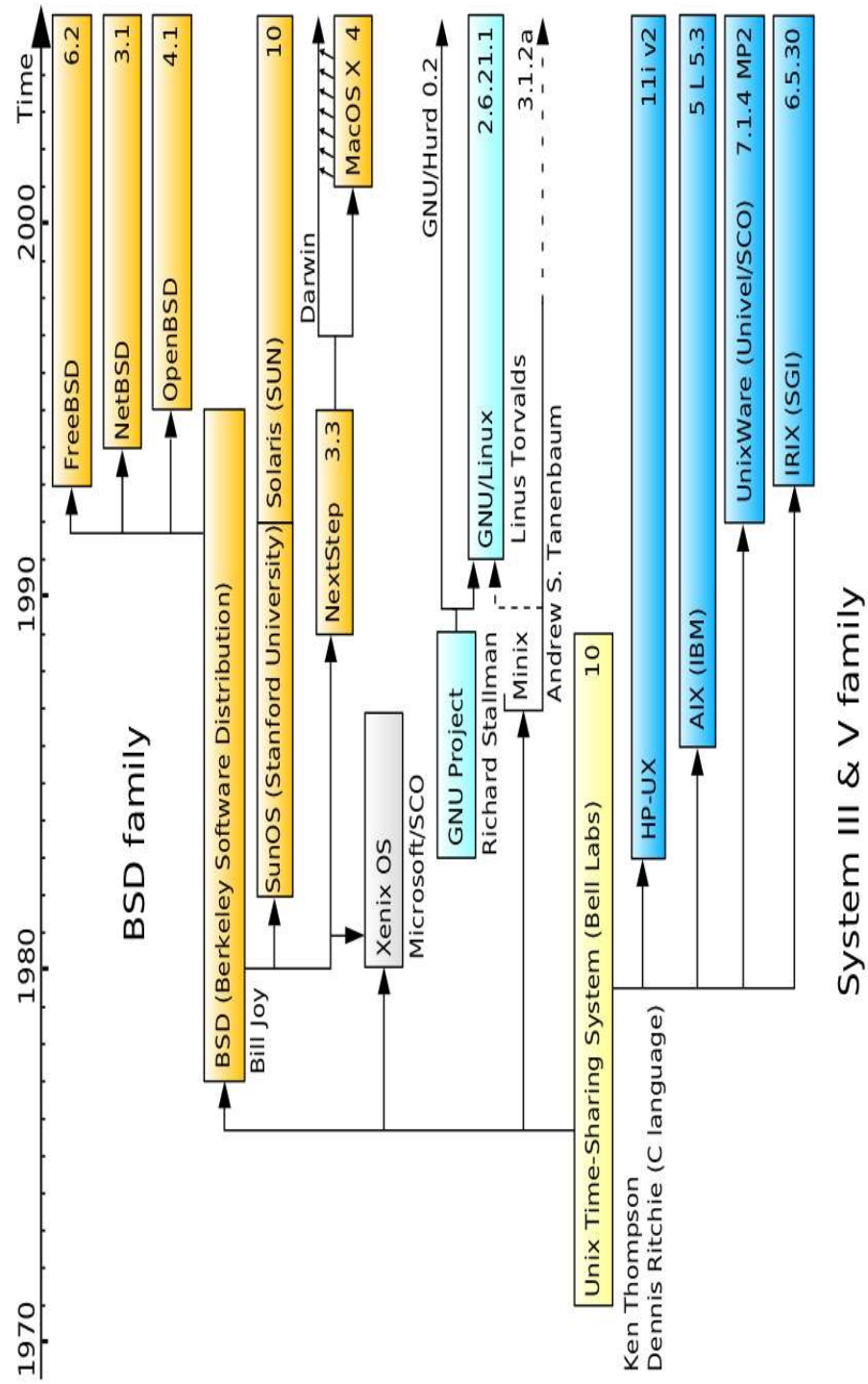
1970
1980
1990
2000
Time
BSD family
FreeBSD 6.2
NetBSD 3.1
OpenBSD 4.1
BSD (Berkeley Software Distribution)
Bill Joy
SunOS (Stanford University) 10
Solaris (SUN)
Darwin
NextStep 3.3
MacOS X 4
Xenix OS
Microsoft/SCO
GNU Project
Richard Stallman
GNU/Hurd 0.2
GNU/Linux 2.6.21.1
Linus Torvalds
Minix 3.1.2a
Andrew S. Tanenbaum
Unix Time-Sharing System (Bell Labs) 10
Ken Thompson
Dennis Ritchie (C language)
HP-UX 11i v2
AIX (IBM) 5 L 5.3
UnixWare (Univel/SCO) 7.1.4 MP2
IRIX (SGI) 6.5.30
System III & V family

Abbildung 1

Die wichtigsten Linien in der Entwicklung, die heute noch eine Rolle spielen, sind „System V“ welches auf den AT&T Quellen basiert und die Grundlage der meisten kommerziellen Unix-Systeme (Solaris, HP-UX, SCO) bildet sowie die „Berkeley Software Distribution“ (kurz BSD) von der UCB[3], die die Grundlage der quelloffenen Derivate NetBSD, OpenBSD und FreeBSD (sowie weiterer) darstellt. Wie das Bild zeigt, ist das weit verbreitete Linux nicht mit diesen beiden Linien verwandt, da der Kernel eine komplette Neuentwicklung darstellt und das System sich aus dem GNU-Projekt[4] mit System-Werkzeugen bedient. Linux orientiert sich hauptsächlich am POSIX-Standard, es finden sich aber auch viele Eigenschaften aus SVID (*System V Interface Definition*) und BSD darin wieder.

Für diesen Teil der Lehrveranstaltung beziehen wir uns meistens auf Linux, da dies weit verbreitet ist und jedem interessierten Anwender kostenfrei zur Verfügung steht. Außerdem lassen sich die meisten Konzepte auch ohne weiteres auf beliebige andere Unix-Systeme übertragen. An anderer Stelle der Lehrveranstaltung wird aber auch explizit auf Sun Solaris® eingegangen.

Grafische Benutzer-Oberflächen

Auf Grund des Alters des Systems spielen grafische Oberflächen unter Unix immer noch eine untergeordnete Rolle, obwohl diese mit dem X Window System schon seit den 80ern weit verbreitet sind. Das *X Window System, Version 11* (kurz X11) stellt allerdings im Wesentlichen ein Netzwerk-Protokoll für die Verteilung von Anwendungen (im X-Jargon *client*) und grafischem Terminal (unter X11 als *server* bezeichnet) bereit. Weiterhin gibt es Bibliotheken, die dieses Protokoll implementieren sowie Schnittstellen-Spezifikationen für Bedien-Elemente (Buttons, Menüs ...) sowie eine Referenz-Implementation vom MIT[5] dieser *widgets*, die allerdings nicht mehr heutigen Ansprüchen entspricht. In den 90ern versuchten einige namhafte Hersteller kommerzieller Unix-Systeme mit dem *Common Desktop Environment (CDE)* einen Standard für grafische Oberflächen von Unix-Systemen auf Basis von X11 zu schaffen, der allerdings heute fast keine Rolle mehr spielt. Stattdessen haben sich im Opensource-Umfeld gleich zwei konkurrierende Desktops entwickelt: das *K Desktop Environment* (KDE, eine Anspielung auf das o.g. CDE) sowie GNOME (ehem. *GNU Network Object Model Environment*).

Die meisten administrativen Tätigkeiten unter Unix werden allerdings nach wie vor auf der Kommandozeile (Shell) ausgeführt (auch wenn grafische Alternativen existieren – die meisten „echten“ Unix-Admins würden diese ohnehin nicht anfassen) und Automatisierung durch Shell-Scripting erreicht – daher wird auf grafische Oberflächen für Unix hier nicht weiter eingegangen.

3 University of California at Berkeley
4 Der eigene Kernel des GNU-Projekts (Hurd) ist bis heute nicht aus der Alpha-Phase herausgekommen.
5 *Massachusetts Institute of Technology*

Das Unix-Dateisystem

Was ist ein Dateisystem?

Der Begriff „Dateisystem“ ist mit mehreren Bedeutungen überladen, die wir im folgenden erklären wollen. Zunächst ist das Unix-Dateisystem eine hierarchische Sammlung von Dateien und Verzeichnissen, die mit Zugriffsrechten und verschiedenen Zeitstempeln versehen sind. Dieser „Dateibaum“ muss aber nicht aus einem Guss sein, sondern kann aus mehreren Teilbäumen zusammengesetzt sein, die die Informationen technisch unterschiedlich speichern (möglicherweise sogar auf verschiedenen Rechnern). Auch der Teil des Systemkerns, der die physikalische Speicherung übernimmt, wird oft als Dateisystem bezeichnet. Zu guter Letzt gibt es Konventionen, die den inhaltlichen Aufbau des Dateibaums festlegen, d.h. bestimmte Informationen (Programme, Bibliotheken, Konfigurations-Dateien) müssen an definierten Orten liegen, damit das System ordnungsgemäß funktioniert und sich die Benutzer zurechtfinden. Auch diesem Aspekt widmen wir einen Abschnitt.

Dateisystem-Konzepte unter Unix

Das Unix-Dateisystem ist eine Baumstruktur mit einer einzigen Wurzel (mit „/“ bezeichnet). Dies unterscheidet Unix zum Beispiel von Windows, wo es das Konzept von gleichberechtigten Laufwerken gibt, von denen jedes ein eigenes Wurzelverzeichnis besitzt. Unterhalb des Wurzelverzeichnisses gibt es (geschachtelte) Unterverzeichnisse und möglicherweise Dateien. Verzeichnisse können weitere Verzeichnisse oder Dateien enthalten, wobei Dateien dann die Nutzdaten enthalten und keine Unterobjekte mehr haben können (d.h. sie sind die Blätter des Dateibaums). Um ein Verzeichnis oder eine Datei auf einem System zu identifizieren, verwendet man Pfade, die die Verzeichnisse von oben nach unten (im Baum) auflisten, wobei die einzelnen Pfadkomponenten durch einen Schrägstrich „/“ getrennt werden (unter Windows ist dies der Backslash „\“). Beginnt der Pfad mit einem „/“ (z.B. `/bin/ls`) dann handelt es sich um einen absoluten Pfad, dessen Auswertung bei der Wurzel beginnen muss, andernfalls um einen relativen Pfad, der von einem aktuellen Standpunkt (Arbeitsverzeichnis) ausgehend ausgewertet wird. Für relative Pfade gibt es die speziellen Bezeichner `..` für das Elternverzeichnis und `.` für das aktuelle Verzeichnis (dies ist unter Windows analog).

Jedes Dateisystemobjekt (Datei oder Verzeichnis) hat unter Unix drei Zeitstempel zugeordnet, die vom Systemkern bei entsprechenden Dateioperationen angepasst werden. Nebenbei bemerkt wird in allen Unix-Systemen die Zeit intern als 32bit Wert gespeichert, der als die Anzahl der vergangenen Sekunden seit dem 01.01.1970 um 0:00 interpretiert wird. Die drei Zeitstempel in den Dateisystemen sind

- die *atime* (*access time*), welche den letzten Lese- oder Schreibzugriff darstellt.
- die *mtime* (*modification time*), welche die letzte Änderung (Schreiboperation) darstellt
- die *ctime* (*change time*[6]), welche die letzte Änderung an Datei-Metadaten (Berechtigungen, Eigentümer, ...) darstellt

Weiterhin hat jede Datei einen eindeutigen Eigentümer (identifiziert durch seine

6 Vorsicht: *ctime* steht nicht etwa für *creation time* – die gibt es unter Unix leider nicht!

numerische UID) und eine Gruppenzugehörigkeit (durch die numerische GID). Diese beiden Informationen werden bei der Auswertung der Zugriffsrechte verwendet, die ebenfalls zu den Metadaten der Dateisystemobjekte gehören. Die Zugriffsrechte werden als 3 Gruppen zu je 3 Bits dargestellt (wobei es noch ein paar weitere Bits gibt, die aber seltener benutzt werden). Die drei Gruppen sind:

- Rechte des Eigentümers (meist mit *u* für *user* abgekürzt)
- Rechte der Gruppe, der die Datei / das Verzeichnis gehört (meist mit *g* für *group* abgekürzt)
- Rechte aller anderen Benutzer (meist mit *o* für *other* abgekürzt)

Jede dieser drei Bitgruppen enthält die folgenden 3 Bits für:

- *r (read)* für Lesezugriffe
- *w (write)* für Schreibzugriffe
- *x (execute)* für Ausführen

Für Verzeichnisse ist die Bedeutung der Berechtigungen leicht unterschiedlich: lesen bedeutet das auflisten der Unterobjekte, schreiben das hinzufügen oder entfernen von Unterobjekten und das x-Bit erlaubt oder verbietet das wechseln in das Verzeichnis (mit *cd* oder *chdir(2)*).

Dargestellt wird dies oft als String aus den obigen Zeichen (der entsprechende Buchstabe bedeutet dabei, dass das jeweilige Recht gewährt wird, ein „-“ bedeutet das Gegenteil. Verwendet man z.B. das Kommando *ls(1)*, um Berechtigungen anzuzeigen, wird den Berechtigungen noch ein weiterer Buchstabe für den jeweiligen Typ der Datei vorangestellt:

- - für „normale“ Dateien
- *d (directory)* für Verzeichnisse
- *l (link)* für symbolische Verweise
- *s (socket)* für *Unix Domain Sockets*
- ...

(siehe *stat(2)* für genauere Informationen)

Hier ein Beispiel:

```
$ ls -l /home/datsche
total 1769208
drwxr-xr-x    7 7d           institut       4096 Dec  6  2006 7d/
drwx------    3 root         root           4096 Dec  6  2006 HH-restore/
...
```

Den Inhalt des Unterverzeichnisses `7d` darf also jeder lesen und hineinwechseln (per *chdir*

(2)), aber darin schreiben darf nur der Eigentümer (der heißt auch `7d` laut der dritten Spalte). Im Verzeichnis `HH-restore` hat hingegen nur der Systemadministrator (*root*) Lese- und Schreibrechte, alle anderen User (auch die, die zur Gruppe *root* gehören), dürfen in diesem Verzeichnis weder lesen noch schreiben.

Die ganze Wahrheit über die Berechtigungen ist, dass es in jeder der drei Gruppen noch ein weiteres Bit gibt. Dieses macht in der Regel nur zusammen mit dem x-Bit bei ausführbaren Dateien Sinn und wenn es gesetzt ist, wird das *x*-Bit in der *ls(1)*-Ausgabe durch ein *s* ersetzt:

```
-rwsr-xr-x  1 root root 98488 Mar 20  2006 /usr/bin/sudo*
```

Ist dieses Bit in der Bitgruppe für den Eigentümer gesetzt, spricht man vom *setuid*-Bit, bei der Gruppen-Gruppe vom *setgid*-Bit und bei den Berechtigungen für sonstige User vom *sticky bit*. Ist dieses Bit gesetzt obwohl das zugehörige *x*-Bit nicht gesetzt ist, macht dies keinen Sinn und wird in der *ls*-Ausgabe durch ein „*S*“ (groß-*s*) gekennzeichnet. Das *setuid*-Bit bewirkt, dass das Programm nicht unter der Kennung des Aufrufers, sondern unter der Kennung des Dateieigentümers ausgeführt wird (für das *setgid*-Bit und die Gruppenrechte bei Ausführung gilt entsprechendes). Dies wird oft benutzt, um Benutzern eingeschränkt Administrator-Rechte zu geben, indem man ihnen ausführbare Programme zur Verfügung stellt, die *root* gehören und das *setuid*-Bit gesetzt haben und somit unter *root*-Kennung laufen – allerdings wird im allgemeinen davon abgeraten, davon regen Gebrauch zu machen (um solche Versuche zu kanalisieren und steuerbar zu machen, dient z.B. oben erwähntes *sudo(1)* Kommando). Die Bedeutung des *sticky bit* wird hier aus Platzgründen nicht erklärt und kann unter http://en.wikipedia.org/wiki/Sticky_bit nachgelesen werden.

Arten von Dateisystemen und Mounts

Wie bereits erwähnt, existiert auf einem Unix-System nur ein Wurzel-Verzeichnis (`/`), unterhalb dessen sich alle zugänglichen Dateien wiederfinden. Es ist zwar sehr wohl möglich, mehrere Festplatten (oder mehrere Partitionen auf einer Festplatte) zu haben, aber diese werden beim Start des Systems auf Unterverzeichnisse des Dateibaums abgebildet (man spricht hier vom montieren, bzw. englisch *mount*). Beim Start eines Unix-Kernels muss also lediglich feststehen, wo die `/`-Partition zu finden ist; diese muss dann allerdings alle notwendigen Kommandos (z.B. *mount(8)*) und Informationen (z.B. *fstab(5)*) enthalten, damit der Dateibaum komplett zusammengesetzt werden kann. Die restlichen Partitionen (oder auch Netz-Laufwerke) werden dann beim Hochfahren des Systems auf beliebige Unterverzeichnisse der Wurzel-Partition (`/` oder *root*-Partition) eingehängt (umgangssprachlich „gemountet“). Der Aufbau des Dateisystems kann mit dem *mount(8)*-Kommando angezeigt werden – Veränderungen darf allerdings nur *root* vornehmen.

Es ist durchaus möglich und üblich, dass nicht alle Partitionen auf die gleiche Art und Weise formatiert sind. Für lokale Festplatten-Partitionen gibt es einerseits die

traditionellen Unix-Dateisysteme (UFS unter Solaris, ext2 unter Linux) und in jüngerer Vergangenheit auch *journaling* Filesysteme (ext2, XFS, ReiserFS ...). Letztere sollen durch atomare Schreibaktionen verhindern, dass Inkonsistenzen entstehen, damit beim Reboot des Systems nach einem Absturz keine aufwändigen Filesystem-Überprüfungen (*fsck(8)*) mehr nötig sind. Auf CD-Roms hingegen kommt meist das Standard-Filesystem ISO9660 (oft mit einer Erweiterung wie Rockridge für Unix oder Jolliet für Windows) zum Einsatz, USB-Sticks oder Disketten sind meistens mit einem einfachen FAT-Dateisystem formatiert und wenn sich auf dem gleichen Rechner auch noch ein Windows-System befindet, können (zumindest unter Linux) auch NTFS-Partitionen im Dateibaum eingehängt sein. Darüber hinaus gibt es noch spezielle Typen von Dateisystemen, die ihren Inhalt beispielsweise verschlüsseln oder in einer Datei eines anderen Dateisystems speichern (*loopback filesystem*).

Andererseits müssen die Dateien eines Dateisystems nicht auf lokalen Geräten gespeichert sein, sondern können auf Netzwerk-Servern in beliebiger Entfernung liegen. Man spricht dann auch bei den verwendeten Netzwerk-Protokollen von Dateisystemen. Ein älterer Vertreter dieser Gattung ist das *Network Filesystem (NFS)* von Sun welches auf SUN-RPC (*Remote Procedure Calls*) basiert und sich semantisch besonders gut für Unix-Systeme eignet, da es die Semantik lokaler Dateisysteme auf ein Netzwerk-Protokoll abbildet. NFS-Dateisysteme werden pro Client-Rechner einmalig für alle User eingehängt; die Autorisierung der Zugriffe erfolgt dabei über die übliche Berechtigungs-Prüfung (s.o.), d.h. in NFS hat jedes Dateisystem-Objekt einen Eigentümer, eine Gruppe und die Bits für die Zugriffsrechte. Die Autorisierung erfolgt aber auf dem Client-System, d.h. es muss ein Vertrauensverhältnis zwischen Client und Server geben, da der Server die Schreib-/Leseoperationen für alle bekannten Clients ausführt und sich darauf verlässt, dass jeder Client lokal überprüft, ob der jeweilige Benutzer auch dazu berechtigt ist. Dies ist bei dem aus der Windows-Welt stammenden SMB (*Server Message Block*) anders – hier muss schon beim mounten eine Authentifizierung vorgelegt werden und alle folgenden Lese- und Schreiboperationen erfolgen dann mit den Rechten des authentifizierten Benutzers. Während NFS zustandslos ist (und somit resistent gegenüber Neustarts des Servers), muss bei SMB eine Session aufrecht erhalten werden.

Zum Thema Automatisierung sollte hier kurz erwähnt werden, dass auf vielen Unix-Systemen ein sog. Automounter läuft, der erst bei Zugriff auf ein Dateisystem dies auch tatsächlich einhängt. Der Vorteil ist z.B. bei Wechselmedien, dass diese nicht erst gemountet werden müssen (OK, dafür gibt es heutzutage bessere Lösungen) und bei Netz-Laufwerken, dass der Boot-Prozess nicht dadurch verhindert wird, dass ein NFS-Server ausgefallen ist.

Dateisystem-Struktur

Im Laufe der Jahre haben sich einige Konventionen für das inhaltliche Layout von Unix-Dateisystemen entwickelt, an die sich die meisten Hersteller (bzw. Linux-Distributoren) halten. Es gibt kleinere Unterschiede zwischen den Systemen, Distributionen und Versionen, aber einige Dinge sind überall gleich. Die folgende Liste zeigt die wichtigsten Verzeichnisse, Dateien und ihre Bedeutungen (es gibt aber natürlich noch viel mehr).

Pfad	*Bedeutung*	*Erklärung*
`/bin`	*binary*	Enthält die minimal notwendigen Programme, um mit dem System zu arbeiten.
`/sbin`	*system binary*	Enthält grundlegende Administrationskommandos, die zum Hochfahren des Systems benötigt werden.
`/usr`	*unix system resources*	Enthält Applikationen und weiterführende Kommandos, sowie das X Window System und ist oftmals auf einer eigenen Partition zu finden. Unter */usr* (sprich *user*) findet sich wieder eine Hierarchie mit *bin*, *lib*, *etc* ...
`/usr/local`		Während Änderungen an */usr* normalerweise nicht empfehlenswert sind (weil diese beim nächsten System-Upgrade verloren gehen könnten), können Applikationen, die nicht zum System gehören hier installiert werden. Unter */usr/local* findet sich wieder eine Hierarchie mit *bin*, *lib*, *etc* ...
`/lib`	*library*	/lib enthält die statischen oder dynamischen Bibliotheken des Betriebssystems (z.B. libc). Unter /usr/lib liegen weitere Bibliotheken und unter /usr/local/lib werden Bibliotheken von lokale installierten Applikationen abgelegt.
`/etc`	*et cetera*	Hier liegen (wie der Name nicht gerade vermuten lässt) die Konfigurations-Dateien des Systems, unter anderem die Passwort-Datei (/etc/passwd), die Start-Scripte für den Boot-Vorgang (/etc/init.d) und die X11-Konfiguation (/etc/X11).
`/tmp` `/var/tmp`	*temporary*	Enthält temporäre Dateien aller Benutzer und ist für jeden schreibbar. Da dieses Verzeichnis (z.B. unter Solaris) allerdings oft im Hauptspeicher liegt, sollten hier nicht allzu viele Daten abgelegt werden. */var/tmp* ist für größere Dateien besser.
`/proc`	*process*	Virtuelles Dateisystem, welches die aktuell laufenden Prozesse des Systems wiedergibt (siehe unter Prozessen). Unter Linux finden sich hier auch Informationen über den Kernel und die Hardware.

Pfad	*Bedeutung*	*Erklärung*
`/var`	*variable*	Hier liegen Arbeitsdaten des Systems, die nicht direkt Nutzerdaten sind (diese liegen unter */home*). Beispiele sind Spool-Bereiche von Druck- und Mailservern sowie Logfiles.
`/home`		Heimatverzeichnisse der Benutzer.

Die Unix-Prozess-Hierarchie

Aufbau eines Prozesses

Unter einem Prozess versteht man unter Unix ein in Ausführung befindliches Programm. Ein Programm ist eine ausführbare Datei, die im Dateisystem liegt und (normalerweise) ausführbaren Code in dem Binärformat des Systems enthält. Ein neuer Prozess wird durch den *fork(2)* Systemaufruf erzeugt wodurch allerdings nur eine Kopie des aufrufenden Prozesses erzeugt wird. Ein ausführbares Programm wird dann (meist direkt nach dem *fork(2)*) mittels des *exec(3)*-Systemaufrufs gestartet. Durch den Aufruf von *fork (2)* entsteht ein Prozess mit einer eindeutigen Prozess-ID (PID), einer Eltern Prozess-ID (PPID; das ist die PID des Prozesses, der den erzeugenden *fork(2)*-Aufruf getätigt hat), und einer effektiven Benutzer- und Gruppen-ID (EUID und EGID), welche u.a. verwendet werden, um Berechtigungsprüfungen bei Dateisystemzugriffen durchzuführen. Benutzer und Gruppe sind zunächst identisch mit Bennutzer und Gruppe des Eltern-Prozesses und können auch nur von Prozessen unter *root*-Kennung geändert werden. Es kann mehrere Prozesse in einem System geben, die dasselbe Programm ausführen; der Programmcode selbst wird dabei nur einmal in den Speicher geladen (dieser ist ohnehin nicht änderbar) und nur die Register der CPU (z.B. der Programmzähler), der Stack und das Datensegment unterscheiden sich bei den beiden Prozessen.

Prozess-Hierarchie

Durch die Eltern-Kind-Relation entsteht eine Baumstruktur. Ähnlich wie beim Start des Systems, wo zunächst das Wurzel-Dateisystem eingehängt wird, welches dann die notwendigen Informationen über die weiteren Dateisysteme enthält, wird vom Kernel auch der erste Prozess erzeugt, der für die Erzeugung aller weiteren Prozesse verantwortlich ist. Dieser sog. *init*-Prozess (siehe *init(8)*) trägt immer die Prozess-ID 1 und ist für den weiteren Verlauf des Boot-Vorgangs verantwortlich. Er startet z.B. diverse System- und Netzwerk-Dienste, die *getty(8)*-Prozesse, die auf den Konsolen auf User-Logins warten sowie möglicherweise einen grafischen Display-Manager (*xdm*, *kdm*, *gdm*), der das Anmelden zu einer grafischen Benutzer-Sitzung erlaubt.

Neben der Benutzer- und Gruppen-Zugehörigkeit und dem aktuellen Arbeitsverzeichnis erbt ein Kind-Prozess noch eine weitere Eigenschaft von seinem Eltern-Prozess: die Umgebungs-Variable (Environment-Variable). Dies ist eine Sammlung von Schlüssel-/ Wert-Paaren, die oftmals zur Steuerung des Programm-Verhaltens benutze werden (es gibt in den meisten Manualseiten eine Sektion *ENVIRONMENT,* welche die

Umgebungsvariable erklärt, die ein Programm auswertet). Da die Umgebungsvariable vom Eltern- zum Kind-Prozess vererbt werden, kann der Eltern-Prozess (z.B. eine Shell) einfach Variable in der eigenen Umgebung setzen und dann einen Kind-Prozess erzeugen, der diese erbt (es gibt aber auch Derivate von *exec(3),* denen man explizit ein Environment übergeben kann). Wichtig ist, dass Kind-Prozesse niemals die Umgebung ihrer Eltern oder deren aktuelles Arbeitsverzeichnis ändern können. Alle Änderungen an Umgebungsvariablen gehen mit Beendigung des (Kind-)Prozesses verloren.

Eine weitere (geerbte) Eigenschaft eines Prozesses sind seine geöffneten Dateien (auch Netzwerk-Verbindungen, Pipes und Geräte gelten als Dateien). Selbst ein Prozess, der keine Dateien geöffnet hat, hat normalerweise drei *filehandles*:

- *stdin* mit der Dateinummer 0, welches normalerweise mit der Eingabe des kontrollierenden Terminals verbunden ist (also der Tastatur)
- *stdout* mit der Dateinummer 1, welches normalerweise Zeichen auf das kontrollierende Terminal ausgibt
- *stderr* – die Standard-Fehlerausgabe zum Melden von Fehlern an den Benutzer – welches ebenfalls am kontrollierenden Terminal hängt.

Ausgabe (*stdout*) und Fehler (*stderr*) landen zwar beide auf dem Terminal, die Trennung macht jedoch Sinn, weil man die beiden Kanäle somit an unterschiedliche Ziele weiterleiten kann (siehe unten).

Geöffnete Dateien werden auch vom Eltern-Prozess an seine Kinder vererbt. Dies erklärt, warum ein Kommando, das auf einer Shell eingegeben wird, seine Ausgabe auf das Terminal umleitet, auf dem auch die Shell läuft (es sein denn, der Benutzer ändert dieses Verhalten explizit durch Umleitungen, siehe unten).

Die aktuell laufenden Prozesse können mittels *ps(1)* aufgelistet werden:

```
$ ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 May31 ?        00:00:01 init [2]
root         2     1  0 May31 ?        00:00:00 [migration/0]
root         3     1  0 May31 ?        00:00:00 [ksoftirqd/0]
root         4     1  0 May31 ?        00:00:00 [migration/1]
root         5     1  0 May31 ?        00:00:00 [ksoftirqd/1]
root         6     1  0 May31 ?        00:00:07 [events/0]
root         7     1  0 May31 ?        00:00:00 [events/1]
root         8     1  0 May31 ?        00:00:00 [khelper]
root         9     1  0 May31 ?        00:00:00 [kthread]
root        12     9  0 May31 ?        00:00:03 [kblockd/0]
root        13     9  0 May31 ?        00:00:00 [kblockd/1]
root        14     9  0 May31 ?        00:00:00 [kacpid]
root        96     9  0 May31 ?        00:00:01 [khubd]
root       180     9  0 May31 ?        00:00:00 [pdflush]
root       181     9  0 May31 ?        00:00:04 [pdflush]
root       182     1  0 May31 ?        00:00:20 [kswapd0]
root       183     9  0 May31 ?        00:00:00 [aio/0]
root       184     9  0 May31 ?        00:00:00 [aio/1]
root       185     1  0 May31 ?        00:00:00 [cifsoplockd]
root       187     9  0 May31 ?        00:00:00 [xfslogd/0]
root       186     1  0 May31 ?        00:00:00 [cifsdnotifyd]
root       188     9  0 May31 ?        00:00:00 [xfslogd/1]
root       189     9  0 May31 ?        00:00:00 [xfsdatad/0]
root       190     9  0 May31 ?        00:00:00 [xfsdatad/1]
root       779     9  0 May31 ?        00:00:00 [kseriod]
```

```
root       857     9  0 May31 ?        00:00:00 [ata/0]
root       858     9  0 May31 ?        00:00:00 [ata/1]
root       872     9  0 May31 ?        00:00:00 [kpsmoused]
...
```

Auf manchen Linux-Installationen gibt es jedoch auch ein Programm namens *pstree(1)* mit dem sich die Prozess-Hierarchie sehr schön veranschaulichen lässt:

```
init─┬─S20xprint─┬─S20xprint───Xprt
     │           └─S20xprint
     ├─acpid
     ├─atd
     ├─3*[automount]
     ├─bonobo-activati
     ├─cifsdnotifyd
     ├─cifsoplockd
     ├─cron
     ├─dbus-daemon-1
     ├─dcopserver
     ├─dictd
     ├─dirmngr
     ├─events/0
     ├─events/1
     ├─evolution-data-
     ├─exim
     ├─3*[gconfd-2]
     ├─5*[getty]
     ├─inetd
     ├─kded
     ├─kdeinit─┬─artsd
     │         ├─evolution-alarm
     │         ├─firefox-bin
     │         ├─gnome-terminal─┬─bash───ssh
     │         │                ├─bash───pstree
     │         │                └─gnome-pty-helpe
     │         ├─kio_file
     │         ├─klauncher
     │         ├─kwin
     │         ├─mozilla-thunder───run-mozilla.sh───
     │         ├─skype
     │         └─soffice.bin
     ├─kdesktop
     ├─kdm─┬─XFree86
     │     └─kdm───csh───x-session-manag─┬─k+
     │                                   └─ssh-agent
     ├─khelper
     ├─khotkeys
     ├─kicker
     ├─kirqd
     ├─2*[kjournald]
     ├─klipper
     ├─klogd
     ├─knotify
     ├─konqueror
     ├─korgac
     ├─ksmserver
     ├─ksoftirqd/0
     ├─ksoftirqd/1
     ├─kswapd0
     ├─kthread─┬─aio/0
     │         ├─aio/1
     │         ├─ata/0
     │         ├─ata/1
     │         ├─kacpid
```

```
      │           ├─kblockd/0
      │           ├─kblockd/1
      │           ├─khubd
      │           ├─kpsmoused
      │           ├─kseriod
      │           ├─2*[pdflush]
      │           ├─rpciod/0
      │           ├─rpciod/1
      │           ├─xfsdatad/0
      │           ├─xfsdatad/1
      │           ├─xfslogd/0
      │           └─xfslogd/1
...
```

Prozesse können sich gegenseitig Signale mittels des Systemaufrufs *kill(2)* senden. Dazu wird geprüft, ob der Aufrufer die gleiche effektive User-ID hat wie der Empfänger des Signals (es sei denn, der Absender ist *root*). Der Name „kill“ lässt darauf schließen, dass die Operation meist tödlich für den Empfänger endet, wobei dies eher historische Gründe hat: früher gab es wenig sinnvolle Dinge, die man in einem Signal-Behandler tun konnte, da die Standard-Bibliotheken meistens nicht reentrant waren, d.h. wenn ein Signal eine Bibliotheksfunktion (z.B. *malloc(3)*) unterbricht und der Signal-Behandler die gleiche Funktion noch einmal aufruft, findet dieser möglicherweise einen inkonsistenten Stand globaler Daten vor. Bei modernen Unix-Systemen stellt dies aber kein Problem mehr dar (weil z.B. die libc reentrant ist), so dass *kill(2)* beliebigen Programmcode im Empfängerprozess ausführen kann (z.B. bei Samba vor Version 2.2 das hoch- oder herunterstellen des Loglevels).

Ein Prozess existiert, bis er sich entweder durch einen Aufruf von *exit(3)* beendet oder er durch ein nicht behandeltes Signal (oder ein nicht behandelbares Signal wie *SIGKILL*) beendet wird. Im ersteren Fall gibt der Prozess einen Status an seinen Erzeuger zurück, der diesen Status mit *wait(2)* abfragen kann (genauer gesagt: abfragen muss, denn sonst bleibt der beendete Kind-Prozess als sog. Zombie in der Prozessliste). Will man einen Prozess starten, auf den man nicht warten will (z.B. weil er ewig laufen soll), muss man als Entwickler dafür sorgen, dass der erzeugende Eltern-Prozess sich beendet. Dies führt dazu, dass der Kind-Prozess als neuen Eltern-Prozess den *init*-Prozess zugewiesen bekommt und dieser ruft regelmäßig *wait(2)* auf, um terminierte Kindprozesse abzuräumen. Wenn man mit der Shell arbeitet, ist oft der Status von beendeten Programmen von Interesse, da dies oftmals Auskunft über Erfolg oder Misserfolg gibt. In den gängigen Shells steht der Rückgabewert des letzten Programmaufrufs meistens in der Shell-Variable `$?`.

Das procfs

Sowohl unter Linux als auch unter anderen Unix-Systemen gibt es ein spezielles, virtuelles Dateisystem, welches unter `/proc` eingehängt wird und welches zu jedem Zeitpunkt den Status aller laufenden Prozesse des Systems wiedergibt (offene Dateien, Hauptspeicher-Allokation, Speicher-Inhalt ...). Die unterschiedlichen Implementationen (von Linux, Solaris, BSD ...) unterscheiden sich im Details allerdings drastisch. So dient das Solaris-proc ausschließlich dazu, Informationen über Prozesse zugreifbar zu machen

bzw. diese zu steuern (Signale können hier z.B. über I/O-Operationen unter */proc* gesendet werden), während das Linux-proc inzwischen ein Wildwuchs aus diversen Prozess-, Betriebssystem- und Hardware-Informationen ist. `/proc` ist somit keine standardisierte Schnittstelle, d.h. die Programme, die darauf zugreifen (z.B. das *ps(1)*-Kommando), sind Betriebssystem-abhängig.

Die Last eines Unix-Systems

Der Scheduler im Systemkern läuft am Ende jeder Zeitscheibe, um den nächsten rechenbereiten Prozess zur Ausführung zu bringen. Dabei führt er Buch über die Anzahl der Prozesse, die zu diesem Zeitpunkt rechenbereit wären. Dies sind Prozesse, die nicht angehalten wurden, oder bereits beendet sind (Zombies) oder auf blockierende I/O von der Festplatte oder dem Netzwerk warten. Die Anzahl dieser rechenbereiten Prozesse bekommt man gewöhnlich als Mittel über die letzten 5, 10 und 15 Minuten zu sehen, wenn man das folgende Kommando absetzt:

```
$ uptime
 16:22:43 up 48 days,  1:17,  5 users,  load average: 0.00, 0.08, 0.04
```

Neben der Uhrzeit und der *uptime* (also der Dauer, die das System schon durchgehend läuft) und der Anzahl eingeloggter Benutzer, sieht man die drei Last-Werte, die folgendermaßen zu interpretieren sind: auf einem System mit einer CPU bedeutet eine Last von 1.0, dass die CPU kontinuierlich beschäftigt ist. Ist die Last dauerhaft über 1.0 (bzw. der Anzahl der CPUs), sollte man entweder Applikationen optimieren (oder auf andere Maschinen verteilen) oder in CPUs investieren.

Es sei aber davor gewarnt, die Last als alleinigen Indikator für das Wohlbefinden eines Systems zu verwenden. Die Last kann sich sehr wohl im moderaten Bereich befinden, aber das System steht trotzdem beinahe still, weil beispielsweise kontinuierliche Festplatten-I/O den Systembus lahmlegt. Andererseits wurden auch schon Server gesichtet, die Last-Werte in den Hunderten hatten, aber trotzdem noch akzeptable Antwortzeiten lieferten, weil jeder Prozess nur ein klein wenig Rechenleistung braucht, z.B. um einen Web-Request zu beantworten. Das richtige interpretieren von Kennziffern der Systemperformance ist eine Kunst, die jahrelange Erfahrung sowohl mit der eingesetzten Hardware als auch dem konkreten Unix-Derivat erfordert.

Interaktion mit der Unix-Shell

Der Begriff *shell* stammt aus der ursprünglichen Vorstellung der Architektur eines Betriebssystems, welches aus einem Kern (*kernel*) und einer äußeren Schale (*shell*) besteht. Die Shell ist dabei die Schnittstelle zur Außenwelt mit der der Benutzer interagiert. Technisch gesehen meint man mit dem Begriff Shell einen Kommandozeilen-Interpreter, der Kommandos entgegen nimmt, diese verarbeitet und das Ergebnis wieder dem Benutzer darstellt. Der Begriff „shell“ wird auch unter grafischen Systemen (wie Windows) für die Benutzerschnittstelle verwendet, unter Unix ist hingegen immer ein textbasierter Kommando-Interpreter gemeint. Auch die Windows-Kommandozeile

(CMD) könnte als einfache Shell bezeichnet werden und die jüngst veröffentlichte Powershell von Microsoft beruft sich auch auf diese lange Historie.

Im Laufe der Entwicklung des Unix-Systems wurden unterschiedliche Shells entwickelt, die teilweise kompatibel sind, teilweise aber auch nicht. Die wichtigsten Familien von Shells sind die Bourne-Shell-Varianten auf der einen Seite und die C-Shell-Varianten auf der anderen Seite[7]. Die Bourne-Shell (benannt nach ihrem Autor Stephen Bourne) hat sich als Standard-Shell für Shell-Scripting durchgesetzt, da sie (oder kompatible Shells) auf quasi jedem Unix-System zur Verfügung steht und auch vom POSIX[8]-Standard gefordert wird. Die ursprüngliche Bourne-Shell (`/bin/sh`) ist jedoch nicht besonders angenehm für den interaktiven Gebrauch, so dass die meisten Administratoren und Benutzer andere Shells für den alltäglichen Gebrauch eingerichtet haben. Im Linux-Umfeld ist die Bash (*Bourne again shell*) als wesentlich komfortablerer Ersatz der (nicht quelloffenen) Original-Shell anzutreffen und wird in diesem Kurs verwendet, da sie auf jedem Linux-System zur Verfügung steht. Alternativen zur Bash wären die Korn-Shell (*ksh(1)*) oder die Z-Shell (*zsh(1)*), die beide in die Reihe der Bourne-Shell-kompatiblen Shells fallen, aber eigene Erweiterungen mitbringen.

Die andere erwähnenswerte Familie ist die der C-Shells die ihren Namen von der Programmiersprache C erhielten, welche als Vorbild für die Syntax der Kommandosprache diente. C-Shells haben immer noch einen Kreis von Anhängern, sind jedoch weniger stark verbreitet.

Den gängigen Unix-Shells ist eines gemeinsam: es gibt einen kleinen Sprachschatz von Shell-internen Schlüsselwörtern (für Schleifen, Fallunterscheidungen, Variablen ...) während die restlichen Kommandos eigene Programme sind, die von der aufrufenden Shell unabhängig sind. Selbst wenn man also als Anwender nur eine Shell kennt (z.B. die *bash*), so wird man trotzdem mit wenig Umgewöhnung auch mit einer andern Shell (z.B. der *tcsh*) zumindest die wichtigsten Aufgaben erledigen können, da die Kommandos beispielsweise zum Kopieren von Dateien oder Auslesen der Routing-Tabelle immer die selben sind.

Erste Schritte

Hat man sich an einem Unix-System im Text-Modus angemeldet oder unter X11 ein Terminal gestartet, wird die sog. Login-Shell des Benutzers gestartet (diese ist in `/etc/passwd` für jeden User definiert und kann meistens durch das Kommando *chsh(1)* geändert werden). Eine interaktive Shell meldet sich für gewöhnlich mit einer Eingabe-Aufforderung (Prompt) zum Dienst. Dies umfasst mindestens ein Sonderzeichen (`$`, `#` oder `%`), meistens aber auch den Namen des eingeloggten Benutzers und des aktuellen Rechners (man kann schließlich auf mehreren Rechnern parallel arbeiten) und manchmal sogar das aktuelle Verzeichnis oder Datum und Uhrzeit o.ä. Wie man den Prompt (die Variable `$PS1`) den eigenen Wünschen anpasst, entnimmt man der Manual-Seite der Shell (s.u.).

7 http://en.wikipedia.org/wiki/Unix_shell listet noch weitere, „exotische“ Shells auf.
8 Portable Operating System Interface

Zunächst zu einigen einfachen Kommandos. Jeder Unix-Prozess kennt immer ein aktuelles Arbeitsverzeichnis – relative Pfade bei Dateioperationen werden immer relativ zu diesem Verzeichnis ausgewertet. Wenn die Shell das aktuelle Verzeichnis nicht sowieso im Prompt ausgibt, kann man dies mit dem Kommando *pwd(1)* (*print working directory*) erfragen:

```
$ pwd
/home/datsche/boesswet
```

Wenn man sich nicht bereits vorher in ein anderes Verzeichnis begeben hat, ist dies normalerweise das Heimatverzeichnis (welches auch für jeden Benutzer in `/etc/passwd` festgelegt wird). Das aktuelle Verzeichnis wechselt man mit *cd* (*change directory*):

```
$ cd /bin
```

Und wenn man sich dort umsehen (also den Inhalt des Verzeichnisses auflisten) möchte, kann man dies mit *ls* (*list*) tun:

```
$ ls
arch*           false*          mbchk*          rm*         touch*
ash@            fdflush*        mkbimage*       rmdir*      true*
bash*           fgconsole*      mkdir*          run-parts*  umount*
cat*            fgrep*          mknod*          rzsh@       uname*
chgrp*          fuser*          mktemp*         sash*       uncompress*
chmod*          gnutar*         more*           scpio*      vdir*
chown*          grep*           mount*          sed*        vi*
cp*             gunzip*         mountpoint*     setpci@     ypdomainname@
cpio*           gzexe*          mt-gnu*         setserial*  zcat*
csh@            gzip*           mt-star*        sh@         zcmp*
dash*           hostname*       mv*             sleep*      zdiff*
date*           ip*             nc*             spax*       zegrep*
dd*             kernelversion@  netcat@         star*       zfgrep*
df*             kill*           netstat*        star_sym*   zforce*
dir*            ksh@            nisdomainname@  stty*       zgrep*
dmesg*          ln*             pdksh*          su*         zless*
dnsdomainname*  loadkeys*       pidof@          suntar*     zmore*
domainname*     login*          ping*           sync*       znew*
echo*           ls*             ps*             tar*        zsh@
ed*             lsmod*          pwd*            tartest*    zsh4*
egrep*          lsmod.modutils@ rbash@          tcsh*
elvis-tiny*     lspci@          readlink*       tempfile*
```

Kommandos (wie das cd oben) können aber auch Argumente übergeben bekommen, in dem man diese einfach mit Leerzeichen getrennt hinter den Kommandonamen stellt:

```
$ ls /bin
...
```

Dieses Kommando liefert die gleiche Ausgabe wie das obige, funktioniert wegen dem absoluten Pfad aber unabhängig vom aktuellen Arbeitsverzeichnis. Die meisten Unix-

Kommandos bekommen Parameter übergeben, wobei sich diese ungefähr in zwei Gruppen unterteilen lassen: Optionen die aus einem „-“ und einem Buchstaben (bzw. manchmal aus „--“ und mehreren Zeichen) und eventuell einem Argument bestehen und Wörter, die meistens Dateinamen, manchmal aber auch Rechnernamen oder sonstige Daten sein können. Datei- und Verzeichnisnamen können in den allermeisten Fällen mehrere angegeben werden:

```
$ ls -l --color=none /bin /etc
...
```

Um ins Heimatverzeichnis zurückzukehren, kann man meistens einfach *cd* (ohne Parameter) verwenden. Das Heimatverzeichnis des ausführenden Users wird übrigens auch der Einfachheit halber mit `~` (sprich Tilde) abgekürzt, d.h. die Kommandos

```
$ cd
```

und

```
$ cd ~
```

sind äquivalent (wobei letzteres niemand schreiben würde). Mit der `~`-Notation kann man aber auch den Weg zu beliebigen anderen Heimatverzeichnissen abkürzen, indem man einfach den Benutzernamen an die Tilde anhängt (z.B. `~donaldduck`).

In der Bash (ebenso wie in anderen modernen Shells) gibt es Hilfsmittel, die das viele Tippen erleichtern sollen. Beispielsweise kann mit der Tabulator-Taste ein angefangenes Wort vervollständigt werden und zwar wie folgt:

- Ist das Wort das erste auf der Zeile, wird im Suchpfad (`$PATH`) nach Kommandos gesucht, die mit der bereits eingegebenen Buchstabensequenz beginnen.
- Bei allen weiteren Worten einer Zeile (ein Wort ist durch Leerzeichen von anderen Wörtern getrennt) wird versucht, diesen als Pfad (absolut oder relativ) zu interpretieren und entsprechend zu erweitern.

In beiden Fällen werden die Wörter ergänzt, so lange dies eindeutig ist. Ist dies nicht der Fall, kann durch wiederholtes drücken der Tab-Taste eine Liste von Alternativen ausgegeben werden. Ein Beispiel:

```
$ fire<tab>
```

(wobei `<tab>` hier als Druck auf die Tabulator-Taste gemeint ist) dürfte auf den meisten aktuellen Linux-Systemen zu `firefox` erweitert werden, während

```
$ ls /usr/s<tab><tab>
```

die Alternativen `sbin`, `share` und `src` anbietet, weil dies die drei Unterverzeichnisse unter `/usr` sind, die mit „s“ beginnen. Der erste Druck auf die Tab-Taste verändert hier noch

nichts, da eine eindeutige Erweiterung nicht möglich ist, erst der zweite Anschlag listet die Alternativen auf.

Ebenso nützlich sind die Cursortasten: mit „auf“ und „ab“ kann man durch die Historie von eingegebenen Kommandos steppen (auch mal probieren: *history*) und mit rechts/links kann man sich in einer Zeile bewegen und Korrekturen vornehmen.

Tip: am besten ausprobieren und auch mal in *bash(1)* nachlesen (es gibt noch mehr Vervollständigungs-Möglichkeiten) – mit den richtigen Tastenkombinationen kann man viel Arbeit sparen.

Hier noch eine Tabelle mit den wichtigsten Kommandos mit denen man die eigene Umgebung erforschen kann:

Kommando	***Bedeutung***
whoami	Gibt den Namen des aktuell angemeldeten Benutzers aus.
who	Gibt die Liste aller am System angemeldeten Benutzer aus.
id	Gibt Namen, Gruppenzugehörigkeiten und numerische Ids des eingeloggten Benutzers aus.
which `<kommando>`	Sucht im Suchpfad für ausführbare Programme nach dem `<kommando>` und gibt dessen vollständigen Pfad aus.
type `<kommando>`	Gibt Informationen darüber aus, ob `<kommando>` ein eingebautes Shell-Kommando oder ein externes Programm ist (es gibt noch ein paar weitere Möglichkeiten).
cat	Hängt alle Dateien in der gegebenen Reihenfolge aneinander (*concatenate*) und gibt das Ergebnis auf die Konsole aus. Oftmals benutzt man dies, um kurze Textdateien anzusehen ohne einen *pager* (s.u.) zu starten.

Kommando	*Bedeutung*
more, less	Der traditionelle *pager* - also ein Programm, welches lange Textdateien seitenweise auf einer Textkonsole ausgibt – ist *more*, weil dieses Wörtchen am unteren Bildschirmrand signalisiert, dass noch mehr Zeilen folgen. In Anspielung darauf wurde vom GNU-Projekt das mächtigere *less* entwickelt. Unter Linux ist *more* meistens ein Alias für *less*. Übrigens: Beenden kann man beide durch drücken der „q“-Taste (*quit*).
ps, top	Wenn man sehen möchte, was auf dem System an Prozessen läuft, hilft das *ps(1)*-Kommando. Ohne Parameter zeigt es lediglich Prozesse in der aktuellen Session – um auch andere Prozesse zu sehen, hilft meistens *ps -ef* (wenn nicht: *ps wuax* ist die BSD-Alternative). *top* gibt eine im Sekundentakt aktualisierte Liste von Prozessen sortiert nach CPU-Verbrauch (oder anderen Parametern) aus. Beendet wird *top* (wie so viele interaktive Unix-Tools) mit „q“.

Online Hilfe (Manualseiten, RTFM[9])

Weiterhin ist die Dokumentation für die Arbeit auf der Shell unverzichtbar. Diese sog. Manualseiten werden mit dem Kommando *man(1)* aufgerufen. Meistens (je nach System) kann man mit den Cursor-Tasten oder den `PgUp`- und `PgDn`-Tasten durch die dargestellte Manualseite scrollen, was jedoch immer funktioniert, ist das Vorwärts rollen mit der Leertaste und das Verlassen mit „q“.

Die Manualseiten (oft auch als *man-pages* bezeichnet) sind in mehrere Sektionen aufgeteilt, die mit Nummern versehen sind. Zwischen den Systemen gibt es leichte Unterschiede (insbesondere zwischen BSD und SysV), aber hier exemplarisch die Bedeutung der Sektionen unter Linux (entnommen aus *man(1)*):

1	*Executable programs or shell commands*
2	*System calls (functions provided by the kernel)*
3	*Library calls (functions within program libraries)*
4	*Special files (usually found in /dev)*

9 Read The Fine Manual :-)

5	*File formats and conventions eg /etc/passwd*
6	*Games*
7	*Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7)*
8	*System administration commands (usually only for root)*
9	*Kernel routines [Non standard]*

Am Ende jeder Manualseite finden sich gewöhnlich Referenzen auf andere Seiten. Da der Name einer Manualseite (entspricht meistens einem Kommando, einem Systemaufruf, einer C-Bibliotheksfunktion o.ä.) nicht unbedingt eindeutig ist, gibt man sowohl in den Referenzen als auch in beliebigen anderen Texten (wie diesem Skript) die Sektionsnummer in Klammern hinter dem Namen der Manualseite an. Ein erfahrener Anwender weiß daher, dass es sich bei *passwd(5)* um die Dokumentation der Passwortdatei (`/etc/passwd`) handelt und nicht um das Programm zum Ändern des Passworts (welches in *passwd(1)* beschrieben ist). Um etwas über *passwd(5)* zu erfahren, reicht unter Linux das Kommando

```
$ man 5 passwd
```

Unter Solaris muss dies etwas ausführlicher als

```
$ man -S 5 passwd
```

geschrieben werden. Das Kommando

```
$ man passwd
```

reicht hier nicht, weil die Sektionen der Reihe nach durchsucht werden und somit *passwd (1)* angezeigt würde.

Für genaueres über das Manualsystem, siehe *man(1)*.

Variable und Maskierung

In der Shell-Programmierung (und -Bedienung) unterscheidet man zwei Arten von Variablen:

- Shell-Variable sind nur der Shell bekannt. Sie können benutzt werden, um z.B. in Scripten die System-abhängige Konfiguration am Anfang des Scripts zu erledigen (z.B. eine Variable, die den Pfad zu einem temporären Verzeichnis enthält), so dass später im Script nur noch Variable auftauchen. Gleiches gilt für Laufvariable von Schleifen oder temporäre Werte.
- Umgebungs-Variable sind technisch anders, da sie Teil der Prozess-Struktur im Unix-Kern sind. Außerdem werden sie an Kind-Prozesse vererbt und werden oftmals verwendet, um globale Konfigurations-Einstellungen vorzunehmen (viele Kommandos

prüfen vor der Ausführung bestimmte Umgebungs-Variable, die ihr Verhalten beeinflussen).

Beide Variablen-Arten „fühlen“ sich zunächst gleich an; sie haben Namen aus Buchstaben, Ziffern und „_“ (das erste Zeichen des Namens darf keine Ziffer sein; Groß- und Kleinschreibung werden unterschieden) und werden in der Bash mit „=“ zugewiesen:

```
$ foo=bar
$ echo $foo
bar
```

Wie das obige Beispiel zeigt, muss zum Ersetzen der Variable ein $-Zeichen vorangestellt werden. Genau genommen werden zunächst in jeder Zeile, die eine Shell ausführt eine Reihe von Ersetzungen vorgenommen, z.B. werden alle Wörter mit einem „$“ davor durch den Inhalt der gleichnamigen Variable ersetzt. Möchte man dies verhindern, muss entweder ein Backslash vor dem Dollar oder die ganze Zeichenkette in einfachen Anführungszeichen stehen (man sagt, der Dollar wird „maskiert“):

```
$ echo \$foo
$foo
$ echo '$foo'
$foo
```

Um aus dieser Shell-Variable nun eine Umgebungs-Variable zu machen (die dann auch Kind-Prozessen zur Verfügung steht), muss sie exportiert werden:

```
$ PS_FORMAT=cmd
$ export PS_FORMAT
```

Zur Kontrolle kann mit dem Programm *env(1)* geprüft werden, ob die Variable in der Umgebung steht. In der Bash kann man Zuweisung und Export auch zusammenfassen:

```
$ export PS_FORMAT=cmd
```

Man sollte sich dabei aber bewusst sein, dass dies von älteren Shells nicht verstanden wird, d.h. Scripte, die diese Notation verwenden sind nicht portabel.

Will man eine Umgebungs-Variable nur für einen Kind-Prozess setzen und sonst nicht, gibt es folgende Kurzschreibweise:

```
$ PS_FORMAT=cmd ps -e
```

Dies setzt die Umgebungs-Variable `PS_FORMAT` (siehe *ps(1)*) auf `cmd` und startet *ps*. Das *ps*-Kommando gibt als Folge der gesetzten Variable nur eine Spalte (mit dem Kommandonamen des Prozesses) aus; nach Ende des Kommandos ist die Variable `PS_FORMAT` aber nicht mehr gesetzt.

Einige wichtige Umgebungs-Variable sind in der folgenden Tabelle erklärt:

Variable	*Bedeutung*
`PATH`	Suchpfad für ausführbare Programme. Die einzelnen Verzeichnisse werden durch „:“ getrennt.
`MANPATH`	Suchpfad für Manual-Seiten. Die einzelnen Verzeichnisse werden durch „:“ getrennt.
`DISPLAY`	Display-Bezeichner des X-Servers, auf dem X11-Programme ihre Ein-/ Ausgabe tätigen sollen. Wird hier nicht weiter erläutert.
`PAGER`	Wird von vielen Programmen (z.B. *man(1)*) verwendet, um den bevorzugten Pager (also *more* oder *less*) des Benutzers zu finden.
`HOME`	Pfad zum Home-Verzeichnis des ausführenden Benutzers
`USER`	Login des ausführenden Benutzers

Eingabe, Ausgabe, Fehler und Rückgabe-Werte

Wie oben erklärt, hat jeder Unix-Prozess normalerweise[10] drei offene , „virtuelle“ Dateien: *stdin*, *stdout* und *stderr*. Diese können (wie im nächsten Abschnitt erklärt) auf der Kommandozeile mit beliebigen konkreten Dateien durch Umleitungen verknüpft werden, sind aber normalerweise mit dem Terminal des Benutzers verbunden.

Die meisten Unix-Kommandos verarbeiten Dateien und entsprechen üblicherweise dem Muster eines Filters, d.h. wenn keine Dateinamen als Argumente angegeben werden, wird die Standard-Eingabe als Eingabe-Datei und die Standard-Ausgabe als Ausgabe-Datei verwendet. Wenn man z.B. nur das Kommando

```
$ cat
```

aufruft, kehrt dies nicht von selbst zurück, da es auf Eingaben des Users vom Terminal wartet. Gibt man dort eine Zeile ein (und beendet diese mit einem Umbruch), wird die gleiche Zeile wieder auf die Standard-Ausgabe ausgegeben. Normalerweise lesen Filter bis zum Dateiende, der bei einem Terminal nur explizit durch die Tastenkombination `Strg-D` am Zeilenanfang erzwungen werden kann.

Oftmals erwarten Kommandos mehrere Dateien als Eingabe, von denen natürlich nur eine durch die Standard-Eingabe ersetzt werden kann. Welche dies ist, wird für gewöhnlich durch einen Querstrich markiert:

```
$ diff - /tmp/foo
```

Dieses Kommando vergleicht den Text von der Standard-Eingabe mit der Datei `/tmp/foo`.

Im Vorgriff auf den nächsten Abschnitt sei hier schon einmal das folgende Konstrukt vorgestellt:

10 Ausnahme sind sog. Daemons, Hintergrundprozesse ohne Terminal

```
$ cat > /tmp/foo
```

Dies ist ein einfacher (sehr einfacher!) Texteditor. Die spitze Klammer lenkt die Ausgabe in die Datei `/tmp/foo` um, aber die Eingabe des Prozesses bleibt am Terminal. D.h. man kann Text eingeben, der in der Datei landet (bis man mit `Strg-D` beendet). Dies kann sich in Situationen als nützlich erweisen, wenn kein Editor verfügbar ist (Harddisk-Schaden) oder zu viel Ressourcen brauchen würde (Maschine unter Last).

Die bisher beschriebenen Ausgabe-Kanäle eignen sich für beliebige Mengen von Text oder sonstigen Daten. Neben diesen Nutzdaten gibt es allerdings auch noch einen Status, den ein Prozess bei seiner Beendigung für seinen Aufrufer hinterlässt – dieser Rückgabewert (*return code*) wird der Funktion *exit(3)* übergeben und steht der Shell dann anschließend in der Variable `$?` zur Verfügung. Die Konvention besagt, dass der Wert 0 einem „OK“ entspricht und alle anderen Werte irgendeine Form von Fehler ausdrücken, die dann in der Manualseite dokumentiert sind.

Sequentielle Ausführung, Pipelines und I/O-Umleitung

Es ist auch möglich, auf einer Zeile mehrere Kommandos anzugeben. Im einfachsten Fall werden diese einfach nacheinander ausgeführt, wenn die einzelnen Kommandos durch ein Semikolon getrennt sind:

```
$ date ; sleep 10 ; date
Wed Jul 18 11:59:55 CEST 2007
Wed Jul 18 12:00:05 CEST 2007
```

Weiterhin ist es aber auch möglich, die Ausgabe eines Kommandos mit dem Symbol „`>`“ in eine Datei zu schreiben oder dessen Eingabe mittels „`<`“ von dort zu lesen:

```
$ /bin/ls -l / > /tmp/xxx
$ cat /tmp/xxx
total 113
drwxr-xr-x    2 root root  4096 Dec  6  2006 bin
drwxr-xr-x    4 root root  1024 Jul 18 04:56 boot
lrwxrwxrwx    1 root root    11 Feb 22 13:37 cdrom -> media/cdrom
drwxr-xr-x    2 root root  4096 Jun  3  2003 daten
drwxr-xr-x   12 root root 32768 Nov 17  2004 dev
drwxr-xr-x  191 root root 12288 Jul 18 14:48 etc
drwxr-xr-x    3 root root  4096 Mar 21  2002 export
drwxr-xr-x   31 root root     0 Jun 29 07:17 home
...
```

Das o.g. Kommando schreibt die Ausgabe von *ls* nach `/tmp/xxx` (das *ls*-Kommando ist mit absolutem Pfad angegeben, damit der gleichnamige Shell-Alias nicht ausgewertet wird). Das nächste Kommando startet *cat* und verknüpft dessen Standard-Eingabe mit der eben erzeugten Datei. Das ist in diesem Fall natürlich überflüssig, weil man *cat* auch einfach einen Dateinamen angeben könnte, aber es gibt Kommandos, die nur von ihrer Standard-Eingabe lesen und keine Dateinamen als Parameter akzeptieren.

```
$ cat < /tmp/xxx
total 113
drwxr-xr-x    2 root root  4096 Dec  6  2006 bin
```

```
drwxr-xr-x    4 root root  1024 Jul 18 04:56 boot
lrwxrwxrwx    1 root root    11 Feb 22 13:37 cdrom -> media/cdrom
drwxr-xr-x    2 root root  4096 Jun  3  2003 daten
drwxr-xr-x   12 root root 32768 Nov 17  2004 dev
drwxr-xr-x  191 root root 12288 Jul 18 14:48 etc
drwxr-xr-x    3 root root  4096 Mar 21  2002 export
drwxr-xr-x   31 root root     0 Jun 29 07:17 home
...
```

Die beiden Symbole können auch gleichzeitig verwendet werden:

```
$ grep bin < /tmp/xxx > /tmp/xyz
```

Dieses Kommando würde den Befehl *grep* starten, der den regulären Ausdruck *bin* in seiner Eingabe sucht (in diesem Fall die Standardeingabe, die mit `/tmp/xxx` verknüpft ist, weil kein anderer Dateiname angegeben wurde) und seinen Output nach `/tmp/xyz` schreibt.

Das Ergebnis des obigen Vorgehens (nämlich die Ausgabe eines Kommandos als Eingabe für ein anderes Kommando zu verwenden) kann man auch einfacher durch sog. *pipes* erzielen. Das folgende Kommando führt das o.g. *ls* aus und reicht seine Ausgabe als Eingabe an *grep* weiter:

```
$ /bin/ls -l | grep bin > /tmp/xyz
```

Der vertikale Strich wird als Pipe-Symbol bezeichnet.

Wie oben erwähnt, hat jeder Prozess neben der Standard-Ausgabe auch eine Standard-Fehlerausgabe. Durch obige Umleitung wird zwar die Standard-Ausgabe des *grep*-Befehls umgeleitet, eine Standard-Fehlerausgabe bleibt jedoch auf dem Terminal. Möchte man diese auch umleiten, verwendet man `2>` statt `>` (dies funktioniert für beliebige Dateinummern, aber nur die ersten 3 sind fix belegt, s.o.):

```
$ /bin/ls -l | grep bin > /tmp/xyz 2> /tmp/grep_errors
```

Es wäre somit auch möglich, beide Ausgaben in dieselbe Datei umzuleiten, aber man spart Tipparbeit, wenn man dies folgendermaßen ausdrückt:

```
$ /bin/ls -l | grep bin > /tmp/xyz_with_errors 2>&1
```

Die Reihenfolge ist dabei entscheidend, da die Shell mit der Auswertung rechts beginnt und somit zunächst feststellt, dass die Fehler dort landen sollen, wo auch die Standard-Ausgabe hingeschrieben wird (`2>&1`); erst im nächsten Schritt wird dann das Ziel der Standard-Ausgabe geändert.

Alle obigen Ausgabe-Konstrukte würden die Ausgabe-Datei löschen, bevor sie die Ausgabe hinein schreiben. Möchte man dies verhindern und an eine bestehende Datei anhängen, verwendet man statt dessen das Symbol `>>`:

```
$ echo „hier ist noch alles OK“ >> /var/tmp/script_log.txt
```

Kommando-Substitution

Eine weitere Methode neben Umleitung und Pipes, um Kommandos dynamisch mit Eingabe zu versorgen, ist die sog. Kommando-Substitution bei der die Standard-Ausgabe eines Kommandos in die Argumente (oder den Namen) eines weiteren Kommandos eingebaut wird. Dies geschieht „traditionell“ mit den *backticks* (umgekehrten Anführungszeichen “`“), aber in neueren Shells wie der Bash kann auch das (schachtelbare) Konstrukt `$()` verwendet werden.

Das folgende Beispiel würde der Shell-Variable `NOW` das aktuelle Datum im Format JJJJMMTT zuweisen:

```
$ NOW=$(date +%Y%m%d)
```

Dies funktioniert wie gesagt an beliebigen Stellen der Kommandozeile. Das folgende Beispiel ist eine etwas umständliche Möglichkeit, das date-Kommando aufzurufen:

```
$ $(echo date) +%s
```

Was geschieht ist folgendes: in einer Subshell (einem Unterprozess der aktuellen Shell) wird der Inhalt der Klammer ausgeführt. Die Ausgabe des echo-Befehls (die Zeichenkette „date“) wird als Ersatz für das gesamte $()-Konstrukt verwendet und mit dem Argument `+%s` wiederum als Befehl interpretiert, d.h. die obige Zeile entspricht dem Kommando `date +%s` welches das aktuelle Datum und die aktuelle Uhrzeit als Unix-Zeitstempel (also Sekunden seit dem 01.01.1970 um 00:00:00) ausgibt.

Job-Kontrolle und Hintergrund-Prozesse

Beim interaktiven Bedienen von Shells hat man oftmals das Problem, dass ein lang laufendes Kommando das Terminal blockiert und man aber eigentlich noch schnell etwas nachsehen möchte. In Zeiten grafischer Oberflächen kann man natürlich einfach ein weiteres Terminal öffnen, aber wenn man nur eine einzige SSH-Verbindung zum Server offen hat, kann sich die Kenntnis der Job-Kontrolle als nützlich erweisen. Ein Kommando im Vordergrund kann mit der Tastenkombination `Strg-Z` angehalten werden, wodurch die Shell sich wieder mit einer Eingabeaufforderung meldet. Möchte man das Kommando im Hintergrund wieder „nach vorne“ holen, genügt das Shell-Kommando *fg* (für *foreground*). Geht man hingegen davon aus, dass das gestoppte Kommando keine Ein- und Ausgabe mehr benötigen wird, kann man es auch mittels *bg* (für *background*) im Hintergrund weiterlaufen lassen. Hat man mehrere Prozesse angehalten, gibt das *jobs*-Kommando einen Überblick; spezifische Prozesse können dann mittels ihrer Jobnummer angesprochen werden. Da dieser Aspekt für die Automatisierung (Scripting) nicht relevant ist, gehen wir hier nicht weiter darauf ein.

Es gib jedoch auch in nicht-interaktiven Umgebungen (Scripten) den Fall, dass man ein Kommando im Hintergrund starten möchte, damit das aufrufende Shell-Script inzwischen weiter arbeiten kann. Die geschieht einfach durch einen Ampersand `&` am Zeilenende:

```
$ grep secret /dev/random > /tmp/result  &
```

```
```

Dieses Kommando sucht den String `secret` in einem Strom von zufälligen Bytes (die vom Pseudo-Gerät `/dev/random` erzeugt werden) und schreibt die Funde nach `/tmp/result`. Da das ganze ziemlich lange dauern kann (genau genommen: ewig, weil der Eingabestrom nie abreißt), wird der (zugegebenermaßen sinnlose) Prozess durch „`&`“ im Hintergrund gestartet, d.h. die Shell gibt sofort wieder eine Eingabeaufforderung aus und der Aufrufer kann weiterarbeiten.

Kontrollstrukturen, bedingte Ausführung, Schleifen

Wie in allen Programmiersprachen gibt es in vielen Shells Kontrollstrukturen für bedingte Ausführung und Schleifen. In Bourne-kompatiblen Shells (wie der Bash) steht ein *if*-Konstrukt zur Verfügung, welches folgendermaßen aussieht:

```
$ if rm /boot/bzImage ; echo „kernel erfolgreich geloescht“ ; else echo „kernel konnte nicht geloescht werden“ ; fi
```

Das Kommando nach dem *if* (bis zum Semikolon „;“) wird ausgeführt und wenn sein Rückgabewert 0 ist, wird der Teil hinter dem *then* ausgeführt, ansonsten der optionale *else*-Teil. Danach geht die Ausführung nach dem *fi* weiter.

Solche zusammengesetzten Kommandos können auch im interaktiven Modus (und erst Recht in Scripten) auf mehrere Zeilen verteilt werden. Ausprobieren!

Oftmals möchte man aber noch gar keine komplexen Kommandos ausführen, sondern als Bedingung in einem *if* lediglich prüfen, ob eine Variable einen bestimmten Wert hat (oder vielleicht leer ist), ob eine Variable eine Zahl größer oder kleiner als eine andere Zahl enthält, ob eine Datei existiert (und Inhalt hat oder schreibbar ist) oder einen logischen Ausdruck aus solchen einfachen Prädikaten. Dazu gibt es das folgende Konstrukt, welches den Ausdruck in eckige Klammern „[]“ stellt:

```
$ if [ -d /tmp -a „$USER“ = „homer“ ]; then echo „ja“; else echo „nein“; fi
```

Dies prüft, ob `/tmp` ein Verzeichnis ist (`-d`) und (`-a` für *and*), ob die Variable `USER` den Wert `„homer“` enthält. Die Variablen-Dereferenzierung ist hier in doppelte Anführungszeichen gestellt, weil eine leere `USER`-Variable sonst zu einem Syntax-Fehler führen würde.

Die ganze Wahrheit über die eckigen Klammern ist, dass `[` ein Link (symbolisch oder hart) auf ein Kommando namens *test* ist. Daher öffnet „*man [*„ auch die Manualseite von *test(1)*.

Eine kürzere Notation für „*if ...;then ... ;else ... ;fi*“ steht durch logische Operationen zur Verfügung:

```
$ rm /tmp/ && echo "yeah"
```

bzw.

```
$ rm /tmp/xxx || echo "yeah"
```

Die Konstrukte `&&` und `||` entsprechen der logischen UND- resp. ODER-Verknüpfung der Rückgabe-Werte der Kommandos. Nachdem die Ausdrücke aber *lazy* evaluiert werden (d.h. wenn bei `&&` schon der erste Teilausdruck falsch war, wird der zweite gar nicht ausgeführt, weil er den Gesamtausdruck auch nicht mehr wahr machen würde und bei `||` wird der zweite Ausdruck nur dann ausgeführt, wenn der erste falsch war, weil andernfalls der Gesamtausdruck sowieso wahr wäre), sind die o.g. Ausdrücke zu den folgenden äquivalent:

```
$ if rm /tmp/ ; then echo „yeah“ ; fi
```

respektive

```
$ if ! rm /tmp/xxx ; then echo „yeah“ ; fi
```

(Das Rufzeichen `!` negiert den Rückgabewert des Kommandos logisch.)

Neben der Verzweigung mittels *if* oder *case* können auch Schleifen mit der Bash ausgeführt werden. Die Gundform der *for*-Schleife ist dabei

```
$ for var in list; do ... done
```

`list` ist dabei eine Liste von Wörtern über die die Variable `var` iteriert. Für jeden Wert, den die Laufvarible `var` annimmt, wird einmal der Schleifen-Inhalt zwischen `do` und `done` ausgeführt. Hier ein einfaches Beispiel:

```
$ for S in wort1 wort2 "Wort drei" ; do echo $S ; done
wort1
wort2
Wort drei
```

Wie man sieht, können Strings mit Leerzeichen in Anführungszeichen gestellt werden, um als einzelnes Wort gewertet zu werden. Viel sinnvoller ist aber die Verwendung der Kommando-Substitution für die Erzeugung der Liste:

```
$ for I in $(seq 13 17) ; do echo $I; done
13
14
15
16
17
```

Das seq-Kommando dient zum Erzeugen von Zahlensequenzen und kann somit verwendet werden, um über ganzzahlige Intervalle zu iterieren (ähnlich wie die *for*-Schleife in konventionellen Programmiersprachen).

Es gibt auch eine *while*-Schleife, die eine Sequenz von Kommandos so oft ausführt, bis ein Ausdruck (ebenfalls ein Kommando) unwahr wird:

```
$ while test ! -w / ; do sleep 1 ; echo -n . ; done
```

```
........
```

Das *test*-Kommando ist dasselbe wie die eckigen Klammern `[]` bei *if*. Die obige Zeile prüft im Sekundentakt, ob das Wurzelverzeichnis für den aktuellen User schreibbar ist, wenn nicht wird ein Punkt ausgegeben und ein Sekunde gewartet.

Bemerkenswert an *while* ist, dass man Daten per Pipeline hineinschreiben kann. So sieht man oft Konstrukte wie das folgende:

```
$ /bin/ls -1 / | while read S ; do ... ; done
```

Dies würde die Ausgabe von *ls* in die *while*-Schleife „pipen“ und landet in der Standard-Eingabe des *read*-Kommandos. Dies liest Zeile für Zeile in die Variable `S`, mit der dann der Schleifen-Körper ausgeführt wird bis *read* einen Status ungleich 0 liefert – dies ist erst dann der Fall, wenn ein End-of-file (EOF) gelesen wird, also wenn ls nichts mehr liefert.

Es sei der Vollständigkeit halber erwähnt, dass es für Schleifen über Dateinamen, die dann nur ein einziges Kommando auf jede Datei ausführen, eine wesentlich effizientere Lösung gibt: das Kommando *xargs(1)*. *xargs* liest Zeilen von seiner Standard-Eingabe und führt für eine konfigurierbare Anzahl von Wörtern (z.B. Dateinamen) einen gegebenen Befehl aus:

```
$ find /tmp/ -mtime +1| xargs rm
```

Dieser Befehl such Einträge unter `/tmp`, die älter als einen Tag sind und gibt deren Namen an *xargs(1)*, welches dann *rm(1)* (löschen) darauf aufruft. Der Vorteil gegenüber einer Schleife ist, dass *rm* immer für eine möglichst große Anzahl von Dateien aufgerufen wird. Das Limit hierbei ist die maximale Länge einer Kommandozeile, die man mit dem funktional äquivalenten, aber naiveren Befehl

```
$ rm $(find /tmp/ -mtime +1)
```

schnell erreichen würde.

Shell-Scripting

Unter Shell-Scripting versteht man die nicht-interaktive Verwendung einer Shell, bei der die auszuführenden Befehle in einer Textdatei hinterlegt sind und sequentiell ausgeführt werden.

Die wichtigsten Grundlagen über einfache und zusammengesetzte Befehle, über Variable und Kontroll-Strukturen wurden in den vorangegangenen Abschnitten bereits erläutert und gelten natürlich auch für Scripte. In diesem Kapitel werden die Besonderheiten beim Shell-Scripting erklärt.

Texteditoren

Es gibt eine beinahe unendlich große Anzahl von Texteditoren für Unix, manche sind

dabei kommerziell, manche quelloffen, manche grafisch und bunt, manche textbasiert und trotzdem bunt, manche grafisch und trotzdem spartanisch und manche sind textbasiert und spartanisch (in diversen Abstufungen). Für das Shell-Scripting ist es erforderlich, einen Texteditor zu haben, den man gut kennt. Dies kann eigentlich jeder beliebige Texteditor sein, aber man sollte immer daran denken, dass man möglicherweise eines Tages in die Situation kommt, an einem möglicherweise fremden System zu sitzen, welches möglicherweise unter Last steht oder beschädigt ist und vielleicht gar keine grafische Oberfläche am laufen hat. Dann sollte man zumindest die wichtigsten Dateioperationen im *vi*, dem wohl am weitesten verbreiteten (und am kontroversesten diskutierten) Unix-Texteditor beherrschen, um noch arbeitsfähig zu sein. Es sei der Fairness halber erwähnt, dass es noch einen zweiten „großen" Editor mit einer eigenen Fan-Gemeinde gibt, nämlich *(x)emacs*.

Im Rahmen dieser Lehrveranstaltung sollten Sie sich in die Grundlagen von *vi*, bzw. dem unter Linux weit verbreiteten *vim* (*Vi Improved*) einarbeiten (Klausur-relevant!). Ein Tutorial findet sich z.B. hier:

http://www.selflinux.org/selflinux/html/vim.html

Eine zweiseitige Kurzreferenz mit den wichtigsten Kommandos zum Ausdrucken gibt es hier zum Download:

http://tnerual.eriogerg.free.fr/vim.html

Anlegen und Ausführen von Shell-Scripten

Ein Shell-Script ist zunächst eine Textdatei, die die auszuführenden Befehle enthält. Um diese wie ein „normales" Kommando ausführen zu können, sind zwei Schritte notwendig:

- Angeben des zu verwendenden Interpreters
- Für den entsprechenden Benutzer-Kreis ausführbar machen

Da es wie bereits erwähnt unter Unix unterschiedliche Shells gibt, muss in einem Shell-Script natürlich der passende Interpreter angegeben werden, da die Befehle sonst von der aktuellen Shell des Users ausgeführt würden, die aber möglicherweise die Syntax des Scripts nicht (richtig) versteht. Dies geschieht durch einen Kommentar in der ersten Zeile des Scripts, der folgendes Format hat:

```
#!/bin/sh
```

Das Doppelkreuz beginnt einen einzeiligen Kommentar, das Ausrufezeichen kennzeichnet diesen als *shell-bang* (manchmal auch *hash-bang*) und der Rest der Zeile ist der Pfad zur Shell, hier die Standard-POSIX-Shell. Wenn man eine lokal installierte C-Shell als Interpreter nutzen möchte, sähe dies möglicherweise so aus:

```
#!/usr/local/bin/tcsh
```

Bevor ein Script aber tatsächlich vom Kernel ausgeführt werden kann, muss es noch

ausführbar gemacht werden. Wie weiter oben beschrieben, ist die Ausführbarkeit ein Recht, welches dem Eigentümer der Datei, der Gruppe oder dem Rest der Welt gegeben oder genommen werden kann. Dies geschieht z.B. durch folgendes Kommando:

```
$ chmod a+x script.sh
```

Dadurch wird das x-Bit für alle User („a“) gesetzt. Dass das Script im Beispiel die Endung `.sh` hat, entspricht zwar gängiger Praxis, ist aber lediglich eine Konvention und nicht verpflichtend.

Kommandozeilen-Argumente

Scripte sollen natürlich oftmals durch Parameter vom Aufrufer gesteuert werden. In den Bourne-Shell kompatiblen Shells stehen die ersten 9 Argumente in den Variablen `$1...$9` zur Verfügung (`$0` ist der Pfad zum Script selbst, `$1-$9` sind der erste bis neunte Parameter). Weiterhin stehen die Variable `$*` und `$@` zur Verfügung, die jeweils *alle* Argumente enthalten. Worin sich diese beiden Variable unterscheiden und welche es noch gibt, findet sich in der *bash(1)* Manualseite unter *Special parameters*.

Modularisierung von Shell-Scripten

Ähnlich wie in anderen Programmiersprachen auch, kann man in Shells Funktionen definieren, die dann an anderer Stelle wieder aufgerufen werden können. In der Bash sieht das z.B. so aus:

```
function replace_whitespaces() {
       echo $1 | tr ' \t' '__'
}
```

Hier sind die Ziffern-Variable die Funktionsparameter (analog den Kommandozeilen-Parametern oben). Diese Funktion würde ihr erstes Argument als String bearbeiten und alle darin enthaltenen Leerzeichen und Tabulatoren mit dem *tr(1)*-Kommando durch Unterstriche ersetzen. Da Funktionsaufrufe (fast) überall auftreten können, wo auch Kommandos stehen können, ist ihr Rückgabe-Wert numerisch, d.h. Textausgaben müssen über Standard-I/O geschehen. Im obigen Beispiel wäre der Status der Funktion gleich dem Status der darin enthaltenen Pipeline, da nicht explizit mittels *return* ein Status gesetzt wird.

Möchte man Funktionen zwischen mehreren Scripten wiederverwenden, kann man diese in eine eigene Datei auslagern und in den jeweiligen Scripten einlesen. Dies geschieht mit dem Schlüsselwort source oder mit einem Punkt .:

```
$ . ~/lib/my_functions.sh
```

Für den interaktiven Gebrauch wird übrigens bei den meisten Shells beim Start eine Datei im Heimatverzeichnis des Benutzers „gesourced“, die bei der Bourne-Shell-kompatiblen `.profile` oder `.bashrc` heisst und bei C-Shells `.login` bzw. `.cshrc` (dass es jeweils zwei pro Shell gibt liegt daran, dass eine nur beim Einloggen und die andere beim Start jeder Shell ausgeführt wird).

Sucht man Fehler in einem Script, kann man die ausführende Shell mit der Option `-x` starten, dann wird jeder ausgeführte Befehl auf die Konsole ausgegeben. Möchte man hingegen nur die Syntax prüfen, kann man das Script mit -n ausführen. In beiden Fällen kann man die Option in die Bang-Zeile des Scripts eintragen oder die Shell explizit mit `/bin/bash -x script.sh` aufrufen.

Zeitgesteuerte Ausführung

Bisher wurde ausschließlich beschrieben, wie man Aufgaben der System-Verwaltung durch Shell-Scripte erledigen lassen kann, aber diese machen natürlich nur dann Sinn, wenn sie auch ausgeführt werden. Manche Scripte werden dazu entwickelt, dem Administrator selbst die Arbeit zu erleichtern, andere Scripte dienen dazu, von „normalen" Benutzern ausgeführt zu werden und wiederum andere Scripte sollen regelmäßig zu definierten Zeitpunkten automatisch ausgeführt werden. Für letztere gibt es auf quasi jedem Unix-System einen *daemon*[11] namens *cron(8)*, der eine Liste von auszuführenden Prozessen mit Datum und Uhrzeit kennt (die sog. *crontab*), und zu den konfigurierten Zeiten die entsprechenden Kommandos ausführt. Jeder User hat eine eigene *crontab*, die er sich mit

```
$ crontab -l
```

ausgeben lassen kann (bzw. root kann dies mit `crontab -l -u <uid>` für beliebige User tun, siehe *crontab(1)*). Das Format der Tabelle ist wie in *crontab(5)* beschrieben, d.h. die ersten 5 Spalten spezifizieren die Ausführungszeit und der Rest der Zeile besteht aus den auszuführenden Kommandos (die von einer `/bin/sh` interpretiert werden und somit auch entsprechende Konstrukte enthalten können). Die Zeitangabe erfolgt der Reihe nach durch

- Minute (0-59)
- Stunde (0-23)
- Tag des Monats (1-31)
- Monat (1-12)
- Wochentag (0-6 wobei 0 Sonntag ist, oder es können *mon, tue, wed, thu, fri, sat, sun* verwendet werden)

In jedem Feld kann * für „egal" stehen oder es können mehrere Werte durch Kommata getrennt angegeben oder mittels „-" Intervalle definiert werden. Möchte man z.B. jeden Donnerstag um 06:00 Uhr morgens das `/var/tmp`-Verzeichnis von allen Dateien befreien, die älter als 7 Tage sind, würde der Eintrag in der *crontab* so aussehen:

```
0 6 * * thu find /var/tmp -mtime +7 | xargs rm -f
```

Monat, Stunde und Minute und einer der beiden Tages-Angaben eines Eintrags müssen passen, damit ein Eintrag ausgeführt wird, d.h. es ist nicht möglich, den ersten Donnerstag jeden Monats mit der folgenden Zeile zu spezifizieren:

11 Hintergrund-Prozess

```
0 6 1-7 * thu find /var/tmp -mtime +7 | xargs rm -f
```

Dies würde am 01. bis 07. Tag jeden Monats und an *jedem* Donnerstag zutreffen.

Eine weitere Möglichkeit zur zeitlichen Planung von auszuführenden Prozessen ist das *at (1)*-Kommando, welches im Gegensatz zu *cron* ein Kommando nur einmalig ausführt. Genaueres steht in der Manualseite.

Die wichtigsten Unix-Kommandos

Im Folgenden werden die wichtigsten Kommandos in Kurzform vorgestellt, wobei die Liste sicher nicht vollständig ist. Manche werden mit besonders häufig eingesetzten Optionen erläutert, aber auch dies kann in keinem Fall als erschöpfend betrachtet werden. Es ist immer ratsam, einen Blick in die jeweilige Manualseite zu werfen.

Dateisystemoperationen

Kommando	***Bedeutung***	***Erklärung***
`cd <directory>`	*change directory*	Wechselt das aktuelle (Arbeitsverzeichnis) nach `<directory>`. Ohne Argument wird ins Heimatverzeichnis des Benutzers gesprungen. `cd` ist ein Shell-Builtin.
`pwd`	*print working directory*	Gibt das aktuelle Arbeitsverzeichnis aus, in das zuvor mit `cd` gewechselt wurde. pwd existiert sowohl als Shell-Builtin als auch als „echtes“ Kommando.
`mkdir <newdir1> <newdir2>` `mkdir -p <path/to/newdir>`	*make directory*	Erzeugt ein neues Verzeichnis `<newdir>`. Mit der Option `-p` werden auch nicht-existierende Eltern-Verzeichnisse des neuen Verzeichnisses angelegt.
`rmdir <dir1> <dir2>` `rmdir <path/to/dir>`	*remove directory*	Entfernt ein leeres Verzeichnis. Es kann zwar mit der `-p`-Option auch rekursiv gelöscht werden, jedoch werden nie Dateien gelöscht (siehe *rm(1)*).

Kommando	*Bedeutung*	*Erklärung*
`rm <file>` `rm -rf <dir>`	*remove*	rm löscht die angegebenen Dateien. Mit `-r` kann dies rekursiv geschehen, d.h. es wird ein ganzer Dateibaum (inklusive der Verzeichnisse) gelöscht. Mit `-f` (für force) werden fehlgeschlagene Löschoperationen ignoriert (diese würden sonst zum Abbruch führen) und Dateien, die für den Benutzer nicht schreibbar sind, werden ohne Nachfrage gelöscht, *wenn er sie schreibbar machen könnte*.
`cp <file> <file_or_dir>` `cp -r <dir1> <dir2>`	*copy*	Kopiert die gegebenen Dateien ins angegebene Zielverzeichnis. Ist das Ziel kein Verzeichnis, aber dessen Elternknoten, wird die Kopie entsprechend umbenannt. Mit der `-r`-Option kann dies auch rekursiv mit ganzen Dateibäumen geschehen.
`mv <file1> <file2...> <dir>`	*move*	Verschiebt Dateien (analog *cp(1)*, aber das Original wird gelöscht). Diese Operation hängt davon ab, ob Quelle und Ziel auf der gleichen Partition liegen: ist dies der Fall, geht die Operation sehr schnell, weil nur der Verzeichnis-Eintrag umgehängt werden muss. Andernfalls wird das Original zum Ziel kopiert und dann gelöscht, was u.U. viel länger dauert.
`touch <file1, file2 ...>` `touch -a <file1, file2 ...>`		Existierende Dateien bekommen als Zeit der letzten Änderung (*mtime*) die aktuelle Zeit eingetragen. Nicht-existierende Dateien werden leer angelegt. Mit der Option `-a` wird statt der *mtime* die *atime* geändert.
`ln <file1, file2 ...> <dst>` `ln -s <file_or_dir1> <dst>`	*link*	Legt eine Verknüpfung (Link) vom Ziel auf die Quelle an. Mit der `-s`-Option wird dies ein symbolischer Link, der sowohl Dateigrenzen überschreiten kann als auch auf Verzeichnisse zeigen kann. Bei *hardlinks* geht dies nicht: sie sind lediglich weitere, nicht unterscheidbare Einträge innerhalb desselben Dateisystems.

Kommando	*Bedeutung*	*Erklärung*

`ls`
`ls <dir>`
`ls -l <file_or_dir>`
	list	Listet den Inhalt eines Verzeichnisses, ohne Angabe das aktuelle Verzeichnis. Mit `-l` werden auch Eigentümer, Gruppe, Änderungsdatum, Dateityp und Rechte ausgegeben. Das `ls`-Kommando hat eine riesige Anzahl von Optionen, so dass sich auf jeden Fall ein Blick in die Manualseite lohnt.

Rechte und Eigentum

Kommando	*Bedeutung*	*Erklärung*

`chown <user> <file_or_dir>`
`chown -r <user> <dir1,...>`
`chown <user:group> <file>`
	change owner	Ändert den Eigentümer einer Datei oder eines Verzeichnisses und evtl. die Gruppe. „Normale“ Benutzer können Datei-Eigentümer nicht ändern (das kann nur *root*), aber die Gruppenzugehörigkeit kann er in allen Gruppen ändern, zu denen er selbst gehört. Die Operation `-r` macht die Änderung rekursiv für ganze Dateibäume.

`chgrp <group> <file_or_dir>`	*change group*	Ändert die Gruppenzugehörigkeit einer Datei analog `chown` (s.o.)
`chmod <mod> <file_or_dir>` `chmod -r ...`	*change mode*	Ändert die Dateiberechtigungen entsprechend der Angabe in `<mod>`. Diese kann in Oktalnotation erfolgen oder durch Angaben im folgende Format: ● `a+x` setzt für alle (also Eigentümer, Gruppe und alle anderen) das x-Bit ● `go-w` löscht das Schreibrecht (w) für Gruppe (g) und alle anderen (o) Für die vollständige Erklärung sollte man in *chmod(1)* nachsehen.

Grundlegende Textverarbeitung

Kommando	*Bedeutung*	*Erklärung*
`more <file>` `less <file>`		Mit `more` können lange Texte seitenweise angezeigt werden, vorwärts blättern mit Leertaste. Unter Linux ist `more` aber meistens nur ein Alias für `less` welches die mächtigere GNU-Implementation ist, die auch rückwärts blättern mit den Cursor-Tasten, `PgUp`, `PgDn` und weitere Features unterstützt.

`grep <regex> <file1,...>`
`grep -l <regex> <file1,...>`
`grep -r <regex> <dir>`
	global regular expression search	Sucht Zeilen in Textdateien, die einem regulären Ausdruck entsprechen: darin entspricht jeder Buchstabe sich selbst, beliebige Buchstaben werden durch „.“ markiert, Wiederholungen werden mit „?“ für 0 oder 1, „*“ für 0 oder mehr und „+“ für 1 oder mehr nach dem jeweiligen Zeichen gekennzeichnet. Normalerweise werden die passenden Textzeilen ausgegeben, mit der Option `-l` werden nur die Dateinamen der Dateien aufgelistet, die passende Zeilen enthalten und mit `-r` kann (nur in der GNU-Implementation!) rekursiv durch ganze Dateibäume gesucht werden.

`head [-n<num>] <file>`
`tail [-n<num>] <file>`
`tail -f <file>`
		head gibt die ersten `<num>` Zeilen einer Textdatei aus – ohne die `-n`-Option sind dies 10 Zeilen. `tail` liefert die letzten `<num>` Zeilen einer Datei und mit `tail -f` (für *follow*) kann im interaktiven Betrieb eine wachsende Textdatei (z.B. ein Logfile) mitgelesen werden.

`wc <file>`
`wc -l <file>`
`wc -c <file>`
`wc -w <file>`
	word count	`wc` zählt die Zeilen, Wörter und Buchstaben einer Textdatei und gibt diese nebeneinander aus. Wünscht man nur einen dieser Werte, erreicht man dies mit der Option `-l` (*lines*) für die Zeilen, `-c` (*characters*) für die Zeichen und `-w` (*words*) für die Wörter.

Kommando	***Bedeutung***	***Erklärung***
`sort [-n] [-r] [-k <n>] <file>`		Sortiert die Zeilen einer Textdatei. Mit `-n` wird numerisch sortiert, mit `-r` wird die Sortierung umgekehrt und mit `-k` gibt man die Sortierspalte(n) an. Mit `-t` kann der Spaltentrenner angegeben werden (normalerweise Leerzeichen).
`sed`	*stream editor*	`sed` erlaubt die scriptgesteuerte Ausführung von Editier-Kommandos auf Textströme, z.B. das Löschen von Zeilen, das Ersetzen von Text usw. Die genaue Syntax der Scriptsprache (in Postfix-Notation!) wird in der Dokumentation erklärt.
`awk`	*Aho, Wheinberger, Kernighan*	Auch `awk` erlaubt das nicht-interaktive Editieren von Texten, wobei es sich im Gegensatz zu `sed` etwas besser für tabellarische Informationen eignet und sich sprachlich an der Programmiersprache C orientiert. Auch hier sei für alles Weitere auf die Manualseite verwiesen.
`m4`		`m4` ist ein Makroprozessor, der ursprünglich für einen Fortran-Dialekt gedacht war, den es aber längst nicht mehr gibt. `m4` hat sich aber als Text-Prozessor gehalten und ist auf vielen Unix-Systemen installiert. Im Gegensatz zu `sed` und `awk` wird hier das Script aber in den zu verarbeitenden Text eingebaut. Alles Weitere steht auch hier in der Dokumentation.

Backup & Archivierung

Kommando	*Bedeutung*	*Erklärung*
`tar c[z]f <arch> <file_or_dir>` `tar x[z]f <arch>`	*tape archive*	`tar` ist das am weitesten verbreitete Archiv-Format unter Unix. Ursprünglich als Format für Magnet-Bänder gedacht, wird es heute ebenso häufig als Dateiformat und manchmal sogar als Netzwerkprotokoll verwendet. mit `tar cf` erzeugt man ein Archiv `<arch>` aus den folgenden Dateien oder Verzeichnissen und mit `tar xf` entpackt man dies wieder. Die zusätzliche Option `z` bewirkt in beiden Fällen die Filterung durch `gzip` für die (De-) Kompression (nur in der GNU-Variante). Verwendet man `j` statt `z`, wird `bzip2` statt `gzip` verwendet.

`gzip [-c] <file>`
`gunzip [-c] <file>`
`compress <file>`
`uncompress <file>`
`bzip2 [-c] <file>`
`bunzip2 [-c] <file>`
		Mit `gzip` wird die angegebene Datei komprimiert (und mit der Endung `.gz` versehen), mit `gunzip` wird eine `gz`-Datei wieder dekomprimiert. Mit der Option `-c` kann man beim (de-)komprimieren die Ausgabe auf die Standard-Ausgabe umleiten, z.B. um diese in eine Pipe zu schicken. `gzip` und `gunzip` stehen nur unter Linux zur Verfügung oder auf kommerziellen Systemen, auf denen sie nachinstalliert wurden.
		`compress` bzw. `uncompress` ist die ältere Unix-Variante, die nur noch selten benutzt wird.
		`bzip2` und `bunzip2` basieren auf einem moderneren Kompressionsverfahren, welches (manchmal) Vorteile bei der Kompression auf Kosten der Geschwindigkeit liefert und stehen überwiegend auf Linux-Systemen zur Verfügung.

Dateien suchen und verarbeiten

Kommando	*Bedeutung*	*Erklärung*
`find <dir> -name <glob>`		`find` ist ein sehr mächtiges Werkzeug zum Auffinden von Dateien. Der links dargestellte Aufruf würde von einem Wurzelverzeichnis `<dir>` ausgehend rekursiv nach Dateien suchen, deren Namen auf den Shell-Glob[12] `<glob>` passen. Es gibt aber noch viel mehr Möglichkeiten (es können sogar logische Ausdrücke aus Prädikaten formuliert werden), die alle in der Manualseite *find(1)* erklärt werden.
`<cmd> | xargs <cmd2>`		`xargs` liest Zeilen von der Standard-Eingabe (z.B. durch eine Pipe) und ruft für eine konfigurierbare Anzahl solcher Zeilen das Kommando `<cmd2>` auf. `xargs` wird oft im Zusammenspiel mit `find` benutzt - `find` hat zwar auf manchen Systemen mit der `-exec`-Option die Möglichkeit, Kommandos auf gefundenen Dateien auszuführen, aber meistens ist es performanter, die `find`-Ausgabe per Pipeline an `xargs` zu schicken, da dies dann das jeweilige Kommando nur einmal für mehrere Dateien aufruft. Da Prozess-Erzeugung unter Unix eine zeitaufwändige Angelegenheit ist, sollte man damit sparsam umgehen.

12 Auch als „Wildcards“ bekannt: * passt auf beliebige Zeichenfolgen, ? nur auf exakt ein Zeichen.

Kommando	*Bedeutung*	*Erklärung*
`locate <str>`		`find` bietet zwar viele Möglichkeiten, um Dateien ausfindig zu machen, aber da bei jedem Aufruf ein (möglicherweise ziemlich großer) Dateibaum traversiert werden muss, kann dies durch hohe I/O-Last langsam sein und das System (oder sogar das Netz) gefährden. Auf vielen Systemen ist daher eine einfache Suche anhand von Dateinamen mittels des `locate`-Kommandos möglich, welches einen Index bemüht, der an Stelle des echten Dateisystems durchsucht wird. Für den sinnvollen Einsatz ist es allerdings unabdingbar, dass der Systemadministrator eine regelmäßige Aktualisierung des Index eingerichtet hat.

Prozess-Verwaltung

Kommando	*Bedeutung*	*Erklärung*

`ps`
`ps wuax`
`ps -ef`
	process status	`ps` ohne Optionen listet die Shell und ihre Kind-Prozesse auf. `ps wuax` ist unter Linux und den BSD-Unixen die Standard-Formulierung für „alle Prozesse und alle Informationen“, `ps -ef` ist das SysV-Äquivalent dazu. `ps` kennt eine Vielzahl von Optionen, die sowohl die Prozessauswahl als auch die anzuzeigenden Informationen beeinflussen, die allesamt in der Manualseite *ps(1)* beschrieben sind.

`kill [-<sig>] <pid>` `kill -l`		Sendet ein Signal an den Prozess mit der ID `<pid>`. Gibt man das Signal nicht an, wird das `TERM`-Signal (15) verwendet. `kill -l` listet die auf dem System verfügbaren Signale. Das Signal 9 (`SIGKILL`) beendet den empfangenden Prozess immer und kann nicht abgefangen werden, `SIGSTOP` hält einen Prozess an, der dann mit `SIGCONT` wieder weiterlaufen kann.

Weitere Werkzeuge

Kommando	*Bedeutung*	*Erklärung*

`expr <n> + <m>`
`expr <n> - <m>`
`expr <n>  \* <m>`
`expr <n> / <m>`
`expr <n> \< <m>`
`...`
`expr <a> \& <b>`
`expr <a> | <b>`
`...`
	expression	`expr` implementiert einfache Ganzzahl-Arithmetik und logische Ausdrücke. Zu beachten ist dabei, dass viele Operatoren für die Shell spezielle Bedeutungen haben und daher mit „\“ maskiert werden müssen. Ausdrücke können Klammern enthalten, aber auch diese müssen vor der Shell maskiert werden, da diese sonst eine Subshell für den Klammerinhalt starten würde.

`crontab` `at`		
`nohup`		
`sendmail`		