Band

2

FACHHOCHSCHULE HAGENBERG

Skriptenserie zu Betriebssysteme und Netzwerktechnologie

UNIX

Zweite Auflage

FACHHOCHSCHULE HAGENBERG

Unterlagen zur Lehrveranstaltung

Stefan Probst
© SSW
Jormannsdorf 99 • A-7431 Bad-Tatzmannsdorf
Telefon +43 699 / 120 55 172
E-Mail: stefan.probst@fh-hagenberg.at

Einleitung

Das vorliegende Dokument enthält die Begleitunterlagen zu der Lehrveranstaltung „Betriebssysteme“. Das Werk einschließlich aller Teile ist urheberrechtlich geschützt. Die komplette oder nur teilweise Verbreitung und Verwendung dieses Dokumentes bedarf der ausdrücklichen Zustimmung des Urhebers.

Danksagung

Besonderer Dank gilt Herrn Klaus Wolfmaier und Rudolf Ramler für die Mitarbeit und Mitgestaltung dieses Dokuments.

Verwendung des Dokuments

SYMBOL-LEGENDE	
ⓘ	Informationen
!	Wichtige Hinweise
🖋	Beispiele
?	Verständnisfragen

In diesem Skriptum werden Sie immer wieder auf verschiedene Symbole und Schreibweisen stoßen.

Spezielle Ausdrücke und Begriffe werden durch einfache Anführungszeichen (‚‘) gekennzeichnet. Kommandos werden innerhalb doppelter Anführungszeichen („“) erwähnt. Variable Teile innerhalb eines Kommandos sind *kursiv* gedruckt, genauso wie betonte Wörter. Am Seitenrand werden Sie immer wieder Symbole finden, die auf spezielle Informationen hindeuten.

Inhaltsverzeichnis

KAPITEL 1

UNIX-Grundlagen und Einführung 1-1
Eigenschaften von UNIX 1-2
Entwicklungsziele von UNIX 1-5
Geschichte von UNIX 1-7
Systemstruktur 1-14
Hauptkomponenten des UNIX-Systems 1-15
Der Kernel 1-20
Freie UNIX-Systeme 1-24

KAPITEL 2

UNIX-Erste Schritte 2-1
Interaktion mit UNIX 2-5
Die Shell 2-7
Verzeichnisse in UNIX 2-8
UNIX-Kommandos 2-12
Die wichtigsten UNIX-Kommandos 2-14
Graphische Benutzeroberfläche 2-20

KAPITEL 3

Das UNIX-Dateisystem 3-1
Einführung in das Dateisystem 3-2
Aufgaben des Dateisystems 3-4
Verschiedene Dateitypen 3-8
Konventionen für Dateinamen 3-12
Verzeichnisse in UNIX 3-13
Standardverzeichnisse 3-16
Aufbau des Dateisystems 3-17
Der Superblock 3-19
Inodes 3-20
Pfadauflösung 3-24
Links 3-26
Hardlinks 3-29
Softlinks 3-30
Unterschiede zwischen Hard- und Softlinks 3-31
Kommandos zur Dateimanipulation 3-34
Dateisysteme einbinden 3-39
Verfügbaren Platenplatz beschränken 3-42
Platten Management 3-43

KAPITEL 4

Benutzerverwaltung in UNIX 4-1
Notwendigkeit der Benutzerverwaltung 4-2
Login 4-4
Benutzerdaten 4-6
Die Shadow-Datei 4-9
Benutzergruppen 4-11
Standardbenutzer 4-13
Benutzerverwaltung 4-15

KAPITEL 5

Zugriffsrechte in UNIX 5-1
Das UNIX Rechtemodell im Detail 5-4
Rechte anzeigen und interpretieren 5-5
Rechte und deren Bedeutung 5-7
Rechte vergeben 5-9
Rechtevergabe steuern 5-14
Standardberechtigungen festlegen 5-16

KAPITEL 6

Die Shell 6-1
Aufgaben der Shell 6-2
Ein-/Ausgabeumleitung 6-6
Pipes 6-10
Metazeichen 6-12
Quoting 6-14
Bourne Again Shell (bash) 6-16
Der Prompt 6-17
Shell-Variablen 6-19
Verwendung von Shell-Variablen 6-20
Umgebungsvariablen 6-21
Interne Kommandos 6-24
Interne Jobkontrolle 6-25

KAPITEL 7
Shellprogrammierung 7-1
Was ist Shellprogrammierung 7-2
Allgemeines zur Shellprogrammierung 7-3
Shellvariablen 7-6
Spezielle Variablen 7-8
Interaktive Eingabe 7-11
Ablaufstrukturen 7-13
Weitere Anweisungen 7-28
KAPITEL 8
Reguläre Ausdrücke 8-1
Einführung in reguläre Ausdrücke 8-2
Reguläre Ausdrücke 8-4
Zeichenersetzung 8-5
Zeichenauswahl 8-6
Sonstige Ausdrücke 8-7
KAPITEL 9
Prozesse 9-1
Was ist ein Prozess 9-2
Prozesse in UNIX 9-4
Prozesse duplizieren 9-5
Prozesse überlagern 9-7
Der init-Prozess 9-10
Prozesstypen 9-11
Prozessverwaltung 9-13
Prozesszustände 9-16
Prozesskommunikation 9-20
KAPITEL 10
Der Bootvorgang 10-1
Der BIOS-Selbsttest 10-2
Der Bootloader 10-5
Initialisierung des Kernels 10-7
Die Datei ‚/etc/inittab' 10-9
Funktion von Runlevels 10-11
Aktionen von Runlevels 10-13
Konfiguration von Runlevels 10-16
Arbeitsweise des Skriptes rc 10-18

Kapitel

1

UNIX – Grundlagen und Einführung

Dieses Kapitel erklärt Grundlagen des UNIX-Betriebssystems und stellt das Betriebssystem und dessen Konzepte vor.

UNIX war von je her ein leistungsstarkes Betriebssystem. Man findet es hauptsächlich in großen Firmennetzwerken und Mainframes. Heutzutage erlebt man wie ein UNIX-ähnliches Betriebssystem (Linux) immer mehr an Popularität gewinnt.

Lehrinhalte und -ziele

Dieses Kapitel beschäftigt sich mit der Einführung in das UNIX-Betriebssystem und erläutert die geschichtliche Entwicklung dieses Betriebssystems.

Sie sollten nach diesem Kapitel über die geschichtliche Entwicklung von UNIX Bescheid wissen und einen groben Einblick in die UNIX-Welt haben.

Eigenschaften von UNIX

Eigenschaften von UNIX$_1$

- Portables, einfach aufgebautes Betriebssystem
  - Läuft auf verschiedenen Maschinen
  - Multitasking / Multiprocessing OS
  - Multiuser OS
  - Dialogorientiert (Shell, Terminals)
- Geeignet für
  - Mikrocomputer
  - Mini-Computer
  - Großrechner

UNIX ist ein portables und einfach aufgebautes Betriebssystem. Ursprünglich wurde UNIX für die PDP-7 von Digital entwickelt. Schnell wurde die Notwendigkeit nach einem portierbaren Betriebssystem erkannt. Darum wurde UNIX in C neu geschrieben und war somit auf verschiedene Maschinen-architekturen portierbar.

Multitasking

UNIX ist ein Multitasking Betriebssystem. Dies bedeutet, dass UNIX mehrere Prozesse gleichzeitig ausführen kann. Die gleichzeitige Ausführung passiert nur scheinbar, da sich die verschiedenen Prozesse eine CPU teilen müssen. UNIX sorgt dafür, dass die CPU-Zeit unter den Prozessen nach Prioritäten aufgeteilt wird. UNIX unterstützt aber auch Multiprozessor-Systeme. Hier werden Prozesse, die getrennt voneinander laufen können, auf verschiedene, im System befindliche CPUs, aufgeteilt. Auf Multiprozessor-Systemen können mehrere Prozesse wirklich gleichzeitig ausgeführt werden.

Multiuser

Eine weitere Eigenschaft von UNIX ist, dass es mehrere Benutzer auf einem System unterstützt. Jeder Benutzer eines UNIX-Systems erhält einen eigenen Bereich, in dem UNIX benutzerspezifische Daten speichern kann. Benutzer können das System an ihre individuellen Bedürfnisse anpassen, ohne die Einstellungen anderer Benutzer zu verändern.

Weitere Eigenschaften

UNIX ist stark dialogorientiert. Dass bedeutet, dass der Benutzer über Dialoge mit dem System kommuniziert. Dies passiert durch Eingaben in der Shell oder über entsprechende Terminals.

Durch die Portabilität ist UNIX für verschiedene Rechnersysteme geeignet. Dazu zählen Mikrocomputer, Mini-Computer, aber auch Großrechner.

Eigenschaften von $UNIX_2$

- Werkzeugkasten
  - Besteht aus hunderten von Dienstprogrammen
  - Flexible, kleine Tools können schnell erstellt werden („small is beautiful“)
- Weitere Eigenschaften
  - Kryptische Befehle (ls, pwd, cat, awk, grep, …)
  - Motto: „Keine Nachricht ist eine gute Nachricht!“ bzw. „Wer will schon schlechte Nachrichten?“ → Wenig Feedback an den Benutzer
- Ein System von Programmierern für Programmierer

Werkzeugkasten

UNIX hat einen eingebauten Werkzeugkasten. Zu dem eigentlichen Betriebssystem werden hunderte von Dienstprogrammen geliefert, die den Benutzer und Administrator bei seiner täglichen Arbeit unterstützen. UNIX bietet die Möglichkeit, schnell flexible und kleine Tools zu erstellen. Mit diesen Tools kann ein Teil der täglichen Arbeit automatisiert werden.

Weitere Eigenschaften

Wenn man mit UNIX arbeitet, stellt man schnell fest, dass das System aus einer Reihe von kryptischen Befehlen besteht, die eine Reihe von verschiedenen Parametern entgegennehmen. Außerdem gibt das System nur die notwendigsten Nachrichten an den Benutzer weiter und spart mit ausführlichen Berichten.

Das Betriebssystem UNIX ist aus diesem Grund ein System von Programmierern für Programmierer; Benutzer, die wissen was sie tun.

Entwicklungsziele von UNIX

Entwicklungsziele von UNIX

- Thompson definiert 7 Richtlinien von UNIX
  1. Hierarchisches Dateisystem
  2. Ausgabe von Daten an Peripherie und Dateien soll gleich sein (stark dateibasiert)
  3. Prozesse und Programme sollen miteinander kommunizieren
  4. Multitasking / Multithreading
  5. Persönliche Shells
  6. Compiler für viele Programmiersprachen
  7. Portabilität

Ken Thompson, der Initiator des Betriebssystems UNIX, definierte bei seiner Entwicklung sieben Richtlinien, die das neue Betriebssystem erfüllen sollte:

1. Hierarchisches Dateisystem: Das Dateisystem folgt einer festen Struktur und ist in einer Hierarchie geordnet. Das Wurzelelement oder der Ursprung wird dabei als „Root“ bezeichnet.

2. Die Ausgabe von Daten an Peripherie und Dateien soll gleich strukturiert sein: Das ganze System sollte stark auf Dateien basieren. Daten sollten an den Drucker oder sonstige Peripheriegeräte genauso ausgegeben werden als wären diese Dateien. In UNIX sind für alle Geräte eigene Dateien im Dateisystem angelegt, die eine Kommunikation zwischen Benutzer und Hardware erlauben.

3. Prozesse und Programme sollen miteinander kommunizieren: Das System bietet Mechanismen an, um eine Kommunikation zwischen einzelnen Prozessen und Programmen zu ermöglichen.

4. Das System sollte Multitasking und Multithreading unterstützen: Das System sollte die gleichzeitige Ausführung mehrerer Programme und Prozesse ermöglichen.

5. Das System sollte eine individuelle Benutzerschnittstelle bieten: Die Benutzerschnittstelle ist nicht Teil des eigentlichen Betriebssystems sondern in einer eigenen Komponente realisiert. Diese Komponente sollte austauschbar sein damit sich Benutzer ihre eigene Benutzerschnittstelle zurechtlegen können.

6. Das System sollte für Programmierer geeignet sein: UNIX wurde hauptsächlich entwickelt, um Programmierer bei deren Arbeit zu unterstützen. Zur damaligen Zeit gab der Programmierer Lochkarten oder Lochstreifen beim Operator ab. Diese wurden dann in den Rechner (damals noch ausgesprochene Batch-Systeme) eingelesen und ein Rechenauftrag nach dem anderen abgearbeitet. Der Programmierer konnte dann nach einiger Zeit seine Ergebnisse abholen.

7. Das System sollte auf unterschiedlicher Hardware funktionieren: Damalige Systeme waren an Hardware gebunden. UNIX sollte erstmals diese Restriktion und Gebundenheit an den Hersteller aufheben.

Mit Hilfe dieser Richtlinien wollte Ken Thompson ein System schaffen, auf welchem mehrere Programmierer im Team und im Dialog mit dem Rechner arbeiten, Programme entwickeln, korrigieren und dokumentieren konnten, ohne von einem Großrechner mit allen seinen Restriktionen abhängig zu sein. Dabei standen Funktionalität, strukturelle Einfachheit und Transparenz sowie leichte Bedienbarkeit im Vordergrund der Entwicklung.

Geschichte von UNIX

Geschichte von UNIX$_1$

- 1969 Ken Thompson @ Bell Laboratories
  - Entwicklung von UNIX für DEC PDP-7
  - Einführung von Threads
  - Echtzeitfunktionen
  - Benutzeroberfläche in Spezifikation integriert
- 1971 in C „umgeschrieben“
  - Rasante Weiterentwicklung durch Benutzer

UNIX ist aus der Notwendigkeit entstanden, für die Programmerstellung ein Rechnerbetriebssystem zu entwickeln. Die Portierbarkeit von UNIX beruht auf der Entwicklung der Programmiersprache C, die es möglich machte, ca. 90% des Betriebssystems in einer höheren Programmiersprache zu entwickeln.

Die ersten Schritte der Entwicklung von UNIX gehen auf das Jahr 1969 zurück, lange bevor es MS-DOS gab. Ken Thompson, von den Bell Laboratories, entwickelte UNIX zunächst als Unterstützung für Programmentwickler. Die Entwicklung richtete sich stark nach den Anforderungen der Programmentwickler, die auf dem System arbeiteten. Die verwendete Hardware Plattform war damals eine PDP-7 von Digital, eine 16-Bit Maschine mit 4096 Speicherworten und einem Fernschreiber.

Auf Grund des geringen Speichervorrats wurde das gesamte System in Assembler realisiert.

Bahnbrechende Neuerungen waren die Einführung von Threads, die Unterstützung von Echtzeitfunktionen und erstmals war die Benutzeroberfläche in die Spezifikation des Betriebssystems integriert.

Um bei künftigen Projekten die Maschinenabhängigkeit durch eine maschinennahe Sprache zu umgehen, entwarf Thompson die Programmiersprache B, aus der dann Dennis Ritchie die Sprache C entwickelte.

UNIX wurde 1971 in C umgeschrieben und auf die PDP-11 übertragen. Von nun an erfolgte die Weiterentwicklung des Systemkerns sowie die meisten Dienstprogramme in dieser Sprache. Die Kompaktheit und strukturelle Einfachheit

des Systems ermunterte viele Benutzer zur eigenen Aktivität und Weiterentwicklung des Systems, so dass UNIX recht schnell einen relativ hohen Reifegrad erreichte. Dies ist deshalb so bemerkenswert, da kein Entwicklungsauftrag hinter diesem Prozess stand und die starke Verbreitung von UNIX nicht auf den Vertrieb oder die Werbung eines Herstellers, sondern primär auf das Benutzerinteresse zurückzuführen ist. Eine ähnliche Entwicklung zeigt sich seit einigen Jahren bei den freien UNIX-Varianten.

Geschichte von $UNIX_2$

- 1975 UNIX Version 6
  - Zur Verwendung an Universitäten
  - Möglichkeit, hardwareunabhängigen C-Code zu schreiben
- 1979 UNIX Version 7
  - Erste 32-bit Version
  - BOURNE Shell
  - Erste Installationen im kommerziellen Bereich
  - Entstehung des BSD (Berkeley Software Distribution)
  - TCP/IP Protokoll eingeführt

1975 wurde die erste offizielle UNIX-Version, vorerst allerdings nur zur Verwendung an Universitäten und für Forschungsabteilungen innerhalb der Bell Laboratories, unter anderem auch an der University of California in Berkeley (UCB) freigegeben. Diese Version wurde auch schnell als UNIX Version 6 bekannt.

Der größte Teil war bereits in C geschrieben und C setzte sich auch unter den Systemprogrammierern langsam durch, die die reichhaltige Funktionalität und einfache Lesbarkeit gegenüber Assemblern bald zu schätzen lernten. Anwendungsprogrammierer hatten nun die Möglichkeit, hardware-unabhängigen Code zu schreiben, die einzige Voraussetzung war ein lauffähiger C-Compiler auf dem Zielsystem.

1979 wurde die erste 32-Bit Version von UNIX freigegeben, bekannt als Version 7. Diese Version beinhaltete die Erweiterung von C auf 32-Bit Datentypen, die Entwicklung der Bourne Shell und die Neuorganisation des UNIX-Kernels, um ihn portabler zu machen. Es erfolgten nun erste Installationen im nicht universitären Bereich.

Mitglieder des Forschungsteams der Bell Laboratories lehrten in Berkeley UNIX und es entwickelten sich sehr bald neue Forschungsaktivitäten am UNIX System. In der Folge entstanden in Berkeley einige UNIX Versionen, die auf Digital VAX 11/780 Systemen liefen und wichtige Kernel Modifikationen beinhalteten wie die virtuelle Speicherverwaltung (Demand Paging) und die Einführung des TCP/IP Protokolls für Kommunikationszwecke. Diese Berkeley Versionen werden unter dem Namen 1BSD, 2.xBSD, 3.xBSD und 4.xBSD geführt, wobei BSD für Berkeley Software Distribution steht.

Geschichte von $UNIX_3$

- 1981 UNIX System 3 / Microsoft XENIX
  - Auf kommerziellen Bereich konzentriert
  - Microsoft und SCO → XENIX
- 1983 AT&T UNIX System V
  - Erweitertes System 3
  - Performanceverbesserungen
  - Semaphoren, Message Queues und Shared Memory eingeführt
  - Als Standard für künftige Entwicklungen

1981 war bereits eine Vertiefung des kommerziellen Interesses an UNIX durch viele Firmen erkennbar, weil es sehr einfach auf die verschiedenen Hardware Plattformen zu portieren war, parallel dazu konnten mittlerweile Prozessoren wesentlich billiger hergestellt werden, wodurch UNIX Portierungen zu einem guten Geschäft wurden.

Microsoft und SCO (Santa Cruz Operation) entwickelten gemeinsam XENIX. Die Bell Laboratories gliederten die UNIX Forschungsgruppe aus und die USG (UNIX Support Group) wurde gegründet, die 1982 erstmals außerhalb der Bell Laboratories eine UNIX Version heraus brachte, die mit UNIX System III bezeichnet wurde.

Kurz darauf brachte SCO XENIX System III heraus und mehrere BSD Versionen wurden auf Motorola 68000 Maschinen realisiert.

Bis 1983 gab es bereits mehr als 100 UNIX basierte Mini- und Mikrocomputer-Systeme und mehr als 100.000 UNIX Anwender.

1983 veröffentlichte die Firma AT&T das UNIX System V. Der Kernel erhielt zwar neue Features, im Prinzip ist es aber ein erweitertes System III mit Performanceverbesserungen (Hashtables und Caches). Darüber hinaus wurden Semaphoren, Nachrichtenwarteschlangen und gemeinsam genutzte Speicher-bereiche eingeführt, die der Prozesskommunikation dienen.

Grosse Überraschung rief die Ankündigung hervor, dass diese Release zu allen zukünftigen Releases von System V aufwärtskompatibel sein würde.

Geschichte von UNIX$_4$

- 1984 AT&T UNIX System V Release 2
  - Gründung des X/OPEN Konsortiums
    - Einbeziehen der Hardwarehersteller
- 1986 POSIX Komitee (IEEE)
  - ISO-Standard für portable Betriebssysteme
- 1987 AT&T UNIX System V Release 3
  - Abspaltung in verschiedene Derivate
- 1989 AT&T UNIX System V Release 4
  - Neufassung des Kernels

1984 veröffentlichte AT&T die zweite Release des System V (SVR2). Die USG hatte sich mittlerweile in USDL (UNIX System Development Laboratories) umbenannt und war verantwortlich für die Veröffentlich für SVR2. In zahlreichen Unterreleases wurden neue Features wie Datensatz- und Dateisperren und neue Speicherverwaltungstechniken eingeführt.

Im gleichen Jahr wurde jedoch auch das X/OPEN Konsortium gegründet, das unter Beteiligung der größten Computerhersteller das Ziel verfolgt, eine umfassende Entwicklungs- und Laufzeitumgebung für portable Applikationssoftware bereitzustellen. Im Gegensatz zu anderen Organisationen wird X/OPEN auch Bereichen gerecht, die über AT&T's Bemühungen einen Standard zu schaffen hinausgehen z.B. graphische Oberflächen (Fenster Schnittstellen), Datenverwaltung, Datenbanken, etc.

X/OPEN verfolgt auch die Strategie, dass es öffentliche Standards anerkennt und unterstützt somit den vollständigen POSIX Standard.

Dieses POSIX Komitee wurde 1986 gebildet. Das erklärte Ziel war es, ein ISO-Standard für portable Betriebssystemumgebungen zu erstellen. 1988 wird dieser Entwurf von der ISO anerkannt und bietet ein konformes Standardprodukt, das von einer Standardisierungsorganisation anerkannt ist.

1987 veröffentlicht AT&T das System V Release 3 (SVR3). Zu den zahlreichen Erweiterungen zählen *Streams* als neue Kommunikationsmöglichkeit, viele Sicherheitsverbesserungen die auf Benutzerebene realisiert wurden und Performanceverbesserungen. Ende 1987 gab es ca. 750.000 UNIX Installationen

und 4,5 Millionen Anwender. Die SVR3 war auch Grundlage für zahlreiche Abspaltungen in andere Derivate.

1989 gibt AT&T letztendlich System V Release 4 frei. Diese Release beinhaltet eine fast vollständige Neufassung des Kernels.

Ursachen dafür waren:

Dilemma der UNIX Anwender:

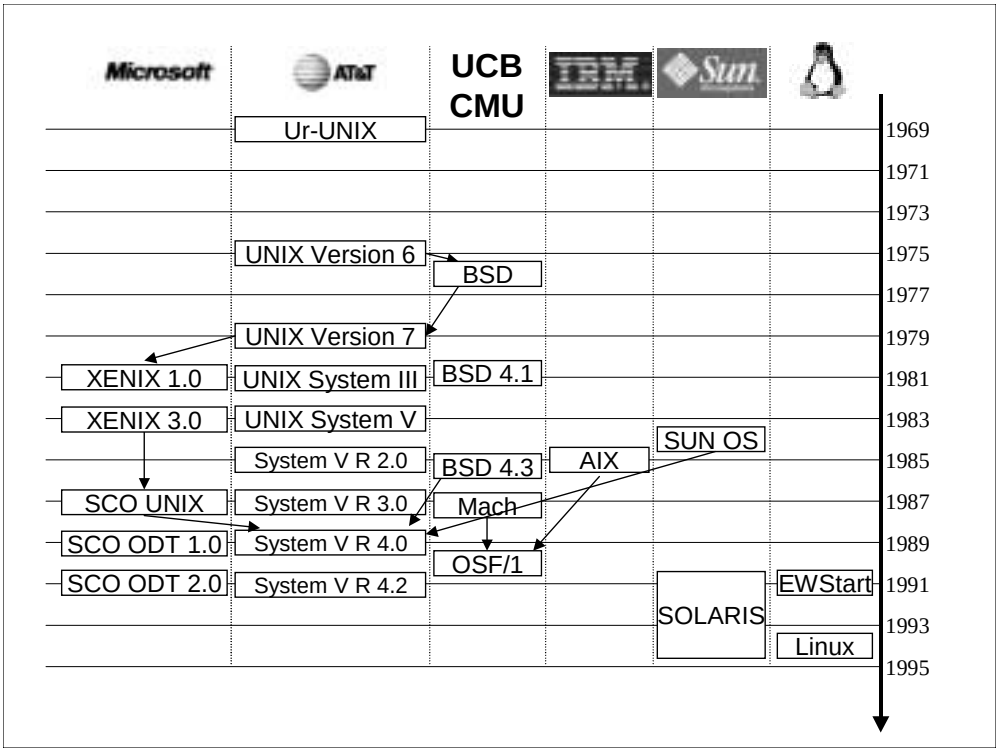
Welche Version oder welche Variante einsetzen? Obwohl als de facto Standard angesehen, ist es aber keine Grundlage für einen allgemeinen Standard.

In den späten 80er Jahren war UNIX Eigentum von AT&T, die Lizenzinhaber wurden zu einem großen Konkurrenten für AT&T's eigenem Geschäft. Sun Microsystems haben durch Erweiterungen in der BSD Variante den Markt für graphische Workstations erobert, XENIX lief eine Zeitlang auf mehr Maschinen als jede andere UNIX-Verison, MIT's Version des X-Window Systems sowie Sun Microsystems NFS (=Network File System) wurden auf BSD-Versionen entwickelt.

Schließlich wandten sich die Anwender an die Standardisierungsorganisationen, in der Hoffnung, dass eine Organisation endlich eine Version des UNIX System V definieren würde, die so an 4.xBSD, XENIX und System V angepasst wäre, dass sich der Anwender nicht mehr entscheiden müsse, welche UNIX Version er wählen sollte.

AT&T brachte die SVID (System V Interface Definition) heraus, das einen Standard für die UNIX Systembibliothek, sowie die Kommandoschnittstelle darstellen soll. Zusätzlich wurde eine Menge von Utilities und Programmen definiert, die überprüfen sollten, ob ein System zu SVID konform war. Hersteller durften ihr System nur dann als System V konform bezeichnen, wenn diese Verifikation positiv verlief.

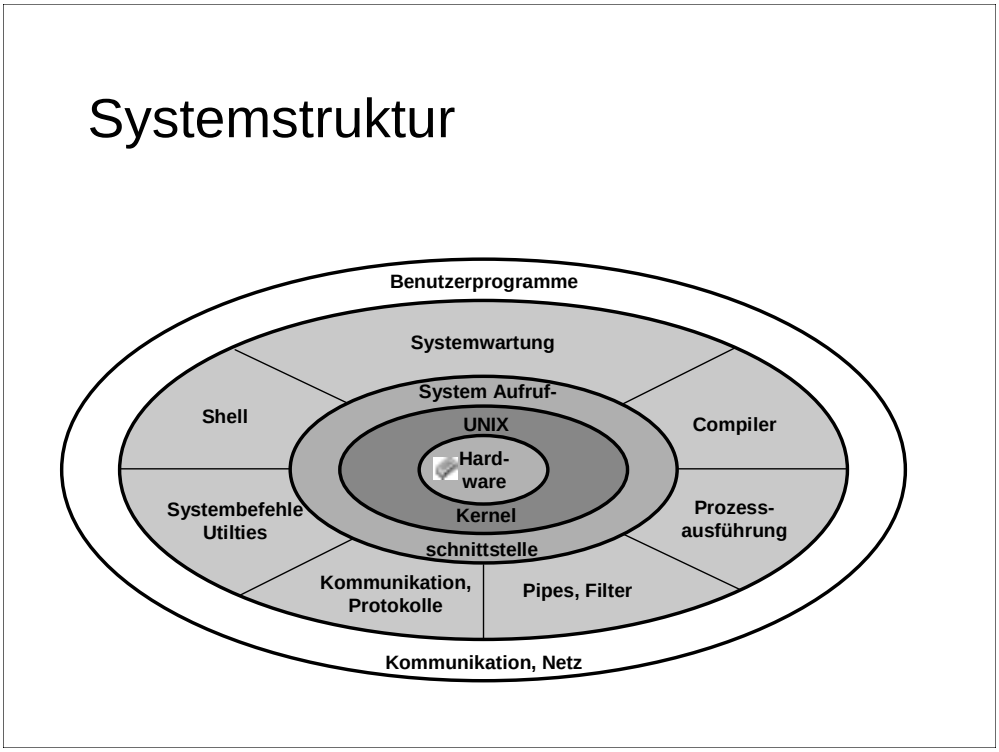
Die Hersteller hatten nun die Wahl, entweder ein angepasstes UNIX System zu verkaufen, das nicht alle Anforderungen der Anwender enthielt, oder ein System anzubieten, das nicht SVID konform war und bei den Anwendern die Angst auslöste, auf die UNIX-Version eines bestimmten Herstellers fixiert zu sein.



UNIX Derivate gibt es von verschiedensten Softwareherstellern. Die Urversion von UNIX wurde von AT&T im Jahre 1969 entwickelt. Die folgende Liste enthält einige bekannte UNIX Implementierungen.

AIX	UNIX Implementierung von IBM
BSD	UNIX Implementierung der University of California at Berkeley
HP-UX	UNIX Implementierung von Hewlett Packard
Mach	UNIX-Implementierung der Carnegie Mellon University
SCO-UNIX	Implementierung für Intel und Motorola Prozessoren, Nachfolger von XENIX
SINIX	Implementierung von Siemens
SUN OS, später Solaris	UNIX Implementierung von Sun Microsystems
ULTRIX, OSF/1	UNIX Implementierung von Digital Equipment, Nachfolger von Mach
XENIX	UNIX-Implementierung von Microsoft
LINUX	Freie Implementierung für unterschiedliche Systemarchitekturen

Systemstruktur



Ein Computersystem besteht aus Hardware, dem Betriebssystem und Anwendungsprogrammen. Es enthält üblicherweise einen Interpreter für Systemkommandos, der als Schnittstelle zum Betriebssystem zu verstehen ist. Zusätzlich werden alle Betriebssysteme mit Texteditoren, Compilern, Linkern, einem Programmlader und anderen Hilfsprogrammen ausgeliefert. Das Betriebssystem enthält ebenfalls eine Programmierschnittstelle, mit der Anwendungsprogramme auf Systemressourcen zurückgreifen können.

UNIX bildet hier eine Ausnahme, da die Anwenderschnittstelle nicht im Betriebssystem integriert ist. Es handelt sich um ein separates Programm, das beliebig ausgetauscht werden kann.

Hauptkomponenten des UNIX Systems

Hauptkomponenten

- UNIX-Kernel
  - Basisbetriebssystem
- UNIX-Systemaufrufschnittstelle
  - User Interface
  - Kommandozeileninterpreter (Shell)
- Kommandos und Werkzeuge
- UNIX-Systemdienste
- UNIX-Programmierschnittstelle

Die Hauptkomponenten eines UNIX-Systems sind:

1. Der UNIX-Kernel, der das Basisbetriebssystem darstellt.
2. Die UNIX-Befehlsschnittstelle, die meistens als Shell bezeichnet wird
3. Kommandos und Werkzeuge, die den Benutzer bei seiner Arbeit unterstützen
4. UNIX-Systemdienste, die zur Konfiguration und Administration des Systems dienen
5. UNIX Programmierschnittstelle die Zugriff auf Systemfunktionen erlaubt.

Die einzelnen Komponenten werden in weiterer Folge erklärt.

Befehlsinterpreter

- Meistens als Shell bezeichnet
- Schnittstelle mit Benutzer
- Eingebaute Programmiermöglichkeiten
  - Shell-Skripte
- Verschiedene Shells
  - C-Shell `csh`
  - bourne Shell `sh`
  - K-Shell `ksh`
  - bourne again Shell `bash`

Befehlsinterpreter

Der UNIX Befehlsinterpreter wird meistens als Shell bezeichnet. Es handelt sich um eine Schnittstelle, die seit der ersten Entwicklung nur unwesentlich verändert wurde. Über diese Schnittstelle kommuniziert der Benutzer mit dem Betriebssystem. Sie verfügt auch über eingebaute Programmiermöglichkeiten, die es erlauben Shellprogramme zu schreiben; die sogenannten Shellskripte. Derzeit finden etliche Shells Verwendung, die bekanntesten sind die C-Shell, die Bourne Shell, die K-Shell oder die Bourne Again Shell.

Es gibt aber auch andere Schnittstellen, die mit Hilfe von Fenstertechniken, Pop-up und Pull-down Menüs, Mausunterstützung, etc., direkt mit dem UNIX-Kernel interagieren.

Kommandos & Werkzeuge

- Zusatzprogramme mit UNIX ausgeliefert
- Nicht Bestandteil des Betriebssystem
- Beispiele
  - Compiler
  - Kommandos (ls, cp,...)
  - Shells

Kommandos und Werkzeuge

UNIX-Kommandos und Werkzeuge sind separate Programme, die zwar mit UNIX ausgeliefert werden, aber nicht Bestandteil des UNIX-Kernels sind.

Beispiele für solche Kommandos und Werkzeuge sind

- verschiedene Compiler, die zusammen mit dem Betriebssystem ausgeliefert werden,
- Kommandos wie ls, cp, etc. die bestimmte Funktionen des Betriebssystem aufrufen und ohne denen ein Arbeiten mit dem System kaum vorstellbar wäre,
- aber auch Shells sind nicht nur Befehlsinterpreter sondern auch Werkzeuge des Systems.

Systemdienste

- Hilfsprogramme
  - Systemadministration
  - Systemkonfiguration
  - Wartung des Dateisystems
  - Datenübertragung über UNIX Netzwerk
- Beispiele
  - Benutzermanagement
  - Systemtuning
  - Wartung des Dateisystems

Systemdienste

UNIX-Systemdienste sind Hilfsprogramme, die den Systemverwalter (Superuser) bei seiner Arbeit unterstützen.

Diese Dienste erlauben die Administration und Konfiguration des Systems, sind aber auch für diverse Wartungsarbeiten zuständig. Systemdienste kümmern sich darüber hinaus um die Datenübertragung über lokale und verteilte UNIX Netzwerke.

Systemdienste werden entweder vom System selbst, oder aber vom Systemadministrator genutzt. Aufgaben des Systemadministrators sind:

- Erstellung und Definition von Benutzereintragungen
- Systemtuning
- Wartung des Dateisystems.

Die Dienste zur Systemadministration sind sowohl als Systemaufrufe für Programmierer als auch als Kommandos implementiert.

Programmierschnittstelle

- Zugriff auf Systemfunktionen
- Ermöglichen Kommunikation mit UNIX-Kernel
- Anprogrammieren des Betriebssystems

Programmierschnittstelle

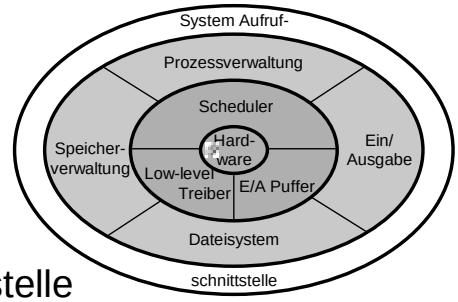
Die UNIX-Programmierschnittstelle erlaubt Anwendungsprogrammierern den Zugriff auf Systemfunktionen (*system calls*), die in Unterprogrammbibliotheken im Dateisystem eingerichtet sind.

Diese Schnittstelle ermöglicht als weitere Aufgabe die Kommunikation mit dem eigentlichen UNIX-Kernel. Dadurch ist es möglich, das Betriebssystem anzuprogrammieren und um eigene Funktionen zu erweitern. Die Interna der verwendeten Funktionen bleiben dabei verborgen und UNIX bietet keinerlei Möglichkeit ohne Systemschnittstelle Kernelfunktionen zu benutzen.

Der Kernel

Kernel$_1$

- Auf unterster Ebene angesiedelt
  - Kommuniziert direkt mit Hardware
- Aufgaben
  - Systeminitialisierung
  - Prozessverwaltung
  - Geräteverwaltung
  - Dateiverwaltung
  - Speicherverwaltung
  - Systemaufrufschnittstelle



Der UNIX-Kernel ist jener Teil des Systems, der auf unterster Ebene angesiedelt direkt mit der Hardware kommuniziert. Er ist der einzige Teil des Betriebssystems, der mit der Hardware direkt kommunizieren kann.

Der Kernel ist größtenteils in C implementiert, nur der hardwareabhängige Teil besteht aus einer Mischung aus C und prozessorspezifischen Assembler.

Zu den Aufgaben des Kernels zählen

1. die Systeminitialisierung,
2. die Prozessverwaltung,
3. die Geräteverwaltung,
4. die Dateiverwaltung,
5. die Speicherverwaltung und
6. die Systemaufrufschnittstelle.

Die einzelnen Aufgaben werden auf den nächsten Seiten detaillierter erläutert.

Kernel$_2$

- Systeminitialisierung
  - Möglichkeit des OS sich selbst zu starten und zu initialisieren
- Prozessverwaltung
  - Erzeugung, Beendigung, Steuerung von Prozessen
  - Verteilt Rechenzeit und Ressourcen an einzelne Prozesse

Systeminitialisierung

Das Betriebssystem muss eine Möglichkeit haben sich selbst zu starten und zu initialisieren. Hierbei wird das Betriebssystem in einen definierten Zustand gebracht, sodass der Benutzer damit interagieren und arbeiten kann. Die Systeminitialisierung beschäftigt sich mit diesen Aufgaben und besteht zum Hauptteil aus dem Bootvorgang.

Prozessverwaltung

Die Prozessverwaltung umfasst die Erzeugung, Beendigung und Steuerungen von Prozessen. Da UNIX ein Multitasking / Multiuser System ist, stellt jener Teil des Kernels sicher, dass jeder Prozess / Anwender gleichmäßig Rechenzeit und Ressourcen zur Verfügung gestellt bekommt. Für diesen Vorgang ist der sogenannte Scheduler zuständig, der jedem Prozess eine bestimmte Zeitscheibe zur Verfügung stellt.

Kernel$_3$

- Speicherverwaltung
  - Verwaltet belegte und freie Teile des Speichers
  - Speichermanagement für Prozesse
  - Ein- und Auslagern von Prozessen zwischen Platte und Hauptspeicher (*swapping*)
- Dateiverwaltung
  - Hierarchisches Dateisystem
  - Kernel stellt Funktionen zur Verwaltung des Dateisystems zur Verfügung

Speicherverwaltung

Die Speicherverwaltung verwaltet belegte und freie Teile des Speichers. Dabei werden Prozessen bei Bedarf Speicher zugewiesen oder entzogen wenn dieser nicht mehr benötigt wird. Darüber hinaus wird die Ein- und Auslagerung von Prozessen zwischen Platte und Speicher organisiert. Dieser Vorgang ist deswegen notwendig, weil im Hauptspeicher nicht alle Prozesse Platz finden und deswegen auf die Festplatte ausgelagert werden müssen, damit andere Prozesse in den Hauptspeicher geladen werden können. Den Vorgang nennt man auch Swapping.

Dateiverwaltung

Eine andere Funktion des Kernels ist die Dateiverwaltung. UNIX hat ein hierarchisches Dateisystem das – bestehend aus beliebig vielen Unterverzeichnissen – Textdateien, Objektdateien, ausführbare Dateien und speziellen Gerätedateien (die Hardwaregeräte repräsentieren) verwalten kann. Es stellt dafür wieder eine große Zahl von Systemfunktionen zur Verwaltung zur Verfügung.

Kernel$_4$

- Geräteverwaltung
  - Systeminitialisierung auf niedrigster Ebene (Urladen)
  - Fehler-, Interrupt- und Ausnahmebehandlung im System
  - Speicherverwaltung: Übersetzen von virtuellen in physikalische Adressen
  - Umschaltung zwischen Kernel- und Anwendermodus
  - I/O Gerätetreiber und -initialisierung durchführen
- Systemaufrufschnittstelle
  - Ermöglicht Prozessen auf Kerneldienste zurückzugreifen

Geräteverwaltung

Die Geräteverwaltung beschäftigt sich mit den maschinenabhängigen Komponenten des Kernels. Sie ist für die Systeminitialisierung auf niedrigster Ebene zuständig, was das Initialisieren der im System eingebauten Hardwarekomponenten einschließt. Die Geräteverwaltung sorgt auch für das Urladen des Systems – das erstmalige Laden des Kernels beim Einschalten des Systems. Darüber hinaus muss die Geräteverwaltung Mechanismen für die Fehler-, Interrupt- und Ausnahmebehandlung installierter Hardware gewährleisten.

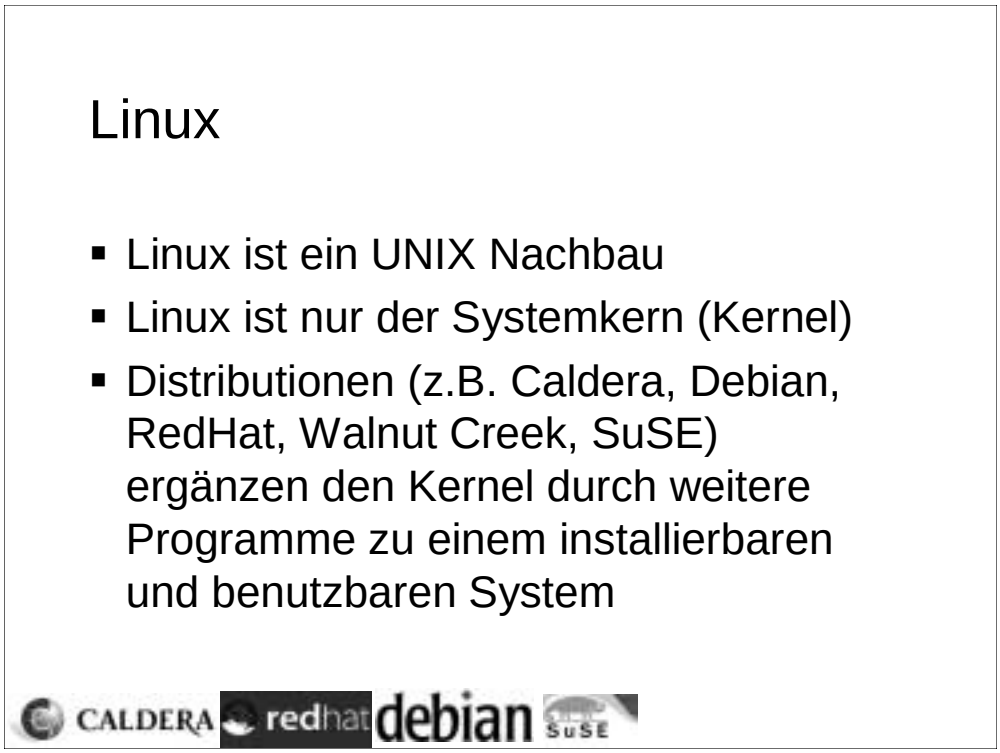
Wie schon im Bereich Speicherverwaltung erwähnt, stellt das System den einzelnen Prozessen mehr Speicher zur Verfügung als tatsächlich vorhanden ist. Das System stellt hier Prozessen *virtuelle* Speicherbereiche zur Verfügung. Diese bestehen aus physikalischem Arbeitsspeicher, aber auch nicht vorhandenen Speicher, der über Speicherauslagerung erreicht wird. Das System muss hierbei die zwei Speicherarten unterschieden und ist für die Übersetzung von virtuellen Speicheradressen in physikalische Speicheradressen verantwortlich.

Andere Aufgaben der Geräteverwaltung ist die Kontext-Umschaltung zwischen Kernel und Anwendermodus oder aber auch das Laden, Ansprechen und die Initialisierung von diversen Gerätetreibern.

Systemaufrufschnittstelle

Der letzte Teil des Kernels, die sogenannte Systemaufrufschnittstelle, ermöglicht Prozessen, auf Kerneldienste zurückzugreifen. Über diese Schnittstelle interagieren diverse Prozesse mit dem Kernel oder delegieren kernel-spezifische Aufgaben an die einzelnen, zuvor erwähnten Teile des Kernels.

Freie UNIX Systeme



Linux

Linux wurde 1991 von Linus Torvals, der damals als Student an der Universität in Helsinki Computerwissenschaften studiert hatte, entwickelt. Linus Torvalds hatte damals mit Minix, einer minimalen Implementierung von UNIX, gearbeitet. Nachdem er das Buch von Andrew Tanenbaum gelesen hatte, begann er mit der Implementierung eines UNIX ähnlichen Kernel. Sein Ziel war in erster Linie ein besseres Minix zu bauen, aber er suchte auch nach einem Betriebssystem, das die damals aktuelle Intel-Prozessorgeneration besser auszunutzen verstand als das übliche MS-DOS. Bereits ein Jahr später stellt er die Version 0.02 mit dem Namen Linux der Internetgemeinde zur Verfügung. Relativ schnell fanden sich viele Leute auf der ganzen Welt, die begeistert Tools und Programme für Linux entwickelten – nicht zuletzt war dies ausschlaggebend für die rasante Verbreitung von Linux.

Linux ist jedoch viel weniger, als man gemeinhin denkt. Eigentlich passt das System auf eine Diskette und beinhaltet nur den Systemkern selbst. Damit alleine kann man nicht viel anfangen, darum ist man schnell dazu übergegangen, so genannte Distributionen zusammenzustellen. Diese Distributionen enthalten neben dem eigentlichen Linux noch Software, damit Linux sinnvoll und benutzbar eingesetzt werden kann.

Linux behauptet von sich, auch auf uralten PCs effizient zu laufen. Immer wieder hört man davon, wie schnell Linux selbst auf Intels nostalgischen 80386 Prozessor laufen soll. Eine sinnvolle Verwendung solcher Rechner, vor allem wenn man bedenkt, dass graphische Oberflächen darauf meist nicht laufen (oder ermüdend langsam) und das Distributionen in ihrer Minimalinstallation doch meist 600 MB Festplattenspeicherplatz brauchen, sei jedoch dahingestellt.

Freie UNIX Systeme



- Stallmann und „Copyleft“
  - Software, die das Gütesiegel der GNU Public License (GPL) erwerben will, muss im Source Code vorliegen und darf ausdrücklich modifiziert sowie für jedermann frei verfügbar sein. Allerdings unterliegen alle Änderungen an GPL Software wiederum den Kriterien der GNU Public License.
- 1984 Richard Stallman
  - gründet Free Software Foundation (FSF) diese das GNU Projekt (GNU is Not Unix) ins Leben ruft. Diese hat das Ziel, ein freies UNIX System zu schaffen.

GNU

Wesentlich zur Verbreitung von Linux hat die zum damaligen Zeitpunkt bereits starke GNU Bewegung beigetragen, die es sich zum Ziel gesetzt hatte, frei verfügbare Software für UNIX-Betriebssysteme zu sammeln und zu schaffen. UNIX war damals ein unübersichtlicher Wald von Derivaten, deren Software nicht oder nur zum Teil miteinander vergleichbar waren.

Die GNU Bewegung wurde 1984 durch Richard Stallman ins Leben berufen. Damals gründete er die Free Software Foundation, die sich mit freier verfügbarer Software beschäftigte.

Hauptprodukte der GNU Bewegung sind die GPL, die GNU Public License, die der Erstellung und Verbreitung freier Software eine rechtliche Grundlage gibt und fast den Status eines Manifestes der Open Source Bewegung besitzt. Außerdem gab es zum Zeitpunkt der Entwicklung von Linux bereits einen der GPL unterstellten C-Compiler (gcc) mit dem Linux portiert wurde. Im Jahre 1993 wurde Linux unter die GNU Public License gestellt.

Die GNU Public License legt fest, dass Software im Quellcode vorliegen muss und der Urheber die Modifikation des Quellcodes ausdrücklich erlaubt. Darüber hinaus muss der Code für jedermann frei verfügbar sein. Allerdings müssen Änderungen an der Software wiederum der GNU Public License unterlegt werden.

Weitere freie UNIX Systeme

- FreeBSD / NetBSD / OpenBSD
  - Entstammen der BSD Linie
  - Dürfen rechtlich nicht UNIX genannt werden
    Lizenzbestimmungen der Berkeley University
  - Bestehen aus Kernel und Dienstprogrammen
  - Kleinere Entwicklungsteams als bei Linux

Andere freie UNIX Systeme

Linux ist zwar das bekannteste freie UNIX-System, keineswegs aber das einzigste. Weitere freie UNIX-Systeme entstammen der Berkeley Software Distribution (BSD), namentlich sind diese FreeBSD, NetBSD und OpenBSD.

Aus lizenzrechtlichen Bestimmungen der Berkeley University dürfen diese Systeme nicht UNIX genannt werden. Die Systeme bestehen aus einen Kernel und zahlreichen Dienstprogrammen, die einen sinnvollen Einsatz des Systems ermöglichen.

Da diese Distributionen aber nicht so großen Anklang gefunden haben wie die Linux-Welt, sind Entwicklungsteams und Lieferumfang erheblich kleiner.

Kapitel

2

Erste Schritte: Das UNIX-Betriebssystem

Dieses Kapitel stellt das UNIX Betriebssystem vor und zeigt den Umgang mit dem Betriebssystem.

Dieses Kapitel führt Sie in den Umgang mit dem UNIX-Betriebssystem ein. Inhalt des Kapitels ist die Vorstellung der Benutzeroberfläche, sowohl textbasiert als auch graphisch, die grundlegendsten Konzepte des UNIX-Betriebssystems und die Einführung in die wichtigsten Kommandos von UNIX.

Lehrinhalte und -ziele

Sie sollten nach diesem Kapitel in der Lage sein, sich an einem UNIX-System anzumelden um mit dem Betriebssystem interagieren zu können. Sie sollten mit der Benutzerschnittstelle vertraut sein und in der Lage sein, grundlegende Kommandos, die in diesem Kapitel vorgestellt werden, abzusetzen und deren Aufgabe zu verstehen. Darüber hinaus sollten Sie über Verzeichnisse und deren Eigenschaften Bescheid wissen.

Erste Schritte

Erste Schritte$_1$

- Multiuser System
  - Anmeldung als erster Schritt erforderlich
    - Schutz der persönlichen Daten
    - Authentifizierung durch Name und Passwort
  - Nach Anmeldung erscheint Prompt einer Shell und wartet auf Kommandos
  - Persönliche Arbeitsumgebung wird geladen
- Beenden der Arbeit
  - Benutzer muss sich vom System abmelden
    - „`logout`“ (nur in Login-Shell), „`exit`“ (Beendet aktive Shell und wechselt zur aufrufenden Shell)

UNIX ist ein Multiuser- und Multitasking System. Das bedeutet, dass in einem bestimmten Augenblick sowohl mehrere Benutzer auf einer UNIX-Maschine gleichzeitig arbeiten können, als auch dass jeder einzelne dieser Benutzer mehrere Programme aufrufen kann, die alle zur gleichen Zeit ausgeführt werden.

Anmelden

Damit jeder dieser Benutzer seine Daten vor dem Zugriff der anderen Benutzer schützen kann, muss man sich, bevor man mit einem UNIX-System arbeiten kann, erst einmal *anmelden*, dass heißt, einen Benutzernamen und das dazugehörige Passwort eingeben. Dadurch kann das System dem Benutzer seine persönliche Arbeitsumgebung (inklusive aller privater Daten) zur Verfügung stellen.

Nach dem Einschalten des Terminals bzw. nach Aufnahme der Verbindung mit dem UNIX-Rechner meldet sich das Betriebssystem mit der Aufforderung, sich zu identifizieren. Der Benutzer gibt darauf den ihm zugewiesenen Login-Namen ein. Dann erscheint die Abfrage des Passwortes. Nun muss der Benutzer sein Passwort eingeben. Das Passwort wird im Gegensatz zu den üblichen Eingaben nicht auf dem Bildschirm ausgegeben. Sie erkennen die erfolgreiche Anmeldung daran, dass die Eingabeaufforderung des Systems, der sogenannte Prompt erscheint. Das System erwartet nun die Eingaben von Kommandos, die dann abgearbeitet werden.

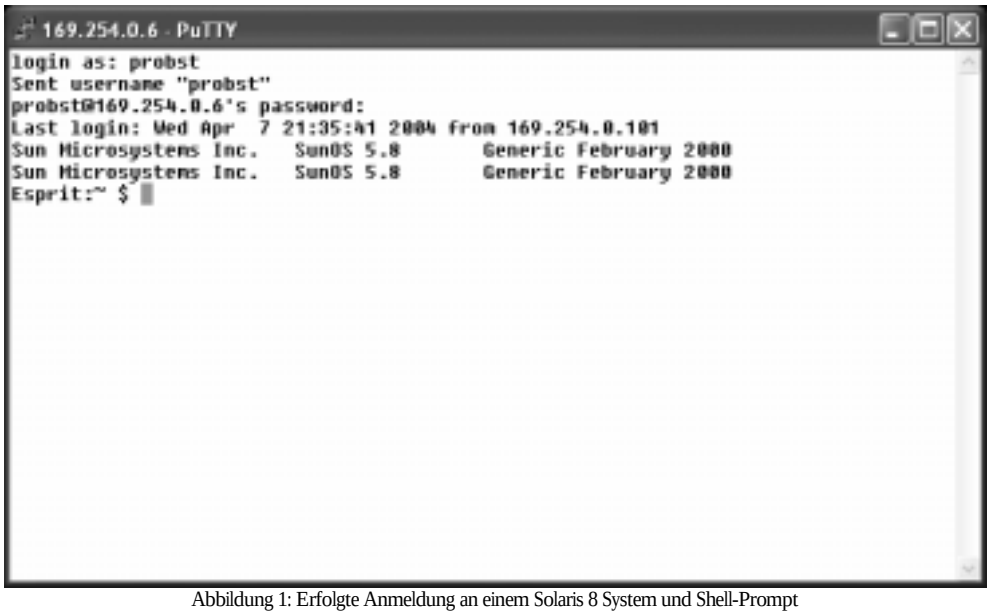


Abbildung 1: Erfolgte Anmeldung an einem Solaris 8 System und Shell-Prompt

Abmelden

Zum Beenden der Arbeit am Rechner muss sich der Benutzer abmelden. Dies geschieht je nach UNIX durch den Befehl „logout“ oder durch den Befehl „exit“ an der letzten aktiven Eingabeaufforderung. Eingabeaufforderungen – oder auch Shells genannt – können ineinander geschachtelt werden. Der Befehl „exit“ beendet hierbei die aktive Shell und wechselt zu der aufrufenden Shell. Wenn keine aufrufende Shell mehr vorhanden ist, wird der Benutzer vom System abgemeldet.

Erste Schritte$_2$

- User und Superuser
  - Normale User voneinander abgeschottet
    - Jeder besitzt eigenen Arbeitsbereich
  - Superuser (Sysadmin) darf alles
    - Zugriff auf Arbeitsbereich der Benutzer
    - Darf Systemdienste starten, die normale Benutzer nicht starten können
- Start und Stop des Systems
  - Vor Ausschalten müssen alle Prozesse terminiert werden und alle Dateien ordnungsgemäß geschlossen werden (Cache entleeren, usw.)
  - „`shutdown`“, „`halt`“, „`reboot`“, ...

Benutzerklassen

UNIX kennt zwei Benutzerklassen. Die normalen Benutzer sind voneinander abgeschottet, d.h. jeder hat seinen eigenen Arbeitsbereich. Der Superuser, normalerweise ist das der Systemadministrator, kann und darf alles. Er kann auf jede Datei zugreifen und Benutzer eintragen oder löschen. Er kann auch Systemdienste aufrufen, die normalen Benutzern verwehrt sind. Normalerweise hat der Superuser den Login-Namen „root“.

Start und Stop des Systems

Bei UNIX Systemen sollte der Rechner nicht einfach ausgeschaltet werden, wie das bei MS-DOS möglich war. Speziell beim Ausschalten müssen alle Benutzerprozesse zuvor abgeschlossen und alle Dateien ordnungsgemäß geschlossen sein. Das leistungsstarke Dateisystem ist beispielsweise mit einem Cache versehen, der Teile von gerade geöffneten Dateien im Speicher behält und erst auf die Platte schreibt, wenn das System gerade unausgelastet ist. Ein einfaches Ausschalten kann deshalb gravierende Auswirkungen haben - angefangen von Inkonsistenzen im Dateisystem bis hin zur Zerstörung des Dateisystems.

Linux und andere UNIX-Systeme werden mit dem Befehl „shutdown“ heruntergefahren. Neuere Systeme – vor allem aber ACPI[1]-kompatible Maschinen – unterstützen auch das Kommando „poweroff“, dass zusätzlich den Rechner ausschaltet.

[1] ACPI: Advanced Configuration and Power Interface, eine Spezifikation die dem Betriebssystem erlaubt, die komplette Kontrolle über Hardwareressourcen und der Stromverwaltung des Computers zu übernehmen.

Interaktion mit UNIX

User Interface

- Textzeilenbasiert
  - Shell
  - Verschiedene Shells
    - Unterscheiden sich nach Umfang und Funktionalität
    - Skript-Funktionalität (Shell Programme)
    - Bourne, Korn Shell , C-Shell, Bourne Again Shell (bash)
- Grafisch
  - X-Window
  - Verschiedene Window-Manager
  - Verschiedene Desktop Environments
  - „`startx`“ zum Starten des X-Window Systems

Die Interaktion mit dem UNIX-System kann auf verschiedene Weise erfolgen. In der Tat ist es so, dass die Benutzerschnittstelle selbst zwar im System definiert ist, aber nicht Teil des Systems ist. Das erlaubt das transparente Wechseln der Benutzerschnittstelle – sofern sich diese an den Standard hält.

Prinzipiell unterscheidet man in UNIX zwischen zwei Arten von Benutzerschnittstellen:

1. Textzeilenbasierte Benutzerschnittstelle

Die textzeilenbasierte Benutzerschnittstelle bietet meist nur einen simplen, einfarbigen Textbildschirm. Vereinfach gesagt, nimmt die Eingabeaufforderung – oder auch Shell genannt - Kommandos von der Standardeingabe entgegen und gibt die Ergebnisse auf der Standardausgabe aus. In Wirklichkeit werden aber eingegebene Kommandos zuvor von der Shell interpretiert und entsprechend verarbeitet. Innerhalb einer Shell können auch eigene Shellprogramme (Shellskripte) erstellt werden.

Es gibt eine Vielzahl verschiedener Shells, die sich durch Umfang und Funktionalität unterscheiden. Die wichtigsten sind:

- Bourne Shell: Diese ist der älteste Vertreter und ist de facto Standard.

- C Shell: Diese Shell erfreut sich außerordentlicher Beliebtheit bei eingefleischten C Programmierern, da sie mit C ähnlichen Programmiermöglichkeiten aufwartet.

- Korn Shell: Diese Shell gehört zur neueren Generation. Sie ist weitgehend kompatibel zur Bourne Shell, bietet aber erweiterte Möglichkeiten im Kommandoeingabenbereich.

- BASH (Bourne Again Shell): Diese Shell ist die Linux Standard Shell. Sie bietet erweiterte Editiermöglichkeiten bei der Kommandoeingabe und Kommandovervollständigung (passende Verzeichnisse oder Dateien werden durch drücken der TAB-Taste im Kommando eingefügt). In neueren Linux Systemen bietet die Shell auch eine farbige Darstellung unterschiedlicher Dateitypen.

2. Graphische Benutzerschnittstelle (GUI)

UNIX bietet auch die Möglichkeit einer graphischen Benutzerschnittstelle. Durch das Eingeben des Kommandos „startx“ wird X-Window – so heißt diese GUI – gestartet. Auch hier ist das Aussehen nicht starr definiert. Durch das Einsetzen verschiedener Window Manager oder Desktop Environments kann das Aussehen der GUI von Benutzer zu Benutzer stark variieren. In der Tat unterschieden sich heutzutage gebräuchliche GUI-Umgebungen kaum mehr von den gewohnten Windows GUIs.

Die Shell

Shell

- Nach erfolgreichem Login erhält Benutzer Shell (Kommandozeileninterpreter)
  - Prompt zeigt Benutzertyp
    - $: Normaler Benutzer
    - #: root / Superuser
    - [*username@host directory*]*usertypechar*
  - Erwartet Eingaben von Kommandos und ruft diese auf.
- Kommandoaufruf abhängig von aktuellem Verzeichnis

Sofern UNIX nicht im graphischen Modus gestartet wird, erhält jeder Benutzer nach einem erfolgreichen Login eine Shell. Welche Shell geladen wird, kann individuell für jeden Benutzer vom Systemadministrator festgelegt werden. Wenn die Shell auf Eingaben wartet, zeigt sie den sogenannten Prompt. Dieser ist abhängig von den Systemeinstellungen und kann natürlich geändert werden. Bei neueren UNIX und Linux-Systemen zeigt die Shell den Prompt im Format [*host:directory*]*usertypechar* (siehe Abbildung 1). Das letzte Zeichen (hier usertypechar genannt) gibt Aufschluss, welche Art Benutzer die Shell benutzt oder welche Art Eingabe erwartet wird.

Kommando-Eingabe:

- $: Dieses Zeichen weist auf einen Standardbenutzer hin.
- #: Dieses Zeichen weist darauf hin, dass der Superuser (root) gerade die aktuelle Shell benutzt

Erweiterte Eingabe:

- >: Dieses Zeichen weist darauf hin, dass die Shell weitere Eingaben zu einem Kommando erwartet.

Die Shell wartet nun auf Kommandos, interpretiert und bearbeitet diese. In welchem Kontext das Kommando jedoch aufgerufen wird bzw. die Auswirkungen des Kommandos hängen größtenteils (sofern nicht anders angegeben) auch vom aktuellen Verzeichnis ab, in dem das Kommando abgesetzt wurde. Verzeichnisse und deren Struktur werden nun in weiterer Folge behandelt.

Verzeichnisse in UNIX

Verzeichnisse$_1$

- UNIX hat ähnlich zu Windows ein hierarchisch aufgebautes Verzeichnissystem
  - Wurzelknoten (Root) wird durch „/“ gekennzeichnet (MS „\“)
  - Keine Laufwerksbuchstaben → Mounten von Dateisystemen
- Homeverzeichnis
  - Verzeichnis, in dem man sich nach Einloggen befindet.
  - Gehört vollständig Benutzer (alle Rechte)
- Aktuelles Arbeitsverzeichnis
  - Ist Verzeichnis, dessen Inhalt gerade bearbeitet wird
  - Kommandos beziehen sich auf dieses Verzeichnis

Auf den ersten Blick ähnelt das Verzeichnissystem von UNIX dem von Windows oder anderen Betriebssystemen. UNIX verfügt über ein hierarchisch aufgebautes Verzeichnissystem. Also gibt es eine Baumstruktur von verschiedenen Verzeichnissen (Directories), in der sich jede Datei irgendwo befindet.

Unterschiede zu MS-Windows

In UNIX werden einzelne Verzeichnisse durch normale Schrägstriche (,/') getrennt – im Gegensatz zur Microsoft-Welt, die Verzeichnisse durch den sogenannten Backslash (,\') voneinander trennt. Das Konzept der Trennung von Verzeichnissen durch den normalen Schrägstrich ist Ihnen vielleicht vom World Wide Web bekannt wo Verzeichnisse in der URL auch mit Schrägstichen getrennt werden.

UNIX besitzt im Gegensatz zu MS-Windows keine Laufwerksbuchstaben. Mehrere Platten und Partitionen können in UNIX an beliebiger Stelle in den Verzeichnisbaum eingehängt werden. Diese Technik erlaubt die Erweiterung bei knappen Plattenspeicher an genau der benötigten Stelle ohne lästiges herumkopieren.

ⓘ Dieses Konzept wurde mittlerweile ansatzweise auch in Windows 2000 übernommen und nennt sich „Dynamische Datenträger“.

Wichtige Verzeichnisse

Da keine Laufwerke vorhanden sind, muss es einen Ursprung oder eine Wurzel für alle Verzeichnisse geben. Dieses spezielle Verzeichnis ist einzig und allein durch den normalen Schrägstrich gekennzeichnet (‚/') und trägt den Namen ‚root'. Dieses Verzeichnis bildet den Stamm des Verzeichnisbaums.

Um die Struktur des Verzeichnissystems verstehen zu können, muss man auch andere spezielle Verzeichnisse kennen und deren Funktionalität verstehen.

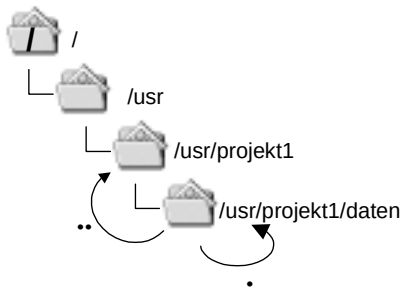
Das Homeverzeichnis ist das dem Benutzer zugeteilte Verzeichnis. Hier landet der Benutzer nach erfolgreichen einloggen. Es gehört dem Benutzer alleine und er kann damit anfangen, was er will. Normalerweise befindet sich das Homeverzeichnis in ‚/home/*username*').

Das aktuelle Arbeitsverzeichnis ist immer jenes, dessen Inhalt gerade bearbeitet wird. Das aktuelle Verzeichnis lässt sich jederzeit wechseln, aber es ist immer nur ein Verzeichnis zu einem bestimmten Zeitpunkt das Aktuelle und alle Kommandos beziehen sich dann auf dieses aktuelle Verzeichnis – sofern nicht anders in der Parameterliste angegeben.

In der Folge wollen wir uns nun mit der Struktur und dem Aufbau von UNIX-Verzeichnissystemen beschäftigen.

Verzeichnisse$_2$

- Pfadangaben
  - Dateien werden über Pfad spezifiziert
  - Absolute Pfadangabe
    - `/usr/projekt1/daten` (beginnt mit root!)
  - Relative Pfadangabe
    - `cd projekt1/daten` (muss in /usr stehen)
- Besondere Einträge
  - . Aktuelles Verzeichnis
  - .. Vaterverzeichnis



Struktur von UNIX Verzeichnissen

Jedes Verzeichnis im Verzeichnisbaum ist über einen Pfad zu erreichen. Der Pfad beschreibt, wie man zu dem gewünschten Verzeichnis hinkommt. Nicht nur Verzeichnisse, auch Dateien können über den Pfad spezifiziert werden. Befindet man sich z.B. gerade in seinem Homeverzeichnis (also ist das Homeverzeichnis der aktuelle Pfad), man möchte aber gerne aus dem Verzeichnis ‚/etc' die Datei foo.txt bearbeiten, so kann man diese im Kommando bei Angabe des Pfads ansprechen (z.B. „cat /etc/foo.txt").

Pfadangaben

Die Angabe des Pfades kann hierbei auf zwei Arten erfolgen

1. Angabe des absoluten Pfads

Die absolute Pfadangabe beginnt immer bei root. Es werden dann die Verzeichnisse in der Reihenfolge angegeben (getrennt durch ‚/'), die durchlaufen werden müssen, um zur Zieldatei zu gelangen.

2. Angabe des relativen Pfads

Die relative Pfadangabe geht bei der Pfadauflösung immer vom aktuellen Arbeitsverzeichnis aus. Sie müssen also beschreiben, wie das gewünschte Verzeichnis von dem Verzeichnis erreicht wird, in dem Sie sich gerade befinden. Dabei können Sie sich besondere Verzeichniseinträge, die in weiterer Folge beschrieben werden, zur Hilfe nehmen.

Besondere Verzeichniseinträge

Es gibt zwei besondere Verzeichniseinträge, die besonders hervorgehoben werden sollten. Bis auf das root-Verzeichnis besitzt jedes Verzeichnis die Einträge ‚.' und ‚..'. Das sind Stellvertreter für Verzeichnisnamen. Das Verzeichnis ‚.' verweist auf sich selbst und bezeichnet so das aktuelle Verzeichnis, ‚..' zeigt auf das übergeordnete Verzeichnis (Vaterverzeichnis). Um zum Vaterverzeichnis – also das Verzeichnis, dass das aktuelle Verzeichnis beinhaltet – zu wechseln, müssen Sie beim Verzeichniswechsel (durch das Kommando ‚cd', dass später in diesem Kapitel behandelt wird) ‚..' angeben.

! Es ist wichtig zu verstehen, das ‚..' ein spezielles Verzeichnis ist, welches auf das Vaterverzeichnis zeigt. Anders als bei MS-DOS (und auch MS-Windows) muss deswegen auf den Befehl cd (Change Directory) das Verzeichnis ‚..' (als Parameter des Kommandos) folgen. Ein Aufruf des Kommandos „cd.." würde zwar unter Windows funktionieren, unter UNIX jedoch fehlschlagen.

Zum Verständnis

Den Unterschied der beiden Pfadangaben soll ein Beispiel verdeutlichen:



Wir befinden uns im Verzeichnis ‚/usr'. Dieses Verzeichnis erreicht man, indem man von root ausgehend in das Verzeichnis usr wechselt. Wir wollen in das Verzeichnis daten, das sich in dem Unterverzeichnis projekt1 befindet.

Sie können mit dem Kommando ‚cd' und der absoluten Pfadangabe – also „cd /usr/projekt1/daten" – in das gewünschte Verzeichnis wechseln. Allerdings würde das auch funktionieren, wenn sie sich irgendwo anders im Verzeichnisbaum befinden.

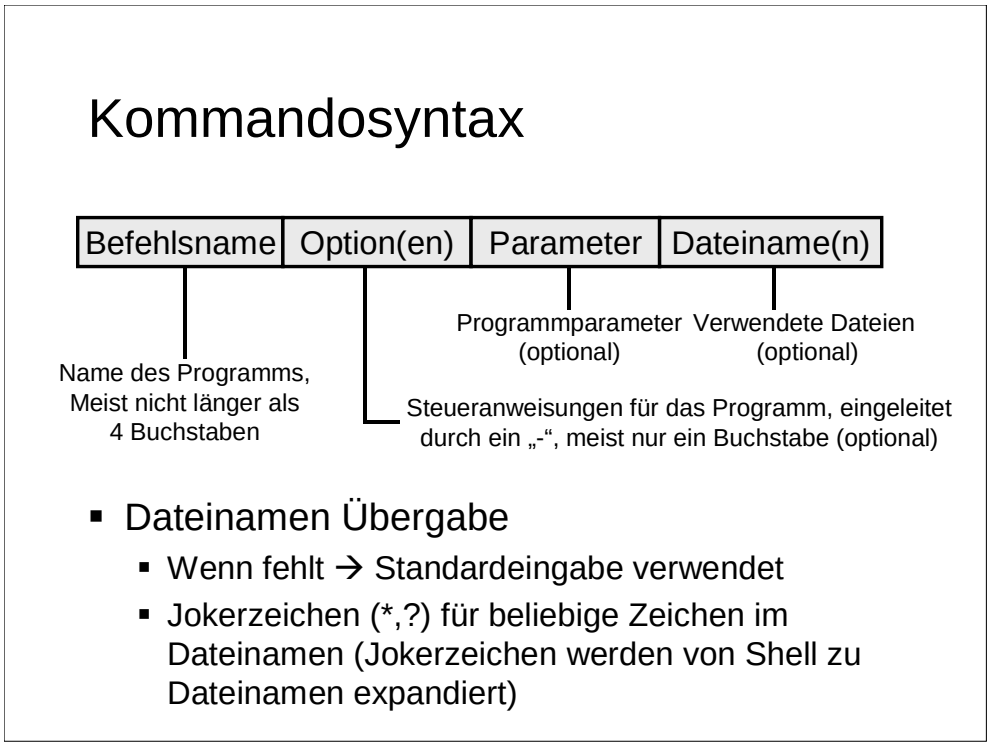
Sie können aber das Kommando auch mit der relativen Pfadangabe aufrufen – also „cd projekt1/daten". Hier *müssen* Sie sich aber unbedingt im Verzeichnis ‚/usr' befinden, sonst werden Sie das gewünschte Verzeichnis nicht erreichen.

Verständnisfrage

? Frage: Ist die Pfadangabe ‚./etc/rc.d' eine absolute oder eine relative?

Antwort: Eine relative! Eine absolute Pfadangabe geht immer von root direkt aus (also ‚/'). Hier wird aber relativ vom aktuellen Verzeichnis – angezeigt durch den Punkt – startend ausgegangen und das Unterverzeichnis ‚etc/rc.d' angesprochen.

UNIX Kommandos



Kommandos sind Befehle, die der Benutzer dem Betriebssystem erteilt. Diese Befehle werden dem Betriebssystem über die Shell durch die Eingabe solcher Kommandos erteilt. Die Eingabe wird durch das Drücken der *ENTER*-Taste abgeschlossen.

Alle Befehle von UNIX haben einen einheitlichen Aufbau:

- Befehlsname: Der Befehlsname gibt den Namen des Programms (Kommandos) an, dass gestartet werden soll. Dieser ist meist sehr kryptisch und nicht länger als vier Buchstaben.

- Option(en): Abhängig vom Befehl, *können* dem Befehlsnamen Optionen folgen. Optionen sind Steueranweisungen für das Programm und werden meist durch ein Minus (‚-') eingeleitet. Anzahl und Art der Optionen hängen vom Befehl ab. Optionen selbst bestehen meist nur aus einem Buchstaben. Mehrere Optionen können entweder durch die Angabe von seperaten Optionen oder durch das hintereinander stellen der einzelnen Optionen (nicht durch ein Leerzeichen getrennt) angegeben werden. Längere Optionsnamen werden durch ein doppeltes Minus (‚--') eingeleitet.

- Parameter: Manche Befehle erwarten Parameter, die Einfluss auf die Auswirkung des Befehles haben. Anzahl und Art der Parameter hängen vom Befehl ab.

- Dateiname(n): Fast jedem Kommando kann eine Liste von Dateien mitgegeben werden, auf die das Kommando dann angewendet wird. Fehlt die Dateiliste, wird in der Regel die Standardeingabe – normalerweise die

Tastatur – als Eingabedatei verwendet. Wie im Kapitel 6 (*Die Shell*) noch genauer gezeigt wird, verwendet UNIX die Jokerzeichen (Wildcards) Stern (‚*') und Fragezeichen (‚?'), um beliebige Zeichen innerhalb des Dateinamens zu kennzeichnen.

ⓘ Wildcards werden auch in DOS und Windows verwendet. Im Gegensatz zu DOS und Windows werden diese Wildcards aber nicht vom Programm sondern von der Shell zu Dateinamen expandiert. Deshalb sieht jedes UNIX-Programm nur eine mehr oder weniger lange Dateiliste.

! Kommandos sind in UNIX ausführbare Dateien. Diese Dateien liegen wie alle anderen Dateien auch in einem bestimmten Verzeichnis. Normalerweise weiß die Shell aber in welchem Verzeichnis sie diese Dateien findet (z.B. ‚/bin'). Dazu verfügt die Shell über sogenannte Umgebungsvariablen die das Verhalten der Shell beeinflussen. Kann aber ein Kommando nicht ausgeführt werden (weil z.B. die zugehörige Umgebungsvariable gelöscht wurde), so muss der voll qualifizierte Kommandoname (Verzeichnispfad abgeschlossen durch ein ‚/', dann direkt gefolgt vom Kommandonamen) angegeben werden (z.B. „/bin/mkdir *newdirectory*").

Im Folgenden werden nun die wichtigsten Kommandos vorgestellt, die für den Umgang mit UNIX unerlässlich sind.

Die wichtigsten UNIX-Kommandos

Kommandos$_1$

- man (Manual)
  - Gibt Erläuterungen zu Befehlen und weitere Hilfe zu den Kommandos
- pwd (Print Work Directory)
  - Gibt aktuelles Arbeitsverzeichnis aus
- cd (Change Directory)
  - Wechselt das aktuelle Verzeichnis

Im Folgenden werden nun die wichtigsten Kommandos aufgelistet und deren Funktionalität kurz erklärt. Dabei werden nur die wichtigsten Parameter und Optionen hier abgebildet. Detailliertere Informationen zu den Kommandos finden Sie in den entsprechenden Online Hilfeseiten (Manual Pages).

Manual Pages (/usr/bin/man)

Das Kommando man bietet Erläuterungen zu fast jedem Befehl des Betriebssystems an. Dazu müssen Sie den Befehl mit „man *Befehlname*“ aufrufen. Das Kommando „man man“ würde zum Beispiel die Online Hilfe zu den Manual Pages anzeigen. Auf einem Linux-System können Sie mit der Leertaste seitenweise blättern, die Enter-Taste zeigt jeweils nur die nächste Zeile an. Durch das Drücken der Taste ‚q’ werden die Manual Pages beendet und zur Eingabeaufforderung zurückgekehrt.

Print Work Directory (/bin/pwd)

Der UNIX-Verzeichnisbaum kann tief strukturiert sein und der Prompt gibt nur spärliche Informationen über das zur Zeit aktuelle Arbeitsverzeichnis. Abhilfe schafft der Befehl „pwd“, der den absoluten Pfad zum aktuellen Arbeitsverzeichnis am Bildschirm ausgibt.

Change Directory (Internes Kommando)

Um das aktuelle Arbeitsverzeichnis zu wechseln (also zur Navigation im Dateibaum) benötigt man das Kommando cd. „cd *Verzeichnis*“ wechselt in das angegebene Verzeichnis. Die Verzeichnisangabe kann hier absolut oder relativ erfolgen. Es können aber innerhalb der Verzeichnisangabe auch Abkürzungen oder Umgebungsvariablen verwendet werden. Wichtigster Vertreter von Abkürzungen

ist das Zeichen Tilde (‚~'), dass das Homeverzeichnis des Benutzers wiedergibt. Umgebungsvariablen sind in der Shell definiert und können durch die Angabe des ‚$'-Zeichens, gefolgt vom Namen der Umgebungsvariablen angesprochen werden. Die Shell ersetzt bereits vor dem Kommandoaufruf die Umgebungsvariable durch ihren definierten Wert.

Wird gar kein Verzeichnisname angegeben, wechselt die Shell in das Homeverzeichnis des Benutzers.

Beispiele:

cd /etc/rc.d	Wechselt in das Verzeichnis /etc/rc.d
cd rc.d	Falls sich die Shell gerade im Verzeichnis ‚/etc' befindet, wechselt sie nun in das Verzeichnis /etc/rc.d
cd ..	Wechselt zum Vaterverzeichnis
cd /	Wechselt zum root-Verzeichnis (allererstes Vaterverzeichnis)
cd ~	Wechselt in das Homeverzeichnis des Benutzers
cd ~/daten	Wechselt in das Unterverzeichnis daten im Homeverzeichnis des Benutzers
cd $HOME/daten	Wechselt in das Unterverzeichnis daten im Homeverzeichnis des Benutzers
cd	Wechselt in das Homeverzeichnis des Benutzers

Kommandos$_2$

- `ls [-Optionen] [pfadname]`
  - Listet Dateien und Verzeichnisse auf
  - Ohne Pfadname wird aktuelles Verzeichnis angezeigt
  - Optionen
    - -d Verzeichnisname anzeigen (statt Inhalt)
    - -a alle Einträge auflisten
    - -l Langform (alle Infos)
    - -i Inode-Nummer anzeigen
    - -n UID-, GID-Nummer anzeigen
    - -R alle Unterverzeichnisse auch anzeigen
    - -C mehrspaltige Ausgabe
    - -p „/“ nach Verzeichnisnamen anzeigen

Inhalt eines Verzeichnisses anzeigen (/bin/ls)

Das Kommando „ls“ dient zum Auflisten von Dateien und Verzeichnissen, mit dazugehörigen Informationen. Durch Parameter wird die Ausgabe gesteuert. Die Angabe eines Pfadnamens legt fest, welches Verzeichnis aufgelistet werden soll. Hier kann wieder ein relativer oder absoluter Pfadname stehen. “ls“ ohne Pfadname listet das aktuelle Verzeichnis auf, sonst muss der Pfad angegeben werden („ls /“ würde z.B. das Wurzelverzeichnis anzeigen).

Im Folgenden werden die wichtigsten Parameter des ls-Kommandos aufgelistet:

-d	Listet den Verzeichnisnamen anstatt des Verzeichnisinhaltes auf
-a	Zeigt alle Einträge, auch „versteckte“ Dateien
-l	Langform (alle Angaben, detaillierte Informationen)
-i	Inode-Nummer angeben
-n	Zeigt anstatt des Besitzers und der Gruppe die jeweilige ID (UID, GID) an
-R	zeigt auch alle Unterverzeichnisse und deren Inhalt an
-C	mehrspaltige Ausgabe (sinnvoll bei Kurzform)
-p	‚/‘ nach Verzeichnisnamen anzeigen



In DOS und MS-Windows können Dateieigenschaften (wie z.B. versteckte Dateien, Systemdateien) mittels Attributen gesetzt werden. Diese Möglichkeit kennt UNIX nicht. Hier gilt aber prinzipiell, dass Dateien die mit einem Punkt (‚.‘) beginnen spezielle Dateien sind. Diese Dateien werden bei einem standardmäßigen ls-Kommando nicht ausgegeben. Wenn Sie diese Dateien auch angezeigt haben möchten, müssen Sie das ls-Kommando mit der Option ‚-a‘ aufrufen.

Kommandos$_3$

- `passwd` (Passwort ändern)
  - Interaktives Ändern des Passwortes
  - Passwörter werden bei Eingabe nicht am Bildschirm angezeigt
  - Hash-Funktion zum Verschlüsseln (crypt)
- `who` (Liste aktiver Benutzer)
  - Zeigt eine Liste mit im System aktiven Benutzern (Login-Name, Terminal, Datum / Uhrzeit der Anmeldung,...)
  - „`who am i`“ gibt Eigeninformationen aus

Passwort ändern (/usr/bin/passwd)

Mit dem Kommando „passwd“ können Sie Ihr Passwort ändern. Dies geschieht interaktiv. Zunächst müssen Sie das bisherige Passwort eingeben. Bei der Eingabe der Passwörter werden diese nicht auf dem Bildschirm angezeigt. Damit Eingabefehler abgefangen werden können, ist das neue Passwort zweimal einzugeben.

Dem Superuser stehen weitere Optionen des passwd-Programms zur Verfügung. So kann er als einziger Passwörter anderer Benutzer verändern, indem er am Ende des Kommandos den Benutzernamen eingibt, dessen Passwort geändert werden soll. Auch kann der Superuser mit Hilfe dieses Kommandos ein Benutzerkonto kurzfristig sperren bzw. die Sperre wieder aufheben.

ⓘ „passwd –l user“ würde das Benutzerkonto user sperren, indem als Passwort ein ungültiger Wert eingetragen wird. In Wirklichkeit wird dem aktuellen Hash-Wert des Passworts ein ‚!’-Zeichen vorangestellt. Somit kann der Wert durch die Hash-Funktion nicht mehr erreicht werden.

„passwd –u user“ hebt die Sperre auf, indem das vorangestellte ‚!’-Zeichen wieder gelöscht wird. Somit ist das alte Passwort durch die Hash-Funktion wieder erreichbar.

Der Passwort-Schutz unter UNIX

Alle Benutzerkonten sind durch ein Passwort geschützt. Dieses Passwort wird aber nicht im Klartext gespeichert, da dies viel zu unsicher wäre. Vielmehr wird eine sogenannte Hash-Funktion benutzt. Dies ist eine mathematische Funktion, die aus dem eingegebenen Passwort eine mehrstellige Zahl errechnet. Der Clou bei der

Sache ist, dass die Funktion nur in eine Richtung funktioniert. Fängt man z.B. irgendwie den errechneten Hash-Code ab, so kann man daraus nicht wieder auf das Passwort schließen.

Bei der Authentifizierung durch den login-Prozess wird neuerlich der Hash-Code des eingegebenen Passworts berechnet. Dieser wird dann mit dem gespeicherten Hash-Code verglichen. Stimmen beide Hash-Codes überein, wird der Zugriff gewährt, ansonsten verweigert, da ein falsches Passwort eingegeben wurde.



Dieses Vorgehen – übrigens auch verwendet unter Windows NT aber mit einem anderen Algorithmus – impliziert, dass die Funktion der Hash-Code Berechnung bekannt ist. Das erlaubt sogenannte Brute-Force-Attacken. Hier wird die Funktion einfach so lange mit variierenden Zeichenketten aufgerufen, bis die Hash-Codes übereinstimmen. Es ist aus diesem Grunde äußerst wichtig, die Datei „/etc/shadow“ vor externen Zugriffen zu schützen, da diese die Hash-Codes der einzelnen Benutzer beinhaltet. Nur wenn dem Angreifer die Hash-Codes nicht bekannt sind, kann eine Brute-Force-Attacke verhindert werden.

Liste aktiver Benutzer (/usr/bin/who)

Das Kommando „who“ gibt an, wer gerade im System eingeloggt und aktiv ist. Zu jedem aktiven Benutzer wird der Login-Name, das Terminal an dem der Benutzer sitzt, Datum und Uhrzeit der Anmeldung angezeigt. Wenn man Informationen zum eigenen Benutzeraccount haben will (z.B. an welchem Terminal sitze ich?), dann retourniert das Kommando „who am i“ die gewünschten Informationen.

Kommandos$_4$

- `tty` (Teletype)
  - Gibt den Namen des aktuellen Terminals (Device-Treibers) aus.
- `finger` (Information über Benutzer)
  - Zeigt Informationen zu einem Benutzer an
  - Ohne Parameter: Liste mit allen Benutzern
  - `finger user`: Detaillierte Informationen zu Benutzer

Namen der Konsole anzeigen (/usr/bin/tty)

Das Kommando tty gibt den Namen des aktuellen Terminals (eigentlich jenen des Device-Treibers) aus. Direkte Ausgabe auf dem Bildschirm können auf dieses Device geleitet werden. Das Kommando unterschiedet auch zwischen den virtuellen Konsolen oder einer X-Window Terminal Session.

Information über Benutzer anzeigen (/usr/bin/finger)

Das Kommando finger zeigt Informationen zu einem Benutzer an. Dieser Benutzer muss nach dem Kommando angegeben werden (z.B. finger root). Wird kein bestimmter Benutzer angegeben, so wird eine Liste mit allen Benutzern im System ausgegeben.

Graphische Benutzeroberfläche in UNIX

Grafische Benutzeroberfläche

- X-Window
  - Netzwerktransparente, geräteunabhängige, fensterorientierte Benutzeroberfläche
- Window Manager (z.B. Enlightment)
  - Definiert Verhalten von Fenstern
  - Fängt Nachrichten an X-Server ab und ändert Daten entsprechend seines Layouts
- Desktop Environments (z.B. KDE, Gnome)
  - Bieten Arbeitsoberfläche
  - Design / Look and Feel einer Oberfläche
  - Meistens inkl. Zusatzprogramme

Als graphische Benutzeroberfläche unter UNIX dient X-Window. Standard in der UNIX-Welt ist das X11-System.

Geschichtliche Entwicklung von X-Window

Der Ursprung von X11 war das Fenstersystem W der Stanford University, 1984 wurde es am Massachusets Institue of Technology (MIT) mit Förderung verschiedener Hardwarehersteller zum X-Window System weiterentwickelt. 1987 wurde für die Verwaltung und Weiterentwicklung das X-Konsortium gebildet, dem fast alle bedeutenden Computerhersteller angehören.

Ziel der Entwicklung von X11 war es, Grafikausgaben in heterogenen Netzwerkumgebungen auf verschiedene Displays verteilen zu können. X11 ist netzwerkfähig und funktioniert nach dem Client-Server-Modell. Ein X-Programm (ein Client) kann irgendwo auf einem Rechner im Netzwerk laufen.

Architektur

X-Window ist in funktionelle Schichten zerlegt. Diese sind:

- X-Server: Der X-Server läuft oberhalb des UNIX-Systems als eigenständiger Prozess, verwaltet die auf der Maschine vorhandene Graphikhardware und kontrolliert die gesamte Eingabe und Ausgabe des Geräts (Tastatur, Maus, Graphikhardware), ist also eine Art Vermittler zwischen Hard- und Software.

- X-Client (alle Anwendungen unter X): Im Einzelnen übermittelt der X-Client X-Protokollelemente an den Server, die darzustellende

Informationen repräsentieren. Ein X-Client steuert ein Hauptfenster und gegebenenfalls mehrere, hierarchisch angeordnete Unterfenster (Scrollbar, Slider, Button, Textfeld, usw.). X-Clients sind eigenständige Prozesse, die auch auf einem anderen System laufen können als der X-Server. Der X-Server liefert an den X-Client Protokollelemente, die die am X-Server getätigten Eingaben übermitteln. X-Clients sind hinsichtlich ihrem äußeren Erscheindungsbild hochgradig konfigurierbar.

- X-Protokoll: Das X-Protokoll, auch X11 Protokoll genannt, dient der Kommunikation zwischen X-Server und X-Clients.

Window Manager

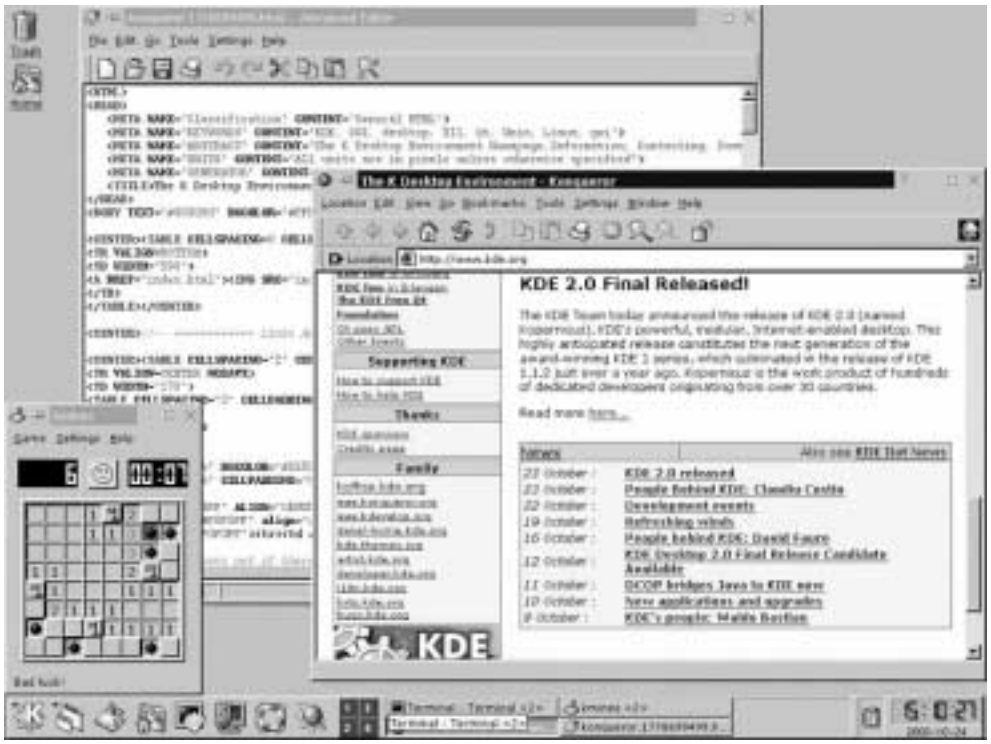
Für das äußere Erscheinungsbild einzelner Clients auf dem Arbeitsplatz ist der Window-Manager verantwortlich, eine mit besonderen Rechten ausgestattete X-Anwendung, die Position und Abmessungen sowie das Aussehen der Fensterrahmen aller an einen X-Server adressierten Fenster steuert. Wesentlich ist dass maximal ein Window-Manager das Erscheinungsbild eines Arbeitsplatzes prägen kann. Läuft dieser auf demselben System wie der X-Server, redet man von einem lokalen Window-Manager. Für das X-Winow System ist der Window-Manager lediglich ein weiterer, eigenständiger X-Client. Dieser Client fängt nur die Nachrichten der Clients an den X-Server ab und ändert deren Daten entsprechend seines Layouts.

Desktop Environments

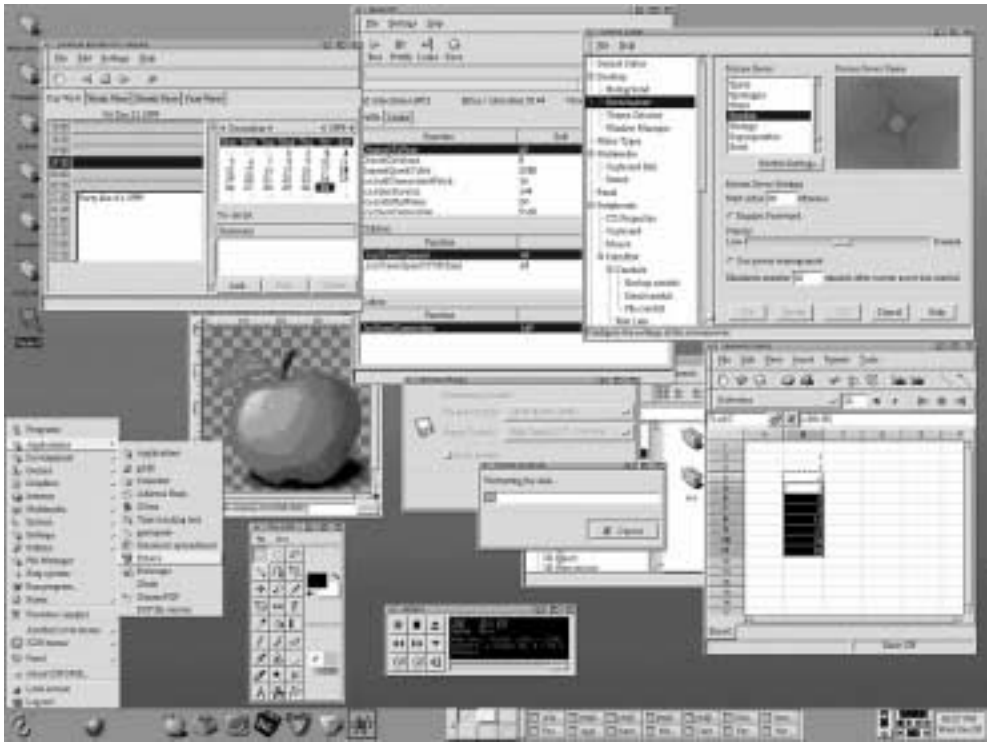
Desktop Environments arbeiten mit Hilfe eines Window Manager und bieten dem Benutzer eine einheitliche Arbeitsoberfläche. Das Desktop Environment definiert das Look and Feel der Benutzeroberfläche. Damit wird gewährleistet, dass alle X-Clients das gleiche Look and Feel haben und der Benutzer somit bei seiner Arbeit mit dem X-Window System unterstützt wird. Meistens haben Desktop Environments Zusatzprogramme, die den Benutzer das Arbeiten erleichtern und auch aus der Windows-Welt bekannt sind (z.B. Taschenrechner, Grafikprogramme, Spiele).

Die zwei populärsten Desktop Environments sind das K-Desktop Environment (KDE) und GNOME. Welches standardmäßig installiert wird, hängt von der Distribution ab. So wird bei RedHat z.B. standardmäßig GNOME installiert, während SuSE das KDE bevorzugt. Bei beiden Systemen kann aber das jeweils andere Desktop Environment nachinstalliert und benützt werden.

Das K-Desktop Environment (KDE)



GNOME



Kapitel

3

Das UNIX-Dateisystem

Dieses Kapitel stellt das UNIX Dateisystem und die dahinterliegenden Konzepte vor.

Dieses Kapitel behandelt das Dateisystem von UNIX. Es werden die Konzepte und die Struktur des Dateisystems erklärt, genauso wie die einzelnen Dateitypen die es in UNIX gibt. Darüber hinaus werden Kommandos zur Manipulation von Dateien behandelt und das Konzept der Links erklärt. Danach wird gezeigt, wie verschiedene Dateisysteme eingebunden werden können und das Dateisystem erweitert werden kann. Abschließend wird das Plattenmanagement behandelt und die damit verbundenen Kommandos.

Lehrinhalte und -ziele

Sie sollten nach diesem Kapitel die Konzepte des UNIX Dateisystems verstanden haben. Darüber hinaus sollten Sie in der Lage sein, das Dateisystem zu verwalten, Dateien zu manipulieren und neue Dateisysteme erstellen und in den Verzeichnisbaum einhängen zu können. Sie sollten auch das Konzept der Links verstanden haben und wissen, welche Eigenschaften die verschiedenen Arten der Links haben und zu welchen Problemen es bei Links kommen kann.

Einführung in das Dateisystem

Einführung

- Hierarchische Struktur
- Linear durchnummerierte Blöcke
- Zugriff auf einzelne Dateien und Verzeichnisse bereits vom System geregelt
  - Jede Datei / jedes Verzeichnis hat Besitzer
  - Besitzer hat alle Rechte, kann Rechte gewähren

Das UNIX Dateisystem

Die Dateisysteme von UNIX haben über zwanzig Jahre Entwicklung hinter sich und dienten als Vorbild für viele andere Betriebssysteme. Den ersten Implementierungen aus den Entwicklungslabors von AT&T mangelte es zwar noch an Performance, doch das Konzept hat sich in der Praxis als extrem tragfähig erwiesen. Bei einer Generalüberholung vor ungefähr 15 Jahren an der Universität von Berkeley entstand dann das BSD Fast Filing System, das auch heute noch viele Workstations verwenden und dessen Ideen sich auch in den Dateisystemen anderer Betriebssysteme wiederfinden.

UNIX abstrahiert auch die zugrundeliegende Hardware und schlägt sich allenfalls auf unterster Betriebssystemebene mit Angaben zur Plattengeometrie wie Köpfen oder Sektoren herum. Seine höheren Schichten betrachten eine Festplatte dagegen als lange Reihe von Blöcken, die linear durchnummeriert sind. Die Umrechnung der linearen Blockadressen in Zylinder, Kopf und Sektor ist entweder Aufgabe eines entsprechenden Gerätetreibers oder des Controllers der Harddisk.

Einführung in das Dateisystem

Das Dateisystem ist hierarchisch strukturiert und in Form eines Baumes aufgebaut. Durch diese Struktur gibt es einen eindeutig definierten Wurzelknoten (Vater aller Unterknoten), der auch als root-Verzeichnis bekannt ist.

Da UNIX ein Multiuser-System ist, muss auch der Zugriff auf einzelne Dateien und Verzeichnisse vom System so geregelt werden, dass kein Benutzer die Daten der anderen Benutzer manipulieren oder vertrauliche Daten unbefugterweise einsehen kann. Das wird dadurch gewährleistet, dass jede Datei und jedes

Verzeichnis einen *Besitzer* (User) hat, der mit Hilfe bestimmter Befehle festlegen kann, welcher der anderen Benutzer des Systems auf welche Weise auf seine Dateien zugreifen darf. Die meisten Dateien und Verzeichnisse in einem UNIX-System gehören naturgemäß dem Systemverwalter und jeder Benutzer kann auf diese Dateien nur lesend zugreifen, sie aber nicht verändern. Beispiele hierfür sind z.B. die zahlreichen Kommandos, die das System zur Verfügung stellt.

Aufgaben des Dateisystems

Aufgaben$_1$

- Datenintegrität
  - Sicherstellung der Konsistenz jeder Datei
  - Festlegung, welche Dateien im Hauptspeicher und welche auf der Platte gespeichert sind
  - Beobachten, welche Prozesse eine Datei öffnen und schließen
- Dateiadministration
  - Beobachten, welche Datenblöcke eine Datei auf der Platte belegt
  - Verwalten der Platten, Partitionen und andere Dateisysteme in denen Daten gespeichert sind

Das UNIX Dateisystem hat mehrere verschiedene Aufgaben. Dazu gehören:

1. Datenintegrität

Die Datenintegrität stellt sicher, das die Konsistenz jeder Datei gewährleistet ist. Dazu gehört die Festlegung, welche Dateien im Hauptspeicher und welche auf der Platte gespeichert sind. Dies ist vor allem deswegen wichtig, da aus Performancegründen Dateien für eine gewisse Zeit in einem Cache-Speicher (hoch-performanter Zwischenspeicher) gehalten werden. Damit können in der Regel viele einzelne Plattenzugriffe gespart werden. Erst wenn die Datei länger nicht mehr bearbeitet wurde oder geschlossen wurde, wird sie tatsächlich auf die Platte gespeichert. Dieses ständige Verändern der Dateien muss natürlich auch beobachtet werden, damit immer Informationen vorliegen, welche Prozesse welche Dateien gerade bearbeiten, öffnen oder schließen.

2. Dateiadministration

Die Dateiadministration beschäftigt sich mit der Verwaltung der Datenblöcke. Dabei wird beobachtet welche Datenblöcke eine Datei auf der Platte belegt. Wird die Datei gelesen, muss von der Dateiadministration dafür gesorgt werden, dass auch die richtigen Blöcke in den Hauptspeicher gelesen werden.

Darüber hinaus kümmert sich die Dateiadministration auch um die Verwaltung der einzelnen Platten im System, deren Partitionen und unterschiedlicher Dateisysteme.

Linux erlaubt aber auch die Einbindung heterogener Dateisysteme (nicht UNIX-Dateisysteme). Diese müssen auch von der Dateiadministration verwaltet werden.

Aufgaben$_2$

- Zugriffsberechtigungen
  - Sicherstellen, dass ein Prozess, der auf eine Datei zugreifen will, die entsprechende Berechtigung hat
- Systemaufrufüberwachung
  - Koordination der Systemaufrufe, die zwischen Anwenderprozess und Kernel aufgerufen werden

3. Zugriffsberechtigungen

Wie schon eingangs erwähnt gibt es für alle Dateien Zugriffsberechtigungen, die Dateien vor unerlaubten Zugriff schützen. Darum muss das Dateisystem dafür Sorge tragen, dass ein Prozess, der auf eine Datei zugreifen will, auch über entsprechende Zugriffsberechtigungen verfügt. Dieser Dateischutz sollte robust und unumgänglich sein - lediglich der Systemadministrator sollte alle Dateien einsehen können.

4. Systemaufrufüberwachung

Die Systemaufrufüberwachung ist zuständig für die Koordination der Systemaufrufe, die zwischen Anwenderprozess und Kernel aufgerufen werden. Immerhin wissen Anwenderprozesse ja nichts über die zugrundeliegende Hardware und müssen darum Anfragen und Anforderungen an den Kernel stellen, der diese dann der Hardware weitergibt und die Ergebnisse den Prozessen liefert.

Aufgaben$_3$

- Koordination des Dateizugriffs
  - Verwaltung einer Liste von geöffneten Dateien
    (wenn mehrere Prozesse Operationen auf derselben Datei ausführen)
- Datei- und Datensatzsperre
  - Steuerung von Sperrungen auf bestimmte Datensätze zwischen kooperierenden Prozessen

5. Koordination des Dateizugriffs

Da UNIX ein Multitasking System ist, ist es möglich, dass mehrere Prozesse versuchen, eine Datei zur selben Zeit zu öffnen. Diese Dateizugriffe müssen koordiniert werden, da eine fehlende Koordination fatale Auswirkungen haben kann. Wenn z.B. ein Prozess gerade eine Datei zum Lesen geöffnet hat und ein zweiter Befehl zur selben Zeit den Auftrag hat, die Datei zu löschen, würde der erste Prozess beim nächsten Leseversuch scheitern. Die Koordination des Dateizugriffs verwaltet eine Liste, die gerade geöffnete Dateien beinhaltet. Besteht nun die Anforderung eines Prozesses, eine bestimmte Datei zu manipulieren, kann das Dateisystem in dieser Liste nachsehen, ob nicht schon ein anderer Prozess die Datei geöffnet hat.

Nur im Zusammenhang mit der Datei- und Datensatzsperre kann sichergestellt werden, dass die Datei in einem konsistenten Zustand bleibt.

6. Datei- und Datensatzsperren

Wenn mehrere kooperierende Prozesse auf eine Datei zugreifen, so kann es hier zum Konsistenzverlust der Datei kommen. Nehmen wir einmal an, zwei Prozesse kommunizieren miteinander über eine Datei. Prozess 1 liefert dabei aktuelle Datenwerte, Prozess 2 verarbeitet diese und schreibt das Ergebnis seiner Verarbeitung wieder in die Datei. Prozess 2 liest nun die Datenwerte aus und startet die Berechnung. Prozess 1 speichert jedoch, kurz nachdem Prozess 2 die Daten gelesen hat, aktuellere Daten in die Datei. Prozess 2 ist mit seiner Berechnung fertig und schreibt das Ergebnis nun mit falschen Berechnungsdaten in die Datei, da sich die Ausgangswerte bereits verändert haben. Würde die Datei nun an einen anderen

Prozess weitergegeben werden, der beide Daten benötigt, würde dieser mit falschen Daten arbeiten - die Datei befindet sich in einem inkonsistenten Zustand.

Deswegen steuert das Dateisystem sogenannte Sperren auf bestimmte Datensätze zwischen kooperierenden Prozessen. Ein Prozess kann nur dann einen Datensatz einer Datei lesen, wenn sich auf diesem keine Sperre befindet - andernfalls muss er warten, bis die Sperre aufgehoben wird.

Verschiedene Dateitypen

Benutzersicht (ls –la)

Zeichen	*Typ*	*Zweck*
-	File	Normale Datei
b	Block device	Blockorientierte Gerätedatei
c	Character Device	Zeichenorientierte Gerätedatei
d	Directory	Verzeichnis
l	Symlink	Querverweis
p	Pipe	Benannte Pipe
s	Socket	Netzwerkverbindung

UNIX basiert sehr stark auf Dateien. In der Tat sind auch diverse Geräte als Dateien im System abgebildet. Das Kommando "ls -la" zeigt detaillierte Informationen zu den Dateien eines Verzeichnisses. Dabei werden in der ersten Spalte der Ausgabe Angaben zu den Dateieigenschaften gemacht. Das erste der 10 Zeichen gibt Informationen über den Dateityp, die anderen 9 Zeichen zeigen die gültigen Dateiberechtigungen an.

In UNIX gibt es mehrere verschiedene Dateitypen, deren Bedeutung in weiterer Folge erklärt wird.

Dateitypen$_1$

- Normale Dateien
  - Dateien im herkömmlichen Sinn (Bytefolgen), keine Datei-Header!
  - Sequentiell oder wahlfreier Zugriff
- Verzeichnisse
  - Strukturierung des Dateisystems
  - Werden wie Dateien auf Platte gespeichert
  - Enthalten im Datenblock Verweise und Namen der Dateien
  - Verzeichnisse werden durch ‚/' getrennt.

1. Normale Dateien

Normale Dateien sind die Dateien im herkömmlichen Sinne. Auch hier gibt es zwei Typen: Binär- und Textdateien. Textdateien speichern den Inhalt in ASCII-Code während Binärdateien den Inhalt in ihrer binären Form (für den Menschen unleserlich) abspeichern. Das System unterstützt keine Dateistruktur (keine Datei-Header), d.h. die Dateien sind lediglich Bytefolgen deren Strukturierung durch das zugehörige Programm vorgegeben ist. Auf solche Dateien kann sequentiell (der Reihe nach) zugegriffen werden oder wahlfrei durch Bytepositionierung.

2. Verzeichnisse

Verzeichnisse (Directories) dienen zur Strukturierung des Dateisystems. Sie bringen sozusagen Ordnung in die unübersichtliche Welt der Dateien. Verzeichnisse selbst werden wie Dateien auf die Platte geschrieben. In ihrem Datenblock enthalten sie jedoch Verweise und Namen aller Dateien, die sich in dem Verzeichnis befinden.

Verzeichnisse sind wie Ordner in Windows. Jedoch werden einzelne Verzeichnisse in der UNIX-Welt durch einen normalen Schrägstrich (‚/') getrennt und nicht wie bei Windows durch den Backslash (‚\'). Die UNIX-Notation ist auch vom WWW bekannt, wo Unterverzeichnisse einer Site in der URL durch ein ‚/' angesprochen werden können.

Dateitypen$_2$

- Spezialdateien (Gerätedateien)
  - E/A-Geräte werden als Spezialdateien eingetragen
  - Standardmäßig im Verzeichnis ‚/dev'
  - Zugriff auf Gerät kann genauso geschützt werden wie Datei
  - Blockorientierte und Zeichenorientierte Geräte
  - Da Geräte wie Dateien betrachtet werden sind keine Laufwerksbuchstaben nötig
  - Spezialdateien
    - /dev/null: Schickt Daten ins schwarze Loch, liefert EOF
    - /dev/zero: Liefert beliebig viele Nullbytes

3. Spezialdateien

UNIX behandelt auch Ein-/Ausgabegeräte als Dateien. Diese werden als Spezialdateien eingetragen und sind meistens im Verzeichnis ‚/dev' zu finden. Diese Eigenschaft erlaubt es, dass Geräte genauso behandelt werden können wie Dateien; es kann von Ihnen genauso gelesen werden bzw. auf die Geräte genauso geschrieben werden als handle es sich um Dateien. Viel wichtiger ist aber, dass der Zugriff auf Geräte genauso geschützt werden kann wie auf Dateien. Abhängig vom Gerät gibt es zwei verschiedene Typen von Spezialdateien. Dies sind einerseits blockorientierte Geräte, die jeweils nur ganze Blöcke von Daten liefern können (z.B. Festplatten) oder aber zeichenorientierte Geräte, die sequentiell einzelne Zeichen liefern (z.B. Drucker).

Da Geräte wie Dateien betrachtet werden, wird man in UNIX auch vergeblich nach Laufwerksbuchstaben suchen. Diese sind auch nicht notwendig, da die einzelnen Geräte in die Verzeichnisstruktur von UNIX eingehängt werden können und diese dann dynamisch erweitern. Es gibt zwei Spezialdateien, die immer vorhanden sind und besonders erwähnt werden sollen:

- /dev/null: Daten, die an diese Datei geschickt werden, landen in einem schwarzen Loch. Dies ist optimal, wenn Ausgaben eines Programms unterdrückt werden sollen. Ein Leseversuch von dieser Datei liefert immer ein End-of-File (EOF).

- /dev/zero: Beim Lesen von diesem Spezialgerät werden beliebig viele Nullbytes geliefert. Man kann dieses Verhalten zum Füllen von Dummy-Dateien verwenden.

Dateitypen$_3$

- Hardlinks
  - Möglichkeit eine Datei unter mehreren Namen anzusprechen → im gleichen Dateisystem
- Softlinks
  - Möglichkeit eine Datei unter mehreren Namen anzusprechen → in unterschiedlichen Dateisystemen
- Benannte Pipes
  - Puffert Daten zwischen zwei Prozessen
- Sockets
  - Netzwerkkommunikation

4. Verweise (Links)

Links erlauben ein und die selbe Datei unter mehreren Namen anzusprechen. Vereinfacht gesagt ist ein Link ein Zeiger auf eine Datei. Es gibt zwei verschiedene Arten von Links, die sich in Ihrer Funktion und Ihrem Aufbau voneinander unterscheiden. Der genaue Unterschied wird später in diesem Kapitel erklärt. Vorerst ist es nur wichtig zu wissen, dass es Hardlinks und Softlinks gibt. Der wesentliche Unterschied ist, dass Hardlinks nur im gleichen Dateisystem erstellt werden können wobei Softlinks auch in unterschiedlichen Dateisystemen erstellt werden können.

5. Benannte Pipes

Pipes sind spezielle Dateien, die zur Prozesskommunikation dienen. Eine Pipe hat zwei Enden: In eines der beiden Enden kann geschrieben werden und vom anderen Ende kann der Inhalt gelesen werden. Eine Pipe puffert also Daten. Sie nimmt Daten von einem Prozess entgegen, speichert diese und stellt diese Daten einem zweiten Prozess zum Lesen zur Verfügung. Sobald der Prozess einen Datensatz liest, wird dieser aus der Pipe gelöscht.

6. Sockets

Sockets sind sehr ähnlich zu Pipes, dienen aber zur Netzwerkkommunikation. Prozesse, die auf unterschiedlichen, vernetzten Rechnern liegen, können über Sockets kommunizieren.

Konventionen für Dateinamen

Konventionen

- Dateinamen
  - Case Sensitiv (TEST, Test, test sind in UNIX unterschiedliche Dateien)
- Sonderzeichen
  - Außer ‚/' sind alle Zeichen erlaubt.
  - Um Probleme vermeiden empfiehlt sich die Verwendung von:
    - Klein- und Großbuchstaben
    - Ziffern
    - Punkt (beliebig viele möglich)
    - Minuszeichen, sofern nicht am Anfang

Dateinamen Konventionen

Dateinamen können in UNIX aus Groß- und Kleinbuchstaben bestehen. Im Gegensatz zu Windows ist UNIX aber Case sensitive, was bedeutet, dass Groß- und Kleinschreibung Auswirkung auf die Dateien haben. Unterschiedliche Schreibweisen bezeichnen auch unterschiedliche Dateinamen. ‚TEST', ‚Test' und ‚test' wären zwar in Windows die gleichen Dateien, in UNIX sind sie aber unterschiedliche Dateien.

Wahl von Dateinamen

Prinzipiell kann in UNIX ein Dateiname aus beliebigen Zeichen bestehen, lediglich der einfache Schrägstrich (‚/') ist nicht erlaubt, da er Verzeichnisse voneinander trennt. Leerzeichen und Sonderzeichen sind erlaubt.

Viele Sonderzeichen haben aber eine Bedeutung und müssen - wenn vergeben - gesondert angesprochen werden. Um Probleme zu vermeiden, empfiehlt es sich darum, Dateinamen auf eine Kombination folgender Zeichen zu beschränken:

- Klein- und Großbuchstaben
- Ziffern
- Punkt (es sind beliebig viele Punkte innerhalb eines Dateinamens möglich)
- Minuszeichen (sofern dieses nicht am Anfang steht)

Verzeichnisse in UNIX (Wiederholung)

Verzeichnisse$_1$

- UNIX hat ähnlich zu Windows ein hierarchisch aufgebautes Verzeichnissystem
  - Wurzelknoten (Root) wird durch ‚/' gekennzeichnet
- Homeverzeichnis
  - Verzeichnis in dem man sich nach Einloggen befindet.
  - Gehört vollständig Benutzer (alle Rechte)
- Aktuelles Arbeitsverzeichnis
  - Verzeichnis dessen Inhalt gerade bearbeitet wird
  - Kommandos beziehen sich auf dieses Verzeichnis

UNIX hat ähnlich zu Windows ein hierarchisch strukturiertes Verzeichnissystem. Durch diese Struktur gibt es einen eindeutig definierten Wurzelknoten (Vater aller Unterknoten), der auch root-Verzeichnis genannt wird. Das root-Verzeichnis wird durch einen einfachen Schrägstrich (‚/') gekennzeichnet und wird auch über dieses Verzeichnis in UNIX angesprochen (z.B. wechselt das Kommando „cd /" ins root-Verzeichnis).

Homeverzeichnis (Home-Directory)

Ein wichtiges Standardverzeichnis in UNIX ist das sogenannte Homeverzeichnis. Dieses Verzeichnis gehört dem aktuellen Benutzer und er besitzt alle Rechte auf dieses Verzeichnis. Jedem Benutzer ist genau ein solches Verzeichnis zugeordnet. In diesem Verzeichnis werden standardmäßig alle Benutzerdateien abgelegt. Nach einem erfolgreichen Login befindet sich der Benutzer in seinem Homeverzeichnis.



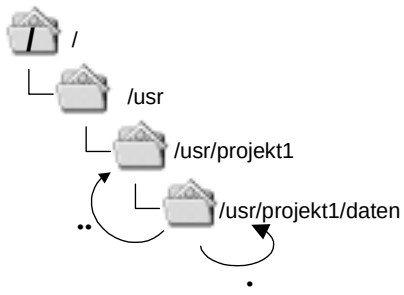
Bei Linux-Systemen befinden sich die Homeverzeichnisse in der Regel im Verzeichnis ‚/home'.

Aktuelles Arbeitsverzeichnis

Das aktuelle Arbeitsverzeichnis ist immer jenes, dessen Inhalt gerade bearbeitet wird. Das aktuelle Verzeichnis kann jederzeit gewechselt werden, aber irgendein Verzeichnis ist zu einem bestimmten Zeitpunkt immer das aktuelle Verzeichnis. Sofern nicht anders angegeben, beziehen sich alle Kommandos immer auf das aktuelle Verzeichnis. Darüber hinaus dient das aktuelle Verzeichnis als Ausgangspunkt für relative Pfadangaben.

Verzeichnisse$_2$

- Pfadangaben
  - Dateien werden über Pfad spezifiziert
  - Absolute Pfadangabe
    - `/usr/projekt1/daten` (beginnt mit root!)
  - Relative Pfadangabe
    - `cd projekt1/daten` (muss in /usr stehen)
- Besondere Einträge
  - . Aktuelles Verzeichnis
  - .. Vaterverzeichnis



Struktur von UNIX Verzeichnissen

Jedes Verzeichnis im Verzeichnisbaum ist über einen Pfad zu erreichen. Der Pfad beschreibt, wie man zu dem gewünschten Verzeichnis hinkommt. Nicht nur Verzeichnisse, auch Dateien können über den Pfad spezifiziert werden. Befindet man sich z.B. gerade in seinem Homeverzeichnis, man möchte aber gerne aus dem Verzeichnis ‚/etc' die Datei foo.txt bearbeiten, so kann man diese im Kommando bei Angabe des Pfads ansprechen (z.B. „cat /etc/foo.txt").

Pfadangaben

Die Angabe des Pfades kann hierbei auf zwei Arten erfolgen:

1. Angabe des absoluten Pfads: Die absolute Pfadangabe beginnt immer bei root. Es werden dann die Verzeichnisse in der Reihenfolge angegeben (getrennt durch ‚/'), die durchlaufen werden müssen, um zur Zieldatei zu gelangen.

2. Angabe des relativen Pfads: Die relative Pfadangabe geht bei der Pfadauflösung immer vom aktuellen Arbeitsverzeichnis aus. Sie müssen also beschreiben, wie das gewünschte Verzeichnis von dem Verzeichnis erreicht wird, in dem Sie sich gerade befinden. Dabei können Sie sich besondere Verzeichniseinträge, die in weiterer Folge beschrieben werden, zur Hilfe nehmen.

Besondere Verzeichniseinträge

Es gibt zwei besondere Verzeichniseinträge, die unbedingt erwähnt werden müssen. Bis auf das root-Verzeichnis besitzt jedes Verzeichnis die Einträge „.' und „..'. Das sind Stellvertreter für Verzeichnisnamen. Das Verzeichnis „.' verweist auf sich selbst und bezeichnet so das aktuelle Verzeichnis, „..' zeigt auf das übergeordnete Verzeichnis (Vaterverzeichnis). Um zum Vaterverzeichnis – also das Verzeichnis, dass das aktuelle Verzeichnis beinhaltet – zu wechseln, müssen Sie beim Verzeichniswechsel (durch das Kommando cd, dass später in diesem Kapitel behandelt wird) „..' angeben.

Standardverzeichnisse

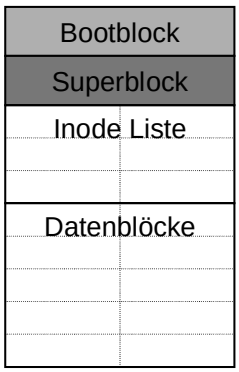
In UNIX gibt es eine Reihe von Standardverzeichnissen, die bereits bei der Installation angelegt werden und deren Inhalt definiert ist. Trotzdem können aber - sollten aber nicht - auch andere Dateien in diesen Verzeichnissen abgelegt werden. Diese Verzeichnisse sind nun in weiterer Folge aufgelistet und kurz erklärt.

Verzeichnis	**Beschreibung / Funktionalität**
/	Hier wird der Default-Kernel und die dazugehörige System-Map gespeichert.
/bin	Hier werden Systemverwaltungsprogramme, die nicht spezifisch für den Superuser sind und auch von anderen Benutzern verwendet werden können, gespeichert.
/boot	Hier werden die LILO-Dateien (Linux Bootmanager), die nicht ausführbar sind und keine Konfigurationsdateien sind, gespeichert. Es können sich hier auch alternative Kernel befinden.
/dev	Hier befinden sich alle Gerätedateien.
/etc	Hier befinden sich die Konfigurationsdateien des Basissystems und des Bootmanagers sowie die Benutzerdatenbank.
/home	Hier befinden sich die Homeverzeichnisse der einzelnen Benutzer.
/lib	Hier werden gemeinsam genutzte Bibliotheken des Systems abgelegt.
/lost+found	Hier befinden sich Dateien, die beim Plattencheck angefallen sind. Diese Dateien beinhalten zumeist Daten, die bei einem eventuellen Dateisystemfehler gerettet werden konnten.
/mnt	Hier befinden sich normalerweise die sogenannten Mount-Points für weitere Geräte wie z.B. CD-ROM oder das Diskettenlaufwerk. Die Datenträger dieser Geräte besitzen ein eigenes Filesystem, dass in diese Mount-Points eingehängt werden kann.
/opt	In diesem Verzeichnis werden optionale Erweiterungspakete wie z.B. KDE oder Netscape Communicator abgelegt. Die Erweiterungspakete selbst werden wieder in Unterverzeichnissen gespeichert.
/proc	Dieses Verzeichnis ist ein besonderes Verzeichnis, da es physisch gar nicht auf der Platte existiert. Es ist vielmehr ein virtuelles Dateisystem, das vom Kernel gemanagt wird und Aufschluss über den aktuellen Zustand des Systems und der Hardware gibt. In den einzelnen Dateien dieses Verzeichnisses findet man Informationen zur vorhandenen Hardware und zum Zustand des Systems.
/root	Das /root-Verzeichnis ist das Homeverzeichnis des Superusers. Es ist aus Sicherheitsgründen von den restlichen Homeverzeichnissen getrennt.
/sbin	Dieses Verzeichnis enthält ausführbare Systemdateien, die nur dem Systemadministrator zugänglich sein sollen. Darüber hinaus werden hier üblicherweise auch Startskripten abgelegt.
/tmp	Hier findet man temporäre Dateien, die während der Laufzeit anfallen und lediglich zwischengespeichert werden müssen.
/usr	Neben dem Verzeichnis /opt werden in diesem Verzeichnis Programme abgelegt. Hier befinden sich normalerweise Programme der Distribution oder Programme, die speziell für das jeweilige System zugeschnitten sind.
/usr/X11R6	Dieses Verzeichnis beinhaltet die graphische Benutzeroberfläche X-Window und deren Dateien. Der Verzeichnisname gibt Aufschluss über die aktuelle Version und Release.
/usr/dict	Hier befindet sich das Wörterbuch zum ispell-Programm.
/usr/doc	Hier ist üblicherweise die Dokumentation zu finden.
/usr/include	Dieses Verzeichnis beinhaltet Header-Dateien zur Unterstützung von Anwendungsentwicklern.
/usr/local	Hier werden lokale Anwendungen und Daten abgelegt.
/usr/info	Verzeichnis für Textdateien des Textinfo-Dokumentationssystems.
/usr/man	Hier befinden sich die manual pages (Online-Hilfe).
/usr/share	In diesem Verzeichnis werden architekturunabhängige Daten in verteilten Systemen mit verschiedenen Rechnerarchitekturen abgelegt. Das Verzeichnis beinhaltet also jenen Teil des verteilten Systems, der auf allen Architekturen gleich ist.
/usr/src	Dieses Verzeichnis ist für Quellcode der Programme des Standardsystems vorgesehen.
/usr/src/linux	Dieses Verzeichnis ist nur auf Linux-Systemen zu finden und enthält - sofern installiert - den Quellcode des Betriebssystems. Dieser Quellcode erlaubt die Generierung von eigenen, an die Hardware angepassten Kernelversionen. Meist ist dieses Verzeichnis aber nur ein symbolischer Link auf ein aussagekräftigeres Verzeichnis (Verzeichnisname beinhaltet meistens Kernelversion).

Aufbau des Dateisystems

Aufbau des Dateisystems$_1$

- Bootblock
  - Bootloader, vor Betriebssystem geladen
- Restliches Dateisystem mit fixer Blockgröße
- Superblock
  - Beinhaltet Geometriedaten der Festplatte
  - Anzahl der Blöcke des Dateisystems (Datenblöcke, Verwaltungsinformationen)



Die Festplatte ist in sogenannte Blöcke geteilt. Die ersten paar Blöcke einer Festplatte bleiben für den Bootloader und ähnliche Dinge reserviert, die vor dem Betriebssystem geladen werden müssen und deshalb außerhalb seiner Reichweite sind. Den Rest der Platte verwaltet UNIX in Form eines Dateisystems. Es operiert immer mit festen Blockgrößen, unabhängig von der Größe der Platte.

ⓘ Bei MS-DOS ist die Größe der Verwaltungseinheiten abhängig von der Größe der Festplatte. Je größer die Partition ist, desto größer auch die Verwaltungseinheiten was sich bei vielen kleinen Dateien ungünstig auf den verfügbaren Speicherplatz auswirkt.

Den Anfang eines Dateisystems bildet der sogenannte Superblock, der die Geometriedaten der Festplatte beinhaltet. Er weiß als einziger Bescheid über die Anzahl der Blöcke im Dateisystem und weiß, welche davon Verwaltungsinformationen enthalten und welche die eigentlichen Daten.

Nach dem Superblock folgt die Inode Liste. Diese beinhaltet die Verwaltungsinformationen und stellt ein Inhaltsverzeichnis aller in dem Dateisystem existierenden Dateien dar. Sie besteht aus einer Folge von Inodes (Dateiköpfen), die jeweils Verwaltungsinformationen zu jeder Datei enthalten.

Dann folgen die eigentlichen Daten im Datenbereich. Hier befinden sich die freien Blöcke, Datenblöcke (Inhalte von Dateien und Verzeichnissen) und Referenzblöcken, die zur Adressierung der Datenblöcke eingesetzt werden. Die Größe der einzelnen Bereiche wird bei der Initialisierung eines physischen Dateisystems festgelegt und kann im nachhinein nicht mehr dynamisch verändert werden.

Jede Datei besitzt einen Inode. Der Zugriff erfolgt entweder sequentiell (nach der Reihe) oder wahlfrei (direkte Adressierung). Das Betriebssystem hat einen Cache Mechanismus implementiert, der einen Teil der Datei im Speicher hält mit dem Ziel, die Plattenzugriffe zu reduzieren. Das erfordert aber das regelmäßige Aktualisieren des Cache-Inhalts da sonst Daten bei einem plötzlichen Stromausfall verloren gehen könnten und dies zur Inkonsistenz des Dateisystems führen könnte.

Der Superblock

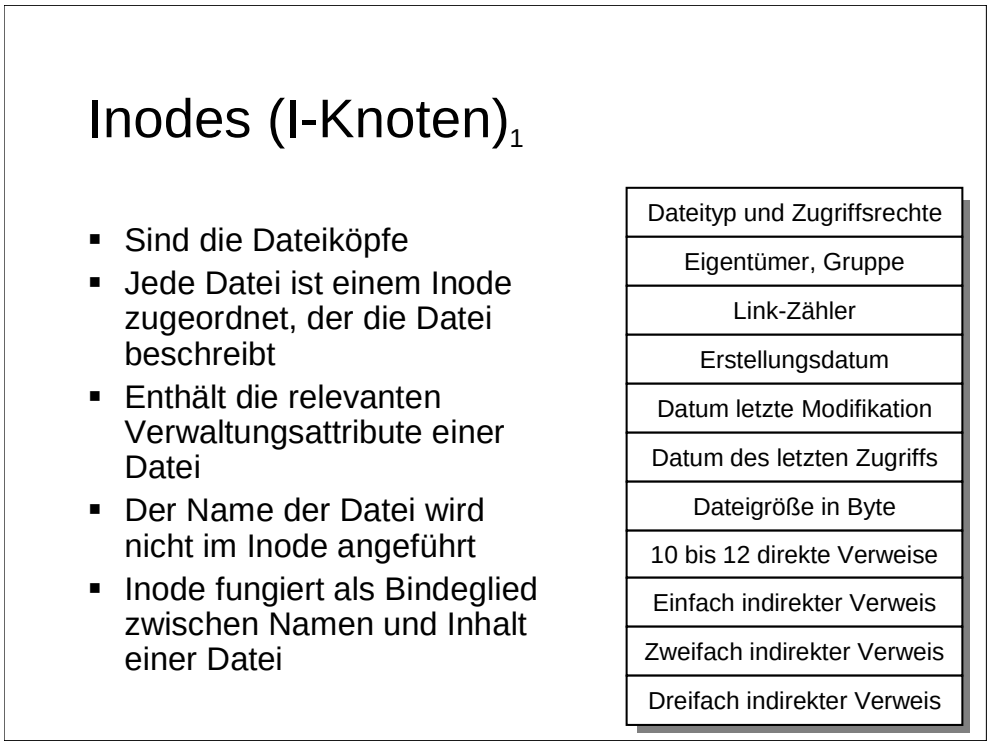
Superblock

- Größe der Inode Liste in Blöcken
- Größe des Dateisystems in Blöcken
- Anzahl der freien Blöcke im Dateisystem
- Liste der freien, im Dateisystem verfügbaren Blöcke
- Liste freier Inodes
- Flag, das anzeigt, ob Superblock modifiziert wurde
- Flag, das anzeigt, ob Dateisystem read-only gemountet ist
- Datum/Uhrzeit der letzten Superblock Aktualisierung
- Gesamtzahl freier Blöcke im Dateisystem
- Gesamtzahl freier Inodes im Dateisystem
- Name des Dateisystems und des Laufwerks

Der Superblock enthält alle Verwaltungsinformationen zu einem physischen Dateisystem. Das sind im Detail:

- die Größe der Inode Liste in Blöcken: Daraus kann die Anzahl der Inodes berechnet werden
- die Größe des Dateisystems in Blöcken
- die Anzahl der freien Blöcke im Dateisystem (die Größe des freien Speichers)
- die Liste freier Inodes
- ein Flag, das anzeigt, ob der Superblock modifiziert wurde
- ein Flag, das angibt, ob das Dateisystem nur gelesen (read-only) werden kann oder auch geschrieben werden kann
- das Datum und die Uhrzeit der letzten Aktualisierung des Superblocks
- die Gesamtzahl freier Blöcke im Dateisystem
- die Gesamtzahl freier Inodes im Dateisystem
- und den Namen des Dateisystems und des Dateisystemlaufwerks (Datei, die Laufwerk wiederspiegelt)

Inodes



Inodes enthalten die relevanten Verwaltungsattribute einer Datei. Sie sind sozusagen die Dateiköpfe jeder Datei. Jede Datei ist einem Inode zugeordnet, der die Datei beschreibt bzw. auf die Datenblöcke zeigt, in der die Daten der Datei gespeichert sind.

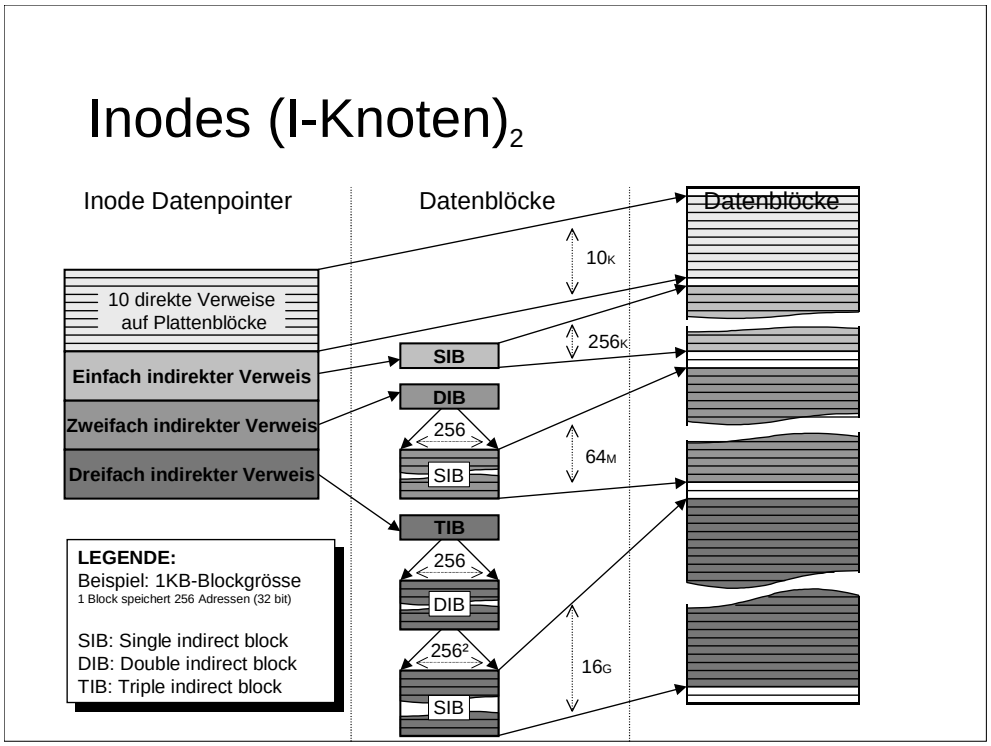
Ein Inode enthält:

- Dateityp und Zugriffsrechte
- Eigentümer und Gruppeninformationen
- Einen Zähler, der die Anzahl der Links speichert, die auf die Datei zeigen
- das Erstellungsdatum der Datei
- das Datum der letzten Modifikation der Datei
- das Datum des letzten Zugriffes auf die Datei
- die Dateigröße in Bytes
- 10 (herkömmliche System V Systeme) bis 12 (BSD-Systeme) direkte Verweise auf die Plattenblöcke, in denen die Daten der Datei gespeichert sind
- einen einfach indirekten Verweis, der auf Datenblöcke zeigt, in denen Daten der Datei gespeichert sind

- einen zweifach indirekten Verweis, der auf Datenblöcke der Datei zeigt und

- einen dreifach indirekten Verweis auf Datenblöcke der Datei.

Zu beachten gilt, dass der Name der Datei nicht im Inode gespeichert wird. Dieser wird im Datenblock der Verzeichnisdatei gespeichert. Der Inode fungiert somit als Bindeglied zwischen den Namen und dem Inhalt einer Datei.



Für die Adressierung der Inhalte einer Datei ergibt sich eine bestimmte Verweisstruktur. Untersucht man die durchschnittliche Länge von Dateien, dann fällt auf, dass kurze relativ häufig sind. Daher versucht UNIX, die Nummern der ersten paar Plattenblöcke, die Daten der Datei enthalten, direkt im Inode zu speichern. Bei einem Zugriff über den Inode stehen diese direkten Datenblöcke also ohne weiteres Lesen von Verwaltungsinformationen zur Verfügung. Für große Dateien ist dieses Verfahren natürlich nicht praktikabel, denn der Inode würde dann ebenfalls sehr groß werden. Wie solche Dateien gespeichert werden, soll nun beschrieben werden.
UNIX speichert immer Verweise auf die ersten 10 bis 12 Plattenblöcke im Inode. Der Einfachheit halber wird in weiterer Folge mit 10 direkten Plattenblock-Verweisen und einer Blockgröße von 1 Kilobyte (d.h. in einem Block kann 1 KB gespeichert werden) ausgegangen. Der Inode enthält somit Verweise auf 10 Datenblöcke, die jeweils Daten der Datei enthalten. Bei einer Blockgröße von 1 KB kann der Inode somit eine Datei von 10 KB ansprechen. Für Dateien, die diese Größe überschreiten, bietet UNIX das Konzept der indirekten Verweise zu Datenblöcken. Ein Inode kann dabei jeweils einen einfachen, zweifachen und dreifachen indirekten Verweis speichern.

Ein indirekter Verweis enthält die Adresse eines anderen Datenblocks. Innerhalb dieses Datenblocks (Single Indirect Block) werden dann weitere Adressen zu anderen Datenblöcken gespeichert. Adressen haben immer die Länge von 32 bit, somit kann ein 1 KB-Block 256 Adressen speichern. Ein einfach indirekter Verweis zeigt somit auf einen Datenblock, der selbst wieder auf 256 Datenblöcke zeigt, die dann Daten der Datei beinhalten. Somit lässt sich mit einem einfach indirekten Verweis bereits eine Datei mit 10 direkten Adressen und 256 zusätzlichen Adressen speichern (in unserem Beispiel wäre das eine Datei der Größe 266 KB).

Reicht dies nicht, so kann ein zweifach indirekter Verweis in Anspruch genommen werden. Dieser funktioniert analog zum einfach indirekten Verweis, nur wird zusätzlich eine Verweisschicht dazwischengeschoben. Ein zweifach indirekter Verweis enthält im Inode einen Verweis auf einen Datenblock. In diesem Datenblock (Double Indirect Block) werden dann Adressen zu einer Reihe von Single Indirect Blocks gespeichert. In einem 1 KB-Block können also 256 Single Indirect Blocks im Double Indirect Block gespeichert werden. Jeder Single Indirect Block speichert dann jeweils 256 Adressen zu Datenblöcken, die letztendlich die Daten der Datei beinhalten. Ein zweifach indirekter Verweis wird erst dann in Anspruch genommen, wenn ein einfach indirekter Verweis nicht mehr ausreicht. Damit lässt sich dann schon eine Datei mit 10 direkten Adressen, 256 Adressen aus dem einfach indirekten Verweis und 65536 (256^2) Adressen aus dem zweifach indirekten Verweis erstellen, was eine Gesamtgröße von mehr als 64 MB (65802 KB) ergibt.

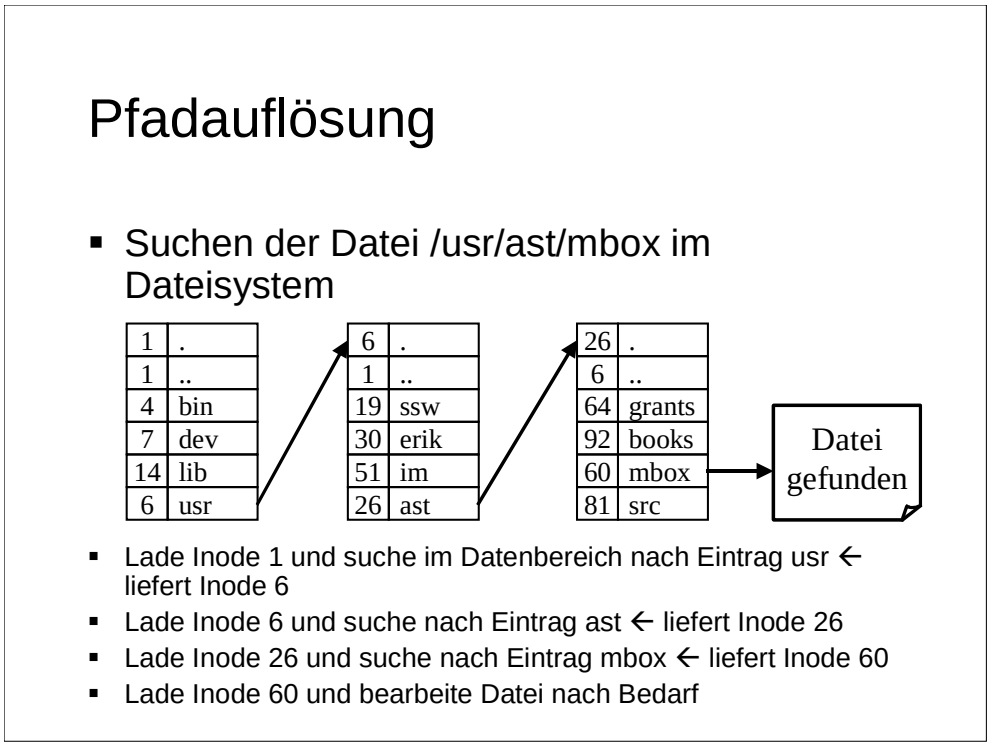
Sollte der Fall eintreten, dass selbst dies nicht ausreicht, ist noch ein dreifach indirekter Block vorgesehen. Hier verweist der Inode auf einen Datenblock (Triple Indirect Block), der wiederum auf eine Reihe von Double Indirect Blocks zeigt. In einem 1 KB-Block können also 256 Double Indirect Blocks gespeichert werden, von denen jeder wiederum auf 256 Single Indirect Blocks zeigt. Jeder dieser Single Indirect Blocks zeigt dann auf 256 Adressen zu Datenblöcken, die dann endlich die Daten beinhalten. Damit lässt sich dann eine Datei mit 10 direkten Adressen, 256 Adressen aus dem einfach indirekten Verweis, 256^2 Adressen aus dem zweifach indirekten Verweis und 256^3 erstellen, was eine Gesamtgröße von mehr als 16 GB ergibt!

Moderne UNIX-Systeme verwenden aber eine Blockgröße von 8 KB, wodurch sich in einem Datenblock bereits 2048 Adressen speichern lassen. Damit lassen sich mit Hilfe eines einfach indirekten Verweises bereits 16 MB große Dateien abspeichern und mit zur Hilfenahme eines zweifach indirekten Verweises mehr als 32 GB.



Durch die Einteilung in Verwaltungsblöcke und Datenblöcke ergeben sich einige Restriktionen im Umgang mit UNIX-Dateisystemen. So ist z.B. die maximale Anzahl von Dateien durch die Anzahl von Inodes beschränkt. Zudem ist die maximale Dateigröße selbst – wie bereits oben gezeigt – auch von der gewählten Blockgröße abhängig.

Pfadauflösung



Unter UNIX existiert zu jedem Prozess eine Verwaltungsstruktur, in der seine Umgebung definiert ist. Dort führt das Betriebssystem unter anderem zwei Einträge, in denen die Inode Nummer des Hauptverzeichnisses und des aktuellen Verzeichnisses hinterlegt sind, die zu dem jeweiligen Prozess gehören. Bekommt das Betriebssystem in einem Systemaufruf einen Pfadnamen übermittelt, so prüft es zunächst, ob es sich um einen absoluten oder einen relativen Pfad handelt. Falls der Name mit einem ‚/' beginnt, wird im Hauptverzeichnis mit der Auflösung begonnen, andernfalls im aktuellen. Dabei wird die erste Komponente des Pfadnamens isoliert und diese im entsprechenden Startverzeichnis gesucht. Sobald sie den gesuchten Namenseintrag dort gefunden hat, kann die zugehörige Inode Nummer ausgelesen werden. Der Inode verweist entweder auf die gesuchte Datei oder, bei einem längeren Pfadnamen, auf ein Verzeichnis, in dem die nächste Komponente zu suchen ist. Bei der Auflösung von Pfadnamen ergibt sich also eine wechselseitige Verkettung: Inodes enthalten Zeiger auf Datenblöcke, und die Datenblöcke eines Verzeichnisses enthalten Zeiger auf Inodes.

Das folgende Beispiel soll nun verdeutlichen, wie Pfade im Dateisystem aufgelöst werden und wie Dateien vom Dateisystem geladen werden.

Suchen der Datei /usr/ast/mbox

Jede Datei ist durch ihren vollständigen Namen (Dateiname und absoluter Pfad) eindeutig im Dateisystem spezifiziert. Wenn das Betriebssystem eine Datei öffnet, so löst es den absoluten Pfad und den Dateinamen der angesprochenen Datei auf. Nehmen wir an, im Verzeichnis ‚/usr/ast' befindet sich die Datei ‚mbox'. Zuerst lädt das Betriebssystem den Inode 1 und kann somit auf die Verzeichnisinformationen des root-Verzeichnisses,

die im Datenbereich gespeichert sind, zurückgreifen. Im Datenbereich wird nun nach dem ersten Verzeichnis, in unserem Falle ‚usr' gesucht. Der Eintrag liefert in unserem Falle den Inode 6 zurück, der dann in weiterer Folge vom Betriebssystem geladen wird. Inode 6 verweist auf das Verzeichnis ‚/usr' und der angesprochene Datenbereich beinhaltet wieder alle darin gespeicherten Dateien und Unterverzeichnisse. Nun wird nach dem zweiten Unterverzeichnis (‚ast') gesucht. Der Eintrag verweist auf den Inode 26, der dieses Verzeichnis referenziert. Der Inode wird geladen und in den zugehörigen Datenbereich wird nun nach dem Eintrag ‚mbox' gesucht. Dieser wird mit dem Verweis auf Inode 60 gefunden. Das Betriebssystem lädt nun diesen Inode und kann mit Hilfe der darin enthaltenen Verweise die zugehörigen Datenblöcke laden – somit kann die Datei nun geladen werden und Prozessen zur Verfügung gestellt werden.

Links

Links

- Verknüpfung zwischen Dateien
- Ermöglicht das Ansprechen einer Datei unter verschiedenen Namen
- Systemfunktion „`ln`“
- Bereits vom Dateisystem aufgelöst → können von allen Programmen benutzt werden

Ein besonders praktisches Konzept von UNIX sind Links. Links sind Verknüpfungen zwischen Dateien, die es erlauben, ein und dieselbe Datei unter verschiedenen Dateinamen und in verschiedenen Verzeichnissen anzusprechen, ohne dass die Datei kopiert werden muss.

ⓘ Seit Windows 95 gibt es sogenannte Verknüpfungen. Diese erfüllen zwar ähnliche Funktionalität wie UNIX Links, sind aber grundsätzlich unterschiedlich. Windows 95-Links sind Spezialdateien, die einfach auf eine andere Datei verweisen. Diese Spezialdateien können aber nur vom Explorer aufgelöst werden. UNIX Links werden bereits vom Dateisystem aufgelöst und können somit von jedem Programm genutzt werden.

Um in UNIX für eine Datei mehrere Dateinamen anzulegen, verwendet man die Systemfunktion ‚ln' (link), die einen Verweis zu einer Datei unter einem anderen Namen anlegt.

Arten von Links

- Hardlinks
  - Verbinden einen Inode mit mehreren Namen
  - Neuer Namenseintrag hat gleichen Inode wie erster
  - Nur innerhalb gleichem Dateisystem möglich
  - Alle Namen gleichberechtigt
- Softlinks
  - Sind Dateien, die im Datenbereich die Pfadangabe jener Datei enthalten, mit der sie verknüpft sind
  - Auch in unterschiedlichen Dateisystemen möglich
  - Eine Art Zeiger auf eine andere Datei

UNIX unterscheidet zwei Arten von Links.

Hardlinks

Das Konzept der Links legt nahe, dass ein und dieselbe Datei unter mehreren Namen angesprochen werden kann, wenn man einfach dieselbe Inode Nummer mehrfach verwendet. Genau dieses Prinzip steht hinter Hardlinks. Mit dem *ln*-Kommando können zusätzliche Verzeichniseinträge für eine Datei an anderer Stelle und unter anderem Namen angelegt werden.

Hardlinks sind auf eine Datei gleichberechtigt. Es spielt keine Rolle, welcher zuerst da war und welcher später mit *ln* eingerichtet wurde. Somit hat das Löschen eines Links keine weiteren Auswirkungen auf die Datei. Die Datei ist lediglich nach dem Löschen des Links unter diesem nicht mehr auffindbar. Erst wenn der letzte Link gelöscht wird und es keine Chance mehr gibt die Datei anzusprechen, wird auch der Inode gelöscht und die zugehörigen Datenblöcke freigegeben. Wie viele Links auf einen Inode zeigen, wird im Inode selbst aufgezeichnet. Dort befindet sich nämlich ein Linkzähler, der erhöht wird, sobald ein neuer Hardlink auf den Inode angelegt wird. Beim Löschen einer Datei wird der Linkzähler um Eins vermindert. Erst wenn der Zähler Null erreicht, gibt das Betriebssystem den belegten Platz frei.

Hardlinks unterliegen allerdings der Einschränkung, die sich aus der Organisation der Inodes ergibt: Da zwei Verzeichniseinträge, die auf diese Weise verbunden sind, die gleiche Inode Nummer verwenden und Inode Nummern für jede Partition getrennt verwaltet werden, müssen Quelle und Ziel bei der *ln*-Verknüpfung im selben Dateisystem, also auf derselben Partition, liegen.

Softlinks

Um auch Links über verschiedene Dateisysteme hinweg erstellen zu können, gibt es die Softlinks (oder symbolische Links). Diese werden mittels der Option ‚-s' erstellt.

Ein Softlink ist ein Verzeichniseintrag, der nicht auf den Inode einer vorhandenen Datei, sondern über einen *Pfadnamen* auf eine Datei verweist. Ein Softlink ist deshalb eine Datei, die in ihrem Datenblock den Namen der Datei enthält, auf die der Link zeigt. Da hier auf eine Datei gezeigt wird, die durch die Angabe des Pfades qualifiziert wird, können auch Dateien in anderen Dateisystemen angesprochen werden.

! Wichtig ist zu verstehen, dass Softlinks eine Art Zeiger auf eine andere Datei sind. Im Datenblock der Softlinkdatei steht der Pfadname der Datei, die angesprochen werden soll. Das Betriebssystem kann so (mittels Pfadauflösung) die angesprochene Datei ausfindig machen. Der Datenbereich des symbolische Links zeigt aber *NICHT* auf den Inode der zu linkenden Datei!

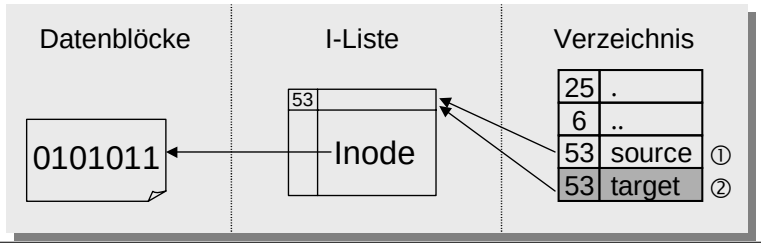
Durch das Abspeichern des Namens der Datei muss die Zieldatei gar nicht vorhanden sein. Beim Erstellen des Links wird auch nicht überprüft, ob sich der Zieldateiname auf eine gültige Datei bezieht. Dadurch kann man symbolische Links schon erzeugen bevor die anzusprechende Datei existiert. Auch lässt sich nachträglich die Zieldatei löschen und durch eine aktuellere ersetzen, ohne dass der Link neu eingerichtet werden muss.

Beim Zugriff auf einen symbolischen Link, der auf eine nicht vorhandene Datei verweist, erhält man eine Fehlermeldung – genauso, als hätte man einen ungültigen Dateinamen eingegeben.

Hardlinks

Hardlinks

- Verschiedene Dateinamen teilen sich ein und denselben Inode
  - Link-Zähler im Inode wird erhöht
- `$ ln` *`source target`*
  - Erzeugt einen Hardlink *target* auf eine bestehende Datei *source*. Verzeichniseintrag *target* zeigt auf gleichen Inode wie *source*.



Bei Hardlinks teilen sich verschiedene Dateinamen ein und denselben Inode. Wenn ein Hardlink erstellt wird, wird im Verzeichnis ein Eintrag erstellt, der auf den selben Inode zeigt wie die ursprüngliche Datei. Dann wird zusätzlich im Inode der Link-Zähler um eines erhöht, damit der Inode weiß, wie viele Dateinamen auf ihn zeigen. Sobald eine Datei gelöscht wird, wird der Link-Zähler um Eins vermindert und erst wenn dieser Null erreicht, wird auch der Inode gelöscht.

Hardlinks werden mit dem Kommando „ln *source target*“ erstellt. Dabei muss die Datei ‚source' bereits existieren, ‚target' wird erstellt.

```
169.254.0.6 - PuTTY
Esprit:~/arbeit/fh-hagenberg/bts3 $ ln source target
Esprit:~/arbeit/fh-hagenberg/bts3 $ ls -la
total 2
drwxr-xr-x    2 probst   staff         512 Sep 29 20:00 .
drwxr-xr-x    4 probst   staff         512 Sep 29 19:59 ..
-rw-r--r--    2 probst   staff           0 Sep 29 20:00 source
-rw-r--r--    2 probst   staff           0 Sep 29 20:00 target
Esprit:~/arbeit/fh-hagenberg/bts3 $
```

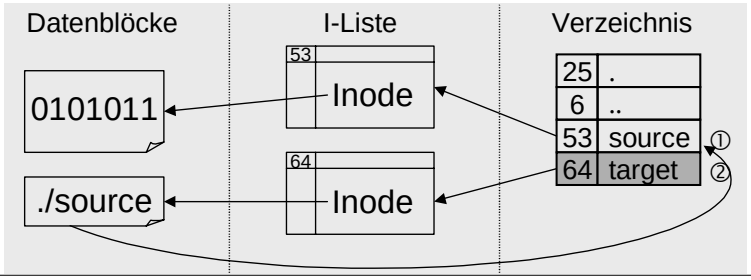
Abbildung 2: Erstellen eines Hardlinks

Wie in Abbildung 2 ersichtlich, ist nach dem Erstellen des Hardlinks kein Unterschied mehr zu sehen. In der Tat kann nun auch nicht mehr festgestellt werden, welcher Dateiname zuerst existiert hat. Wichtig ist aber, dass Buch darüber geführt wird, wie viele Dateinamen nun auf den Inode zeigen. Dies ist in der zweiten Spalte der Ausgabe ersichtlich.

Softlinks

Softlinks

- Verschiedene Dateinamen haben verschiedene Inodes
  - Datenbereich des neuen Inodes zeigt auf Pfadangabe der ursprünglichen Datei
- `$ ln -s` *`source target`*
  - Erzeugt einen Softlink *target* auf eine bestehende Datei *source*. Es wird ein eigener Inode angelegt, dessen Datenbereich auf *source* zeigt.



Bei Softlinks haben die verschiedenen Dateinamen auch verschiedene Inodes. Der Dateiname des Softlinks zeigt auf einen Inode im Dateisystem, in dem auch der Softlink abgelegt ist. Der Datenbereich dieses Inodes enthält den bei der Erstellung angegebenen Pfadnamen der Datei, auf die der Link zeigt. Über diesen Dateinamen kann das Betriebssystem die Ursprungsdatei dann auffinden.

Softlinks werden auch mit dem *ln*-Kommando erzeugt, jedoch unter Angabe des Parameters ‚-s'. Das Kommando „ln –s *source target*" würde einen Softlink ‚target' auf die Datei ‚source' erstellen. Dabei ist es unerheblich, ob die Datei ‚source' bereits existiert oder nicht. Beim Erzeugen des Softlinks wird ein eigener Inode angelegt, dessen Datenbereich den vollständigen Pfadnamen der Datei ‚source' enthält.

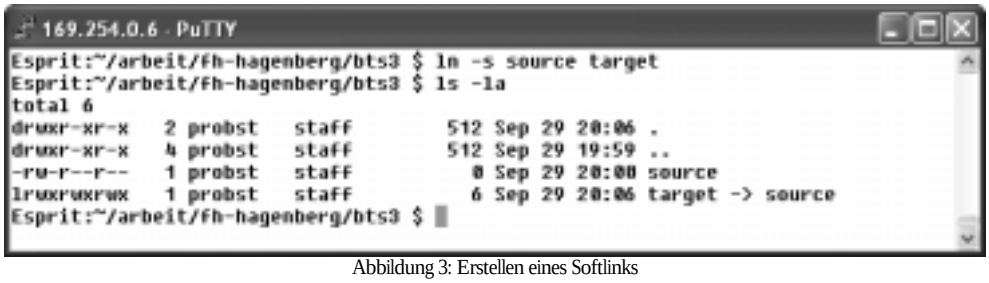
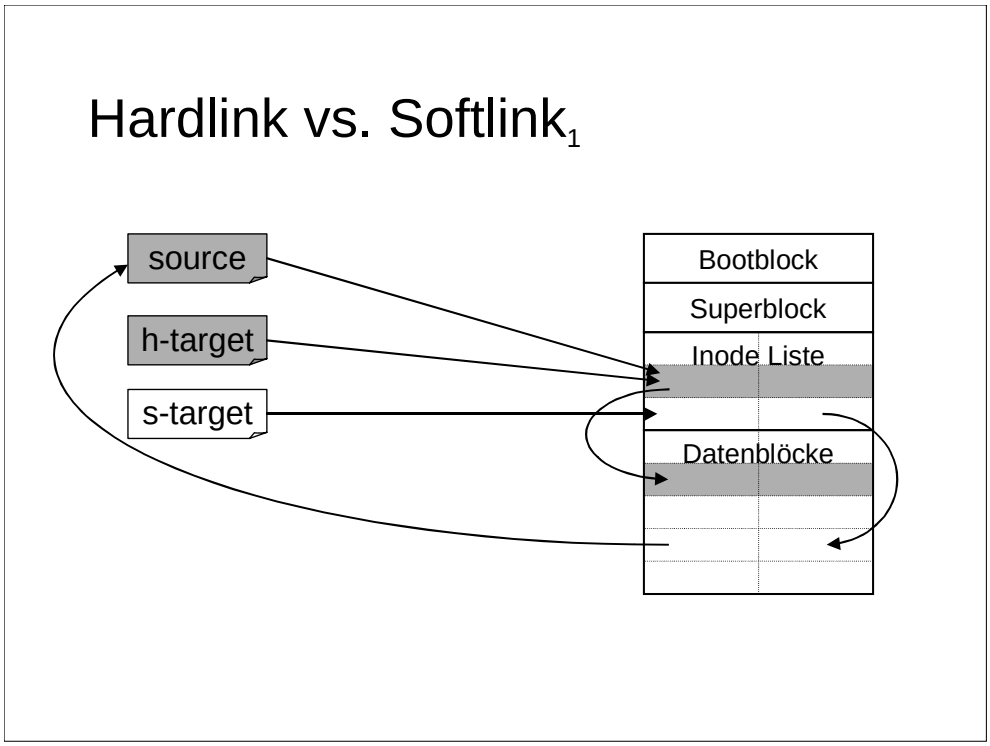


Abbildung 3: Erstellen eines Softlinks

Wie in Abbildung 3 zu sehen, ist ein deutlicher Unterschied zwischen der ursprünglichen Datei und dem Softlink zu erkennen. Auch der Linkzähler wurde nicht erhöht.

Unterschiede zwischen Hard- und Softlinks

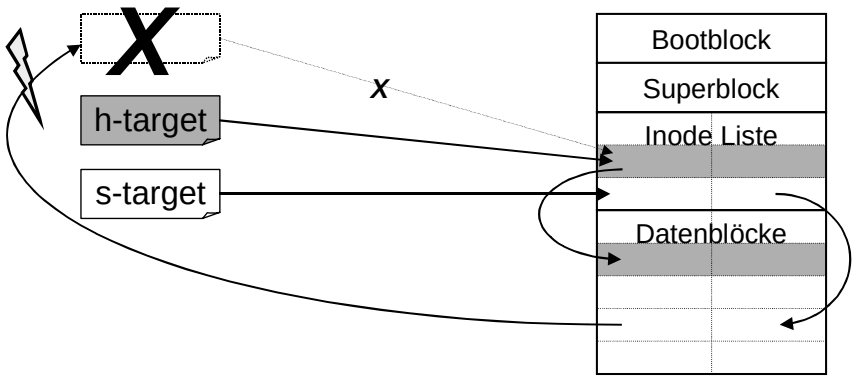


Der Unterschied zwischen Hard- und Softlinks soll nun anhand eines Beispieles näher erläutert werden. In einem Verzeichnis befindet sich eine Datei mit dem Namen ‚source'. Darüber hinaus befinden sich noch zwei Links auf die Datei, ein Hardlink (h-target) und ein Softlink (s-target).

Die Datei ‚source' zeigt auf einen bestimmten Inode in der Inode-Liste. Dieser Inode wiederum zeigt mit Hilfe seiner Verweise (direkte und indirekte Verweise) auf Datenblöcke, die dann die Daten der Datei beinhalten.

Wie in der Grafik ersichtlich, zeigt der Hardlink ‚h-target' auf denselben Inode, somit hat er die gleichen Verweise auf die Datenblöcke wie die ursprüngliche Datei. Der Softlink ‚s-target' hingegen hat zeigt auf einen eigenen Inode der wiederum eigene Datenblöcke referenziert. In diesen Datenblöcken steht der Dateiname der Datei ‚source'. Mit Hilfe dieser Information ist das Betriebssystem in der Lage, die Datei ‚source' ausfindig zu machen, erhält somit auch den dazugehörigen Inode und kann somit auf die Datenblöcke, die dann die Daten der Datei beinhalten, zugreifen.

Hardlink vs. Softlink$_2$



Problematik: *source* wird gelöscht
Daten können nach wie vor durch Hardlink (h-target) angesprochen werden
Softlink (s-target) zeigt ins Leere (Daten können nicht angesprochen werden, obwohl Datenblock noch existiert)

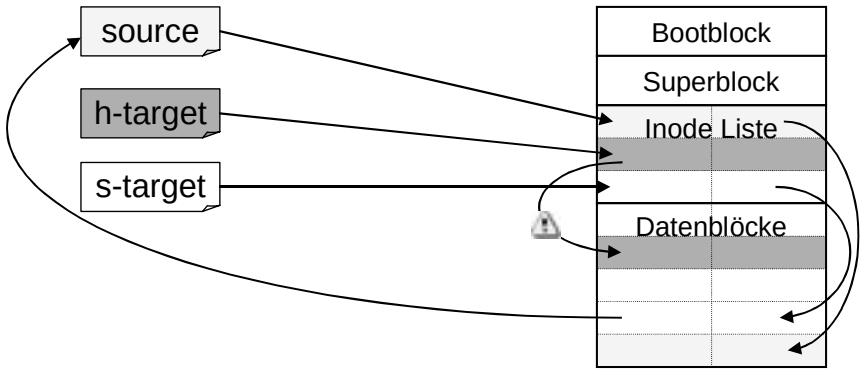
Unterschiede zwischen Hard- und Softlinks lassen sich erst in speziellen Situationen erkennen. Nehmen wir zum Beispiel einmal an, dass die Datei ‚source' gelöscht wird. Wir haben nun folgende Situation:

Die Datei ‚source' ist gelöscht. Da aber immer noch ein Hardlink auf den Inode zeigt, wird der Inode und die darin referenzierten Datenblöcke *NICHT* gelöscht. Obwohl der Dateiname ‚source' nicht mehr existiert, kann der Inhalt der Datei noch immer über den *Hardlink* bearbeitet werden, da dieser noch auf den alten Inode zeigt und somit die Referenzen zu den Datenblöcken hat.

Beim Softlink hingegen hat sich einiges verändert. Auch hier hat sich am Inode und am Datenblock des Links nichts geändert. Im Datenblock steht nach wie vor der Dateiname ‚source' referenziert. Will das Betriebssystem nun über den Softlink auf die Datei zugreifen, so wird der Dateiname zwar dem Betriebssystem bekannt gegeben, dieses kann aber den Dateinamen nicht mehr auflösen, da es den Dateinamen nicht mehr gibt. In diesem Fall würde ein Zugriffsversuch über den *Softlink* fehlschlagen. Die Daten können also über den Softlink nicht mehr angesprochen werden, obwohl es den ursprünglichen Inode und die Datenblöcke mit allen Daten noch immer gibt!

ⓘ Existiere in unserem Beispiel aber kein Hardlink mehr, der auf die Datei ‚source' zeigen würde, so würden auch der Inode und die Datenblöcke der Datei ‚source' nicht mehr existieren.

Hardlink vs. Softlink$_3$



Problematik: Neue Datei *source* wird angelegt (gleicher Name, gleicher Pfad)
Hardlink (h-target) spricht Daten der **alten** Datei *source* an, nicht der neuen!
Softlink (s-target) spricht Daten der neuen Datei *source* an, obwohl die Datei zum Erstellungszeitpunkt des Links noch nicht existiert hat.

Hard- und Softlinks unterscheiden sich voneinander aber noch in einem Detail. Nehmen wir an, die Datei ‚source' wurde gelöscht (siehe voriges Beispiel). Über den Hardlink kann die alte Datei trotzdem noch immer angesprochen werden. Tritt nun der Fall ein, das eine neue Datei im Verzeichnis erzeugt wird, die genau den selben Dateinamen hat wie die alte Datei – also auch ‚source' – dann passiert folgendes:

1. Es wird in der Inode Liste ein *neuer* Inode angelegt.
2. Dieser Inode reserviert *leere* Datenblöcke, in die er die Daten der Datei ablegt.

Die Situation ist nun folgende:

Der Hardlink zeigt immer noch auf den Inode der *alten* Datei ‚source', somit existieren noch immer die Daten dieser Datei und können noch immer über den Hardlink bearbeitet werden.

Der Softlink hingegen, der in seinem eigenen Datenblock ja den Dateinamen ‚source' gespeichert hat, zeigt nun auf die *neue* Datei ‚source', obwohl die Datei zum Erstellungszeitpunkt des Links noch gar nicht existiert hat.

! Es ist nun wichtig zu verstehen, dass der Hardlink nun andere Daten referenziert als der Softlink. Bei Unachtsamkeit können sich hier schwer erkennbare Fehler einschleichen, vor allem wenn Benutzer immer noch der Meinung sind, der Hardlink zeige auf die gleiche Datei wie der Softlink.

Kommandos zur Dateimanipulation

Dateimanipulationen$_1$

- `cat datei [datei]`
  - Liest die angegebene(n) Datei(en) und gibt diese auf der Standardausgabe aus.
  - `cat dat1 dat2 > dat3`
    - fügt die Inhalte aus *dat1* und *dat2* zusammen und speichert diese in *dat3*.
- `cd [directory]`
  - Wechselt das Arbeitsverzeichnis.
  - Ohne Parameter wechselt cd ins Homeverzeichnis.

Im Folgenden werden nun die wichtigsten Kommandos zur Dateimanipulation behandelt. Dabei werden nur die wichtigsten Parameter und Optionen angegeben und deren Funktionalität kurz erklärt. Detailliertere Informationen zu den Kommandos finden Sie wie immer in den entsprechenden Online Hilfeseiten (Manual Pages).

Anzeigen eines Dateiinhaltes (/bin/cat)

Das Kommando „cat *dateiliste*“ liest den Inhalt der angegebenen Datei(en) aus und gibt diese auf der Standardausgabe aus. Durch Ausgabeumleitung (nähere Informationen siehe Kapitel 6) können Inhalte aus mehreren Dateien in eine Datei zusammenkopiert werden.

Change Directory (Internes Kommando)

Das Kommando ‚cd' wechselt das aktuelle Arbeitsverzeichnis. „cd *Verzeichnis*“ wechselt in das angegebene Verzeichnis. Die Verzeichnisangabe kann hier absolut oder relativ erfolgen.

Das Kommando ‚cd' wurde bereits in der Einführung der UNIX-Kommandos detailliert betrachtet.

Dateimanipulationen$_2$

- `cp [optionen] quelle(n) ziel`
  - Kopiert Dateien und Verzeichnisse
    - Mehrere Dateien in ein Verzeichnis
    - Duplizierung eines gesamten Verzeichnisbaums
  - Wichtigste Optionen:

-i	Das eventuelle Überschreiben von Zieldateien muss vom Benutzer bestätigt werden
-p	Preserve: Neben dem Inhalt der Datei werden auch alle sicherheitsrelevanten Informationen, wie Benutzer und Gruppe, mit kopiert.
-r	Rekursives Abarbeiten aller Dateien im angegebenen und aller darunter liegenden Verzeichnisse
-R	Rekursives Abarbeiten unter Berücksichtigung spezieller Dateien (/dev/zero)

Dateien kopieren (/bin/cp)

Das Kommando ‚cp' kopiert Dateien und Verzeichnisse. Es gibt zwei prinzipielle Kopiervorgänge. Entweder es werden mehrere Dateien in ein Verzeichnis kopiert oder ein gesamter Verzeichnisbaum wird dupliziert (Option –r). Dem Kommando kann eine Reihe von Optionen übergeben werden, gefolgt von einer oder mehreren Quellen und einem Ziel. Werden mehrere Dateien (>=3) angegeben, so wird die letzte als Zielangabe interpretiert und alle vorigen Dateien als Quelldateien. Ist die Zielangabe ein Verzeichnis, werden die Quellen in das angegebene Verzeichnis kopiert, ist die Zielangabe aber eine Datei, schlägt der Kopierversuch fehl (anders als bei DOS).

ⓘ Werden in MS-DOS mehrere Dateien als Quelle angegeben und eine Zieldatei (also kein Verzeichnis) dann kopiert MS-DOS im Unterschied zu UNIX die Inhalte aller Quelldateien und fügt diese in eine Zieldatei ein.

Im Folgenden werden die wichtigsten Optionen des Kommandos aufgelistet:

-i	Falls das Kopieren eine existierende Datei überschreiben würde, wird vorher vom Benutzer eine Erlaubnis zum Überschreiben der existierenden Datei eingeholt.
-p	Preserve: Neben dem Inhalt der Datei werden auch alle sicherheitsrelevanten Informationen (wie z.B. Benutzer, Gruppeninformationen) mitkopiert.
-r	Diese Option bewirkt das rekursive Abarbeiten aller Dateien im angegebenen und aller darunter liegenden Verzeichnissen. Mit Hilfe dieser Option können ganze Verzeichnisbäume dupliziert werden.
-R	Diese Option hat die selbe Wirkung wie –r, jedoch werden spezielle Dateien (wie z.B. /dev/zero) berücksichtigt. Solche Dateien würden während dem Kopieren Probleme verursachen und müssen besonders behandelt werden.

Dateimanipulationen$_3$

- `mv [optionen] quelle(n) ziel`
  - Verschiebt Dateien und Verzeichnisse oder benennt diese um
  - Wichtigste Option:

-i	Das eventuelle Überschreiben von Zieldateien muss vom Benutzer bestätigt werden

- `rm [optionen] datei(en)`
  - Löschen von Dateien oder Verzeichnissen
  - Wichtigste Optionen:

-i	Das Löschen jeder Datei muss bestätigt werden
-f	Löschen ohne Nachfrage bei schreibgeschützten Dateien
-r 	Rekursives Löschen, es werden auch alle Dateien in darunter liegenden Verzeichnissen gelöscht

Dateien umbenennen / verschieben (/bin/mv)

Dateien können mit dem Kommando ‚mv' umbenannt oder in ein anderes Verzeichnis verschoben werden. Das Kommando selbst ist ähnlich aufgebaut, wie das ‚cp'-Kommando. Dem Kommandonamen können Optionen folgen, danach wird eine Dateiliste erwartet. Die Auswertung der Dateiliste erfolgt analog zu der des ‚cp'-Kommandos. Befindet sich die Quell- und Zieldatei im selben Verzeichnis, so wird die Quelldatei in die Zieldatei umbenannt. Befinden sich die Dateien hingegen in unterschiedlichen Verzeichnissen, so wird die Quelldatei in das Zielverzeichnis verschoben. Die wichtigste Option des Kommandos ist ‚-i', dass vor einem eventuellen Überschreiben beim Verschieben (oder Umbenennen) nach Erlaubnis fragen würde. Fehlt die Angabe, würden eventuell bestehende Dateien ohne Nachfrage überschrieben werden.

Dateien löschen (/bin/rm)

Das Kommando ‚rm' löscht im Normalfall Dateien. Es können aber – durch Angabe von Optionen – auch ganze Verzeichnisse (inkl. Inhalt) durch dieses Kommando gelöscht werden. Dem Kommandonamen selbst können Optionen folgen, zumindest wird jedoch eine Dateiliste der zu löschenden Dateien erwartet. Die wichtigsten Optionen des Kommandos sind:

-i	Bevor eine Datei gelöscht wird, muss der Löschvorgang vom Benutzer nochmals bestätigt werden. Diese Option empfiehlt sich, wenn mit Platzhaltern (à la ‚*' und ‚?') gearbeitet wird.
-f	Diese Option unterdrückt das Einholen einer Bestätigung wenn schreibgeschützte Dateien gelöscht werden sollen. Im Normalfall würde das rm-Kommando vor dem Löschen solcher Dateien nachragen, ob dies wirklich erwünscht ist.
-r	Rekursives Löschen. Diese Option ist wohl die gefährlichste und sollte besonders bedacht angewandt werden. Diese Option löscht nicht nur die Dateien, die in der Dateiliste angegeben wurden, sondern auch alle darin befindlichen Verzeichnisse (und natürlich die Dateien, die in diesen Verzeichnissen liegen).

Dateimanipulationen$_4$

- `ln [optionen]` *`quelle ziel`*
  - Erstellt einen Link mit dem Namen *ziel* auf die Datei *quelle*.
  - Wichtigste Optionen:

-f	Unterdrückt die Nachfrage, falls die Datei schreibgeschützt ist.
-s	Erzeugt einen „Symbolischen Link“

- `mkdir name`
  - Erstellt ein neues Verzeichnis mit dem angegebenen Namen
- `rmdir name`
  - Löschen eines leeren Verzeichnisses mit dem angegebenen Namen

Links auf Dateien erstellen (/bin/ln)

Die in diesem Kapitel besprochenen Links können mit Hilfe des ‚ln'-Kommandos erstellt werden. Dem Kommandonamen können Optionen folgen. Danach wird eine Quelldatei und eine Zieldatei erwartet. Je nach gewählter Option muss die Quelldatei existieren (Hardlink) oder auch nicht (Softlinks). Die wichtigsten Optionen des Kommandos sind:

-f	Diese Option unterdrückt die Nachfrage einer Bestätigung, falls eine Datei mit dem Zielnamen bereits existiert und überschrieben werden soll.
-s	Diese Option bewirkt die Erzeugung eines Softlinks. Wird die Option nicht angegeben, so erzeugt das Kommando einen Hardlink.

Verzeichnis erstellen (/bin/mkdir)

Das Kommando ‚mkdir' erzeugt ein neues Verzeichnis, sofern der Schreibschutz des übergeordneten Verzeichnisses dies zulässt. Dem Kommando muss ein Verzeichnisname folgen. Die Standardeinträge ‚.' und ‚..' werden im neuen Verzeichnis automatisch erstellt.

Leeres Verzeichnis löschen (/bin/rmdir)

Das Kommando ‚rmdir' löscht ein Verzeichnis, sofern dieses *leer* ist und keine weiteren Unterverzeichnisse enthält. Verzeichnisse, die nicht leer sind oder weitere Unterverzeichnisse enthalten, können mit dem Kommando ‚rm –r *verzeichnisname*' gelöscht werden. Dem Kommando ‚rmdir' folgt der Verzeichnisname der gelöscht werden soll. Dieser kann entweder absolut oder relativ angegeben werden.

Dateimanipulationen$_5$

- `more (dateiname(n))`
  - Zeigt die angegebene Datei am Bildschirm seitenweise aus.
    - Falls keine Datei angegeben wird, wird Standardeingabe genommen
  - Nach jeder Seite wird auf einen Tastendruck gewartet.
    - Leertaste: Seitenweise weiterblättern
    - Return-Taste: Nächste Zeile
    - h: Hilfemenü
    - b: Zurückblättern
    - /: Suchen
    - q: Beenden
- `less (dateiname(n))`
  - Wie ‚more', jedoch erweiterter Befehlssatz.
  - Nähere Informationen → man Pages

Dateiinhalte anzeigen (/bin/more)

Das Kommando ‚more' bietet neben ‚cat' und ‚less' eine weitere Möglichkeit, Dateiinhalte zu betrachten. Jedoch stoppt hier die Ausgabe nach einer Bildschirmseite. Durch Tastendruck kann weitergeblättert werden oder abgebrochen werden.

Dem Kommando folgt eine Dateiliste, die Dateien enthält, die angezeigt werden sollen. Wird keine Dateiliste angegeben, so wird die Standardeingabe als Quelle hergenommen.

Nach der Ausgabe einer Seite wartet das Kommando auf einen Tastendruck. Im Folgenden ist eine Liste der erlaubten Zeichen und deren Funktion verzeichnet:

Leertaste	Blättert auf die nächste Seite
Return-Taste	Zeigt die nächste Zeile
H	Zeigt ein Hilfemenü auf dem Bildschirm
B	Blättert eine Seite zurück
/	Möglichkeit zur Suche im Text
Q	Beenden

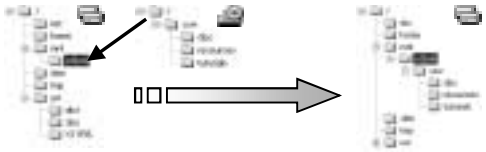
Dateiinhalte anzeigen (/usr/bin/less)

Das Kommando ‚less' hat die gleiche Funktion wie das more-Kommando und ist diesem auch in der Bedienung sehr ähnlich, bietet aber einen erweiterten Befehlssatz. Dieser ist auf der entsprechenden Hilfeseite ersichtlich oder kann in den man Pages nachgeschlagen werden.

Dateisysteme einbinden

Dateisysteme einbinden$_1$

- UNIX kennt keine Laufwerksbuchstaben
  - Mehrere Festplatten, CD-ROMs, Bandgeräte, Diskettenlaufwerke, etc.
  - Einhängen des Dateisystems in den Verzeichnisbaum
    - Verzeichnisbaum erweitert sich um die Verzeichnisse des neuen Dateisystems



UNIX erlaubt selbstverständlich die Verwendung mehrerer Platten, Bandgeräte, Diskettenlaufwerke, etc., die aber nicht über Laufwerksbuchstaben – wie sie aus DOS oder Windows bekannt sind – angesprochen werden. Vielmehr können alle diese Geräte in das Dateisystem an einer beliebigen Stelle eingebunden werden. Dabei erweitern sie das vorhandene Dateisystem an der eingebundenen Stelle.

Jede Platte enthält ein vollständiges Dateisystem. Die Koppelung erfolgt durch das Ersetzen eines vorhandenen Verzeichnisses durch das Dateisystem der eingebundenen Platte. Dabei ist der ursprüngliche Inhalt des Verzeichnisses nicht mehr zugänglich, ist aber nach dem Aushängen des Dateisystems wieder verfügbar.

Durch das Aushängen wird das Dateisystem wieder aus dem Verzeichnisbaum entfernt. Es lassen sich auch Dateisysteme anderer Rechner einbinden (über das NFS oder Network File System, dass im Kapitel Vernetzung näher behandelt wird). Dateisysteme, die beim Booten des Rechners eingebunden werden sollen, stehen in der Datei ‚/etc/fstab' (oft auch /etc/vfstab).

Dateisysteme einbinden$_2$

- `mount [optionen] [gerät] [directory]`
  - Einbinden eines Gerätes *gerät* in den Verzeichnisbaum *directory*
  - Solange Datenträger gemountet ist, darf er nicht gewechselt werden
- `umount gerätename`
  - Abmelden eines Gerätes *gerätename* aus dem Dateisystem
  - Dateipuffer wird auf Platte geschrieben
  - Nach umount können Datenträger gewechselt werden

Im folgenden werden nun die beiden Kommandos zum Ein- und Aushängen von Dateisystemen behandelt.

Dateisysteme einbinden (/bin/mount)

Das Kommando ‚mount' bindet ein Dateisystem, dass sich auf einem Gerät befindet, in den Verzeichnisbaum ein. Dem Kommandonamen können Optionen folgen, danach werden ein Gerätename und ein Directory erwartet. Das Directory identifiziert den Mountpoint, das ist jenes Verzeichnis, dass durch das Dateisystem des Gerätes erweitert / ersetzt wird. Ohne Angabe von Parametern erhält man eine Liste der eingebundenen Geräte. Die zum Zeitpunkt des Bootvorgangs einzubindenden Dateisysteme lassen sich mit ‚mount –a' auch später einbinden. Lassen sich diese Systeme nicht automatisch einbinden (z.B. Disketten, Wechselplatte, CD-ROM, usw.), können sie später durch „mount *Directory*" eingebunden werden, ohne dass eine Geräteangabe nötig wäre, sofern sie in der Datei ‚/etc/fstab' aufgeführt sind. Je nach UNIX-Variante hat das mount-Kommando noch weitere Parameter, z.B. den Typ des Dateisystems oder Nur-Lesezugriff. So lassen sich auch andere Typen von Dateisystemen einbinden, z.B. unter Linux auch MS-DOS oder Windows Partitionen.

Austauschbare Datenträger dürfen nicht gewechselt werden, solange das Dateisystem dieses Datenträgers eingebunden ist. Das ursprüngliche Dateisystem erweitert sich um die Dateien und Verzeichnisse des neuen Datenträgers. Würden diese plötzlich verschwinden kann es zu Problemen in der Dateisystemverwaltung kommen.

Dateisysteme aushängen (/bin/umount)

Das Kommando ‚umount' meldet eine Platte aus dem Dateisystem ab. Es darf kein Benutzer oder Prozess mehr offene Dateien auf dieser Platte haben, sonst schlägt das Abmelden fehl. Beim Abmelden werden auch alle Dateipuffer auf die Platte geschrieben. Wechseldatenträger (Diskette, Wechselplatte) sind unbedingt vor dem Herausnehmen abzumelden.

Verfügbaren Plattenplatz beschränken

Disk Quotas

- Große Probleme wenn Festplatte voll → Absturz, Dateisystem Fehler,...
- Quotas beschränken zur Verfügung stehenden Speicherplatz (pro Partition)
  - Benutzerquota
    - Benutzer steht bestimmter Speicherplatz zur Verfügung
  - Gruppenquota
    - Speicherplatz der Mitgliedern einer Gruppe zur Verfügung steht
- Hard-Limit
  - Absolute Grenze
- Soft-Limit
  - Grenze, die bis zum Hard-Limit überschritten werden kann

Da UNIX im laufenden Betrieb immer wieder Prozesse auf die Harddisk auslagert, kann es zu großen Problemen kommen, wenn die Festplatte voll wird. UNIX bietet aber eine Möglichkeit an, den Plattenplatz der einem Benutzer zur Verfügung steht, zu begrenzen, nämlich durch sogenannte Quotas.

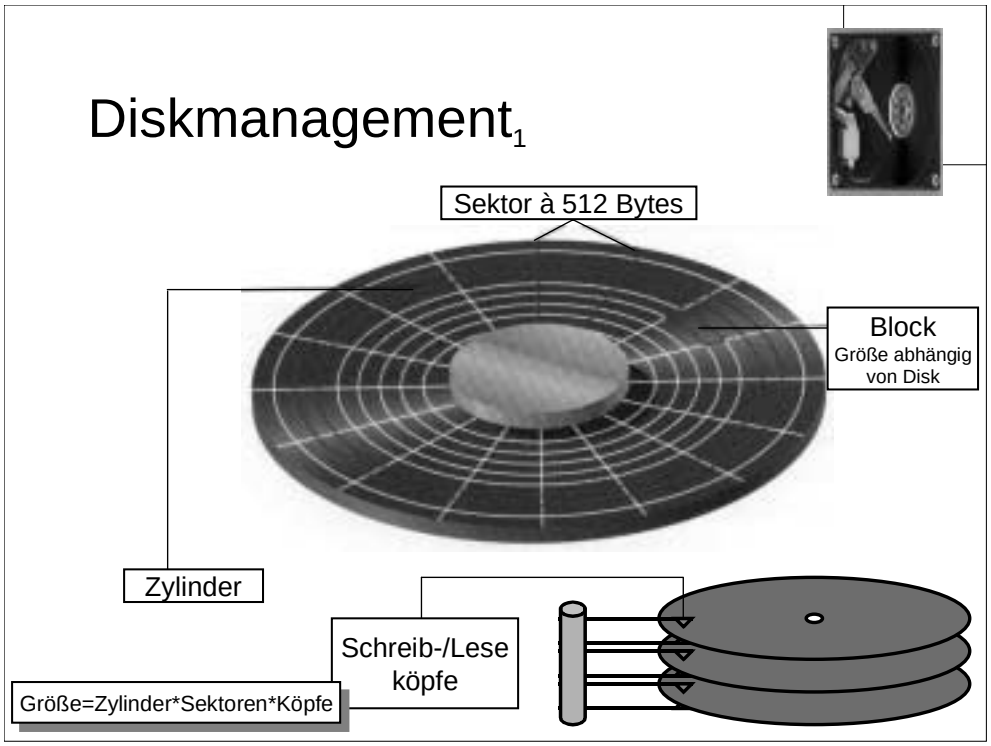
Quotas beschränken den zur Verfügung stehenden Speicherplatz pro Partition. Dabei wird angegeben, wie viel Platz von Benutzern oder Gruppen auf einer Partition in Anspruch genommen werden kann. Prinzipiell werden zwei Arten von Quotas unterschieden:

- Benutzerquotas: Diese Art von Quotas legt fest, wie viel Platz einem Benutzer auf einer Partition zur Verfügung steht. Ist die festgelegte Grenze erreicht, kann der Benutzer keine Dateien mehr in der betroffenen Partition ablegen.

- Gruppenqutoas: Diese Art Quota gibt die Summe des Speicherplatzes an, den alle Mitglieder dieser Gruppe gemeinsam belegen dürfen. Wird die Grenze erreicht, können Mitglieder der betroffenen Gruppe auf der Partition keine Daten mehr ablegen.

Damit Benutzer aber rechtzeitig gewarnt werden, unterscheidet UNIX zwischen dem Soft- und dem Hardlimit. Wird das Softlimit erreicht, wird der Benutzer beim Speichern von Dateien entsprechend gewarnt. Er kann trotzdem noch Dateien auf die Partition speichern, solange bis er das Hardlimit erreicht. Dieses Limit kann nicht mehr überschritten werden.

Platten Management



Festplatten bestehen aus einer Reihe von übereinander gestapelten Magnetplatten. Auf mehreren konzentrischen Spuren werden die Daten Bit für Bit seriell in Form von magnetisierten Punkten geschrieben und gelesen. Der schnell rotierende Plattenstapel und ein beweglicher Zugriffsmechanismus mit je einem Schreib-Lese-Kopf pro Plattenoberfläche (also zwei pro Magnetplatte) ermöglichen sowohl ein sequentielles Lesen und Schreiben von Datenbeständen als auch einen schnellen direkten Zugriff zu jedem beliebigen Datensatz. Die Platten selbst sind in Spuren aufgeteilt, die Daten speichern können. Die Spuren sind wiederum in Sektoren unterteilt. Üblich ist eine Einteilung in Sektoren mit jeweils 512 Byte. Ein Sektor innerhalb einer Spur wird als Block bezeichnet. Die auf dem Plattenstapel genau übereinanderliegenden Spuren ergeben jeweils einen Zylinder. Die Anzahl der Zylinder ergibt sich somit aus der Anzahl der Spuren multipliziert mit der Anzahl der Schreib-/Leseköpfe. Der Zugriffsarm mit den Schreib-/Leseköpfen wird dabei auf solche Zylinder eingestellt. Dann können alle Daten auf die entsprechenden Spuren geschrieben oder von den entsprechenden Spuren gelesen werden.

Die Größe einer Platte errechnet sich dabei aus der Anzahl der Sektoren multipliziert mit der Anzahl der Zylinder.

Diskmanagement$_2$

- Partitionieren
  - Logische Aufteilung der Festplatte in getrennte Bereiche
    - z.B. für verschiedene Dateisysteme
    - Mehrere Betriebssysteme auf einer Festplatte
  - mind. 1 Partition erforderlich
- UNIX-Kommando: `fdisk`
- DOS-Kommando: `fdisk`

Partitionieren

Bevor eine Festplatte mit einem Dateisystem versehen werden kann, muss sie erst partitioniert werden. Durch das Partitionieren wird die Festplatte in Datenbereiche aufgeteilt. Grundsätzlich gibt es zwei Arten von Partitionen:

- Primäre Partitionen: Verweisen direkt auf den Bereich der Festplatte, der die Dateien enthält. Meistens sind Betriebssysteme in primären Partitionen installiert, die meisten setzen dies sogar zwingend voraus. Eine Festplatte kann bis zu vier primäre Partitionen enthalten.

- Erweiterte Partitionen: Diese enthalten im Gegensatz zu primären Partitionen keine Dateien, sondern eine Art Container für weitere Partitionen. Diese werden dann als logische Laufwerke eingerichtet. Dadurch ist es möglich, die Festplatte in mehr als nur vier logische Bereiche zu trennen.

Jede Festplatte muss aber mindestens über eine Partition verfügen, zumeist ist dies eine primäre Partition! Während der Installation von Linux werden Sie aufgefordert, Partitionen für das neue Betriebssystem einzurichten. Dabei wird auch noch eine spezielle Partition ins Spiel gebracht, die sogenannte *swap*-Partition. Diese wird für Swapping-Zwecke vom Betriebssystem verwendet.

Unter UNIX kann über das Kommando ‚fdisk' die Partitionstabelle der einzelnen Platten eingesehen werden bzw. verändert werden. Dieses Kommando gibt es übrigens auch unter DOS (Windows) unter dem gleichen Namen. Das Kommando ist jedoch mit äußerster Sorgfalt anzuwenden und es ist vorher auf jeden Fall die Systemdokumentation zu konsultieren.

Diskmanagement$_3$

- Formatieren
  - Versieht eine Partition mit einem Dateisystem
  - UNIX-Kommando: `mkfs`
  - DOS-Kommando: `format`
- Prüfen des Dateisystems
  - Prüfen Dateisystem auf Fehler
  - UNIX-Kommando: `fsck`
  - DOS-Kommando: `scandisk`

Formatieren

Bevor die Partitionen einer Festplatte aber genutzt werden können, müssen diese zuerst mit einem Dateisystem versehen werden. Diesen Vorgang nennt man auch Formatieren.

Das UNIX-Kommando dazu heißt ‚mkfs'. Gebrauch und Parameter muss man der jeweiligen Dokumentation entnehmen. Auch unter DOS oder Windows muss eine Partition erst formatiert werden. Hierzu wird das Kommando ‚format' hergenommen.

! Wie auch das Kommando ‚fdisk' ist das Kommando zum Formatieren einer Partition mit äußerster Vorsicht zu verwenden. Bei unachtsamer oder unsachgemäßer Anwendung werden alle Daten unwiederbringlich gelöscht.

Prüfen des Dateisystems

Zum Überprüfen der Platten, eigentlich des Dateisystems, gibt es das Kommando ‚fsck'. Es wird normalerweise beim Booten automatisch ausgeführt. Für den Fall des manuellen Aufrufs sollte man auch hier die Dokumentation des jeweiligen Systems konsultieren.

! Das Prüfen von Dateisystemen sollte eigentlich nur auf ‚read-only' gemounteten Dateisystemen passieren, da sonst Probleme auftreten können.

Kapitel

4

Benutzerverwaltung in UNIX

Dieses Kapitel erläutert die Benutzer- und Gruppenverwaltung in UNIX.

UNIX bietet die Möglichkeit, zwischen einzelnen Benutzern des Systems zu unterscheiden. Jeder eingetragene Benutzer eines UNIX-Systems hat einen eigenen Datenbereich, das Homeverzeichnis, in dem er seine eigenen Dateien und seine eigene Arbeitsumgebung verwalten kann. Auch die Prozesse, die ein Benutzer ausführt, laufen unter seiner persönlichen Kennung ab. Nur der Benutzer selbst kann die eigenen Prozesse kontrollieren – mit Ausnahme des Systemverwalters. Dieses Kapitel soll zeigen, wie Benutzer auf einem UNIX-System administriert werden und welche Tools und Programme zur Benutzerverwaltung bereitstehen.

Lehrinhalte und -ziele

Nach diesem Kapitel sollten Sie in der Lage sein, verschiedene Benutzer in einem UNIX-System anzulegen und zu administrieren. Sie sollten in der Lage sein, Gruppen anzulegen und Benutzer zu Gruppen zuzuordnen. Darüber hinaus sollten Sie verstehen, wie Benutzer und Gruppeninformationen im System abgelegt werden.

Notwendigkeit der Benutzerverwaltung

Warum Benutzerverwaltung

- Mehrere Benutzer können sich am System anmelden
  - Speichern der persönlichen Daten und Einstellungen
  - Verschiedene Benutzer verschiedene Rechte
  - Jeder Benutzer eigene Daten
  - Nicht jeder darf sich im System anmelden
- Sicherheit auf Systemebene (definiertes Sicherheitsmodell)
- Jeder Benutzer Login-Name und Passwort → Kombination identifiziert den Benutzer
- Management der Daten durch die Benutzerverwaltung

UNIX bietet die Möglichkeit, dass sich mehrere Benutzer an einem System zur gleichen Zeit anmelden können. Dabei werden persönliche Daten und Einstellungen der Benutzer gespeichert und sind in weiteren Sitzungen verfügbar. Somit kann sich jeder Benutzer das System zurechtschneidern, ohne die Systemumgebung anderer Benutzer zu beeinflussen.

In solchen Multiuser Systemen spielt Sicherheit eine ganz markante Rolle. Es gibt auf jedem System verschiedenste Benutzer die unterschiedliche Aufgaben zu erledigen haben. Je nach Aufgabe werden gewisse Rechte im System benötigt, Zugriff auf unterschiedliche Ressourcen gewährt und unterschiedliche Programme gestartet. Natürlich muss gewährleistet sein, dass persönliche und sensible Daten nicht von anderen Benutzern gelesen werden können. Letztendlich muss auch gewährleistet werden, dass nur bestimmte Benutzer sich am System anmelden dürfen und nicht jeder die Ressourcen, die das System bereitstellt, nutzen kann.

UNIX bietet diese Sicherheit auf Systemebene. Durch ein fest definiertes Sicherheitsmodell wird gewährleistet, dass nur berechtigte Personen Zugang zum System und zu Ressourcen bekommen. Das Sicherheitsmodell wird im nächsten Kapitel noch näher erläutert.

Diese Sicherheit wirkt auf Benutzerebene, d.h. das Betriebssystem muss in der Lage sein, unterschiedliche Benutzer zu erkennen. Dabei erhält jeder Benutzer einen Login-Namen (auch Benutzerkonto oder Account genannt) und ein Passwort. Die Kombination dieser Daten authentifiziert einen Benutzer im System. Das Management dieser Daten erfolgt durch die Benutzerverwaltung.

Benutzerverwaltung

Die Benutzerverwaltung ist in UNIX ebenso offen aufgebaut, wie der Rest des Systems. Die Benutzer werden über eine zentrale Datei, nämlich ‚/etc/passwd' verwaltet. Jeder Benutzer ist mindestens einer Gruppe zugeordnet. Gruppen werden über die Datei ‚/etc/group' verwaltet. Um zusätzliche Sicherheit zu gewährleisten, findet man auf vielen UNIX-Systemen auch eine Datei namens ‚/etc/shadow'. Diese Datei trennt die Benutzerinformationen von den Passwörtern und ist im Normalfall nur dem Superuser zugänglich.

Wenn sich ein Benutzer am System anmeldet (Login), wird auf diese Dateien zugegriffen. Da einzelne Prozesse für ihre Tätigkeit immer wieder Informationen aus diesen Dateien benötigen, sind die Dateien ‚/etc/passwd' und ‚/etc/group' für alle Benutzer lesbar, jedoch schreibgeschützt. Lediglich der Superuser und das Programm ‚passwd' dürfen auf diese Datei schreiben. Die Datei ‚/etc/shadow' ist hingegen nur vom Systemadministrator und Superuser-Prozessen lesbar.



Jeder Prozess, der unter UNIX gestartet wird, läuft unter der Kennung des Benutzers, der den Prozess gestartet hat. Es ist jedoch möglich, dass Prozesse – unabhängig von der Person, die ihn gestartet hat – unter der Kennung des Eigentümers laufen (SUID-Bit). Mit dieser Möglichkeit können Systembenutzer einen Prozess starten, der unter der Kennung des Benutzers ‚root' läuft (sofern ‚root' Eigentümer des Prozesses ist) und die Benutzerverwaltungsdateien ändert. Für nähere Informationen bezüglich Sicherheit und SUID-Bit wird auf das nächste Kapitel verwiesen.

Login

Login

- `login` ist eigener Prozess
- Prozess erhält Login-Name und Passwort
- Prozess sucht Benutzer und vergleicht Passwort
- Bei erfolgreicher Anmeldung wird ein Programm gestartet (normalerweise eine Shell)
- Daten über Benutzer werden in Datei `/etc/passwd` und `/etc/shadow` gespeichert

Nachdem UNIX gebootet hat, wird als erstes der login-Prozess gestartet. Hier wird auf die Eingabe eines Benutzernamens und des dazugehörigen Passworts gewartet. Der login-Prozess durchsucht nun die Datei ‚/etc/passwd' nach einem passenden Benutzernamen. Wird kein passender Benutzer gefunden, so wird der Benutzer abgewiesen und neuerlich auf die Eingabe eines Benutzernamens und ein Passwort gewartet.

Wird der Benutzer in der Datei ‚/etc/passwd' gefunden, so wird überprüft, ob in dieser Datei auch ein Passwort eingetragen ist. Falls dieses vorhanden ist, wird das eingegebene Passwort mit dem Gespeicherten verglichen. Stimmen die Passwörter überein, so wird das im Benutzereintrag spezifizierte Programm gestartet. In der Regel ist dies die Shell. Stimmt das Passwort jedoch nicht überein, wird der Benutzer abgewiesen und der login-Prozess wartet neuerlich auf die Eingabe eines Benutzernamens und Passworts.

! UNIX speichert Passwörter NICHT im Klartext, da dies zu unsicher wäre. Vielmehr wird ein mathematischer Einweg-Algorithmus verwendet, der aus dem eingegebenen Passwort einen sogenannten Hash-Code berechnet. Die Berechnung funktioniert dabei nur in eine Richtung, d.h. aus dem Hash-Code kann nicht mehr auf das ursprüngliche Passwort geschlossen werden. Selbst der Superuser kann nicht das Passwort eines Benutzers einsehen. Bei jedem Login wird aus dem eingegebenen Passwort mit Hilfe des gleichen Algorithmus der Hash-Code berechnet und es werden dann lediglich die Hash-Codes miteinander verglichen.



Neuere UNIX-Systeme speichern das Passwort (oder eigentlich den Hash-Code) getrennt von den restlichen Benutzerdaten in der Datei ‚/etc/shadow'. Der Login-Prozess vergleicht den Hash-Code in diesem Falle mit dem der in der Shadow-Datei gespeichert ist. Die Datei ‚/etc/passwd' enthält dann keine Passwortinformationen mehr.

Benutzerdaten

Daten über Benutzer

- Folgende Daten in einer Zeile getrennt durch ‚:' gespeichert, Zeile darf max. 511 Zeichen enthalten
  - Login-Name
  - Passwort (immer 13 Zeichen), bei modernen Systemen ein x
  - User ID (0 - 50 000; 1-99 reserviert)
  - Group ID (0 - 50 000, 100 ist Sammelgruppe)
  - Kommentar (z.B. Name des Benutzers)
  - Home-Verzeichnis
  - Programm das gestartet werden soll

Benutzerrelevante Informationen werden in der Datei ‚/etc/passwd' gespeichert. Jeder Prozess ist in der Lage, diese Datei zu lesen, verändern darf sie jedoch nur der Superuser.

Aufbau der Datei ‚/etc/passwd'

Die Datei ist eine normale Textdatei und enthält für jeden Benutzer genau eine Zeile mit sieben Feldern. Die einzelnen Felder sind durch einen Doppelpunkt voneinander getrennt. Die Länge der einzelnen Felder ist variabel, die Zeilenlänge darf dabei aber nicht 511 Zeichen überschreiten. Jede Zeile der Datei repräsentiert ein Benutzerkonto, jedes Feld spezifische Benutzerinformationen.

Eine Zeile in der Datei hat folgendes Format:

```
Login-Name:Passwort:UID:GID:Kommentar:Home-Verzeichnis:Programm
```

Im folgenden werden nun die einzelnen Felder näher beschrieben:

Login-Name: Der Login-Name ist der Name, den der Benutzer beim Login angeben muss. Er repräsentiert sozusagen das Benutzerkonto und ist einem Benutzer eindeutig zugewiesen. Dieser Name ist im System eindeutig und kann nicht mehrmals verwendet werden.

Passwort: In älteren UNIX-Versionen wird in diesem Feld der Hash-Code des Benutzerpasswortes gespeichert. Der Hash-Code ergibt sich aus einer mathematischen Berechnung und ist immer 13 Zeichen lang. Ist dieses Feld leer, dann muss kein Passwort eingegeben werden. Bei modernen UNIX Systemen

findet man hier aber nur ein ‚x'. In diesem Fall ist der Hash-Code des Passwortes in der Datei ‚/etc/shadow' abgespeichert.

UID: Die UID ist die sogenannte User Identification Number. Jedem Benutzer wird eine eindeutige UID zugeordnet. Der Wertebereich liegt hierbei von 0 bis 50.000 wobei es keine zwei gleichen UIDs im System geben kann. Die UID 0 ist für den Superuser reserviert, die UIDs 1 – 99 für interne Zwecke. Reguläre Konten beginnen meist bei einer definierten Grenze (z.B. 500). Das System selbst arbeitet bei Sicherheitsauswertungen nicht mit dem Login-Namen sondern mit der UID und der GID. Dies verhindert, dass ein neuer Benutzer, der zufällig den Login-Namen eines früheren Benutzers erhält, auch auf dessen Dateien und Ressourcen zugreifen darf, falls diese noch immer im System gespeichert sind.

GID: Die GID ist die Group Identification Number. In diesem Feld wird festgehalten, zu welcher Gruppe der Benutzer gehört. Auch hier gibt es wieder einen festen Wertebereich (0 bis 50.000) wobei die GIDs von 1 – 99 meist für interne Zwecke verwendet werden.

Kommentar: In diesem Feld werden allgemeine Hinweise zum Benutzer gespeichert. Normalerweise sind dies der vollständige Name des Benutzers, die Abteilung, etc. Dieses Feld wird meist von Mail- und News-Programmen abgefragt um automatisch den Absender einzutragen.

Home-Verzeichnis: Beim Erstellen eines Benutzers wird ihm auch ein Arbeitsverzeichnis, das sogenannte Home-Verzeichnis, eingerichtet. Nach einem Login befindet sich der Benutzer normalerweise in diesem Verzeichnis. Das Feld Home-Verzeichnis enthält den absoluten Pfad zu diesem Verzeichnis.

Programm: In diesem Feld wird das Programm oder der Prozess angegeben, der bei einem erfolgreichen Login gestartet werden soll. Normalerweise wird hier die Shell eingetragen. Da UNIX unterschiedliche Shells bereitstellt, kann hier die vom Benutzer bevorzugte Shell eingetragen werden, die dann auch gestartet wird. Fehlt hier der Eintrag, wird die Standard-Shell gestartet. Es kann aber auch sein, dass ein Benutzer nur mit einem einzigen Programm arbeiten soll (und darf). Dann kann hier dieses Programm eingetragen werden, dass auch automatisch nach dem erfolgten Login gestartet wird. Wird dieses Programm beendet, wird auch die Sitzung geschlossen und es erscheint neuerlich der Login-Prozess.

Beispiele

Im Folgenden werden nun zwei Beispiele behandelt, die Einträge in die Datei ‚/etc/passwd' repräsentieren. Ziel ist es, die Felder der Einträge zu verstehen und die Auswirkungen der Einträge zu erkennen.

```
Karl:x:235:100:Karl Müller:/usr/karl:/bin/sh
```

Dieser Eintrag deutet darauf hin, dass es im System ein Konto ‚Karl' gibt. Das Passwort dieses Benutzers ist dabei in der Datei ‚/etc/shadow' gespeichert. Der Benutzer hat die UID ‚235' und gehört der Gruppe ‚100' an. Der vollständige Name des Benutzers ist ‚Karl Müller' und sein Home-Verzeichnis liegt in ‚/usr/karl'. Nach erfolgreichem Login wird die Shell ‚/bin/sh' gestartet.

```
Hans:7894561237894:500:500:TEST:/home/test:/usr/prg
```

Dieser Eintrag deutet darauf hin, dass es im System ein Konto ‚Hans' gibt. Der Hash-Code des Benutzerpassworts ist dabei direkt in der Datei gespeichert. Der Benutzer hat die UID ‚500' und gehört auch der Gruppe ‚500' an. Der Kommentar deutet darauf hin, dass es sich lediglich um ein Testkonto handelt, dessen Home-Verzeichnis in ‚/home/test' liegt. Nach erfolgreichem Login wird das Programm ‚/usr/prg' gestartet. Wird dieses beendet, so wird die Sitzung geschlossen und der Login-Prozess erscheint neuerlich am Bildschirm.

Die Shadow-Datei

Warum /etc/shadow

- Jeder User braucht Leserechte auf /etc/passwd → Sicherheitslücke
- Nur Superuser darf /etc/shadow lesen und schreiben
- Speichert u.a. Hash-Codes der Passwörter

Wie schon zuvor erwähnt, speichern neuere UNIX-Systeme die Passwörter getrennt von den restlichen Benutzerinformationen in der Datei ‚/etc/shadow'.

Sicherheitsgewinn durch ‚/etc/shadow'

Die Verwendung der Shadow-Datei ist ein großer Sicherheitsgewinn, da diese Datei – im Gegensatz zur Datei ‚/etc/passwd' nur von *root* gelesen werden kann. Wie schon zuvor erwähnt, werden Passwörter nicht im Klartext gespeichert sondern als Parameter für einen mathematischen Einweg-Algorithmus hergenommen, der mit Hilfe des Passworts einen Hash-Code berechnet. Man kann zwar aus dem Hash-Code nicht mehr auf das ursprüngliche Passwort schließen, jedoch muss der Algorithmus zur Berechnung des Hash-Codes bekannt sein, damit das System durch andere Programme (z.B. Administrationstools) erweitert werden kann. Dieses Faktum ermöglicht sogenannte Brute-Force Attacken. Hierbei berechnet ein Programm aus beliebigen Zeichenkombinationen so lange Hash-Codes, bis ein Code gefunden wird, der mit dem Hash-Code des Passworts übereinstimmt. Ist das der Fall, so hat man auch das Passwort gefunden.

Wie man sieht, steht und fällt das Sicherheitskonzept der verschlüsselten Passwörter mit dem Schutz der Passwort Hash-Codes. Da die Datei ‚/etc/shadow' nun nicht mehr lesbar ist, können andere Benutzer auch nicht die Hash-Codes auslesen. Das kann einzig und allein der Superuser.

Aufbau der Datei ‚/etc/shadow'

Die Datei ist ähnlich aufgebaut die wie Datei ‚/etc/passwd', sie enthält also mehrere Felder die durch Doppelpunkte voneinander getrennt sind.

Eine Zeile besteht aus 9 Textfeldern und hat folgendes Format:

```
Login-Name:Passwort:letzte Änderung:Min:Max:Vorwarnzeit:
Inaktiv:Verfall:Kennzeichen
```

Im folgenden werden nun die einzelnen Felder beschrieben:

Login-Name: Hier steht derselbe Name wie in der Datei ‚/etc/passwd'. Dieser Eintrag verbindet die beiden Dateien.

Passwort: Normalerweise steht hier ein 13 stelliges Passwort. Moderne Systeme haben aber stärkere Verschlüsselungsalgorithmen und darum kann hier auch eine längere Zeichenkette stehen. Es gilt: Je stärker der Algorithmus, desto länger ist der Hash-Code. Anstelle des Passwortes kann hier aber auch ein Sperrvermerk – angezeigt durch das Zeichen ‚*' – stehen. Wird dieser Sperrvermerk gefunden, so kann das Konto nicht benutzt werden.

Letzte Änderung: Hier ist der Tag der letzten Passwortänderung verzeichnet. Gezählt wird in Tagen ab dem 01.01.1970, dem offiziellen Entstehungstag von UNIX.

Min: Hier steht die minimale Gültigkeitsdauer des Passworts in Tagen. Ein Passwort kann innerhalb dieser minimalen Gültigkeitsdauer nicht verändert werden, erst nachdem diese Frist abgelaufen ist, kann das Passwort wieder verändert werden. Meist steht in diesem Feld eine ‚0' die ein sofortiges Ändern des Passworts erlaubt.

Max: Hier steht die maximale Gültigkeitsdauer des Passworts in Tagen. Der Benutzer *muss* vor Ablauf dieser Frist sein Passwort ändern, ansonsten kann er sich nicht mehr im System anmelden.

Vorwarnzeit: Der Benutzer wird in der Regel über die Ablauffrist seines Passwortes informiert. Die Vorwarnzeit gibt die Zeitspanne (Anzahl der Tage) an, in der ein Benutzer im vorhinein auf die notwendige Änderung des Passworts hingewiesen werden soll.

Inaktiv: Hier wird die Zeitspanne (Anzahl der Tage) angegeben, in der ein Benutzerkonto unbenutzt sein darf. Wird die Anzahl überschritten dann wird auch das Konto gesperrt.

Verfall: Hier wird die Gültigkeitsperiode eines Kontos angegeben. Das Datum wird dabei absolut angegeben und bei Erreichen dieses Datums wird das Konto gesperrt.

Kennzeichen: Dieses Feld ist reserviert für künftige Verwendung und meistens leer.

Damit nicht alle Werte beim Anlegen eines Benutzers von Hand eingegeben werden müssen, können in der Datei ‚/etc/default/useradd' Standardwerte gespeichert werden.

Benutzergruppen

Benutzergruppen

- Erleichtern Verwaltung von Benutzern
- Gruppen können Rechte zugewiesen werden
- Ein Benutzer kann zu mehreren Gruppen gehören
- Mindestens aber eine Gruppe (= Hauptgruppe in `/etc/passwd`)
- Gruppen in `/etc/group` gespeichert
  - Aufbau: Gruppenname:Passwort:GID:Benutzernamen

Oftmals können gewisse Benutzer, die gleiche Tätigkeiten im Unternehmen durchführen, zu Benutzergruppen zusammengefasst werden. Benutzergruppen zeichnen sich dadurch aus, dass sie dieselben Ressourcen eines Systems benutzen und die gleichen Ziele verfolgen.

UNIX kann gleichartige Benutzer zu Benutzergruppen zusammenfassen und erleichtert somit die Verwaltung von Benutzern. Dabei können Rechte sowohl an einzelne Benutzer, aber auch an ganze Benutzergruppen zugewiesen werden. Es ist somit möglich, dass ein Benutzer keinen individuellen Zugriff auf eine Datei hat, aber einer Gruppe angehört die diese Datei bearbeiten darf. Somit darf der Benutzer über die Gruppenrechte die Datei bearbeiten.

Ein Benutzer kann mehreren Gruppen zugeordnet werden, gehört aber immer mindestens einer Benutzergruppe (Hauptgruppe) an. Die Hauptgruppe wird in der Datei ‚/etc/passwd' gespeichert während zusätzliche Gruppenzuordnungen in der Datei ‚/etc/group' gemacht werden. Die Datei ‚/etc/group' enthält aber *ALLE* Gruppen und gibt somit Aufschluss, welche Gruppen innerhalb eines Systems existieren. Die Datei kann von Benutzern nur gelesen werden, das Schreiben ist dem Superuser vorbehalten.

Die Datei enthält einzelne Datensätze, die jeweils einer Gruppe verschiedene Benutzer zuordnen. Jeder dieser Datensätze hat folgendes Format:

```
Gruppenname:Passwort:GID:Benutzernamen
```

Im Folgenden werden nun die einzelnen Felder näher beschrieben:

Gruppenname: Hier wird der Gruppenname angegeben. Dieser Name wird auch im ‚ls –l' Kommando angeführt.

Passwort: Es ist möglich, für Gruppenzugehörigkeit ein Passwort zu vergeben. Diese Möglichkeit wird aber normalerweise nicht verwendet.

GID: In diesem Feld wird die Group Identification Number der Benutzergruppe angegeben. Hier sind Werte von 0 bis 50.000 möglich, wobei die GIDs von 1 – 99 meist für interne Zwecke verwendet werden.

Benutzernamen: Dieses Feld enthält eine Liste von Benutzern, die der Gruppe angehören. Einzelne Benutzer werden hier durch Beistriche voneinander getrennt.

Beispiele

Im Folgenden werden nun zwei Beispiele erläutert, die Einträge in die Datei ‚/etc/group' behandeln. Ziel ist es, die Felder der Einträge zu verstehen und die Auswirkungen der Einträge zu erkennen.

```
employees::300:mary,john,andy,lucy
```

Dieser Eintrag definiert eine Benutzergruppe namens ‚employees'. Für diese Benutzergruppe ist kein Passwort vorgesehen. Die Gruppe wird durch die Group Identification Number ‚300' referenziert. Die Benutzer ‚mary', ‚john', ‚andy' und ‚lucy' sind die Mitglieder dieser Gruppe.

```
test:7894561237894:529:testworker
```

Dieser Eintrag deutet darauf hin, dass es im System eine Gruppe ‚test' mit der Group Identification Number ‚529' gibt, für die ein Passwort vergeben ist. Der Hash-Code dieses Passwortes ist im zweiten Feld gespeichert. Die Gruppe hat nur ein Mitglied nämlich den Benutzer ‚testworker'.

Standardbenutzer

Standardbenutzer

- Bereits während der Installation angelegt → Standardbenutzer (meist Prozesse)
  - root UID=0
    - Superuser
  - daemon
    - Für Systemserverprozesse zuständig
  - bin
    - Gehören die ausführbaren Dateien der user-Kommandos
  - sys
    - Gehören bestimmte Systemdateien
  - Weitere

In UNIX gibt es einige Benutzerkonten, die bereits während der Installation des Systems angelegt werden. Diese sind die sogenannten Standardbenutzer. Jeder dieser Standardbenutzer hat dabei eine bestimmte Aufgabe und oftmals stehen hinter diesen Benutzer gar keine realen Personen sondern Prozesse die bestimmte Privilegien benötigen. In weiterer Folge werden nun einige dieser Standardbenutzer aufgelistet:

- root: Die Benutzerkennung root bezeichnet den Systemadministrator oder Superuser. Ein Benutzer der unter dieser Kennung arbeitet unterliegt keinen Sicherheitsrestriktionen mehr und darf alles im System machen. Um das System vor Viren und anderen Attacken zu schützen, ist es sehr ratsam wenn diese Kennung nur für administrative Zwecke benutzt wird. In UNIX arbeitet üblicherweise selbst der Systemadministrator mit einem normalen Benutzerkonto und übernimmt für administrative Zwecke kurzfristig die root-Kennung (Kommando ‚su').

- daemon: Dieses Konto wird benötigt, um System-Server Prozesse auszuführen. Hintergrundprozesse (auch Dämonen-Prozesse genannt) benötigen oftmals Zugriff auf Ressourcen. Dieser Zugriff kann mit dem Konto daemon reglementiert werden.

- bin: Dieses interne Konto wird für die ausführbaren user Kommandos verwendet. Dem Konto bin gehören alle user Kommandos im System.

- sys: Diesem internen Konto werden normalerweise Systemdateien zugeschrieben.

- Weitere: Je nach System gibt es zahlreiche weitere Standardkennungen wie z.B. adm für die Verwaltung von Accounting, cron für die zeitliche Ausführung von Hintergrundprozessen, usw. Die Anzahl und Funktion der weiteren Standarduser entnehmen Sie im Normalfall dem Systemhandbuch.

Mit Ausnahme von ‚root' werden diese Benutzerkonten nur selten für Logins benutzt, sodass diese Benutzer zwar in der Passwortdatei existieren, deren Passworteintrag aber deaktiviert ist.

Benutzerverwaltung

Benutzer/Gruppen anlegen

- Benutzer: Händisch, Shell-Skript oder ‚useradd'
- Gruppen: Händisch, Shell-Skript oder ‚groupadd'
- Distributionsabhängiges Tool (z.B. YaST, Linuxconf)
- Folgende Schritte notwendig
  - Benutzer in `/etc/passwd` eintragen
  - Evt. auch Eintrag in `/etc/shadow`
  - In `/etc/group` Eintrag für Standardgruppe
  - Home-Verzeichnis erzeugen (+ eventuell .profile-Datei)
  - Neuen Benutzer als Eigentümer für Home-Verzeichnis und .profile-Datei setzen (‚chown')

Es ist Aufgabe des Systemadministrators, die Benutzer seines Systems zu verwalten. Dazu gehört das Anlegen von Benutzerkonten, das Sperren und Löschen solcher Konten aber auch das Festlegen von Sicherheitsrichtlinien (z.B. nach wie vielen Tagen das Passwort eines Benutzers geändert werden muss). In weiterer Folge werden wir uns mit den einzelnen Aufgaben näher vertraut machen.

Benutzer anlegen

In UNIX gibt es mehrere Möglichkeiten um Benutzer zu erstellen:

Manuelles Erstellen eines Benutzers

Beim manuellen Erstellen von Benutzern muss der Systemadministrator selbst einzelne Konfigurationsdateien mit Hilfe eines Editors manipulieren

1. Zuerst muss der neue Benutzer in die Datei ‚/etc/passwd' eingetragen werden. Dazu wird ein neuer Datensatz (vorzugsweise am Ende der Datei) angelegt und die Felder entsprechend dem Aufbau der Datei ausgefüllt. Die Passwortvergabe bedarf allerdings spezieller Anstrengungen. Je nachdem ob eine Shadow-Datei verwendet wird oder nicht, ist zusätzlich noch die Datei ‚/etc/shadow' zu editieren.

   Wird keine Shadow-Datei verwendet, so ist es am besten, man lässt das Passwort-Feld leer, da man sonst selbst den Hash-Code des Passworts berechnen müsste. Nach Erstellen des Kontos sollte sich der Systemadministrator entweder mit dem neuen Kontonamen am System anmelden und ein Passwort vergeben oder das passwd-Kommando mit Angabe des neuen Kontos aufrufen.

ⓘ Wenn das passwd-Kommando vom Superuser gestartet wird, kann zusätzlich ein Parameter angegeben werden, der das zu ändernde Konto spezifiziert. „passwd test“ würde z.B. das Passwort des Kontos ‚test' ändern.

Wird eine Shadow-Datei verwendet, so muss in dieser ein zum neuen Benutzerkonto korrespondierender Eintrag erstellt werden. Hier sollte wiederum das Feld in dem der Hash-Code des Passworts gespeichert wird, leer bleiben, die anderen Felder sind aber auszufüllen. Wiederum sollte sich der Systemadministrator nach Erstellen des Kontos entweder am System mit dem neuen Kontonamen anmelden und ein Passwort (z.B. mit dem Kommando „passwd“) vergeben oder das passwd-Kommando mit Angabe des neuen Kontos aufrufen.

2. Der Benutzer wird in der Datei ‚/etc/passwd' natürlich auch einer Gruppe (Hauptgruppe) zugeordnet. Diese Zuordnung muss sich natürlich auch in der Datei ‚etc/group' wiederspiegeln. Falls die zugewiesene Gruppe bereits existiert, muss der Benutzer nur der Benutzerliste der Gruppe hinzugefügt werden. Soll aber zusammen mit dem Benutzer auch eine neue Gruppe im System angelegt werden, so ist auch ein neuer Datensatz in der Datei ‚etc/group' anzulegen und die Felder entsprechend der Eigenschaft dieser Gruppe auszufüllen. Es ist auf jeden Fall darauf zu achten, dass die GID der Datei ‚etc/group' mit der GID der Datei ‚etc/passwd' übereinstimmt.

3. Der neue Benutzer kann auch anderen Gruppen zugeordnet werden. Dabei ist jeweils die Benutzerliste der Gruppe um den neuen Benutzer zu erweitern.

4. Als vierter Schritt muss das Homeverzeichnis des Benutzers erstellt werden. Der Pfad für das Homeverzeichnis wird in der Datei ‚/etc/passwd' spezifiziert. Gegebenenfalls ist es auch nötig, im Homeverzeichnis des Benutzers eine Datei namens ‚.profile' anzulegen. Mit Hilfe dieser Datei kann die Umgebung dem Benutzer angepasst werden (Einträge in dieser Datei bewirken z.B. das setzen von Shell-Umgebungsvariablen oder beinhalten Kommandos die nach dem Login ausgeführt werden sollen).

   ⓘ Es hat sich in der Praxis als sinnvoll erwiesen, dass der Systemadministrator allgemein gültige Dinge in einer ‚.profile'-Vorlagendatei definiert. Wenn ein neuer Benutzer angelegt wird, wird diese Datei einfach in sein Homeverzeichnis kopiert. Der Systemadministrator spart sich dadurch administrativen Mehraufwand der durch das individuelle Einstellen der Umgebung entstehen würde.

5. Als letzter Schritt muss dafür gesorgt werden, dass der Benutzer alle Rechte auf sein Homeverzeichnis und die darin befindlichen Dateien besitzt. Dazu dient das Kommando ‚chown', mit Hilfe dessen man den Besitzer einer Ressource festlegen oder ändern kann. Wird dieser Schritt nicht durchgeführt so ist immer noch der Systemadministrator Besitzer der Ressource und dem Benutzer ist es dann meist nicht erlaubt, Änderungen an der Umgebung vorzunehmen oder Programme zu installieren.

Erstellen eines Benutzers mittels eines Shellskripts

Shellskripte erlauben die Automatisierung von manuellen Tätigkeiten. Es ist leicht vorstellbar, dass die oben erwähnten 5 Schritte zum manuellen Hinzufügen von Benutzern mit Hilfe eines Shellskripts automatisiert werden können. Dabei führt das Skript sequentiell die einzelnen Schritte durch, ohne dass eine Aktion seitens des Systemadministrators nötig wäre. Der Systemadministrator stößt nur mehr das Skript an und übergibt dem Skript die für seine Aufgabe notwendigen Informationen in Form von Parametern.

Das Erstellen von Skripts wird im Kapitel 7 „Shell Programmierung“ näher behandelt.

Verwendung des Kommandos ‚useradd'

Die meisten Systeme bieten heute ein Standard-Tool, dass vom Systemadministrator aufgerufen werden kann und die Einträge in die entsprechenden Dateien erstellt. Je nach System heißt dieses Kommando „useradd“ oder „adduser“. Das Tool erwartet nur einen Namen des Benutzerkontos das angelegt werden soll. Optional können aber noch andere Einstellungen wie zu vergebende UID, Standardgruppe, Homeverzeichnis, Passwort, etc. angegeben werden. Ruft man das Kommando ohne Parameter auf, so listet es die möglichen Optionen und Parameter auf. Natürlich gibt es zu dem Kommando auch entsprechende Manual-Pages die weitere Hilfestellung zum Kommando bieten.

Verwendung sonstiger Tools

Fast jede Distribution bietet heutzutage eine komfortable Möglichkeit, Benutzer und Gruppen des Systems zu verwalten. Diese Tools bieten meist eine Oberfläche und erlauben auch spätere Änderungen der Benutzerdaten. Leider hat sich bis dato noch kein Standard herauskristallisiert und darum setzt jede Distribution eigene Tools ein, die sich von der Bedienung und vom Aussehen stark voneinander unterscheiden. Darüber hinaus ist es nicht gewährleistet, dass diese Tools beim Aufsetzen des Systems automatisch mitinstalliert werden oder das dasselbe Tool in der nächsten Version der Distribution noch eingesetzt wird.

Benutzer sperren / löschen

- Sperren
  - In `/etc/passwd` oder `/etc/shadow` Eintrag für Passwort z.B. auf ‚*' setzen (kann nicht als Ergebnis bei der Verschlüsselung entstehen)
- Löschen
  - Löschen des Home-Verzeichnisses
  - Löschen des Mail-Verzeichnisses (`/var/mail`)
  - Löschen des Benutzers aus `/etc/passwd`
  - Bereinigung der Datei `/etc/group`

Benutzer sperren

Manchmal ist es aus Sicherheitsgründen sinnvoll, gewisse Benutzer zu sperren. Ist es z.B. vorhersehbar, dass ein bestimmter Benutzer das System für längere Zeit nicht nutzen wird so sollte sein Benutzerkonto gesperrt werden um den Missbrauch des Kontos vorzubeugen.

Wie schon zuvor in diesem Kapitel erwähnt, wird jeder Benutzer durch eine Kombination aus Kontonamen und Passwort authentifiziert. Um einen Benutzer zu sperren, genügt es somit, sein Passwort so zu manipulieren, dass er sich nicht mehr am System anmelden kann. Dazu muss einfach die Datei in der das Passwort gespeichert ist (‚/etc/passwd' oder ‚/etc/shadow') in einem Editor geöffnet werden und das entsprechende Feld das den Hash-Code des Passworts enthält verändert werden. Trägt man hier einen Wert ein der durch die Berechnungsfunktion für den Hash-Code nie erreicht werden kann (z.B. ‚*'), ist es nicht mehr möglich das Konto zu benutzen. Das Konto kann dann jederzeit durch Zurücksetzen des Passworts (leeres Feld) reaktiviert werden.



In der Praxis hat es sich als sehr sinnvoll erwiesen, dem Hash-Code einfach ein Sonderzeichen (z.B. ‚!') voranzustellen. Dadurch wird der eigentliche Hash-Code nicht gelöscht, es ist aber durch das Sonderzeichen nicht mehr möglich den Wert durch die Hash-Funktion zu erreichen. Soll das Konto reaktiviert werden, so ist einfach das Sonderzeichen wieder zu löschen. Das Konto kann dann wieder mit dem *alten* Passwort benutzt werden.

Benutzer löschen

Manchmal ist es notwendig, ein Benutzerkonto zu löschen. Damit verliert ein Benutzer permanent das Nutzungsrecht eines UNIX-Systems. Ein späteres reaktivieren des Kontos ist nicht mehr möglich. Um einen Benutzer zu löschen sind folgende Schritte notwendig:

1. Löschen des Homeverzeichnisses: Dateien sind in UNIX immer einem Eigentümer zugeordnet der alle Rechte über die Datei hat. Jedem Benutzer ist in der Regel auch ein Homeverzeichnis zugeordnet, über dessen Inhalt der Benutzer alle Rechte hat. Im Normalfall gehören alle dort darin gespeicherten Dateien dem korrespondierenden Benutzer. Im ersten Schritt müssen alle diese Dateien und auch das Homeverzeichnis des Benutzers gelöscht werden. Geschieht dies nicht so „verwaisen“ diese Dateien und sind später keinem Eigentümer mehr zugeordnet.

2. Löschen des Mailverzeichnisses: Jeder Benutzer bekommt automatisch eine Datei im Mailverzeichnis, sobald das erste Mal ein Mail zugestellt wird. Diese Datei trägt den Namen des Benutzerkontos und ist standardmäßig im Verzeichnis ‚/var/mail’ oder ‚/var/spool/mail’ abgelegt. Außerdem kann es sein, dass der User zeitlich gesteuerte Prozesse (sogenannte Cron-Jobs) definiert hat. Diese speichern Ihre Daten im Verzeichnis ‚/var/spool/cron’ in eine Datei die den Namen des Benutzerkontos trägt. Falls diese Datei vorhanden ist, ist diese ebenfalls zu löschen.

3. Löschen des Benutzers aus der Datei ‚/etc/passwd’ ggf. ‚/etc/shadow’: Nachdem die einzelnen Benutzerdateien gelöscht wurden kann mit dem eigentlichen Entfernen des Benutzerkontos begonnen werden. Dazu muss einfach die den Benutzer betreffende Zeile in der Datei ‚/etc/passwd’ entfernt werden. Wird das Passwort in die Datei ‚/etc/shadow’ gespeichert, so muss auch in dieser die den Benutzer betreffende Zeile gelöscht werden.

4. Löschen der Gruppenzuordnung (Datei ‚etc/group’): Im letzten Schritt muss dann noch die Benutzer-Gruppenzuordnung gelöscht werden. Dazu wird die Datei ‚/etc/group’ geöffnet und der Benutzer jeweils aus der Benutzerliste der zugeordneten Gruppen gelöscht. Manchmal kann es vorkommen, dass einer Gruppe nur der zu löschende Benutzer und sonst niemand zugeordnet ist. In diesem Falle kann auch die Gruppe gelöscht werden. Dies wird durch das Entfernen der ganzen Zeile erreicht.

Kapitel

5

Zugriffsrechte in UNIX

Dieses Kapitel erläutert die UNIX Sicherheitsmechanismen auf Dateiebene und die damit verbundenen Zugriffsrechte.

Sicherheit wird immer mehr zu einem sehr wichtigen und weit diskutierten Problem. Immer wieder hört man davon, dass sich Personen unerlaubterweise Zugriff zu sensiblen Daten verschaffen. Meist liegt das aber an ungenügend implementierten Sicherheitsmechanismen im System oder an Administrationsfehlern. Dieses Kapitel befasst sich mit den UNIX Sicherheitsmechanismen, erläutert die Zugriffskontrolle auf Dateiebene und beschäftigt sich mit der Administration der Zugriffskontrolle.

Lehrinhalte und -ziele

Nach diesem Kapitel sollten Sie die Zugriffskontrollen und Dateiberechtigungen von UNIX verstehen und erklären können. Sie sollten in der Lage sein vergebene Rechte auf Dateien richtig interpretieren zu können, aber auch gegebenenfalls Zugriffsrechte zu ändern und zu steuern.

Zugriffsrechte in UNIX

Zugriffsrechte in UNIX

- Schutz privater Dateien vor Zugriff anderer Systembenutzer
- Eigentümerbasierte Zugriffskontrolle
- Granularität der Rechte
  - Eigentümer (user)
  - Benutzergruppe (group)
  - Alle anderen Anwender (others)
- Auswertung der Rechte
  - 1. Prüfen der Eigentümerschaft
  - 2. Prüfen der Zugehörigkeit zur Gruppe
  - 3. Prüfen der Zugehörigkeit zu anderen Anwendern

Sicherheit gewinnt im Zeitalter vernetzter Computer zunehmend an Bedeutung. Sensible und geheime Dokumente werden immer öfters elektronisch erfasst und auf Datenträgern abgelegt. Das Betriebssystem muss daher mit entsprechenden Mechanismen aufwarten, die den Schutz von sensiblen Informationen gewährleisten können.

UNIX hat seit je her ein ausgefeiltes Sicherheitsmodell implementiert. Dabei gewährleistet das Betriebssystem selbst Schutz vor unerlaubten Zugriffen anderer Systembenutzer auf private Dateien. Das System realisiert dabei eine leicht abgeänderte Form der Eigentümerbasierten Zugriffskontrolle, auch unter dem Begriff *Discretionary Access Control Model* (kurz DAC) bekannt.

Innerhalb des DAC-Modells wird sichergestellt, dass jede im Betriebssystem befindliche Ressource einem eindeutigen Eigentümer zugeordnet ist. Der Eigentümer einer Ressource hat dabei alle Rechte auf seine Ressource und darf frei darüber verfügen. Er ist übrigens auch der einzige, der anderen Benutzern Rechte auf seine Ressource gewähren oder wieder entziehen kann.

UNIX hebt sich vom DAC-Modell insofern ab, als dass es im System einen speziellen Benutzer (den Superuser) gibt, der keinen Sicherheitsrestriktionen unterliegt. Dieser Benutzer ist automatisch Eigentümer *aller* im System befindlichen Ressourcen und darf deshalb über die Ressourcen auch frei verfügen oder die Zugriffsrechte auf die Datei ändern. Es ist nicht möglich, dem Superuser Rechte zu entziehen.

UNIX bietet aber auch gruppenorientierte Zugriffsrechte. Der Eigentümer einer Ressource kann dabei bestimmen, welche im System befindlichen Gruppen zusätzlich Zugriff auf seine Ressource bekommen können.



Die Wissenschaft ist sich uneinig darüber, ob das UNIX-Sicherheitsmodell in die Kategorie DAC oder RBAC (Rollenbasierte Zugriffskontrolle fällt). In RBAC werden Rechte an Rollen vergeben anstatt an Benutzer. Vertreter der RBAC-Philosophie meinen, dass der Eigentümer einer Ressource einer Rolle „Eigentümer“ zugeordnet werden kann. Gerade Benutzergruppen erinnern stark an Rollen weil hier keiner individuellen Person ein Recht gegeben wird sondern einer Gruppe von Personen.

Fest steht jedoch, dass das UNIX-Sicherheitsmodell keiner dieser beiden Modellen entspricht. Vielmehr ist es eine Kombination beider Modelle.

Granularität der Rechte

Entsprechend dem oben beschriebenen Sicherheitsmodell kann jeder Benutzer, der Zugriff auf eine bestimmte Ressource haben will, in eine der folgenden Kategorien eingeordnet werden:

Eigentümer (owner): Falls der Benutzer auch der Eigentümer der Datei ist, so fällt er bei der Auswertung der Zugriffsregeln in die Kategorie „Eigentümer“. Die Auswertung erfolgt somit anhand der Zugriffsregeln, die für diese Kategorie definiert wurde.

Benutzergruppe (group): Jede Ressource ist auch einer Gruppe zugeordnet. Für diese Gruppe können Zugriffsrechte vergeben werden. Ist der Benutzer Mitglied dieser Gruppe gelten die auf Gruppenebene definierten Zugriffsregeln.

Alle anderen Anwender (others): Ist ein Benutzer weder Eigentümer noch Mitglied der Benutzergruppe, so fällt er in die Kategorie „alle anderen Anwender“. Diese Kategorie hat meistens keine oder nur sehr eingeschränkte Rechte auf eine Ressource.

Auswertung

Bei der Auswertung wird zuerst geprüft, ob der Benutzer in die Kategorie Eigentümer fällt. Ist dies der Fall, so gelten für ihn alle auf dieser Ebene definierten Rechte. Ist er nicht Eigentümer, so wird geprüft, ob der Benutzer derjenigen Gruppe angehört, für die Rechte auf die Datei definiert sind. Falls ja, so gelten für ihn die Gruppenrechte. Es kann durchaus vorkommen, dass ein Benutzer sowohl Besitzer als auch Mitglied der Gruppe ist. In diesem Fall gelten die auf Besitzerebene definierten Rechte, die Gruppenrechte werden ignoriert, selbst wenn diese mehr erlauben würden als auf Besitzerebene definiert ist. Sollte der Benutzer keiner der beiden zuvor erwähnten Kategorien angehören, so gelten für ihn automatisch die Rechte die für die Kategorie „*others*“ definiert wurden.

Das UNIX Rechtemodell im Detail

Rechte in UNIX

- Benutzer – Gruppenzuordnung
  - Jeder Benutzer gehört mind. einer Gruppe an
  - Kann mehreren Gruppen zugeordnet werden
- Eigentümer kann Rechte für eine Gruppe oder für andere vergeben
- Durchsetzung der Rechte wird vom Betriebssystem garantiert
- Probleme mit Gruppen
  - Gruppen als Mitglieder anderer Gruppen nicht möglich!

In UNIX wird jeder Benutzer mindestens einer Gruppe zugeordnet. Um die Administration zu erleichtern, kann der Benutzer mehreren Gruppen zugeordnet werden. Das hat den Vorteil, dass Rechte auf Gruppenebene vergeben werden können.

Vergabe von Rechten

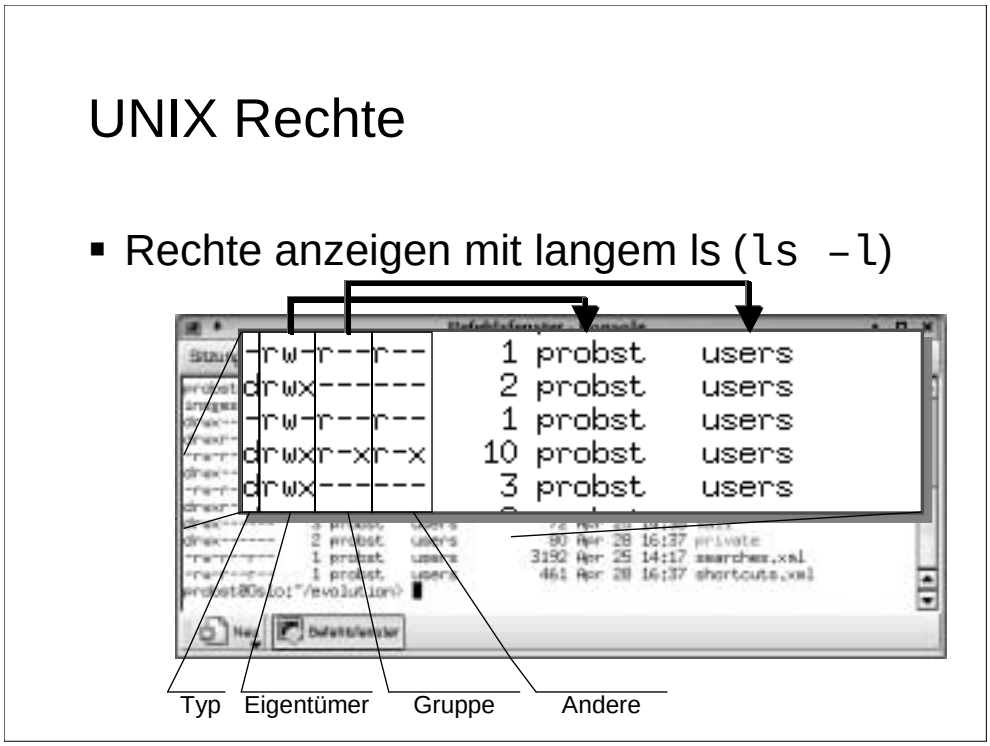
Es ist nur dem Eigentümer (mit Ausnahme des Superusers) vorbehalten, Rechte auf seine Ressource an andere weiterzugeben. Der Eigentümer hat dabei die Möglichkeit, Rechte an die der Ressource zugeteilten Gruppe zu vergeben oder an alle anderen Benutzer im System. Darüber hinaus kann der Eigentümer auch die Gruppe ändern, für die die Gruppenrechte geltend gemacht werden.

Die Durchsetzung der Rechte wird dabei vom Betriebssystem garantiert.

Probleme mit Gruppen

Mitglieder, die voraussichtlich die selben Ressourcen im System benutzen und die gleichen Aufgaben zu bewerkstelligen haben, sollten in Gruppen zusammengefasst werden. Leider ist diese Aufgabe aber nicht immer trivial zu lösen, da es in UNIX standardmäßig nicht möglich ist, Gruppen als Mitglieder anderer Gruppen zu definieren. Man stelle sich vor, dass im System zwei Gruppen existieren, die für eine bestimmte Teilaufgabe die selbe Ressource benutzen müssen. Für jede Ressource können aber Rechte nur für den Eigentümer, für *eine* Gruppe und für alle anderen definiert werden. In diesem Fall müsste man speziell eine neue Gruppe definieren, die genau diese Ressource benutzen darf und die Mitglieder der beiden anderen Gruppen dieser neuen Gruppe zuordnen.

Rechte anzeigen und interpretieren



Aufschluss über die auf eine Ressource definierten Rechte gibt das lange ls-Kommando. Führt man dieses Kommando aus („ls –l“), erhält man eine Liste mit Dateien und die darauf definierten Rechte. Diese stehen in der ersten, dritten und vierten Spalte der Ausgabe.

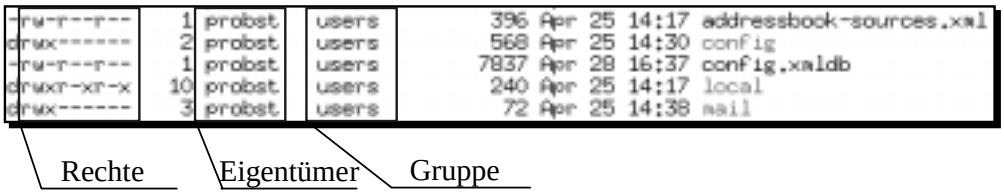


Abbildung 5-1: Rechte auf Dateien

Die erste Spalte der Ausgabe enthält Informationen zum Dateityp und die darauf definierten Berechtigungen. Die Berechtigungen sind jeweils in 3 Dreiergruppen zu unterteilen. Die erste Dreiergruppe gibt Aufschluss über die Berechtigungen des Besitzers der Ressource, die zweite Dreiergruppe gibt Aufschluss über die Berechtigungen die für die zugehörige Gruppe definiert wurden und die letzte Gruppe gibt Aufschluss über Berechtigungen die für alle anderen Benutzer definiert wurden. Die Dreiergruppe der Berechtigungen können jeweils ein ‚r' für Leserecht, ein ‚w' für Schreibrecht und ein ‚x' für Ausführrecht der Datei enthalten. Die einzelnen Rechte werden später in diesem Kapitel behandelt.

Die dritte Spalte gibt Aufschluss über den Eigentümer der Ressource. Hier wird im Normalfall der Name des Benutzerkontos des Eigentümers angezeigt. In der vierten Spalte steht im Normalfall der Name der Gruppe für die das Gruppenrecht gilt. Wie schon im Kapitel 4 „Benutzerverwaltung in UNIX“ erläutert wurde,

wertet UNIX auf Ebene der UID und GID Berechtigungsanfragen aus. Das hat den einfachen Grund, da es möglich ist, dass ein bestimmtes Benutzerkonto gelöscht wird und eine gewisse Zeit später für einen anderen Benutzer ein Konto mit demselben Namen erneut angelegt wird. In diesem Fall würde der neue Benutzer alle Rechte des alten Benutzers „erben“. Um in der Ausgabe anstatt den Benutzer- und Gruppennamen die jeweilige UID und GID anzeigen zu lassen, muss das ls-Kommando zusätzlich mit dem Parameter ‚-n’ aufgerufen werden (also „ls –ln“). Wenn ein Besitzer einer Datei oder eine zugehörige Gruppe nicht mehr existiert, zeigt UNIX automatisch die UID oder GID an. So können „herrenlose“ Ressourcen im Dateisystem ausfindig gemacht und gelöscht, oder neuen Besitzern zugewiesen werden.

Verschiedene Rechte und deren Bedeutung

Rechte

Recht	*Datei*	*Verzeichnis*
r (read)	Datei kann gelesen werden	Verzeichnisinhalt kann angezeigt werden
w (write)	Datei kann geschrieben werden	Verzeichniseinträge können erstellt werden
x (execute)	Datei kann ausgeführt werden	In Verzeichnis kann gewechselt werden

Wie schon erwähnt, gibt es drei verschiedene Berechtigungen (read, write, execute) für jede Ressource. Die Auswirkung der Berechtigung hängt dabei von der Art der Ressource ab.

Read-Recht

Das Read-Recht erlaubt es, eine Datei zu lesen. Die Datei kann geöffnet werden und ihr Inhalt betrachtet werden. Fehlt dieses Recht, hat der Benutzer keinerlei Möglichkeit, den Dateiinhalt zu betrachten. Das Read-Recht alleine erlaubt jedoch keine Modifikation der Datei.

Wird dieses Recht auf ein Verzeichnis angewendet, so darf der Benutzer den Verzeichnisinhalt einsehen, d.h. er darf sich z.B. mit dem ls-Kommando Informationen über die in diesem Verzeichnis gespeicherten Dateien verschaffen. Fehlt dieses Recht so hat der Benutzer keinerlei Möglichkeit sich den Inhalt des Verzeichnisses auflisten zu lassen.

Write-Recht

Das Write-Recht erlaubt es, in eine Datei zu schreiben. Ein Programm kann beispielsweise den Inhalt einer Datei durch Anfügen neuer Datensätze erweitern. Fehlt dieses Recht, so kann der Dateiinhalt nicht verändert werden.

In Kombination mit dem Read-Recht kann die Datei mit einem beliebigen Editor geöffnet und verändert werden. Fehlt das Read-Recht, können nur neue Datensätze angehängt werden oder bestehende Datensätze überschrieben werden. Die Datei selbst und ihr Inhalt kann in keinem Editor geöffnet werden.

Wird dieses Recht auf ein Verzeichnis angewendet, so kann der Benutzer neue Einträge im Verzeichnis erstellen. Dieses Recht erlaubt es, in einem Verzeichnis neue Dateien oder Unterverzeichnisse anzulegen oder Dateien in das Verzeichnis zu kopieren.

eXecute-Recht

Das eXecute-Recht (gekennzeichnet durch den Buchstaben ‚x') erlaubt das Ausführen einer Datei. Dieses Recht sollte nur auf ausführbare Dateien (z.B. Shellskripts) gegeben werden. Nur wenn dieses Recht vorhanden ist, startet die Shell einen neuen Prozess und führt die Datei aus.

Wird dieses Recht auf Verzeichnisse angewendet, so ist es dem Benutzer erlaubt, in dieses Verzeichnis zu wechseln. Fehlt dieses Recht so hat der Benutzer keine Möglichkeit, das Verzeichnis und die darunter liegenden Verzeichnisse zu seinem aktuellen Arbeitsverzeichnis zu machen. Der Benutzer hat auch keine Möglichkeit, auf Dateien zuzugreifen die in Verzeichnissen liegen, auf die er kein eXecute-Recht besitzt.

SUID-Kennzeichnung

Die SUID-Kennzeichnung ist ein spezielles Recht, das je nach Anwendung auf Benutzerebene oder Gruppenebene eine andere Auswirkung hat. Wird eine solche Kennzeichnung auf Benutzerebene auf eine ausführbare Datei gesetzt, so startet der damit verbundene Prozess nicht unter der Kennung des Aufrufers, sondern unter der Kennung des Eigentümers der Datei. Wird die Kennzeichnung auf Gruppenebene gesetzt, so startet der Prozess under der Kennung der zugeordneten Benutzergruppe.

ⓘ Oftmals wird dieses Recht auf spezielle Kommandos gesetzt, die einen höher priviligierten Zugriff auf Systemdateien benötigen. Ein Beispiel hierfür ist das Kommando „passwd", welches Schreibzugriff auf die Dateien ‚/etc/passwd' und ‚/etc/shadow' benötigt. Da diese Dateien nur der Benutzer ‚root' manipulieren darf, muss das entsprechende Kommando unter den Rechten dieses Benutzers laufen.

! Allerdings ist dieses Recht immer wieder eine potenitelle Gefahrenquelle, da die damit verbundenen Kommandos mit höherem Recht laufen, als dem Aufrufer eigentlich zustehen würde. Ein Angreifer wird immer versuchen, solche Kommandos entsprechend zu manipulieren, damit er entsprechend höhere Privilegien im System erreicht.

Sticky-Kennzeichnung

Wird diese Kennzeichnung auf Verzeichnisse angewendet, dürfen Benutzer *nur* ihre *eigenen* Dateien löschen, auch wenn sie ausreichend Rechte hätten andere Dateien ebenfalls zu löschen. Diese Kennzeichnung wird oftmals auf öffentlichen Verzeichnissen angewendet, wo Benutzer Dateien gegenseitig freigeben können. Allerdings ist es nur dem Eigentümer erlaubt, seine Dateien wieder zu löschen.

Rechte vergeben

Rechtevergabe

- Eigentümer und Superuser können Zugriffsrechte auf Dateien ändern
- Rechte können relativ oder absolut vergeben werden
- Relativ
  - Bestehende Rechte werden erweitert oder vermindert
- Absolut
  - Bestehende Rechte werden mit den zugewiesenen Rechten überschrieben
- 2 Modi
  - Symbolischer Modus
  - Direkter Modus

Nur der Eigentümer oder der Superuser können Rechte auf Ressourcen vergeben. Dazu dient das Kommando ‚chmod'. Das Kommando kann auf verschiedene Art und Weise aufgerufen werden, je nachdem wie Rechte vergeben werden sollen. Rechte können absolut vergeben werden oder relativ. Bei der absoluten Rechtevergabe werden bestehende Rechte mit den zugewiesenen Rechten überschrieben, d.h. es ist unrelevant welche Rechte zum Zeitpunkt der Rechtevergabe bereits existieren. Die relative Rechtevergabe berücksichtigt die aktuellen Rechte und erweitert oder vermindert diese.

Das Kommando ‚chmod' kann in zwei Modi aufgerufen werden. Im symbolischen Modus können Rechte über sogenannte Symbole vergeben werden. Der Aufruf des Kommandos in diesem Modus ähnelt stark einer mathematischen Gleichung und muss zumindest einen Operanden und eine Berechtigung enthalten, meistens wird noch ein Bereich vorangestellt. Im Direktmodus definiert eine drei bis vierstellige Oktalzahl die Rechte. Jede Stelle dieser Zahl definiert einen Benutzerbereich, der Wert jeder Stelle definiert *absolut* die Berechtigung.

Rechtevergabe im symbolischen Modus

Bei dieser Art von Rechtevergabe erwartet das chmod-Kommando die Angabe eines Bereiches, eines Operanden und der Berechtigung. Der Bereich gibt dabei an, welche Kategorie von Benutzer (u = Eigentümer, g = Gruppe, o = Alle anderen) von der Rechtevergabe betroffen sind. Bereiche können kombiniert werden (z.B. ‚go' würde das Recht für die Gruppe und für alle anderen setzen). Wird kein Bereich spezifiziert, gilt das Recht automatisch für alle drei Bereiche (käme also der Bereichsangabe ‚ugo' gleich). Der Operand gibt an, ob das Recht hinzugefügt werden soll (‚+'), weggenommen werden soll (‚-') oder absolut gesetzt werden soll (‚='). Zum Schluss wird noch die Berechtigung selbst erwartet. Abhängig von den

angegebenen Buchstaben (beliebige Kombination ist natürlich möglich) werden dann die unterschiedlichen Rechte gesetzt:

- r: Read-Recht: Erlaubt das Lesen einer Datei oder das Anzeigen des Verzeichnisinhaltes.
- w: Write-Recht: Erlaubt das Schreiben in eine Datei oder das Recht neue Dateien in einem Verzeichnis anzulegen.
- x: Execute-Recht: Erlaubt das Ausführen einer Datei oder das Wechseln in ein Verzeichnis.
- s: SID-Bit: Spezielles Recht, dass beim Ausführen der Datei einen anderen Benutzer zur Rechteauswertung hernimmt. Wird das SUID-Bit gesetzt, so wird die Datei unter der Kennung des Eigentümers ausgeführt, wird das SGID-Bit gesetzt, so wird die Datei so ausgeführt als würde der aktuelle Benutzer der zugehörigen Gruppe angehören.
- t: Sticky-Bit: Spezielles Recht, das definiert, dass ein Benutzer in einem Verzeichnis nur die Dateien löschen darf, deren Eigentümer er ist, auch wenn er die nötigen Rechte besitzen würde, alle anderen Dateien zu löschen. Das Sticky-Bit-Recht kann nur ohne Angabe eines Bereichs gesetzt werden!

Beispiele

chmod +x *datei*	Ausführungsrecht auf *datei* für Benutzer, Gruppe und alle anderen Benutzer setzen.
chmod go-w *datei*	Entzieht das Schreibrecht auf *datei* für Gruppe und alle anderen Benutzer.
chmod g+rwx *datei*	Gewährt der Gruppe alle Rechte auf *datei*.
chmod ugo-rwx *datei*	Entzieht allen Benutzern alle Rechte auf *datei*.
chmod u=rwx *datei*	Gewährt dem Eigentümer von *datei* alle Rechte.
chmod +t *verzeichnis*	Setzt das Sticky-Bit auf *verzeichnis*.
chmod u+s *admintool*	Setzt das SUID-Bit für die Datei *admintool*. Wenn irgendein Benutzer diese Datei aufruft, läuft der Prozess unter der Kennung des Eigentümers.
chmod g+s *admintool*	Wenn ein irgendein Benutzer die Datei *admintool* als Prozess startet, läuft der Prozess so, als würde der Benutzer Mitglied der zugeordneten Gruppe sein.

Direktmodus$_1$

- Unterschiedliche Berechtigungen mit einem Kommando setzen
- `chmod` *`modus dateiname`*
  - *Modus*: Kombination von Oktalzahlen
  - Jede Stelle steht für Bereich (1. User, 2. Group, 3. Others)
  - Optional 4. Wert (1. Stelle): Spezielle Zugriffsrechte
    - UID/GID: Beim Ausführen läuft Prozess unter der Kennung des Eigentümers (UID) oder der zugeordneten Gruppe (GID), nicht unter eigener Kennung!
    - Sticky-Bit: Falls gesetzt, darf nur der Eigentümer einer Datei die Datei in einem Verzeichnis löschen, auch wenn andere Benutzer entsprechende Rechte hätten.

Rechtevergabe im Direktmodus

Das chmod-Kommando erlaubt neben dem symbolischen Modus auch den Direktmodus, bei dem unterschiedliche Rechte für unterschiedliche Benutzerkategorien gleichzeitig gesetzt werden können. Das Kommando erwartet hier einen *Modus*, der eine Kombination von Oktalzahlen darstellt und eine Dateiliste (Dateien für die Berechtigung gesetzt werden soll).

Der Modus selbst ist eine Dreierkombination von Oktalzahlen. Jede Stelle steht für einen Bereich, die erste für den Eigentümer, die zweite für die zugehörige Gruppe und die dritte für alle anderen Benutzer. Optional kann noch ein vierter Wert angegeben werden wo die speziellen Zugriffsrechte wie das SID-Bit und das Sticky-Bit gesetzt werden können.

Beide Modi erlauben die Berechtigungsvergabe gleichermaßen, es ist somit der Vorliebe des Benutzers zu überlassen für welchen Modus er sich entscheidet. Jedoch wird das Konzept des Direktmodus auch bei der Vergabe der Standardberechtigung verwendet, weshalb die Kenntnis des Direktmodus unerlässlich ist.

Direktmodus$_2$

Recht	*Oktalzahl*
Read	4
Write	2
Execute	1
SUID	4
SGID	2
STICKY	1

Pro Bereich wird eine Oktalzahl angegeben.

Zugeteiltes Recht ergibt sich aus der Summe der Einzelrechte:
rwx = read + write + execute →
4+2+1 = 7

777 → rwx rwx rwx (ugo)
754 → rwx r-x r-- (ugo)
4711 → rws --x --x (SID,ugo)
2712 → rwx --s -w- (GID,ugo)
1712 → rwx --x -wt (Sticky, ugo)

Zahlenwerte im Direktmodus

Wie schon zuvor erwähnt wird jeder Kategorie von Benutzern eine Oktalzahl zugewiesen. Da es drei Kategorien oder Bereiche gibt (user, group, others), ist die Oktalzahl standardmäßig dreistellig. Die erste Stelle ist für den Eigentümer reserviert, die zweite Stelle für die zugehörige Gruppe und die dritte Stelle für alle anderen Benutzer. Für Spezialrechte, wie das SID-Bit oder das Sticky-Bit, ist optional eine weitere Stelle vorgesehen, die als *erstes* (also vor den Benutzerrechten) übergeben werden muss.

Jedes Recht hat einen bestimmten Wert (read=4, write=2, execute=1). Die Addition der Rechte ergibt den Wert, der jeder Stelle übergeben werden muss. Alle Rechte würden sich aus der Summe von 4 (read) + 2 (write) + 1 (execute) = *7* ergeben. Auch bei den Spezialrechten wird ähnlich vorgegangen. Sollen z.B. alle Spezialrechte vergeben werden, so ist in der ersten Stelle (die Benutzerrechte sind dann in den Stellen zwei bis vier angegeben) die Summe von 4 (SUID) + 2 (SGID) + 1 (Sticky-Bit) = *7* anzugeben.

Beispiele

chmod 777 *datei*	Alle Benutzer eines Systems erhalten alle Rechte auf *datei*.
chmod 754 *datei*	Der Eigentümer von *datei* erhält alle Rechte, Mitglieder der zugehörigen Gruppe erhalten Lese- und Ausführrechte, alle anderen Benutzer erhalten nur das Leserecht.

chmod 4711 *datei*	Auf *datei* wird das SUID-Bit gesetzt, sodass jeder Benutzer *datei* unter der Kennung des Eigentümers ausführt. Nur der Eigentümer selbst darf jedoch die Datei lesen oder schreiben, ausgeführt darf sie von jedem Benutzer im System werden.
chmod 2712 *datei*	Auf *datei* wird das SGID-Bit gesetzt, sodass für jeden Benutzer die Rechte gelten, als würde er der zugehörigen Gruppe angehören. Dabei hat der Eigentümer alle Rechte auf die Datei, die Gruppe darf die Datei ausführen und alle anderen dürfen in die Datei schreiben.
chmod 1712 *verzeichnis*	Setzt das Sticky-Bit auf *verzeichnis*. Dabei darf der Besitzer im Verzeichnis alles machen, die zugehörige Gruppe darf in das Verzeichnis wechseln (nicht aber den Inhalt anzeigen lassen) und alle anderen dürften neue Einträge im Verzeichnis erstellen. Da allen anderen jedoch das execute-Recht fehlt, können sie dieses Recht nicht in Anspruch nehmen.

Gerade das letzte Beispiel soll vor Augen führen, dass auch „sinnlose“ Rechte vergeben werden können. Besitzt ein Benutzer z.B. Schreibrechte auf ein Verzeichnis aber keine Ausführrechte, so kann er das Schreibrecht nicht in Anspruch nehmen. Würde dem Benutzer hingegen zusätzlich noch das Ausführrecht auf das Verzeichnis gegeben werden, so könnte er vom Schreibrecht Gebrauch machen.

Rechtevergabe steuern

Steuerung der Berechtigungen

- Eigentümer kann weitergegeben werden
  - Benutzer wird neuer Eigentümer, alter Eigentümer verliert Rechte an Datei
  - `chown` *`username dateiliste`*
  - Befehl kann nur vom Superuser ausgeführt werden
- Gruppenzugehörigkeit einer Datei ändern
  - `chgrp` *`gruppenname dateiliste`*
  - Befehl kann nur vom Superuser oder vom Eigentümer ausgeführt werden, sofern dieser auch Mitglied der neuen Gruppe ist.

Weitergabe der Eigentümereigenschaft

Normalerweise kann nur der Eigentümer Rechte mit Hilfe des chmod-Kommandos vergeben. Es ist aber auch möglich, dass der Eigentümer einer Datei verändert werden kann. Diese Weitergabe der Eigentümereigenschaft kann dabei nur vom Superuser erfolgen.

Mit Hilfe des Kommandos ‚chown' wird ein neuer Eigentümer bestimmt. Die Rechte des alten Eigentümers werden dabei an den neuen Eigentümer übertragen wobei der alte Eigentümer seine Rechte an der Datei verliert. Das Kommando erwartet den Benutzernamen des neuen Eigentümers und eine Liste von Dateien, deren Eigentümer gewechselt werden soll.



Das Kommando „chown newuser *" würde z.B. alle Dateien eines Verzeichnisses dem Benutzer *newuser* zusprechen. Dieser ist nach erfolgtem Ausführen des Kommandos neuer Eigentümer und hat vollständige Verfügungsgewalt über die Dateien.

!

Im obigen Beispiel würden nur alle sichtbaren Dateien dem Benutzer zugesprochen werden. Versteckte Dateien (also Dateien die mit einem „.' beginnen) sind vom Eigentümertransfer nicht betroffen.

Gruppenzugehörigkeit ändern

Wie schon zuvor erwähnt, ist jede Datei einer Gruppe zugeordnet (zugehörige Gruppe). Diese Gruppenzugehörigkeit kann natürlich geändert werden. Die Änderung kann nur vom Superuser erfolgen oder vom Eigentümer, sofern dieser auch Mitglied der neuen Gruppe ist. Ein Eigentümer kann seine Datei keiner

Gruppe zusprechen, deren Mitgliedschaft er nicht besitzt. Das Kommando zum Ändern der Gruppenzugehörigkeit ist ‚chgrp'. Dem Kommando folgt der Name der neuen Gruppe und eine Liste von Dateien, deren Gruppenzugehörigkeit geändert werden soll.



Das Kommando „chgrp newgroup *" würde z.B. alle Dateien eines Verzeichnisses der Gruppe *newgroup* zusprechen. Mitglieder dieser Gruppe haben nach erfolgtem Ausführen des Kommandos alle Rechte, die auf Gruppenebene der Datei definiert sind.

Standardberechtigungen festlegen

Berechtigungen für neue Dateien

- Berechtigungen für neue Dateien können festgelegt werden
- Auf bestehende Dateien hat Kommando keinen Einfluss
- Kommando: `umask` *`Berechtigungsmaske`*
  - Berechtigungsmaske ist Komplement zu Zugriffsrechte (definiert, was nicht gesetzt werden darf
    → 777 - Berechtigungsmaske = Berechtigung)
    umask 022: 777-022=755 → rwx r-x r-x
    umask 027: 777-027=750 → rwx r-x ---
    umask 177: 777-177=600 → rw- --- ---
- umask ohne Berechtigungsmaske liefert aktuelle Einstellungen

Oftmals ist es sehr angenehm, wenn man eine Standardeinstellung der Rechte für neue Dateien einstellen kann. Berechtigungen auf neue Dateien müssen dann nicht jedes Mal durch das chmod-Kommando explizit eingestellt werden. Zur Einstellung von Standardberechtigungen dient das Kommando ‚umask'. Dieses Kommando hat allerdings keinen Einfluss auf schon existierende Dateien. ‚umask' erwartet einen dreistelligen Zahlenwert für Eigentümer, Gruppe und alle anderen Benutzer. Es wird genau das Komplement der Oktalzahlenangabe des chmod-Kommandos erwartet, d.h. hier wird festgelegt, welches Recht *nicht* gesetzt werden soll. Das vergebene Zugriffsrecht ergibt sich somit aus allen Rechten minus der erwarteten Berechtigungsmaske.

Beispiele

umask 022	777 - 022 =755	Von allen möglichen Rechten werden dem Eigentümer keine Rechte, der Gruppe und den restlichen Benutzer das Schreibrecht weggenommen. Auf neue Dateien hat der Eigentümer alle Rechte, die Gruppe und restlichen Benutzer dürfen die Datei lesen und ausführen.
umask 027	777 - 027 =750	Von allen möglichen Rechten werden dem Eigentümer keine Rechte, der Gruppe das Schreibrecht und den restlichen Benutzern alle Rechte weggenommen. Auf neue Dateien hat der Eigentümer alle Rechte, die Gruppe darf sie lesen und ausführen und alle restlichen Benutzer haben keine Rechte darauf.

umask 177	777 - 177 =600	Von allen möglichen Rechten wird dem Eigentümer das Ausführrecht, der Gruppe und den anderen Benutzern alle Rechte weggenommen. Alle neuen Dateien können vom Eigentümer gelesen oder geschrieben werden, alle anderen haben keine Rechte darauf.

Kapitel

6

Die Shell

Dieses Kapitel befasst sich detailliert mit der UNIX-Shell. Die Shell ist die Schnittstelle zwischen dem System und dem Benutzer. Darüber hinaus ist dieses Kapitel die Grundlage für die danach behandelte Shellprogrammierung.

Die Shell ist der Kommandozeileninterpreter von UNIX. Sie ist aber gleichzeitig auch die wichtigste Schnittstelle zwischen System und Benutzer. Eingegebene Kommandos werden von der Shell interpretiert und entsprechend verarbeitet. Innerhalb einer Shell können auch eigene Shellprogramme (Shellskripte) erstellt werden. Diese dienen zum Automatisieren von Vorgängen, die sonst manuell durchgeführt werden müssen. Da die Shell nicht direkt in das Betriebssystem integriert ist, kann der Benutzer zwischen verschiedenen Shells auswählen.

Lehrinhalte und -ziele

Nach diesem Kapitel sollten Sie in der Lage sein, die Konzepte der Shell zu verstehen und die Shell richtig zu bedienen. Sie sollten wissen welche Aufgaben die Shell zu bewerkstelligen hat und wie sie dies macht. Sie sollten das Konzept der Ein-/Ausgabeumleitung verstehen und Pipes verwenden können. Darüber hinaus sollten Sie die Bedeutung der einzelnen Metazeichen kennen und wissen wie diese von der Shell ausgewertet werden bzw. wie diese Auswertung beeinflusst werden kann. Letztendlich sollten Sie mit Shellvariablen und internen Kommandos umgehen können und deren Konzept verstehen.

Aufgaben der Shell

Was kann die Shell

- Shell ist Kommandointerpreter
  - z.B. sh, csh, ksh, bash
- Kommando vom Terminal entgegennehmen
- Programm suchen (PATH) und starten
- Steuerung der Ein- und Ausgabe am Terminal
- Ersetzen von Sonderzeichen und Befehlssubstitution
- Ablaufsteuerung (bedingte Anweisungen, Schleifen, etc. → Shell-Programmierung)
- Shell nicht direkt in das BS integriert → daher austauschbar

Die Shell ist der Kommandointerpreter von UNIX. Da die Shell selbst nicht direkt in das Betriebssystem integriert ist, kann sie durch andere Shells ausgetauscht werden. In der Praxis haben sich vier Shells durchgesetzt. Diese sind die Bourne Shell (sh), die C-Shell (csh), die Korn Shell (ksh) und die Borne Again Shell (bash), die eine Erweiterung der Bourne Shell darstellt.

Die Shell selbst nimmt eine Kommandozeile vom Terminal entgehen, entschlüsselt das Kommando und teilt es in seine Komponenten (Kommandoname, Optionen, Parameter) auf. Dann sorgt sie dafür, dass das gewünschte Programm gestartet wird. Da Programme, vor allem aber Kommandos, in unterschiedlichen Verzeichnissen im Dateisystem abgelegt sind, muss die Shell in der Lage sein, das gewünschte Programm zu lokalisieren. Dies funktioniert entweder dadurch, dass der Benutzer den Pfad dem Kommando voranstellt (entweder absolut oder relativ) oder man nützt die Suchfunktion der Shell. Dabei durchsucht die Shell diejenigen Verzeichnisse, die im Suchweg spezifiziert sind. Der Suchweg wird durch die Shellvariable ‚PATH' vorgegeben. Shellvariablen bieten die Möglichkeit, das Verhalten der Shell zu beeinflussen. Dies wird später in diesem Kapitel noch ausführlicher behandelt.

Die Shell ist darüber hinaus für die Steuerung der Ein- und Ausgabe des Terminals verantwortlich. UNIX ist als Dialogsystem verfasst und war ursprünglich für den Betrieb mit Fernschreibern und Datensichtgerät konzipiert. Diese sogenannten Terminals findet man heute noch in den *logischen* Terminals. Beispiele für solche Terminals sind die Konsolen, die dem Bildschirm und der Tastatur des PCs zugeordnet sind, den Pseudo-Terminals, die einer Telnet-Verbindung zugeordnet werden oder den X-Terminal-Fenstern der graphischen Benutzeroberfläche. Dabei wird die Eingabe von der Standardeingabe (Tastatur), auch als *stdin* bezeichnet,

entgegengenommen während die Ausgabe an die Standardausgabe (Bildschirm), auch als *stdout* bezeichnet, und die Fehlerausgabe an die Standardfehlerausgabe (Bildschirm), als *stderr* bezeichnet, gesendet. Die Shell kann aber angewiesen werden, bei Bedarf die Ein- und Ausgaben an andere Geräte oder Dateien umzuleiten.

Eine weitere Aufgabe der Shell ist die Ersetzung von Sonderzeichen und die Befehlssubstitution. Kommandoeingaben können durch Sonderzeichen erweitert und der Kommandostring kann selbst durch die Shell expandiert werden. Diese Mechanismen erlauben die Verwendung von Metazeichen (Jokerzeichen) oder die Verwendung von Sonderzeichen (z.B. ‚~' als Referenz zum Home-Verzeichnis).

Die Shell selbst bietet eine Möglichkeit, Vorgehensweisen und Tätigkeiten zu automatisieren (durch sogenannte Shellskripte). Dafür bietet die Shell viele Strukturen, wie man sie von Programmiersprachen her kennt wie z.B. Tests, bedingte Anweisungen, Schleifen, Variablen, Rechenanweisungen. Diese Möglichkeiten werden im Kapitel 7 „Shellprogrammierung" noch genauer behandelt.

Die Shell bietet ferner noch Möglichkeiten zum Erzeugen von Kind- und Hintergrundprozessen. Dies ist einerseits für Shellskripte relevant, andererseits erlaubt es aber auch das Starten von Serverprozessen oder Prozessen, die im Hintergrund ablaufen und nicht mit dem Benutzer interagieren (z.B. Batch-Prozesse oder komplexe Rechenoperationen). Dies wird durch das Anhängen des Zeichens ‚&' nach dem Kommando erreicht (z.B. „ls –la &").

Was macht die Shell

- Shell ist ein Programm das von *stdin* liest und auf *stdout* schreibt
- Shell meldet sich mit Prompt ($ bzw. #) in PS1
- Kommando wird nach Return ausgeführt
  - Shell trennt die Argumente eines Kommandos (IFS)
  - Kommandotrenner (;) oder Trennzeichen (&,||,<,>)
  - Variablen werden durch ihre Werte ersetzt
  - Kommandosubstitution wird durchgeführt (`...` bzw. $(...))
  - *stdin*, *stdout*, *stderr* wird entsprechend umgeleitet
  - Variablenzuweisungen werden durchgeführt
  - Jokerzeichen (Metazeichen) werden ersetzt
  - Die Shell führt das Kommando aus

Prinzipiell kann die Shell als ein Programm beschrieben werden, dass Anweisungen von der Standardeingabe (stdin) liest, für deren Verarbeitung sorgt und das Ergebnis oder Meldungen die im Verarbeitungsprozess anfallen an die Standardausgabe (stdout) weitergibt. Wird das Dateiende erreicht (End Of File, EOF), z.B. durch Eingabe der Tastenkombination <CTRL>+<D>, terminiert die Shell. Das Beenden der letzten Login-Shell kommt mit einem Abmelden des Benutzers vom System gleich.

Sobald die Shell bereit ist, Kommandoeingaben entgegenzunehmen, meldet sie sich mit einem Bereitschaftszeichen, dem sogenannten Prompt. Das angezeigte Zeichen ist dabei abhängig von der erwarteten Eingabe und vom angemeldeten Benutzer. Im einfachsten Fall ist dies ein ‚$'-Zeichen, dass eine Eingabe von einem normalen Benutzer erwartet. Wird eine Eingabe vom Superuser erwartet, wird das ‚#'-Zeichen als Prompt angezeigt. Erwartet ein Kommando oder Programm noch weitere Eingaben (z.B. weitere Parameter), dann erscheint ein ‚>' als Prompt (sowohl bei normalen Benutzern als auch beim Superuser). Da UNIX ein offenes System ist und alles beliebig modifiziert werden kann, ist es natürlich auch möglich das Erscheinungsbild des Prompts zu ändern.

ⓘ Vor allem auf Linux-Systemen findet man häufig die Einstellung, dass zusätzlich zum Bereitschaftszeichen noch der Name des Benutzers (der Login-Name), der Name des Rechners (Host-Name) und das aktuelle Verzeichnis angegeben wird. Jedoch sollte das Bereitschaftszeichen selbst immer als letztes Zeichen des Prompts angezeigt werden!

Bearbeitung der Kommandozeile durch die Shell

Ein Kommando wird erst ausgeführt, wenn der Benutzer am Ende der Kommandozeile die ENTER-Taste drückt. Die Shell liest bis zum ersten Kommandotrenner (‚;') oder zu einem der folgenden Trennzeichen (‚&', ‚||', ‚>', ‚<') und stellt fest, ob Variablenzuweisungen erfolgen sollen oder die Ein- bzw. Ausgabe umgelenkt werden muss.

Dazu zerlegt die Shell die Kommandozeile in einzelne Argumente. Sie trennt die einzelnen Argumente durch eines der Zeichen, die in der Shellvariablen IFS (Internal Field Separator) stehen, normalerweise sind dies Leerzeichen, Tabs und Newline-Zeichen. Dann werden Variablenreferenzen, die in der Kommandozeile stehen, durch ihre Werte ersetzt. Da manchmal eine solche Ersetzung unerwünscht ist, werden Kommandos, die in einfachen Hochkomma ('...') stehen oder bei der bash in „$ (...)" angegeben werden, ausgeführt ohne das eine Ersetzung stattfinden würde. Falls eine Ein-/Ausgabeumleitung spezifiziert wurde, wird die Standardeingabe (stdin), die Standardausgabe (stdout) oder die Standardfehlerausgabe (stderr) auf die angegebenen Zieldateien umgelenkt. Falls in der Kommandozeile noch Zuweisungen an Variablen stehen, werden diese vor dem Start des Kommandos noch ausgeführt. Bevor das eigentliche Kommando ausgeführt wird, sucht die Shell nach Jokerzeichen und ersetzt diese durch passende Dateinamen. Jedem Kommando wird so eine mehr oder weniger lange Liste von passenden Dateinamen mitgegeben. Letztendlich führt die Shell das zuvor bearbeitete und aufbereitete Kommando aus.

Ein-/Ausgabeumleitung

Ein-/Ausgabeumleitung[1]

- Jedem Terminal sind 3 Dateikanäle zugeordnet
  - Standardeingabe (*stdin*)
  - Standardausgabe (*stdout*)
  - Standard-Fehlerausgabe (*stderr*)
- Eingabeumleitung
  - Programm liest aus Datei, nicht von Tastatur
  - Umleitung erfolgt durch ‚<'
- Ausgabeumleitung
  - Ergebnis wird in Datei geschrieben, nicht auf Terminal
  - Umleitung erfolgt durch ‚>' bzw. ‚>>'

Jedem Terminal sind drei Dateikanäle zugeordnet. Diese sind:

1.) Standardeingabe: Die Standardeingabe (meist als *stdin* bezeichnet) liefert einen Zeichenstrom an die Shell. Sofern nicht anders definiert ist die Standardeingabe die angeschlossene Tastatur. Der Standardeingabe ist der Handle 0 zugeordnet.

2.) Standardausgabe: Die Standardausgabe (meist als *stdout* bezeichnet) zeigt Ergebnisse und Rückmeldungen der Shell am Bildschirm an. Der Standardausgabe ist der Handle 1 zugeordnet.

3.) Standard-Fehlerausgabe: Die Standard-Fehlerausgabe (meist als *stderr* bezeichnet) funktioniert ähnlich wie stdout, nur werden hier Fehlermeldungen der gestarteten Prozesse ausgegeben. Der Standard-Fehlerausgabe ist der Handle 2 zugeordnet.

Alle diese Ein- und Ausgabekanäle können umgeleitet werden. Bei der Eingabeumleitung werden Zeichen anstatt von der Tastatur von der angegebenen Datei gelesen, bei der Ausgabeumleitung werden Meldungen in die angegebene Datei geschrieben anstatt sie am Bildschirm anzuzeigen.

In weiterer Folge wird die Ein- und Ausgabeumleitung genauer behandelt.

Eingabeumleitung

Bei der Eingabeumleitung liest der Prozess seine Daten nicht mehr von der Tastatur sondern aus einer Datei. Dies geschieht solange, bis das Dateiende erreicht ist. Das Dateiende wird durch ein End Of File (EOF) gekennzeichnet.

Will man bezwecken, dass ein Kommando anstatt von der Tastatur seine Eingaben aus einer Datei bezieht, so ist das Kommando folgendermaßen anzugeben:

```
Kommando < Datei
```

Die Eingabeumleitung selbst erfolgt durch das Zeichen ‚<', gefolgt von einem Dateinamen. Da in UNIX jedes Gerät dateibasiert abgespeichert wird, kann der Benutzer von jedem beliebigen zeichenorientierten Gerät (z.B. ein anderes Terminal) die Eingabedaten an ein Kommando weiterleiten.

Ausgabeumleitung

Bei der Ausgabeumleitung wird die Ausgabe eines Programms nicht auf dem Bildschirm ausgegeben, sondern in eine Datei geschrieben. Dies geschieht solange, bis der Prozess mit der Ausgabeumleitung beendet wird. Existiert die für die Umleitung angegebene Datei bereits, wird diese überschrieben. Will man bezwecken, dass ein Kommando seine Ausgabe in eine Datei schreibt anstatt auf dem Bildschirm, so ist das Kommando folgendermaßen anzugeben:

```
Kommando > Dateiname
```

Wie schon erwähnt, wird eine eventuell existierende Datei bei der Ausgabeumleitung überschrieben. Wenn man jedoch die Ausgabe an eine bestehende Datei anhängen will, so ist das ‚>'-Zeichen doppelt anzugeben.

```
Kommando >> Dateiname
```

Die Ausgabe wird dann an die bestehende Datei angehängt, sofern diese existiert. Ist dies nicht der Fall, wird – ähnlich wie bei der normalen Umleitung – eine neue Datei angelegt und in diese geschrieben.

Die Ausgabeumleitung sorgt nur dafür, dass reguläre Ausgaben anstatt am Bildschirm in die Datei geschrieben werden. Fehlermeldungen erscheinen nach wie vor auf dem Bildschirm. Will man diese auch umlenken, so muss man Gebrauch von der Fehlerausgabeumleitung machen.

Ein-/Ausgabeumleitung$_2$

- Umlenken der Fehlerausgabe
  - Fehlermeldungen werden in Datei geschrieben
  - Umleitung erfolgt durch ‚2>'
- Kombinationen
  - *Kommando < Eingabedat > Ausgabedat 2> Fehlerdat*
- Spezialziel/Quelle
  - /dev/null
  - /dev/zero

Fehlerausgabeumleitung

Die Fehlerausgabeumleitung funktioniert analog zur Ausgabeumleitung. Hier werden jedoch Fehlermeldungen umgeleitet, die von der normalen Ausgabeumleitung nicht betroffen sind. Dazu wird anstatt des Zeichens ‚>' die Zeichenfolge ‚2>' angegeben, da stderr die Handlernummer 2 hat. Will man nun bezwecken, dass ein Kommando seine Fehlermeldungen in eine Datei schreibt anstatt auf dem Bildschirm, so ist das Kommando folgendermaßen anzugeben:

```
Kommando 2> Dateiname
```

Kombination von Umleitungen

Natürlich sind alle Umleitungen beliebig kombinierbar. Das Kommando

```
Kommando < Eingabedat > Ausgabedat 2> Fehlerdat
```

würde z.B. seine Eingaben aus der Datei *Eingabedat* beziehen, seine Ausgaben in *Ausgabedat* schreiben und die Fehler in die Datei *Fehlerdat*.

Spezielle Umleitungen

Es gibt in einem UNIX-System zwei Spezialdateien, die speziell für die Umleitung von Ein- und Ausgaben interessant sind. Diese Dateien werden bei einer Ausgabeumleitung *NICHT* überschrieben.

- Datei „/dev/null": Ausgaben, die an diese Datei geschickt werden, verschwinden in einem schwarzen Loch. Sie können an diese Datei irgendwelche Daten ohne eine Auswirkung schicken. Diese

Ausgabeumleitung wird dazu verwendet, um Ausgaben zu unterdrücken. Wird diese Datei zur Eingabe verwendet (Eingabeumleitung), dann liefert die Datei ein End-of-File (EOF) Zeichen.

- Datei „/dev/zero“: Die Ausgabeumleitung an diese Datei bewirkt nichts, ähnlich der Datei „/dev/null“. Wird jedoch die Datei zur Eingabe verwendet, liefert diese Datei unendlich viele Nullbytes. Dieses Verhalten kann dafür verwendet werden, um Programme oder Shellskripte zu testen.

Pipes

Pipes

- Pipe verbindet Ausgabe eines Programms und Eingabe eines zweiten Programms über einen temporären Puffer
- Mehrere Kommandos können hintereinander geschaltet werden = Filter
- `Kommando1 | Kommando2 | Kommando3`
- Beispiele
  - `ls –la | more`
  - `head [-n] [files]` (Filter)
  - `tail [-n] [files]` (Filter)
  - `wc [-lwc] [files]` (Filter)

Eine Pipe verbindet zwei Kommandos über einen temporären Puffer, d.h. die Ausgabe vom ersten Programm wird als Eingabe vom zweiten Programm verwendet. Man kann sich eine Pipe auch so vorstellen, dass die Ausgabe des ersten Programms in eine temporäre Datei umgeleitet wird und diese Datei und die darin enthaltenen Daten durch Umleitung als Eingabe für das zweite Programm hergenommen werden. Dabei wird die Reihenfolge genau eingehalten. Was als erstes in den Puffer geschrieben wird, wird als erstes vom zweiten Prozess wieder gelesen. Die Pufferung und Synchronisation wird dabei vom Betriebssystem vorgenommen.

Eine Pipe wird durch das Zeichen ‚|' definiert, das zwischen zwei Kommandos angegeben wird.

```
Kommando1 | Kommando2
```

Dabei wird zuerst das erste Kommando (Kommando1) ausgeführt, die Ausgabe zwischengepuffert und dann als Eingabe des zweiten Kommandos (Kommando2) hergenommen.

Es können natürlich auch mehrere Kommandos hintereinander durch Pipes verbunden werden. Dabei wird die Ausgabe vom aktuellen Prozess gepuffert und als Eingabe des nächsten Prozesses hergenommen, bis alle Prozesse abgearbeitet wurden.

```
Kommando1 | Kommando2 | Kommando3 | Kommando4
```

Solche Kommandofolgen werden auch als Filter bezeichnet, weil in UNIX-Systemen einige Kommandos zum Filtern von Ausgaben implementiert sind, die

meistens in solchen langen Kommandofolgen eingesetzt werden um die Ausgabe entsprechend zu kürzen. Beispiele für solche Filter sind:

head [-n] [datei(en)]: Dieses Kommando liefert die ersten *n* Zeilen aus den angegebenen Dateien. Wird keine Datei angegeben, liest das Kommando von der Standardeingabe. Anstatt der Standardeingabe kann natürlich auch eine Eingabeumleitung oder eine vorgeschaltete Pipe die Daten liefern.

tail [-/+n] [datei(en)]: Dieses Kommando liefert die letzten *n* Zeilen aus den angegebenen Dateien. Wird keine Datei angegeben, liest das Kommando von der Standardeingabe. Anstatt der Standardeingabe kann natürlich auch eine Eingabeumleitung oder eine vorgeschaltete Pipe die Daten liefern.

tee [-i] [-a] [datei]: Dieses Kommando kopiert Zeichen von der Standardeingabe in die Standardausgabe und schreibt die Daten gleichzeitig in die angegebene Datei. Wird keine Datei angegeben, so wird die Ausgabe nochmals auf die Standardausgabe kopiert. Über Ausgabeumleitung kann mittels diesem Kommandos die Ausgabe eines Kommandos verdoppelt werden (1x Pipe, 1x Ausgabeumleitung in Datei).

Beispiel: `ls –la | tee out.txt >out2.txt`

wc [-lwc] [datei(en)]: Dieses Kommando zählt – je nach angegebener Option – Zeilen, Worte oder Zeichen in einer Datei. Wird kein Dateiname angegeben, liest das Kommando von der Standardeingabe. Die Daten können natürlich auch über Eingabeumleitung oder eine vorgeschaltete Pipe bezogen werden. Normalerweise zählt man damit in Skripten Ergebnisse.

more: Dieses Kommando gibt Inhalte oder Textzeilen seitenweise aus. Das Kommando eignet sich sehr gut dazu, Verzeichnisinhalte seitenweise aufzulisten. Das Kommado „ls –la | more“ würde z.B. die Inhalte des Verzeichnisses in eine Pipe schreiben, die dann vom more-Kommando gelesen wird. Das more-Kommando liest solange, bis die Bildschirmseite gefüllt ist, wartet dann auf eine Eingabe und setzt die Ausgabe dann fort.

Es gibt auch noch andere Filter (z.B. less, tr) die noch im Kapitel 7 Shellprogrammierung genauer behandelt werden.

Metazeichen

Metazeichen

- Helfen mehrere Dateien an ein Kommando zu übergeben (=Joker, Wildcards), Expandieren Dateinamen
- Werden in der Shell ersetzt (nicht vom Kommando)
- Können beliebig oft an beliebiger Stelle in Dateinamen stehen
  - * beliebige Zeichenfolge (auch leere)
  - ? beliebiges Zeichen
  - [...] Auswahl von Zeichen (z.B. [abcd] = [a-d])
  - [!...] Auswahl von Zeichen die nicht vorkommen
  - \ Metazeichen werden nicht ersetzt
- Bei Ein-/Ausgabeumleitung werden Metazeichen generell nicht ersetzt

Metazeichen haben die Aufgabe, Dateinamen zu expandieren. Dabei werden sie von der Shell verwendet, um die Eingabe zu erleichtern und den Tippaufwand zu reduzieren. Mit Hilfe der Metazeichen ist es möglich, nicht alle Dateinamen eintippen zu müssen sondern die Dateien nach bestimmten Kriterien vorher auszuwählen. Diese Metazeichen sind auch als Jokerzeichen oder Wildcards bekannt.

Wichtig zu wissen und zu verstehen ist die Tatsache, dass die Metazeichen vor dem eigentlichen Kommandoaufruf selbst von der Shell durch passende Dateinamen ersetzt werden. Bei MS-DOS wird dem Kommando auch das Metazeichen übergeben was bedeutet, dass dieses sich um die Ersetzung kümmern muss. Ein UNIX-Kommando sieht aber nur eine Dateiliste mit variabler Länge. Dies ist insofern wichtig zu verstehen, da bei der Shellprogrammierung darauf acht gegeben werden muss, dass die Kommandos Rücksicht auf Dateiersetzung nehmen müssen und beliebig viele Dateinamen akzeptieren sollten. Im Programm sind aber auch keine Systemaufrufe nötig, die auf die Verzeichnisinformation zurückgreifen, es wird lediglich eine Schleife benötigt, welche die einzelnen Dateien nacheinander bearbeitet.

Metazeichen sind Zeichen mit erweiterter Bedeutung. Die Shell ersetzt die Metazeichen durch alle Dateinamen des aktuellen Verzeichnisses, die auf das Muster passen. Dabei können die Metazeichen beliebig oft an beliebiger Stelle im Dateinamen stehen (z.B. *abc*def*).

Es gibt unterschiedliche Metazeichen die jeweils eine eigene Bedeutung haben. Diese lassen sich natürlich je nach Anforderung auch beliebig kombinieren.

‚*' steht für jede beliebige Zeichenfolge, ausgenommen eines Punktes am Beginn des Dateinamens.

‚?' steht für ein beliebiges Einzelzeichen, ausgenommen eines Punktes am Beginn des Dateinamens.

‚['...,]' Die eckige Klammer wird bei der Dateinamenverarbeitung durch ein in der Klammer angegebenes Zeichen ersetzt. Dabei wird jeweils bei der Ersetzung nur eins der angegebenen Zeichen hergenommen. Der Bereich kann entweder durch die Angabe jedes möglichen Zeichens (z.B. [abcd]) oder durch ein bis-Zeichen ‚-' (z.B. [a-d]) angegeben werden.

Wird als erstes Zeichen ein ‚!' gefunden, so gibt man innerhalb der Klammer die Zeichen an, die nicht vorkommen dürfen (Negation!). Es gelten somit alle Zeichen die nicht in der Klammer angegeben wurden.

Beispiele:

[abc] Zeichen wird ersetzt durch a oder b oder c.

[1-3] Zeichen wird ersetzt durch 1 oder 2 oder 3.

[!0-9] Es sind alle Zeichen gültig, sofern es sich nicht um Ziffern handelt.

‚\' Hebt den Ersetzungsmechanismus für das folgende Zeichen auf. Somit kann ein Zeichen, dass normalerweise ein Metazeichen wäre aber trotzdem Teil eines Dateinamens ist, direkt angesprochen werden.

Beispiel:

\? Das Fragezeichen wird nicht durch ein beliebiges Zeichen ersetzt, sondern direkt im Dateinamen übernommen (z.B. dateinamemit?.txt).

! Bei der Umleitung von Ein- und Ausgaben werden Metazeichen in den Dateinamen hinter dem Umleitungszeichen nicht ersetzt. Zum Beispiel würde das Kommando „ls –la > ausgabe?" eine Datei ‚ausgabe?' erzeugen, die dann mit „rm ausgabe\?" gelöscht werden müsste.

Wird ein ‚\' am Ende der Zeile angegeben, unterdrückt dies sogar das Return-Zeichen. Das Kommando kann in der folgenden Zeile fortgesetzt werden. Dabei erscheint dann der ‚>'-Prompt.

Quoting

Quoting

- Um Sonderzeichen als Argumente zu übergeben, werden Anführungszeichen verwendet – keine Ersetzung
  - "..." Keine Ersetzung der Metazeichen (*, ?,[]), jedoch Shell-Variablen werden ersetzt
  - '... ' unterdrückt jede Substitution
  - `...` Kommandos zwischen Back-Quotes (Accent Grave) werden ausgeführt und Ergebnis als Parameter übergeben (in bash auch $(...))

Da die Shell Metazeichen ersetzt, muss es eine Möglichkeit geben, Sonderzeichen als Argumente zu übergeben ohne das diese ersetzt werden. Dies ist mit dem sogenannten Quoting möglich. Beim Quoting werden Anführungszeichen verwendet, die auch ineinander geschachtelt werden können. Dabei haben die drei verschiedenen Kategorien von Anführungszeichen, nämlich doppelte Anführungszeichen (“), einfache Anführungszeichen (’) und Backquotes (`), unterschiedliche Bedeutung.

Beispiele

"..." Innerhalb der doppelten Anführungszeichen erfolgt keine Ersetzung der Metazeichen * ? [], jedoch werden Shellvariablen und Ergebnisse von Kommandos (Backquotes) ersetzt.

```
echo Der * wird hier durch alle Dateinamen ersetzt

echo "Der * wird hier nicht ersetzt"
```

'...' Das einfache Anführungszeichen unterdrückt jede Ersetzung. Alles was sich innerhalb dieser Anführungszeichen befindet, wird unverändert weitergegeben.

```
echo 'Weder * noch 'pwd' werden ersetzt.'

echo 'Weder * noch pwd werden ersetzt.'
```

`...` Zwischen Backquotes (auch Accent Grave genannt) gesetzte Kommandos werden ausgeführt und das Ergebnis wird dann als Parameter übergeben (d.h. die Ausgabe des Kommandos wird als Parameter des aufzurufenden Programms gewertet.

```
echo "Aktuelles Verzeichnis: `pwd`"
```



In der bash kann anstatt der Backquotes auch die Kommandofolge in $(...) angegeben werden.

Bourne Again Shell (bash)

Bourne Again Shell (bash)

- Erweiterung der Original Bourne Shell
- Falls bash Login-Shell (in `/etc/passwd` eingetragen), werden folgende Dateien ausgeführt
  - `/etc/profile`
  - `~/.bash_profile`, `~/.bash_login`, oder `~/.profile` (nur eine dieser Dateien)
- Außerdem wird `~/.bashrc` ausgeführt, falls vorhanden (bei jedem Start, nicht nur bei Login)

Die bash (Bourne Again Shell) ist die Standard-Shell bei fast allen Linux-Systemen. Sie ist vollständig kompatibel zur originalen Bourne Shell, hat aber etliche Erweiterungen erfahren.

Startdateien

Wenn die Shell das erste Mal aufgerufen wird, arbeitet sie zunächst die Datei ‚/etc/profile' ab. Hier kann der Systemverwalter für alle Anwender gültige Kommandos und Variablen eintragen. Anschließend sucht die bash nach einer der folgenden Dateien im Home-Verzeichnis des Benutzers und führt diejenige aus, die zuerst gefunden wird: „*.bash_profile*' (Standard), „.bash_login', „.profile'.

Danach wird die Datei „.bashrc' ausgeführt. Wird die bash jedoch nicht als Login-Shell gestartet so führt sie ausschließlich nur die Datei „.bashrc' aus.

Die Ausführung der Dateien lässt sich über zwei Kommandozeilen-Parameter steuern. Mit dem Parameter ‚-noprofile' wird veranlasst, dass die bash keine der oben genannten Startdateien ausführt. ‚-norc' bewirkt, dass die persönliche Konfigurationsdatei „.bashrc' ignoriert wird.

Der Prompt

Prompt

- Erscheinung definiert durch Variable PS1 und PS2.
- Möglichkeiten (Einleitung durch ‚\'):

h	Rechnername ohne Domäne
H	Vollständiger Rechnername
t	Aktuelle Uhrzeit
u	Benutzername (Login-Name)
W	Aktuelles Verzeichnis
w	Aktuelles Verzeichnis mit Shortcuts
#	Fortlaufende Nummer des Befehls
$	Benutzerkennung (‚#' oder ‚$')

Die bash gibt einen Eingabeprompt aus, häufig in der Form ‚[*Benutzername@Hostname aktuelles Verzeichnis*]$'. Die Erscheinung und angezeigte Information des Prompts ist über die Shellvariable ‚PS1' konfigurierbar. Der oben angegebene Prompt würde aus der Einstellung „PS1=[\u@\h \W]\$" resultieren. Um andere Einstellungen auszuprobieren, muss die Variable neu belegt werden. Neben jedem beliebigen Text können folgende Konstanten verwendet werden, die durch einen Backslash eingeleitet werden müssen:

h Rechnername ohne Domäne

H Vollständiger Rechnername (inkl. Domäne)

t Aktuelle Uhrzeit

u Benutzername (Login-Name)

W Aktuelles Verzeichnis

w Aktuelles Verzeichnis, mit Angabe von Shortcuts (‚~' für Home-Verzeichnis,...)

Fortlaufende Nummer des aktuellen Befehls

$ Benutzerkennung: „#" für root, „$" für normale Benutzer

Die Neubelegung in der aktuellen Shell erfolgt durch Zuweisung eines neuen Werts an die PS1-Variable. Dabei muss geachtet werden, dass die Zuweisung nicht durch die Shell ersetzt wird, d.h. der zugewiesene Wert muss in einfachen Anführungszeichen angegeben werden.



Beispiele:

Zuweisung:	*Ausgabe:*
PS1='[\u@\h \W]\$ '	[probst@Esprit /bin]$
PS1='[\t \w]\# \$ '	[13:37:36 ~]36 $
PS1='\$ '	$

! Der zugewiesene Wert bezieht sich nur auf die aktuelle Shell. Wird eine neue Shell gestartet bzw. die aktuelle Shell beendet, so wird der Prompt wieder zurückgesetzt. Will man das Aussehen des Prompts permanent ändern, so muss die Zuweisung in einer der Startdateien der bash erfolgen.

Shellvariablen

Shellvariablen$_1$

- Möglichkeit, Werte (Zeichenketten) mitzuführen
- Gültigkeit:
  - Ab Zuweisung ist Variable gültig
    - Zuweisung: `name=wert`
  - Gilt nur in aktueller Shell
  - Zugriff in anderen Shells mit ‚*export*‘
    - Exportieren: `export name`

Wie schon oftmals zuvor erwähnt, gibt es in der Shell eine Möglichkeit, Variablen zu deklarieren und mitzuführen. Die Werte dieser Shellvariablen werden dabei immer als Zeichenketten interpretiert, es gibt keine verschiedenen Datentypen. Nur in Ausnahmefällen werden diese Zeichenketten als Zahlen oder logische Werte interpretiert. Shellvariablen erinnern stark an das Konzept von Variablen in Programmiersprachen. Jedoch werden Shellvariablen nicht deklariert sondern es wird ihnen mit einer einfachen Zuweisung ein Wert zugewiesen.

Zuweisung

Die Shellvariable ist ab dem Zeitpunkt ihrer Zuweisung gültig. Die Zuweisung erfolgt hierbei durch die Angabe des Variablennamens gefolgt von dem Zuweisungsoperator ‚=’ und einem Zeichenkettenwert (`name=wert`). Dabei darf vor und nach dem Zuweisungsoperator kein Leerzeichen stehen.

Gültigkeitsdauer

Ab der Zuweisung existiert die Shellvariable solange die Shell existiert. Die Variable hat nur Gültigkeit in der Shell in der sie definiert wurde. Wird eine neue Shell aufgerufen oder die Shell beendet, so kann auf die Variable nicht zugegriffen werden. Will man jedoch auf die Variable auch in anderen Shells zugreifen, dann muss die Variable exportiert werden. Dies geschieht durch die Eingabe von „export *variablenname*“.

Verwendung von Shellvariablen

Shellvariablen$_2$

- Verwendung:
  - ‚`set`' listet alle definierten Variablen auf
  - Zugriff über ‚$*name*'
    - Liefert Wert der Variablen *name*
    - Falls Variable noch nicht gesetzt, wird leere Zeichenkette eingesetzt.
- Umgebungsvariablen
  - Spezielle Bedeutung
  - Von Programmen als vorhanden vorausgesetzt

Mit dem Shellkommando ‚set' (ohne Parameter) kann man abfragen, welche Variablen zurzeit in der Shell gesetzt sind und welche Werte diese haben. Bei einer großen Anzahl von Variablen kann es sinnvoll sein, die Ausgabe über eine Pipe seitenweise auszugeben (Kommando „set | more").

In der Shell können Variablen an jeder Stelle verwendet werden. Immer wenn die Shell bei der Analyse einer Kommandozeile ein Wort findet, das mit dem Dollarzeichen (‚$') beginnt, so wird dieses Wort als Shellvariable interpretiert. Wird ein solches Wort gefunden, so wird die Shellvariable durch ihren entsprechenden Wert ersetzt. Wird jedoch keine passende Variable gefunden, so wird gar nichts eingesetzt.

Beispiel:

Eingabe:	*Ausgabe:*
NAME=myname	
echo $NAME	myname
echo NAME	NAME
echo Der Wert $xyz wird eingesetzt.	Der Wert wird eingesetzt
xyz=1	
echo Der Wert $xyz wird eingesetzt.	Der Wert 1 wird eingesetzt.

Umgebungsvariablen

Shell-Umgebungsvariable

- `IFS` Internal Field Separator
- `PATH` Liste der Verzeichnisse, in denen nach Kommandos gesucht wird
- `PS1` Aussehen des Prompts
- `HOME` Home-Verzeichnis
- `HOSTNAME` Name des Rechners
- `HOSTTYPE` Typ des Rechners
- `PWD` aktuelles Verzeichnis
- `PPID` id des Vaters, der Shell erzeugte
- etc.

Umgebungsvariablen haben eine spezielle Bedeutung, weil viele Programme sie abfragen oder sie schon beim Login eines Benutzers automatisch mit Werten belegt werden. Diese Umgebungsvariablen sind ganz normale Shellvariablen, sind jedoch normiert und werden von gewissen Programmen als vorhanden vorausgesetzt. In weiterer Folge sind die wichtigsten dieser Umgebungsvariablen aufgeführt und kurz erklärt:

IFS (Internal Field Separator): Die Variable enthält eine Liste von Zeichen, die Trennzeichen zwischen Worten darstellen. Voreingestellt enthält diese Liste die Zeichen Leerzeichen, Tabulator und Zeilenende.

PATH: Wenn man in UNIX ein Kommando eingibt, so wird es nacheinander in den in dieser Variablen aufgeführten Verzeichnissen gesucht. Üblicherweise enthält diese Variable eine Liste von Verzeichnissen. Die Namen der Verzeichnisse sind dabei durch einen Doppelpunkt voneinander getrennt und werden genau in der aufgeführten Reihenfolge durchsucht.



Diese Variable ähnelt sehr stark der Pfad-Variablen von MS-DOS. Jedoch gibt es einen wichtigen Unterschied. Bei MS-DOS ist das aktuelle Verzeichnis immer automatisch das erste Verzeichnis im Suchpfad. In UNIX dagegen muss das aktuelle Verzeichnis ausdrücklich mit dem Namen „.' im Suchpfad aufgeführt sein, wenn es auch bei einem Kommandoaufruf durchsucht werden soll. Wird das aktuelle Verzeichnis nicht in der PATH-Variablen definiert, so werden ausführbare Dateien im lokalen Verzeichnis nicht gefunden – sofern diese nicht mit ihrem vollständigen Pfad angesprochen werden.

PS1 (Prompt String 1): Wie schon zuvor in diesem Kapitel angeführt, definiert diese Variable das Aussehen des Prompts. Es wurden dort bereits Beispiele und ausführliche Informationen zu dieser Shellvariable gegeben.

PS2 (Prompt String 2): Wenn eine Fortsetzungszeile angefordert wird, verwendet die Shell den in der Variablen PS2 definierten Promptstring. In der Regel steht dieser auf ‚> '.

HOME: In der Variablen *HOME* ist der absolute Pfad des Home-Verzeichnis des aktuellen Benutzers gespeichert. Dadurch kann durch die Eingabe des Kommandos „cd $HOME" automatisch in das Home-Verzeichnis gewechselt werden.

HOSTNAME: Diese Variable beinhaltet den Rechnernamen.

Im Folgenden sind einige Shellvariablen aufgeführt, die häufig genutzt werden jedoch nicht auf jedem System vorhanden sein müssen.

HOSTTYPE: Diese Variable gibt Informationen über die Rechnerarchitektur des zugrundeliegenden Systems aus.

PWD: Diese Variable enthält das aktuelle Arbeitsverzeichnis.

PPID: Diese Variable enthält die Prozess-ID des Vaters der aktuell genutzten Shell.

Es gibt natürlich eine Reihe anderer Shellvariablen. Die genaue Funktion hängt jedoch stark vom benutzen System und den eingesetzten Programmen ab.

Spezielle Variablen

Es gibt jedoch auch Variablen mit speziellem Zweck. Diese Variablen werden fast ausschließlich in Shellskripten verwendet.

$0: Diese Variable enthält den Namen des aufrufenden Befehls, also bei einem Shellskript den Namen der Skriptdatei.

$1 bis $9: Diese Variablen enthalten die Parameter 1 bis 9 des Aufrufers. Wurde das Kommando mit mehr als 9 Parameter aufgerufen, so sind die weiteren Parameter nicht direkt ansprechbar. Zugriff zu diesen Variablen kann über das Shellkommando „shift" erfolgen. Diese Möglichkeit wird im Kapitel 7 Shellprogrammierung noch näher behandelt.

$*: Diese Variable gewährleistet Zugriff auf alle Parameter der Kommandozeile als ein zusammenhängendes Wort. Dieser Parameter entspräche der Konstruktion „$1 $2 $3...".

$@: Diese Variable liefert alle Parameter als eine Folge von einzelnen Worten. Dieser Parameter entspräche der Konstruktion „$1", „$2", „$3",...

$#: Diese Variable gibt Aufschluss über die Anzahl von übergebenen Parametern.

$-: In dieser Variablen speichert die Shell die derzeit eingeschalteten Optionsbuchstaben.

$?: Beim Beenden eines UNIX-Kommandos hinterlässt dieses eine Zahl, den sogenannten Exit-Status. Dieser gibt Aufschluss darüber, ob das Kommando normal beendet worden ist oder durch einen Fehler abgebrochen wurde. Per Konvention deutet 0 auf ein fehlerfreies Ende hin, jede andere Zahl bedeutet eine fehlerhafte Abarbeitung des Befehls. In der Shellvariablen $? steht der Exit-Status des zuletzt ausgeführten Befehls.

$$: Jedes Kommando wird in UNIX als eigener Prozess ausgeführt. Jeder Prozess ist dabei durch eine eindeutige Prozessnummer identifiziert. Die Prozessnummer der Shell steht in der Variablen $$.

Interne Kommandos

Interne Kommandos

- `alias` Namen für Befehle definieren
- `echo` Ausgabe am Bildschirm
- `hash` Liste der gemerkten Pfade zu den Programmen und Informationen wie oft Programm aufgerufen wurde
- `logout` Kann nur in Login-Shell aufgerufen werden; meldet den aktuellen Benutzer vom System ab

Interne Kommandos zeichnen sich dadurch aus, dass sie keine ausführbare Datei benötigen und immer auf jedem System verfügbar sind. Im Folgenden werden die wichtigsten internen Kommandos vorgestellt und erklärt.

alias: Das Kommando alias weist einem Befehl oder einer Befehlsfolge einen neuen Namen zu. Das Kommando ist dann sowohl unter dem neuen als auch dem alten Namen ansprechbar. Besonders sinnvoll und angenehm ist es, wenn man Alias-Namen für Kommandos mit sehr langen Parameterlisten übergibt. Gerade für MS-DOS Umsteiger lassen sich mit diesem Kommando DOS ähnliche Befehle realisieren:

```
alias dir='ls –la'
alias cd..='cd ..'
alias md='mkdir'
```

echo: Das Kommando echo gibt Text am Bildschirm aus. Soll der Zeilenumbruch am Ende der Ausgabe unterdrückt werden, verwendet man den Parameter –n.

hash: Das Kommando erwartet keine Parameter und zeigt eine Liste der gemerkten Pfade zu Programmen an. Um die Zugriffe auf Programme zu beschleunigen, verwaltet die bash einen internen Cache der Pfade auf bereits gestartete Programme enthält. Das erspart die zeitaufwendige Suche des Pfades. Das Kommando zeigt auch an, wie oft das Programm gestartet wurde.

logout: Dieses Kommando kommt einem „exit"-Aufruf auf der letzten aktiven Shell gleich. Das Skript „.bash_logout' wird ausgeführt und die Login-Shell wird beendet.

Interne Jobkontrolle

Interne Jobkontrolle

- Gestoppte Prozesse (Tastenkombination [STRG]+[Z]) mit der Shell verwalten
  - `jobs`: Liste aller im Hintergrund befindlichen Prozesse
  - `fg` *`jobnummer`*: Prozess im Vordergrund fortsetzen
  - `bg` *`jobnummer`*: Prozess im Hintergrund ausführen

Es gibt aber auch einige Befehle zur internen Jobkontrolle. **fg**, **bg** und **jobs** sind solche Befehle. Ein laufendes Programm kann mit der Tastenkombination <STRG>+<Z> in den Hintergrund verschoben werden. Dort wird es zunächst gestoppt (suspended). Das Kommando **jobs** gibt nun Aufschluss über alle derzeit im Hintergrund befindlichen Prozesse. Dabei wird jedes Programm mit seiner internen Jobnummer angegeben. Mit „**fg *Jobnummer***“ wird der Prozess wieder in den Vordergrund geschickt. Soll ein angehaltenes Kommando dagegen im Hintergrund weiterlaufen, verwendet man den Befehl „**bg *Jobnummer***“.

Kapitel

7

Shellprogrammierung

Dieses Kapitel befasst sich mit der Möglichkeit, Skripte und kleine Programme innerhalb der UNIX-Shell zu erstellen.

Die Shell ist der Kommandozeileninterpreter von UNIX. Sie ist aber gleichzeitig auch die wichtigste Schnittstelle zwischen System und Benutzer. Eingegebene Kommandos werden von der Shell interpretiert und entsprechend verarbeitet. Innerhalb einer Shell können auch eigene Shellprogramme (Shellskripte) erstellt werden. Diese dienen zum Automatisieren von Vorgängen, die sonst manuell durchgeführt werden müssen. Da die Shell nicht direkt in das Betriebssystem integriert ist, kann der Benutzer zwischen verschiedenen Shells auswählen.

Lehrinhalte und -ziele

Nach diesem Kapitel sollten Sie in der Lage sein, die Möglichkeiten der Shellprogrammierung zu kennen und verschiedene Shellskripte zu programmieren. Dabei sollten Sie die zur Verfügung stehenden Konstrukte kennen und richtig anwenden können.

Was ist Shellprogrammierung?

Motivation und Eigenschaften

- Shell bietet Ablaufstrukturen ähnlich wie Programmiersprachen
- Textdateien die Linux-Kommandos und Kontrollstrukturen einhalten bezeichnet man als Shellskripte
- Shellskripte sollen die Bedienung, die Administration und die Wartung des Systems erleichtern
- Shellskripte erweitern Funktionalität des Systems
- Shellskripte werden interpretiert

Die Shell dient nicht nur der Kommunikation mit dem Benutzer, sondern bietet eine Möglichkeit, Vorgänge zu automatisieren. Dabei bietet sie Konstrukte die von Programmiersprachen her bekannt sind. Diese Funktionalität nennt man auch Shellskripte.

Shellskripte sind Textdateien, die einzelne Anweisungen und weitere UNIX-Kommandos enthalten. Dieses Skript wird dann von der Shell gelesen und abgearbeitet. Die Shell arbeitet interpretierend mit allen sich daraus ergebenden Vor- und Nachteilen. Mit Hilfe solcher Shellskripte kann die Funktionalität des Systems erweitert werden. Im Normalfall greifen solche Shellskripte auf bestehende Kommandos zurück und reihen diese in einer Art und Weise, dass Prozesse und Vorgehen automatisiert werden können. Diesen Shellskripten sind in Punkto Funktionalität keine Grenzen gesetzt.

Shellskripte liegen immer in einem lesbaren Format vor, da sie reine Textdateien sind. Diese Textdateien können leicht weitergegeben werden und auf anderen Systemen eingesetzt werden. Shellskripte sind nicht an ein Derivat oder eine Distribution gebunden sondern an einen bestimmten Shell-Typ. Dabei kann im Skript festgelegt werden, für welche Shell das Skript geschrieben worden ist.

ⓘ Shellskripte können mit Batch-Dateien (.bat-Dateien) aus der DOS-Welt verglichen werden. Jedoch bieten Shellskripte erheblich mehr Funktionalität und bieten neben Schleifen, Bedingten Anweisungen und Mehrfach-Auswahl auch die Möglichkeit, Variablen mitzuführen und zu verwalten.

Allgemeines zur Shellprogrammierung

Allgemeines$_1$

- Alle Shellskripte in eigenem Verzeichnis verwalten (z.B. `~/bin`)
- Shellskript als Textdatei erstellen
- Ausführungsrechte setzen
  - `chmod u+x` *`filename`* oder
  - `chmod 7**` *`filename`*
- Shellskripte können wie Programme verwendet werden

Das Shellskript kann alle Möglichkeiten der Shell nutzen, die auch bei der interaktiven Eingabe möglich sind. Insbesondere kann auch von der Ein-/Ausgabeumleitung Gebrauch gemacht werden. Es lassen sich selbstverständlich innerhalb eines solchen Skripts auch weitere Shellskripte aufrufen, zusätzlich zu allen Befehlen der Shell. Um Skripte universeller gestalten zu können, ist es möglich, dem Skript Parameter zu übergeben.

Konventionen

Beim Arbeiten mit Shellskripten empfiehlt es sich, gewisse Konventionen einzuhalten.

Shellskripte sollten prinzipiell in einem eigenen Verzeichnis verwaltet werden. Dazu empfiehlt es sich, im Home-Verzeichnis des Benutzers ein neues Unterverzeichnis anzulegen, dass auch in der PATH-Variablen der Shell referenziert wird.

Shellskripte werden als Textdatei erstellt. Sie können über eine Sub-Shell aufgerufen werden (z.B. „sh *dateiname*“) oder aber direkt über den Namen, sofern das Ausführrecht für die Datei vergeben worden ist. Dies geschieht mit dem Kommando ‚chmod'. Dieses Kommando wurde bereits im Kapitel 5 „Zugriffsrechte“ näher erläutert. Wenn das Ausführrecht gesetzt worden ist, können Shellskripte wie normale Programme verwendet werden. Trotzdem kann das Skript noch mit einem Texteditor geöffnet und verändert werden.

Leer- und Tabzeichen können beliebig verwendet werden, was Einrückung und eine übersichtliche Gestaltung des Skripts erlaubt.

Allgemeines$_2$

- Kommentare
  - # Alles was hinter dem # steht wird als Kommentar bezeichnet
    Sonderbedeutung: *#!/bin/sh*
  - : null command (Keine Ausführung aber Ersetzung)
- Testen
  - Aufruf des Skripts über Sub-Shell (sh *filename*)
  - Optionen
    - -v (verbose): Befehle werden ausgegeben
    - -x (execute): Ersetzungen werden ausgegeben

Kommentare

Wie herkömmliche Programme sollten auch Shellskripte kommentiert werden. Kommentare werden durch die Zeichen ‚#' eingeleitet. Dabei kann das ‚#' an beliebiger Stelle stehen, der Kommentar gilt bis zum Zeilenende. Der Doppelpunkt (‚:') ist das sogenannte *null command* und retourniert immer einen Exit-Status von 0. Shellvariablen können nach dem ‚:' platziert werden und werden von der Shell durch ihre Werte ersetzt, angegebene Kommandos werden jedoch nicht ausgeführt. So kann man Variablenersetzung quasi testen.

! Es gibt jedoch eine spezielle Sonderform von Kommentaren. In der ersten Zeile des Skripts sollte ein Kommentar vermerkt sein, der zur Festlegung der verwendeten Shell bestimmt ist.

`#!`*Pfad+Shellname*

Beispiele:

`#!/bin/sh`	Skript wurde für Bourne Shell entwickelt
`#!/bin/bash`	Skript wurde für Bourne Again Shell entwickelt

Da verschiedene Benutzer möglicherweise unterschiedliche Shells benutzen, muss dafür gesorgt werden, dass Shell-spezifische Kommandos richtig interpretiert werden können. Dieser spezielle Kommentar legt fest, welche Shell für die Interpretation und Ausführung des Skripts verwendet wird.

Testen eines Shellskripts

Die einzelnen Ersetzungsmechanismen machen es manchmal nicht leicht, das Verhalten des Skripts vorherzubestimmen. Dadurch ist es notwendig, dass Skripte vor Ihrem Einsatz auch auf Korrektheit getestet werden können. Zum Testen bieten sich daher einige Möglichkeiten an, die in weiterer Folge behandelt werden.

Die erste Möglichkeit ist das Einfügen von echo-Kommandos anstelle der vorgesehenen Kommandos. Die Ersetzung findet bei diesen Kommandos genauso statt wie bei allen anderen Befehlen. Man kann so die Ersetzung visuell darstellen und prüfen, ob das Skript korrekt damit umgeht.

Die zweite Möglichkeit besteht darin, die interpretierende Sub-Shell über Optionen aufzurufen. Dabei bieten sich folgende Optionen an:

- -v (verbose): Die Shell gibt alle zu bearbeitenden Befehle aus.
- -x (execute): Die Shell gibt die Ersetzungen aus
- -n (execute): Die Shell gibt die Befehle zwar aus, sie werden jedoch nicht ausgeführt.

! Kritische Aufrufe (z.B. das Löschen-Kommano ‚rm') sollten vor dem endgültigen Einsatz erst mal durch davor gesetzte echo-Befehle entschärft werden. Nachdem das Skript getestet worden ist, können die echo-Befehle gelöscht werden und die richtigen Befehle eingesetzt werden.

Shellvariablen

Shellvariablen

- Namen frei wählbar, nehmen Zeichenketten auf, max. 200 Zeichen
  - Zeichenkette nur aus Ziffern = kann als Integer Zahl interpretiert werden (z.B. `expr`)
- Drei Formen der Verwendung von Variablen
  - Variablendeklaration (VAR=) → Shell-Variable wird leere Zeichenkette zugewiesen
  - Wertzuweisung (VAR=`Wert` oder VAR=`"Wert"`)
- Wertreferenzierung ($VAR) Wert der Variable wird mit dem $-Zeichen ermittelt
- Variablen sind nur in Shell sichtbar, in der sie definiert wurden
  - `export` VAR → auch in allen Sub-Shells sichtbar

Shellvariablen wurden bereits im Kaptitel 6 „Die Shell“ eingehend erläutert. Hier ist lediglich eine kurze Wiederholung der wichtigsten Eigenschaften verzeichnet.

Shellvariablen bestehen aus einem frei wählbaren Bezeichner (dem Namen), dem eine beliebige Zeichenkette zugeordnet wird. Der Name muss dabei mit einem Buchstaben beginnen und kann max. 200 Zeichen umfassen. Der zugewiesene Wert ist immer eine Zeichenkette. Besteht die Zeichenkette jedoch nur aus Ziffern, kann diese von bestimmten Kommandos als Integer-Zahl interpretiert werden.

Die Verwendung von Variablen in der Shell bringt einige Besonderheiten mit sich. Von Programmiersprachen ist es bekannt, dass Variablen vor Ihrer Verwendung zuerst deklariert werden müssen. In der Shell werden Variablen aber nicht explizit deklariert. Vielmehr ist in der Wertzuweisung die Deklaration enthalten. Bei der Wertzuweisung wird einem Variablennamen ein Wert zugewiesen. Somit ist der Shell die Variable bekannt. Wird kein Wert zugewiesen (z.B. ‚VAR=’), dann weist die Shell der Variablen eine leere Zeichenkette zu. Zugriff auf den Wert der Variablen erfolgt durch ein vorangestelltes ‚$’-Zeichen. Dieses Zeichen bewirkt, dass die Shell den Variablennamen auflöst und diesen mit dem aktuellen Wert ersetzt. Zwischen dem $ und dem Variablennamen darf kein Leerzeichen stehen (z.B. ‚$VAR’).

Eine weitere Besonderheit ist der Gültigkeitsbereich von Variablen. Diese gelten nur in der Shell, in der sie definiert wurden. Soll die Variable auch in verzweigenden Sub-Shells sichtbar sein, so ist die Variable zuerst mit dem export-Befehl sichtbar zu machen.

Jede verzweigende Sub-Shell läuft dabei in einer eigenen Umgebung und vergisst somit die darin definierten Variablen bei Beendigung. Variablen können nur von einer Vater-Shell an eine Sub-Shell übergeben werden (mit Hilfe des export-Kommandos), jedoch nicht umgekehrt. Bei der Zuweisung von Werten an eine Variable ist die intakte Shellersetzung zu beachten. Es ist somit möglich, bestehende Variablen zu erweitern.

`PATH=$PATH:`*`neuerPfad`*

Verwendung von Shellvariablen

Um Shellvariablen von Kommandos und Befehlen unterscheiden zu können, sollten diese immer groß geschrieben werden. Dabei ist zu beachten, dass UNIX in allen Bereichen zwischen Groß- und Kleinschreibung unterscheidet.

Soll der Wert der Variablen mit einer Zeichenkette konkateniert werden, ist der Name in geschweifte Klammern zu schreiben. Damit kann die Shell erkennen, wo der Variablenname endet.

`VAR=`pwd``	Weist VAR aktuelles Verzeichnis zu (z.B. /bin)
`echo ${VAR}old`	Gibt Inhalt von VAR aus und hängt ‚old' dran (/binold)
`echo $VARold`	Gibt Inhalt von VARold aus (leere Zeichenkette)

Spezielle Variablen

Vordefinierte / Spezielle Variable

- Vordefinierte Variable siehe Shell (oder env)
- Spezielle Variable
  - $- gesetzte Shell-Optionen
  - $$ Prozess-ID
  - $? Exit-Status des letzten Kommandos
- Parameterzugriff
  - $0 Name des Shellskripts
  - $1-9 1-9ter Parameter
  - $# Anzahl der Parameter
  - $* Alle Parameter als eine Zeichenkette
  - ‚shift' eliminiert ersten Parameter, Folgeparameter werden um eine Stelle nach links verschoben

Die Shell beinhaltet eine ganze Reihe von vordefinierten Variablen, die vom Betriebssystem oder von anderen Programmen benötigt werden. Eine Liste dieser Variablen ist im Kapitel 6 „Die Shell" zu finden. Dort sind auch die speziellen Variablen erklärt, die ihren Einsatz im Gebiet der Shellprogrammierung finden. Diese speziellen Variablen werden jedoch hier zur Wiederholung nochmals kurz angeführt.

Spezielle Variablen

Mit Hilfe dieser speziellen Variablen kann man Informationen bezüglich der Shell und anderer Kommandos beziehen.

- $-: Diese Variable liefert die gesetzten Shelloptionen. Welche Optionen gesetzt sind und deren Auswirkungen hängen dabei von der Shell ab. Die verwendete Shell kann durch die Variable $0 ermittelt werden.

- $$: Diese Variable liefert die eindeutige Prozesskennung (PID) des aktuellen Prozesses. Wird diese Variable von der Shell aufgerufen, so wird hier die PID der Shell eingesetzt, innerhalb eines Skripts würde hier die PID des Skripts stehen.

- $?: Jeder Prozess übermittelt an die Shell einen Exit-Status der Aufschluss gibt, wie der Prozess beendet wurde. Im Erfolgsfall steht hier eine ‚0', sonst eine andere Zahl. Der Exit-Status des letzten Prozesses kann mit der Variable ‚$?' ermittelt werden.

Parameterzugriff

Jedem Prozess können eine Reihe von Parametern übermittelt werden. Es obliegt dann dem gestarteten Prozess diese Parameter auszuwerten. Um Shellskripte flexibler gestalten zu können, sollte von Parametern Gebrauch gemacht werden. Der Zugriff auf solche Parameter erfolgt wieder über spezielle Variablen.

- $0: Diese Variable enthält den Namen des Prozesses. Im Normalfall steht hier der Dateiname der ausführbaren Datei bzw. der Skriptdatei.

- $1-$9: Die Variablen $1-$9 enthalten den ersten bis neunten Parameter, der an den Prozess übergeben worden ist. Über diese Variablen kann ein Parameter an einer Stelle direkt referenziert werden. Durch die Metazeichenersetzung ist es aber möglich, dass einem Kommando mehr als nur neun Parameter übergeben werden. Auf diese Parameter kann durch das Kommando ‚shift' zugegriffen werden. ‚shift' eliminiert den ersten Parameter und schiebt alle Parameter um eine Stelle nach vorne. An die letzte, nun frei gewordene Stelle, wird nun der nächste Parameter (also beim ersten Mal der Parameter 10) kopiert. Um beliebig lange Parameterlisten abarbeiten zu können, empfiehlt es sich, in einer Schleife nur mit Parameter $1 zu arbeiten und solange ‚shift' aufzurufen, bis keine weiteren Parameter mehr vorhanden sind.

- $*: Diese Variable liefert alle übergebenen Parameter in *einer* Zeichenkette. Die Parameter werden dabei durch ein Leerzeichen (blank) voneinander getrennt.

- $#: Diese Variable liefert die Anzahl der übergebenen Parameter.

Weitere Parameterzugriffsvariablen sind im Kapitel 6 verzeichnet.

Beliebige Anzahl von Parametern bearbeiten

Wie schon zuvor erwähnt, kann eine beliebige Anzahl von Parametern mit Hilfe des ‚shift' Kommandos bearbeitet werden. Direkter Zugriff ist nur auf die ersten neun Parameter möglich. Das shift-Kommando löscht dabei den ersten Parameter und rückt mit allen folgenden Parametern um eine Stelle auf. Dadurch wird die letzte (neunte) Stelle frei. Dort kopiert das shift-Kommando den nächsten Parameter (also beim ersten Mal den 10ten, beim zweiten Mal den 11ten usw.) hinein.

Bei einer beliebigen Anzahl von Parametern empfiehlt sich die Verwendung einer Schleife, die so lange shift aufruft, bis die Anzahl der Parameter 0 ist.

```
while [ $# -gt 0 ]              # solange noch Parameter vorhanden
do
  Kommandofolge
  shift
done
```

Namens- und Parameterersetzung

Die einfache Parameterersetzung besteht darin, dass der Wert der Variablen textuell ersetzt wird. Diese Möglichkeit wurde bereits ausführlich erläutert.

Zusätzlich gibt es aber noch die Möglichkeit, Voreinstellungen zu vereinbaren und auf fehlende Parameter zu reagieren. Bei dieser Art von Parameterersetzung wird zuerst geprüft, ob die referenzierte Variable existiert und abhängig von deren Existenz erfolgt dann die Ersetzung. Diese Ersetzung funktioniert jedoch nur bei Variablenreferenzierung und muss sich daher innerhalb des Referenzierungszeichens ‚$' und geschweiften Klammern ‚{', ‚}' befinden.

`${variablenname:-neuerWert}`: Ist die Variable ‚*variablenname*' bereits deklariert, so wird dieser Wert eingesetzt. Ist die Variable jedoch noch nicht bekannt, so wird stattdessen ‚*neuerWert*' eingesetzt. Der Wert der Variablen bleibt jedoch unverändert!

`${variablenname:=neuerWert}`: Ist die Variable ‚*variablenname*' bereits deklariert, so wird dieser Wert eingesetzt. Ist die Variable jedoch noch nicht bekannt, so wird eine neue Variable mit dem Namen ‚*variablenname*' deklariert und der neuen Variablen der Wert ‚*neuerWert*' zugewiesen und dieser Wert dann eingesetzt. Falls die Variable also zuvor noch nicht existiert hat, dann existiert sie nach Ausführung dieses Statements.

`${variablenname:?fehlerMeldung}`: Ist die Variable ‚*variablenname*' bereits deklariert, so wird dieser Wert eingesetzt. Ist die Variable jedoch noch nicht bekannt, so wird die Fehlermeldung ‚*fehlerMeldung*' ausgegeben und das Shellskript abgebrochen. Die Fehlermeldung wird an *stderr* geschickt.

`${variablenname:+neuerWert}`: Ist die Variable ‚*variablenname*' bereits deklariert, so wird statt diesem Wert der Wert ‚*neuerWert*' eingesetzt. Ist die Variable jedoch noch nicht bekannt, so bleibt dieser Zustand erhalten und es wird nichts eingesetzt. Der Wert der Variablen bleibt jedoch auf jeden Fall erhalten.

Beispiele

	Eingabe	**Ausgabe**
Zuweisung:	W1='Max'	
Einfache Substitution	echo $W1	Max
String-Konkatenation	echo ${W1}.Mustermann	Max.Mustermann
Bedingte Substitution	echo ${W1:-"Test"} echo ${W2:-"Test"}	Max Test
Bedingte Zuweisung	echo ${W1:="Test"} echo ${W2:= "Test"} echo $W2	Max Test Test
Abbruch-Zuweisung	echo ${W1:?"Fehler"} echo ${W3:? "Fehler"}	Max bash: W3: Fehler
Nur-Wenn-Definiert Zuweisung	echo ${W1:+"Test"} echo ${W3:+ "Test"} echo $W1	Test *<< keine Ausgabe >>* Max

Interaktive Eingabe

Interaktive Eingabe

- Interaktive Eingabe von Daten in Shellskript
  - `read variable [variable ...]`
  - Liest Werte von *stdin* und weist Werte den angegebenen Variablen zu
  - Getrennt durch *IFS* Zeichen
  - Falls mehr Variablen als Werte → Rest mit Leerstrings belegt
  - Falls mehr Werte als Variablen → Letzte Variable bekommt Rest der Werte als Zeichenkette zugewiesen

Manchmal ist es notwendig, dass ein Benutzer neben Parameter auch zur Laufzeit mit dem Skript interagieren kann. Dazu bietet die Shell das read-Kommando an, dass Werte von der Standardeingabe (*stdin*) liest und den angegebenen Variablen zuweist. Das Kommando selbst hat folgende Syntax:

`read variable1 variable2 ... variable`$_n$

Das Kommando liest dabei eine Zeile von der Standardeingabe und weist die einzelnen Eingaben den angegebenen Variablen zu. Dabei wird die Eingabe in Felder getrennt. Feldtrenner sind die Zeichen, die in *IFS* definiert sind. Wurden mehr Variablen angegeben als Felder eingegeben wurden, so werden die Variablen nach Möglichkeit gefüllt, die restlichen Variablen werden mit leeren Zeichenketten belegt. Wurden mehr Werte als Variablen angegeben, so wird der letzten Variablen die restlichen Werte als Zeichenkette zugewiesen.

Das read-Kommando bezieht seine Eingaben von der Standardeingabe. Wurde das Shellskript jedoch mit einer Eingabeumleitung aufgerufen (‚*<datei*'), so werden die Eingaben aus der angegebenen Datei bezogen. Dabei werden Eingaben zeilenweise aus der Datei abgearbeitet.

! Obwohl bei der Eingabeumleitung die Eingabedatei zeilenweise abgearbeitet wird, bedeutet das nicht, dass einer Variablen immer eine ganze Zeile zugeordnet wird. Vielmehr wird die Zeile auf Feldtrenner durchsucht und dann entsprechend den angegebenen Variablen aufgeteilt. Eine Zeile in der Eingabedatei entspricht somit einer langen Eingabe über die Tastatur.

Da das Shellskript in einer eigenen Sub-Shell läuft, kann die Variable *IFS* entsprechend der erwarteten Eingabe beliebig umdefiniert werden, ohne dass die Vater-Shell oder später aufgerufene Shells oder Kommandos beeinflusst werden. Nach Beendigung des Skripts steht in der Variablen *IFS* wieder der ursprüngliche Wert.

Beispiel

Shellskript ‚interaktiveEingabe'

```
#!/bin/bash

IFS=','
echo "Eingabe von 3 Werten (getrennt durch Komma):"
read A B C                              # liest Werte von Eingabe
echo Eingabe war: $A $B $C
```

Ablaufstrukturen

Ablaufstrukturen$_1$

- ‚test *Bedingung*' oder ‚[*Bedingung*]'
  - Prüft Bedingung, liefert 0 falls wahr, 1 falls falsch und 2 falls Fehler im Argument
- if – then – else
  - ```
    if bedingung
      then kommandos
    fi
    ```
  - ```
    if bedingung
      then kommandos
    else kommandos
    fi
    ```
  - ```
    if bedingung1
      then kommandos
    elif bedingung2
      then kommandos
    elif ...
    fi
    ```

Shellskripte unterscheiden sich von Batch-Programmen insofern, dass dem Benutzer ausgefeilte und umfangreiche Ablaufstrukturen angeboten werden. Diese werden in diesem Abschnitt näher behandelt. Zusammen mit den Shellvariablen und den Ein-/Ausgabefunktionen ‚echo', ‚cat' und ‚read' hat man nahezu die Funktionalität einer Programmiersprache.

Bedingungen testen

Fast alle Ablaufstrukturen brauchen die Möglichkeit zum Testen von Bedingungen. Dies kann auf zwei verschiedene Arten bewerkstelligt werden. Entweder man gebraucht das Kommando ‚test', oder man schließt die Bedingung in eckige Klammern ein. Dies funktioniert jedoch nur in einem bash Shellskript. Dabei muss vor und nach jeder Klammer ein Leerzeichen stehen.

`test` *`Bedingung`*

`[` *`Bedingung`* `]`

Bis auf ihre unterschiedliche Schreibweise agieren jedoch beide Arten gleich. Die angegebene Bedingung wird überprüft und je nach Ergebnis wird ein unterschiedlicher Wert zurückgeliefert. Trifft die Bedingung zu, liefert der Aufruf ‚true' (0), sonst ‚false' (1). Wird die Bedingung fehlerhaft angegeben (Syntax-Fehler), so wird der Wert 2 zurückgegeben.

Die Bedingung erlaubt den Vergleich von Dateien, Zeichenketten und Zahlen (Integer). Das zu übergebende Argument (= die Bedingung) besteht aus einer Option und einem Operanden, der je nach gewünschter Operation eine Datei, eine
    ```

Variable oder ein Wert sein kann. Es ist auch möglich, mehrere Bedingungen miteinander logisch zu verknüpfen (*UND*, *ODER*, *NICHT*).

Die in der Bedingung zu übergebenden Optionen hängen von der gewünschten Operation (Einsatzgebiet) ab. Im Folgenden sind die einzelnen Optionen angeführt, gegliedert nach Einsatzgebiet.

Testoptionen auf Dateien: Diese Optionen können nur auf Dateien angewendet werden. Existiert die Datei nicht, wird automatisch ein *false* zurückgeliefert.

Syntax: „`test` *`option dateiname`*“

-b	Prüft ob es sich bei der Datei um eine blockorientierte (Geräte)Datei handelt.
-c	Prüft, ob es sich bei der Datei um eine zeichenorientierte (Geräte)Datei handelt.
-d	Prüft, ob es sich bei der Datei um ein Verzeichnis handelt.
-e	Prüft, ob die angegebene Datei im Verzeichnissystem existiert.
-f	Prüft, ob es sich bei der Datei um eine reguläre (normale) Datei handelt.
-g	Prüft, ob das SGID-Bit auf die Datei gesetzt ist.
-k	Prüft, ob das sticky-Bit auf die Datei gesetzt ist.
-L	Prüft, ob es sich bei der Datei um einen symbolischen Link handelt.
-p	Prüft, ob es sich bei der Datei um eine benannte Pipe handelt.
-r	Prüft, ob die Datei vom aktuellen Benutzer gelesen werden kann.
-s	Prüft, ob die Datei nicht leer ist, d.h. ob sie über einen Inhalt verfügt.
-S	Prüft, ob es sich bei der Datei um eine Socket-Datei handelt.
-u	Prüft, ob das SUID-Bit auf die Datei gesetzt ist.
-w	Prüft, ob die Datei vom aktuellen Benutzer geschrieben werden kann.
-x	Prüft, ob die Datei das Ausführrecht besitzt.
-O	Prüft, ob der Aufrufer auch gleichzeitig der Eigentümer der Datei ist.
-G	Prüft, ob der Aufrufer der zugehörigen Gruppe der Datei angehört.

Optionen für Dateivergleiche: Diese Optionen erlauben den Vergleich zweier Dateien.

Syntax: „`test` *`dateiname1 option dateiname2`*”

-nt	newer than: Prüft, ob *dateiname1* neuer ist als *dateiname2*. Für den Vergleich wird das letzte Änderungsdatum hergenommen, nicht das Erstellungsdatum!
-ot	older than: Prüft, ob *dateiname1* älter ist als *dateiname2*. Für den Vergleich wird das letzte Änderungsdatum hergenommen, nicht das Erstellungsdatum!
-ef	Prüft, ob die beiden Dateien dieselbe Inode-Nummer haben (Hardlinks).

Optionen für Zeichenketten: Diese Optionen erlauben Prüfungen von Zeichenketten.

Syntax: „`test` *`option zeichenkette`*"

-z	Liefert ‚true' wenn die Zeichenkette null Zeichen lang (leer) ist.
-n	Liefert ‚true' wenn die Zeichenkette zumindest ein Zeichen enthält (nicht leer ist).

Optionen für den Vergleich von Zeichenketten: Diese Optionen erlauben den Vergleich zweier Zeichenketten.

Syntax: „`test` *`zeichenkette1 option zeichenkette2`*"

=	Liefert ‚true' wenn die beiden Zeichenketten gleich sind.
!=	Liefert ‚true' wenn die beiden Zeichenketten unterschiedlich sind.

Optionen für den Vergleich von Zahlenwerten: Diese Optionen erlauben den Vergleich zweier ganzen Zahlen (Integer).

Syntax: „`test` *`zahl1 option zahl2`*"

-eq	equals: Liefert ‚true' falls beide Zahlen den gleichen Wert haben.
-ne	not equals: Liefert ‚true' falls beide Zahlen einen unterschiedlichen Wert haben.
-lt	less than: Liefert ‚true' falls *zahl1* kleiner als *zahl2* ist.
-le	less or equal than: Liefert ‚true' falls *zahl1* kleiner oder gleich *zahl2* ist.
-gt	greater than: Liefert ‚true' falls *zahl1* größer als *zahl2* ist.
-ge	greater or equal than: Liefert ‚true' falls *zahl1* größer oder gleich *zahl2* ist.

Logisches Verknüpfen von Bedingungen: Mit Hilfe der folgenden Optionen lassen sich einzelne Bedingungen logisch miteinander verbinden.

! *bedingung*	Negieren der Bedingung: Liefert ‚true' wenn die Bedingung selbst ‚false' liefert und umgekehrt.
bedingung1 –a *bedingung 2*	Logisches UND: Liefert ‚true' wenn beide Bedingungen ‚true' ergeben. Falls eine der beiden Bedingungen ‚false' ergibt, liefert das logische UND auch ‚false'.
bedingung1 –o *bedingung2*	Logisches ODER: Liefert ‚true' wenn eine der beiden Bedingungen ‚true' ergibt oder wenn beide ‚true' ergeben. ‚false' wird nur dann geliefert, falls beide Bedingungen ‚false' ergeben.

Bedingte Anweisung (if – then – else)

Mit Hilfe der bedingten Anweisungen ist es möglich, gewisse Kommandos nur dann auszuführen, wenn eine zuvor angegebene Bedingung den Wert ‚wahr' (=‚true') ergibt. Optional kann definiert werden, welche Kommandos ausgeführt werden sollen falls die Bedingung nicht erfüllt ist oder falls eine andere Bedingung erfüllt ist.

Das Konzept der bedingten Anweisungen ist auch in jeder Programmiersprache zu finden und wird meist als if-Anweisung bezeichnet.

Bedingte Anweisungen beginnen mit dem Wort ‚if'. Jede weitere Anweisung muss entweder in einer eigenen Zeile stehen oder durch einen Strichpunkt von den anderen Anweisungen getrennt werden. Trotzdem verhält sich eine bedingte Anweisung wie eine einzige Anweisung. Erst wenn die if-Anweisung beendet wird (mit ‚fi'), wird die Anweisung von der Shell ausgewertet und bearbeitet.

Wenn-Dann Anweisung:

```
if bedingung
then
  kommandofolge
fi
```

Die einfachste if-Anweisung ist die sogenannte Wenn-Dann Anweisung. Hier wird die *Kommandofolge* dann ausgeführt, wenn eine bestimmte *Bedingung* erfüllt ist. Ist die Bedingung nicht erfüllt, wird die *Kommandofolge* übersprungen.

Nach dem ‚if' folgt eine Bedingung. Hier kann der vorher vorgestellte test-Befehl hergenommen werden. Intern erwartet UNIX jedoch den Wert ‚true' (0) für Bedingung erfüllt oder den Wert ‚false' (1) für Bedingung nicht erfüllt. Da jedes Kommando mit einem Errorcode terminiert (0 für fehlerfrei, anderer Wert zeigt Fehler während Bearbeitung an), kann hier aber auch eine beliebige Folge von Kommandos angegeben werden.

Das Wort ‚then' muss entweder in einer eigenen Zeile stehen oder durch einen Strichpunkt von der Bedingung getrennt sein.

Danach folgt eine Liste von Kommandos, die nur dann abgearbeitet werden, falls die Bedingung ‚true' ergeben hat. Dabei muss jedes Kommando in einer eigenen Zeile stehen oder man verwendet das Kommando-Trennzeichen „;' zwischen den einzelnen Kommandos.

Das Wort ‚fi' definiert das Ende der if-Anweisung und verlässt auch den Kommandoblock. Alle Kommandos, die nach diesem Wort angegeben wurden, werden wieder unabhängig vom Ergebnis der Bedingung ausgeführt.

Das if-Konstrukt ist auch von herkömmlichen Programmiersprachen bekannt. Anders als bei diesen befindet man sich jedoch solange im bedingten Anweisungsteil, solange dieser nicht geschlossen wird (z.B. durch ‚fi' (Ende der if-Anweisung, ‚else' oder ‚elif' für alternative Pfade).

Wenn-Dann-Sonst Anweisung:

```
if bedingung
then
  kommandofolge1
else
  kommandofolge2
fi
```

Bei der Wenn-Dann-Sonst Anweisung wird spezifiziert, was auszuführen ist wenn die Bedingung erfüllt ist und was auszuführen ist, wenn die Bedingung nicht erfüllt ist. Die *Kommandofolge1* wird nur dann ausgeführt, wenn die *Bedingung* erfüllt ist. Ist die *Bedingung* nicht erfüllt (und nur dann), wird *Kommandofolge2* ausgeführt. Anweisungen die ausgeführt werden sollen, falls die Bedingung nicht erfüllt ist, werden durch das Wort ‚else' eingeleitet. In den darauf folgenden Zeilen werden – solange bis das Wort ‚fi' gefunden wird – diese Anweisungen dann spezifiziert.

Mehrstufige Wenn-Dann-Sonst Anweisung:

```
if bedingung1
then
  kommandofolge1
elif bedingung2
then
  kommandofolge2
elif bedingung3
  ...
else
  kommandofolge_x
fi
```

Bei der mehrstufigen Wenn-Dann-Sonst Anweisung ist es möglich, mehrere verschiedene Bedingungen in einer Anweisung zu verpacken. Hierbei wird die *Kommandofolge1* nur dann ausgeführt, wenn *Bedingung1* ‚true' ergibt. Falls diese jedoch ‚false' ergibt (und nur dann), wird geprüft, ob *Bedingung2* ‚true' ergibt. Wenn dies der Fall ist (und nur dann), so wird *Kommandofolge2* ausgeführt. Dieses Spiel kann beliebig oft wiederholt werden. Trifft keine dieser weiteren Bedingungen zu, so wird die Kommandofolge nach dem Wort ‚else' abgearbeitet. Die Angabe des else-Zweigs ist jedoch nicht zwingend erforderlich.

! Es ist wichtig zu verstehen, dass nur in einen der angegebenen Wenn-Dann Zweige verwiesen werden kann. Trifft *Bedingung1* zu, wird gar nicht mehr geprüft ob auch *Bedingung2* zuträfe, es werden nur die Kommandos zwischen dem Wort ‚then' und dem nächsten Vorkommen des Wortes ‚elif' oder ‚else' ausgeführt.



Beispiel

Das folgende Beispiel zeigt ein Skript namens ‚explore'. Dem Skript wird ein Dateiname übergeben. Handelt es sich bei der Datei um eine normale Datei, wird deren Inhalt ausgegeben. Handelt es sich bei der Datei um ein Verzeichnis, so wird dessen Inhalt mit dem Kommando ‚ls' präsentiert. Wird keine Datei als Parameter angegeben, dann wird interaktiv nach einem Dateinamen gefragt.

```
#!/bin/bash

if test $# -eq 0
then
  echo -n "Bitte Namen eingeben: "
  read DATEINAME
else
  DATEINAME=$1
fi

if test -f $DATEINAME
then
  less $DATEINAME
elif test -d $DATEINAME
then
  ls -CF $DATEINAME
else
  echo "Datei kann nicht \
    angezeigt werden."
fi
```

Wird nur ausgeführt, falls kein Parameter übergeben wurde (test $# -eq 0)

Wird nur ausgeführt, falls angegebene Datei eine reguläre Datei ist (test –f dateiname)

Wird nur ausgeführt, falls angegebene Datei ein Verzeichnis ist (test –d dateiname)

Wird nur ausgeführt, falls angegebene Datei weder eine Datei noch ein Verzeichnis ist (keine der beiden obigen Bedinungen hat ‚true' ergeben)

Ablaufstrukturen$_2$

- Case-Anweisung
  - `case` *`selector`* `in`
    `Muster1)` *`Kommandofolge1`* `;;`
    `.....`
    `MusterN)` *`Kommandofolgen`* `;;`
    `esac`
  - Übereinstimmung selector-Muster wird gesucht, entspr. Kommandofolge ausgeführt und mit der Ausführung nach dem case-Konstrukt fortgefahren
  - Metazeichen sind in Muster erlaubt, im selector nicht
  - * ist default Muster, muss als Letztes stehen
  - Das Zeichen ‚|' bildet eine Oder-Bedingung

Mehrfachauswahl (case-Anweisung)

Stehen mehrere Zweige zur Verfügung, kann anstatt einer langen, mehrstufigen Wenn-Dann-Sonst Anweisung auch von der case-Anweisung Gebrauch gemacht werden. Bei dieser Auswahl findet ein Mustervergleich (pattern matching) statt. Passt das Vergleichsobjekt auf das Muster, so wird in den nach dem Muster angegebenen Zweig verwiesen. Die Mehrfachauswahl erlaubt zudem die Verwendung von Metazeichen (‚*', ‚?', ‚[]') in den Mustern und mehrere Muster für eine Auswahl. Dabei wird aber immer nur in einen Zweig verwiesen. Dies ist derjenige, dessen Muster als erstes auf das Vergleichsobjekt passt.

Das Vergleichsobjekt (oder auch *selector* genannt) wird der Reihe nach mit den einzelnen Mustern verglichen. Stimmt das Muster mit dem Vergleichsobjekt überein, so wird die nachfolgende Kommandofolge ausgeführt. Die Kommandofolge selbst wird durch doppelte Semikolons („;;') abgeschlossen. Ist die Kommandofolge ausgeführt, so setzt die Shell die Ausführung nach der case-Anweisung, gekennzeichnet durch das Wort ‚esac', fort.

Das Muster ‚*' passt auf jedes Vergleichsobjekt und stellt somit eine Standardauswahl dar. Wird kein anderes passendes Muster gefunden, ‚*' passt immer! Da jedoch Muster für Muster durchsucht wird, muss das ‚*'-Muster als letztes angegeben werden.

Soll auf eine Auswahl mehr als ein Muster (zwei oder mehrere) zutreffen, so sind die einzelnen Muster durch ein ‚|' zu trennen. Dieses Zeichen steht für eine Oder-Bedingung.

Die Mehrfachauswahl hat folgende Syntax:

```
case selector in
  muster1) Kommandofolge1 ;;
  muster2) Kommandofolge2 ;;
  ....
  musterN) KommandofolgeN ;;
esac
```



Beispiel

Das folgende Beispiel präsentiert ein Menü am Bildschirm. Der Benutzer hat die Möglichkeit, eine der angebotenen Funktionen auszuwählen. Die Eingabe wird ausgewertet und das entsprechende Programm gestartet.

```
#!/bin/bash

clear                          # Löscht alles am Bildschirm
echo " 0 -> Ende "
echo " 1 -> Datum und Uhrzeit "
echo " 2 -> aktuelles Verzeichnis "
echo " 3 -> Inhaltsverzeichnis "
echo –n "Ihre Eingabe: "
read EINGABE
case $EINGABE in
  0) exit ;;                   # Falls Eingabe 0 war
  1) date ;;                   # Falls Eingabe 1 war
  2) pwd ;;                    # Falls Eingabe 2 war
  3) ls –l ;;                  # Falls Eingabe 3 war
  *) echo " Falsche Eingabe " ;;     # sonst
esac
```

Ablaufstrukturen$_3$

- For-Anweisung
  - `for` *`selector`* `in` *`liste`*
    `do`
    `kommandofolge`
    `done`
  - `for` *`selector`*
    `do`
    `kommandofolge`
    `done`

Im Folgenden werden nun die verschiedenen Schleifenvarianten erläutert. Schleifen erlauben die wiederholte Ausführung gewisser Kommandos.

For-Schleife

Mit Hilfe der for-Schleife kann man eine Anweisungsfolge in der Shell für unterschiedliche Belegungen einer Variablen ausführen.

Aufruf mit Werten

```
for selector in value1 value2 … value_n
do
    kommandofolge
done
```

Im Kopf der Schleife wird eine Variable (*selector*) angegeben und eine Liste von Werten (*value1 – value$_n$*) die der Variablen nacheinander zugeordnet werden. Beim ersten Schleifendurchlauf wird der erste Wert zugewiesen, beim zweiten Durchlauf der zweite Wert usw. bis die Schleife mit allen Werten durchlaufen worden ist. Dabei ist es möglich, Werte verschiedenen Typs (z.B. Integer und Zeichenketten aber auch Dateinamen) zu kombinieren. Auf den aktuellen Wert kann natürlich per Referenz zugegriffen werden.

Im weiteren Verlauf wird nun die for-Schleife anhand eines Beispieles vorgestellt. Dabei werden im Kopf Werte verschiedenen Typs zugewiesen. Da in der Shell alle Werte als Zeichenketten interpretiert werden, stellt dies kein Problem dar. Die zugewiesenen Werte werden dann innerhalb der Schleife ausgegeben.

Beispiel

```
for i in 1 2 ja nein "lange Zeichenkette"
do
   echo $i
done
```

Ausgabe:

```
1
2
ja
nein
lange Zeichenkette
```

Die Eigenschaft, dass unterschiedliche Werte möglich sind, erlaubt auch die Verwendung von Metazeichen und Dateinamen. Metazeichen werden von der Shell durch passende Dateinamen ersetzt (z.B. ‚*' liefert alle nicht-versteckten Dateien im Verzeichnis).

Das folgende Skript betrachtet den Inhalt aller Dateien des aktuellen Verzeichnisses mit dem cat-Kommando:

```
for file in *
   cat $file
done
```

Aufruf ohne Werte

```
for selector
do
   kommandofolge
done
```

Wird keine Werteliste angegeben, so nimmt die Shell als Werteliste die übergebenen Parameter. Jeder Parameter wird als Zeichenkette interpretiert und der Schleifen-Variable übergeben. Wird kein Parameter übergeben, wird die Schleife niemals durchlaufen.

! Mit Hilfe dieser Form von for-Schleifen kann sehr gut der Metazeichen-Ersetzungsmechanismus veranschaulicht werden. Wird das Skript

```
for PARAM
do
   echo $PARAM
done
```

mit dem Parameter ‚*' aufgerufen, dann ersetzt die Shell den ‚*' vor Ausführung des Skripts durch alle im Verzeichnis befindlichen Dateien. Dann erst wird das Kommando mit einer Liste von Dateinamen (durch die Shell ersetzt) aufgerufen.

Ablaufstrukturen$_4$

- Wiederholungsanweisung
  - `while` *`Bedingung`* *`#(Solange)`*
    `do`
    `Kommandofolge`
    `done`
  - `until` *`Bedingung`* *`#(Bis)`*
    `do`
    `Kommandofolge`
    `done`

Neben der for-Schleife gibt es noch zwei andere Schleifen, die auch von diversen Programmiersprachen bekannt sind.

Abweisende Schleife

Die abweisende Schleife (while-Schleife) dient dazu, eine Anweisung zu wiederholen, solange eine angegebene Bedingung gilt. Eine while-Schleife ist dabei ein abweisendes Konstrukt, da der Schleifenrumpf niemals betreten wird, wenn die Bedingung gleich beim ersten Mal falsch ist.

```
while Bedingung
do
    Kommandofolge
done
```

Ergibt die im Kopf angegebene Bedingung den Wert ‚true' (oder Exit-Status ‚0'), dann wird die Schleife durchlaufen, solange bis die Bedingung einen anderen Wert (‚false' oder einen Wert ungleich ‚0') liefert. Ist es garantiert, dass der Wert immer ‚true' oder ‚0' liefert, so wird die Schleife endlos lange wiederholt (Endlosschleife) und kann nur durch das ‚break'-Kommando abgebrochen werden. Dieses Kommando wird weiter unten in diesem Kapitel behandelt.



Beispiel

Im folgenden Beispiel wird gezeigt, wie man mittels einer Schleife eine beliebige Anzahl von Parametern durchlaufen kann.

```
while test $# -ne 0     # Solange Parameteranzahl nicht 0 ist
do
   shift                # 1 Parameter löschen, Parameter „aufrücken“
done
```

Bedingte Schleife

Im Gegensatz zur while-Schleife wird die bedingte Schleife (until-Schleife) solange ausgeführt, bis die angegebene Bedingung erfüllt ist – also ‚true' oder den Wert ‚0' ergibt. Da auch hier die Bedingung im Kopf steht, kann es möglich sein, dass die Schleife selbst nie betreten wird (abweisendes Konstrukt).

```
until Bedingung
do
   Kommandofolge
done
```



Beispiel

Im folgenden Beispiel wird gezeigt, wie man mittels einer until-Schleife eine beliebige Anzahl von Parametern durchlaufen kann.

```
until test $# -eq 0     # Bis Parameteranzahl 0 ist gehe in Schleife
do
   shift                # 1 Parameter löschen, Parameter „aufrücken“
done
```

Ablaufstrukturen$_5$

- `select` Menügesteuerte Verzweigung
- `exit` (wie in Shell)
- `break [n]` Verlassen von n Schleifen
  - Bsp: *break 1* bezieht sich auf die erste umschließende Schleife
- `continue [n]` Beginn mit neuem Durchlauf in der n-ten umfassenden Schleife
  - Bspl: *`continue 1`* bezieht sich auf die erste umschließende Schleife

Select Anweisung

Die Select-Anweisung bietet eine Kombination aus menügesteuerter Verzweigung und Schleife. Die Wortliste wird als nummerierte Liste auf dem Standardfehlerkanal ausgegeben. Mit dem PS3-Prompt wird daraufhin die Eingabe von der Tastatur angefordert. Eine leere Eingabe führt zu einer neuen Anzeige des Menüs.

```
select VAR in Wortliste
do
    Kommandofolge
done
```

Zuerst wird die Auswahl auf den Standardfehlerkanal repräsentiert. Jeder Wert in der Wortliste kann nun durch die Eingabe einer Zahl ausgewählt werden. Dabei wird jedem Wert eine Zahl (aufsteigend beginnend mit 1) zugeordnet. Wird diese Zahl gedrückt, so beinhaltet die Variable ‚REPLY' die Eingabe, die im Statement angegebene Variable (in unserem Falle ‚*VAR*') jedoch den dazugehörigen Wert. Wird keine passende Zahl übergeben, so ist auch die angegebene Variable leer.

Das Menü wird solange bearbeitet, bis die Schleife mit der ‚break'- oder ‚return'-Anweisung abgebrochen wird. Es ist auch möglich, mit <Strg>+<D> das Menü sofort zu verlassen. Wird keine Wortliste angegeben, so werden die Werte der Parameter $0 bis $9 verwendet.



Beispiel

Das folgende Beispiel repräsentiert eine Liste und gibt zu jeder gedrückten Option die Nummer und den Wert aus.

```
select EING in eins zwei drei ende
do
   echo –n "Eingabe: "
   echo $EING
   echo –n "Optionswert: "
   echo $REPLY
   if test $EING = "ende"
   then
      break
   fi
done
```

Exit Anweisung

Wie schon bei der Shell kann auch das Shellskript mit „exit“ abgebrochen werden. Vom Terminal aus kann ein Skript aber auch mit der <Entf>-Taste abgebrochen werden, sofern das Signal nicht abgefangen wird.

Break Anweisung

Die break-Anweisung dient zum Verlassen einer Schleife. Die Schleifenausführung wird dabei unterbrochen und es werden die Kommandos nach der Schleife ausgeführt. Da Schleifen ineinander geschachtelt werden können, kann optional ein Wert eingegeben werden, der die jeweilige Schleife spezifiziert. Der Zählvorgang der Schleifen beginnt dabei bei der innersten Schleife (Wert 1).

Continue Anweisung

Soll eine Schleife nicht abgebrochen werden, aber darauf folgende Kommandos nicht mehr ausgeführt werden sondern sofort der nächste Schleifendurchlauf eingeleitet werden, so ist dies mit der Anweisung „continue“ möglich. Wie bei der break-Anweisung kann optional die Schleife durch eine Zahl identifiziert werden (1 ist dabei die innerste Schleife).

Weitere Anweisungen

Weitere Anweisungen

- `expr`-Anweisung erlaubt das Auswerten arithmetischer Ausdrücke
  - Operatoren und Argumente müssen dabei durch Leerzeichen getrennt werden
  - Drei verschiedene Kategorien von Operatoren:
    - Arithmetische
    - Relationale
    - Logische
- `trap`-Anweisung
  - Fängt angegebene Signale ab und führt die angegebenen Kommandos aus
  - `trap kommandosliste signale`

Expr (Ausdrücke auswerten)

Die ‚expr'-Anweisung erlaubt das Auswerten von arithmetischen Ausdrücken, die als Argumente übergeben werden. Argumente und Operatoren müssen dabei durch Leerzeichen getrennt angegeben werden. In den meisten Fällen sind Argumente Integer-Zahlen, die dann mittels Operatoren mathematisch bearbeitet werden. Dabei gibt es drei verschiedene Kategorien von Operatoren: arithmetische, relationale und logische.

Das Ergebnis der Operation wird auf die Standardausgabe geschrieben. Auch hier gilt Punkt vor Strichrechnung und das Beachten von Klammern. Da die Klammern und der Stern auch von der Shell verwendet werden, müssen diese Operationszeichen immer durch einen vorangehenden Backslash (‚\') geschützt werden. Damit die Operatoren von der Shell erkannt werden, müssen sie von Leerzeichen umgeben sein.

`expr` *`arg1 operator arg2`* `[` *`operator arg3`* `... ]`

In weiterer Folge sind nun Listen mit den verschiedenen Operatoren verzeichnet.

Arithmetische Operatoren

Diese Operatoren werden zu mathematischen Berechnungen genutzt. Dabei wird der Operator zwischen den beiden Argumenten angegeben.

+	Addiert *arg1* zu *arg2*.
-	Subtrahiert *arg2* von *arg1*.

*	Multipliziert *arg1* mit *arg2*.
/	Dividiert *arg1* durch *arg2*.
%	Gibt den Rest zurück, der bei der Division von *arg1* und *arg2* entstanden ist.

Relationale Operatoren

Diese Operatoren werden zu Vergleichsoperationen benutzt. Das Ergebnis kann entweder ‚true' oder ‚false' ergeben.

=	Liefert den Wert ‚true' falls beide Argumente gleich sind.
==	Liefert den Wert ‚true' falls beide Argumente gleich sind (wie ‚=')
!=	Liefert den Wert ‚true' falls beide Argumente unterschiedlich sind
>	Liefert den Wert ‚true' falls *arg1* größer ist als *arg2*.
>=	Liefert den Wert ‚true' falls *arg1* größer oder gleich *arg2* ist.
<	Liefert den Wert ‚true' falls *arg1* kleiner ist als *arg2*.
<=	Liefert den Wert ‚true' falls *arg1* kleiner oder gleich *arg2* ist.

Logische Operatoren

Diese Operatoren werden zum logischen Vergleich zweier Argumente benutzt. Je nach Operation kann das Ergebnis *arg1*, *arg2*, oder 0 ergeben.

|	Logisches Oder: Wenn *arg1* einen Wert ungleich 0 besitzt ist das Ergebnis *arg1*, sonst *arg2*.
&	Logisches Und: Wenn *arg1* und *arg2* einen Wert ungleich 0 besitzen, ist das Ergebnis *arg1*, sonst 0.

Schlüsselwörter

Mit Hilfe von Schlüsselwörtern können spezielle Funktionen und Operationen durchgeführt werden.

index *zeichenkette character-list*	Liefert die Position des ersten Vorkommen eines Zeichens aus der *character-list* in der *zeichenkette*. Falls keine Übereinstimmung gefunden wird, liefet das Kommando den Wert 0.
length *zeichenkette*	Liefert die Länge der angegebenen *zeichenkette*.
quote *token*	Anweisung an das expr-Kommando das nachfolgende *token* als eine Zeichenkette zu interpretieren, selbst wenn diese normalerweise ein Schlüsselwort oder ein Operator wäre.
substr *zeichenkette start länge*	Liefert einen Teil einer *zeichenkette*, beginnend bei *start* mit einer maximalen Länge von *länge* Zeichen. Ist diese Zeichenkette nicht lieferbar (falsche Argumente), so liefert substr den Wert null.



Beispiel

Mittels der while-Schleife ist es möglich, in der Shell einen primitiven Zähler zu implementieren (Zähl-Schleife). Dazu wird jedoch der Befehl ‚expr' benötigt, der mittels einer arithmetischen Operation die Schleifendurchläufe zählt. An der Ablaufgeschwindigkeit und der Art und Weise der Implementierung sieht man jedoch deutlich die Grenzen der Leistungsfähigkeit.

```
ZAEHLER=0
while test $ZAEHLER –le 10
do
   ZAEHLER=`expr $ZAEHLER + 1`
   echo –n $ZAEHLER
done
```

Abfangen von Signalen (trap)

In UNIX können jedem Prozess verschiedene Signale gesendet werden. Signale sind eine Möglichkeit, wie verschiedene Prozesse miteinander kommunizieren können. Dabei sendet ein Prozess einem anderen ein Signal, wobei das Betriebssystem (eigentlich der Kernel) für die Übermittlung des Signals sorgt. Der Empfängerprozess reagiert dann auf das Signal, normalerweise dadurch dass er terminiert.

Für Shellskripte gibt es die Möglichkeit, auf verschiedene Signale zu reagieren bzw. diese zu ignorieren. Das ist beispielsweise nützlich, wenn ein Shellskript nicht durch den Benutzer von der Tastatur aus abgebrochen werden soll, oder aber wenn es eine Möglichkeit geben soll, dem Skript mitzuteilen, auf ein Signal von außen zu reagieren.



Manche Shellprogramme speichern ihre Einstellungen in Konfigurationsdateien, die auch von außen verändert werden können. Nach einer solchen Veränderung könnte man an den Prozess ein bestimmtes Signal schicken, damit dieser die Konfigurationsdatei neu einliest und entsprechend den Änderungen arbeitet.

Das Reagieren oder Ignorieren von solchen Signalen kann mit dem Kommando ‚trap' realisiert werden.

```
trap "kommandoliste" signale
```

Der trap-Anweisung wird eine *Kommandoliste* und eine Signalliste (*signale*) übergeben. Wird dem Prozess eines der Signale übermittelt, so unterbricht der Prozess seine Ausführung und bearbeitet die in der *Kommandoliste* angegebenen Anweisungen. Nachdem diese Anweisungen bearbeitet wurden, setzt der Prozess die Bearbeitung an der Stelle fort, an der er unterbrochen wurde. Soll das Signal ignoriert werden, so ist als *Kommandoliste* eine leere Zeichenkette zu übergeben.

Es gibt unterschiedliche Signale mit unterschiedlicher Bedeutung. Normalerweise werden diese vom Betriebssystem generiert und an einen Prozess gesendet, es

besteht aber auch die Möglichkeit, solche Signale zu generieren und an einen Prozess zu übermitteln (Kommando ‚kill').

Nr	Name	Aktion	Beschreibung
1	SIGHUP	exit	Leitung zum Terminal wird unterbrochen
2	SIGINT	exit	Interrupt, Unterbrechung am Terminal
3	SIGQUIT	exit	Abbruch am Terminal, Quit
4	SIGILL	core	Illegale Anweisung im Prozess
5	SIGTRAP	core	Trace Point gesetzt (bei Debugging von Prozessen)
6	SIGABRT	core	Abbruch des Prozesses
8	SIGFPE	core	Gleitkomma-Fehler
9	SIGKILL	exit	Beenden des Prozesses durch Abbruchsignal
10	SIGBUS	core	Fehler am Bus-System
11	SIGSEGV	core	Fehler beim Speicherzugriff
12	SIGSYS	core	Fehler beim Systemaufruf
13	SIGPIPE	exit	Fehler bei einer Pipe-Bearbeitung
14	SIGALARM	exit	Ablauf eines Zeitintervalls
15	SIGTERM	exit	Abbruchsignal durch Software
16	SIGUSR1	exit	Benutzerdefiniertes Signal
17	SIGUSR2	exit	Benutzerdefiniertes Signal
18	SIGCLD	ignore	Ende eines Sohn-Prozesses
19	SIGPWR	ignore	Spannungsausfall (von USV verwendet)
20	SIGWINCH	ignore	Änderung der Fenstergröße
21	SIGURG	ignore	Meldung von Socket
22	SIGPOLL	ignore	Abfragbares Ereignis
23	SIGSTOP	stop	Prozess wurde angehalten
24	SIGSTP	stop	Prozesshalt, von Benutzer
25	SIGCONT	ignore	Gestoppter Prozess wird fortgesetzt
26	SIGTTIN	stop	Hintergrundprozess wartet auf Eingabe
27	SIGTTOU	stop	Hintergrundprozess schreibt Ausgabe
28	SIGVTALARM	exit	Virtueller Timer abgelaufen
29	SIGPROF	exit	Prozessüberwachungstimer abgelaufen
30	SIGXCPU	core	CPU-Zeit abgelaufen
31	SIGXFSZ	core	Maximale Dateigröße überschritten
32	SIGIO		Socket I/O möglich

Sollen mehrere Signale die gleiche Kommandoliste bearbeiten (oder ignoriert werden) können diese – getrennt durch ein Zeichen aus der Internal Field Separator (IFS) Variablen – in einer Liste angegeben werden.



Beispiele

Im Folgenden wird anhand von Beispielen die Verwendung des trap-Kommandos erklärt.

```
trap "" 2          # Signal 2 ignorieren
...                # Weitere Kommandos, Singal 2 wird ignoriert
trap 2             # Signal 2 wieder beachten
```

```
trap "rm –f $tmp; exit" 0 1 2 15   # Wenn Signal 0, 1, 2 oder 15 kommt, temporäre Datei löschen und Beenden
```

Senden von Signalen an Prozesse

Signale können manuell generiert werden und an Prozesse gesendet werden. Dies geschieht durch die Verwendung des Kommandos ‚kill'.

`kill` *`signal prozess`*

Das kill-Kommando erwartet hierbei die Angabe eines Signals (dieselben Signale wie oben angeführt). Dann wird eine Prozess-ID erwartet, die angibt, an welchen Prozess das Signal gesendet werden soll. Nähere Informationen über Prozesse und Prozess-IDs sind im Kapitel „Prozesse“ angeführt.

Kapitel

8

Reguläre Ausdrücke

Dieses Kapitel befasst sich mit der Möglichkeit von UNIX nach Zeichenketten mit Hilfe von regulären Ausdrücken zu suchen.

Reguläre Ausdrücke sind eine Möglichkeit, in Dateien oder bestehenden Daten Übereinstimmungen von Suchbegriffen zu finden. Diese Ausdrücke bestehen definieren ein Muster, nach dem dann gesucht wird. Um die Flexibilität während der Suche zu gewährleisten, erlauben reguläre Ausdrücke die Verwendung von Metazeichen. Dabei sind Metazeichen zumeist Platzhalter, geben aber auch erweiterte Regeln wie zum Beispiel Wiederholungen von Mustern an.

Die in diesem Kapitel vorgestellten regulären Ausdrücke sind nur mit Hilfe der GNU-Version des grep-Kommandos veranschaulicht. Da die Spezifikation der regulären Ausdrücke nicht standardisiert ist, kann es vorkommen das manche hier angegebenen Ausdrücke auf anderen UNIX-Derivaten nicht funktionieren.

Lehrinhalte und -ziele

Am Ende dieses Kapitels sollten Sie mit regulären Ausdrücken vertraut sein. Sie sollten in der Lage sein, in einer Datei nach Übereinstimmungen eines Musters suchen zu können. Diese Muster sollten Sie mit regulären Ausdrücken angeben können. Sie sollten auch in der Lage sein, die einzelnen Metazeichen zu verwenden und sinnvoll zu kombinieren.

Einführung in reguläre Ausdrücke

Allgemeines

- Reguläre Ausdrücke sind Suchmuster
  - Sucht nach Übereinstimmungen in einer Datei oder in einer Eingabe
  - Kombination aus herkömmlichen Zeichen und Metazeichen
- Verwendung
  - grep-Kommando (global regular expression print)
    - Ausgabe von Zeilen einer Datei, die Suchmuster enthalten
    - *grep regulärerAusdruck [Optionen] [Datei(en)]*
    - Auf Shell-Ersetzung achten → einfache Hochkomma setzen
  - andere
    - Abwandlungen von grep (egrep, fgrep,...)

Ein regulärer Ausdruck ist ein Suchmuster, um übereinstimmende Zeichenketten in einer Eingabe oder einer Datei zu finden.

Reguläre Ausdrücke werden von verschiedenen Kommandos verwendet. Das wichtigste Kommando ist dabei ‚grep' und dient zum Suchen von Zeichenketten in einer Datei oder vom Eingabestrom der Standardeingabe.

Suchen von Zeichenketten (grep)

‚grep' steht für global regular expression print und ist ein Tool zum Suchen von Zeichenketten, die mit regulären Ausdrücken definiert sind. Dabei erwartet das grep-Kommando ein Suchmuster, das dann in den angegebenen Dateien gesucht wird. Die Zeilen der Dateien, die das Suchmuster enthalten, werden auf die Standardausgabe ausgegeben. Wird keine Datei angegeben, so sucht grep in der Standardeingabe.

Da das Kommando oftmals Sonderzeichen zur Angabe von Suchmustern verwendet, muss auf die Shellersetzung acht gegeben werden. Es empfiehlt sich, die erwarteten regulären Ausdrücke in Hochkomma zu setzen.

Ein regulärer Ausdruck ist nichts anderes als ein Suchmuster, um übereinstimmende Muster in einer Eingabe zu finden.

`grep regulärerAusdruck [Optionen] [Dateiname(n)]`

Das Kommando erwartet zuerst einen regulären Ausdruck, der das Suchmuster spezifiziert. Dann können verschiedene Optionen an das Kommando übergeben werden.

-q	quiet: Die Ausgabe wird unterdrückt, lediglich Rückgabewerte werden ausgegeben. Das Kommando stoppt bei der ersten Übereinstimmung.
-n	Zeilennummern werden ausgegeben
-c	count: Es wird die Anzahl der Zeilen ausgegeben, die dem regulären Ausdruck entspricht.
-l	Es werden nur die Namen der Dateien ausgegeben, in denen der reguläre Ausdruck gefunden wurde.
-i	Groß-/Kleinschreibung wird nicht unterschieden.
-v	Es werden alle Zeilen ausgegeben, in denen der reguläre Ausdruck *nicht* gefunden wurde.

Es gibt noch eine Reihe anderer Optionen, die aus den entsprechenden Online Hilfeseiten (man-Pages) zu entnehmen sind.

Im folgenden werden nun die regulären Ausdrücke, die das grep-Kommando für seine Funktionalität benötigt, näher erklärt.

! ‚grep' ist eines der Kommandos, die stark von Derivat zu Derivat variieren. Im folgenden wird die GNU-Version des grep-Kommandos behandelt. Diese Version ist auf den meisten der heutzutage erhältlichen Linux-Distributionen im Einsatz. Es darf jedoch nicht davon ausgegangen werden, dass die hier verzeichneten regulären Ausdrücke auch auf anderen Systemen ausgewertet werden können.

Reguläre Ausdrücke

Verwendung

- Eingabe des Musters mit Hilfe verschiedener Metazeichen
  - Metazeichen sind ähnlich zur Shell, aber nicht die gleichen!
- Oftmals bezieht sich Metazeichen nur auf vorangestelltes Zeichen oder die vorangestellte Gruppe
  - bugg\?y → ‚?' bezieht sich auf g
    Lösungen: bugy, buggy aber nicht bugggy!

Wie schon zuvor erwähnt, dienen reguläre Ausdrücke zur Spezifikation von Suchmustern, nach denen dann in den angegebenen Dateien oder von der Standardeingabe gesucht wird. Die Möglichkeiten zur Angabe von Suchmustern sind dabei sehr umfangreich. Reguläre Ausdrücke sind eine Kombination aus herkömmlichen Zeichen gepaart mit zahlreichen Metazeichen.

Mit Hilfe der regulären Ausdrücke können Suchmuster spezifiziert werden. Die Möglichkeiten sind dabei sehr umfangreich. Dabei wird ein regulärer Ausdruck durch Metazeichen und einem Suchmuster spezifiziert. Die in Frage kommenden Metazeichen sind nun in weiterer Folge aufgelistet. Jedes Metazeichen wird im Folienteil durch ein Beispiel erklärt.

Wichtig ist, das die hier aufgelisteten Metazeichen nur für reguläre Ausdrücke gelten und nicht mit den Metazeichen der Shell verwechselt werden dürfen.

Zeichenersetzung

Reguläre Ausdrücke$_1$

.	Genau ein beliebiges Zeichen
*	Vorangegangenes Zeichen kann beliebig oft wiederholt werden (0 oder mehrmals)

testfile

```
hallo unix,
das ist eine Testdatei.
hello user,
this is unix.
helo
heo
```

- grep 'h.llo' testfile
  → hallo unix,
  → hello user,
- grep 'hel*o' testfile
  → hello user,
  → helo
  → heo

Platzhalter für genau ein beliebiges Zeichen

Ein Punkt („.') steht in einem regulären Ausdruck für ein beliebiges Zeichen. Der Punkt markiert jedoch genau ein Zeichen, oder eine Position. Statt dem Punkt wird beim Vergleich jedes beliebige Zeichen eingesetzt.



Im oben gezeigten Beispiel wird der Ausdruck ‚h.llo' durch hallo und hello ersetzt, da der Punkt sowohl durch ein ‚a' als auch durch ein ‚e' ersetzt werden kann. Jedoch muss das ‚h' an erster Stelle stehen, und ‚llo' beginnend an dritter Position in der Übereinstimmung. Lediglich die zweite Stelle (gekennzeichnet durch den Punkt) ist beliebig.

Voriges Zeichen beliebig oft wiederholen

Ein Zeichen kann mit dem Stern (‚*') beliebig oft wiederholt werden. Dabei bezieht sich der Stern aber auf das vorige Zeichen (oder die vorige Zeichengruppe). Ein Stern hätte keine Bedeutung, es muss immer ein Zeichen vorangestellt sein.



Im oben gezeigten Beispiel passt der Ausdruck ‚hel*o' auf hello, helo und heo. Der ‚*' nach dem ‚l' zeigt an, dass das ‚l' beliebig oft wiederholt werden kann. Beliebig oft heißt auch, dass das Zeichen überhaupt nicht (also kein mal) vorkommen darf.

Zeichenauswahl

Reguläre Ausdrücke$_2$

[ ]	Eines der in der Klammer angegebenen Zeichen
[^]	Keines der in der Klammer angegebenen Zeichen

testfile

```
hallo unix,
das ist eine Testdatei.
hello user,
this is unix.
helo
heo
```

- grep 'h[ae]llo' testfile
  → hallo unix,
  → hello user,
- grep 'h[^e]llo' testfile
  → hallo unix,

Platzhalter für ein bestimmtes Zeichen

Wie schon oben erwähnt, steht der Punkt für irgendein beliebiges Zeichen. Manchmal ist es notwendig, die möglichen Zeichen zu beschränken. Dies kann mit der eckigen Klammer (‚[' ... ‚]') angegeben werden. Dabei werden in der eckigen Klammer all diejenigen Zeichen angegeben, die an der Stelle für den Vergleich hergenommen werden. Bereiche können mit ‚-' angegeben werden (z.B. [1-9]). Einzelne Zeichen werden *NICHT* durch Leerzeichen oder sonstige Zeichen voneinander getrennt!

Will man genau die Zeichen angeben, die nicht passen sollen, so ist gleich nach der Klammer ein ‚^' anzugeben (also [^ ...]). Beim Vergleich werden alle Zeichen bis auf die in der Klammer angegebenen hergenommen.

In dem in der Folie gezeigten Beispiel passt der Ausdruck ‚h[ae]llo' sowohl auf ‚hallo' als auch auf ‚hello'. An der zweiten Stelle muss also eines der in der Klammer angegebenen Zeichen (also ‚a' oder ‚e') vorkommen. Auf den Ausdruck ‚h[^e]llo' passt jedoch nur mehr ‚hallo', da sonst keine Übereinstimmung gefunden wird wo an erster Stelle ein ‚h', an zweiter Stelle ein beliebiges Zeichen außer ‚e', an dritter und vierter Stelle ein ‚l' und an fünfter Stelle ein ‚o' steht.

Sonstige Ausdrücke

Reguläre Ausdrücke$_3$

^	Muster steht am Anfang der Zeile
$	Muster steht am Ende der Zeile
\	Nachfolgendes Zeichen wird nicht ersetzt

testfile

```
hallo unix,
das ist eine Testdatei.
hello user,
this is unix.
helo
heo
```

- grep '^hello' testfile
  → hello user,
- grep 'unix.$' testfile
  → hallo unix,
  → this is unix.
- grep 'unix\.$' testfile
  → this is unix.

Muster am Anfang der Zeile suchen

Zeilenanfang und Zeilenende sind in regulären Ausdrücken durch ein jeweils spezielles Metazeichen gekennzeichnet. Soll der Ausdruck am Zeilenanfang stehen, so ist auch das Metazeichen, dass für den Zeilenanfang steht, anzugeben. Der Zeilenanfang wird durch das Zeichen ‚^' gekennzeichnet.

Im oben angeführten Beispiel passt auf den Ausdruck ‚^hello' die dritte Zeile der *testfile*-Datei, weil dort das Wort ‚hello' gleich zu Beginn der Zeile steht.

Muster am Ende der Zeile suchen

Das Zeilenende ist durch ein ‚$' markiert. Dieses muss natürlich auch an der Position vorkommen, an der sich das Zeilenende befindet (also ganz zum Schluss).

In dem oben angeführten Beispiel passen auf den Ausdruck ‚unix.$' die Zeilen ‚hallo unix,' und ‚this is unix.'. Das ist deswegen so, weil der Punkt ja für ein beliebiges Zeichen steht, also auch für einen Beistrich oder einem anderen Zeichen.

Nachfolgendes Zeichen nicht ersetzten

Durch die Angabe eines Backslashes (‚\') wird das darauffolgende Metazeichen als normales Zeichen zum Vergleich hergenommen.

Um nur die Zeile mit unix. zu bekommen, muss der reguläre Ausdruck folgendermaßen heißen: ‚unix\.$'. Der Backslash vor dem Punkt zeigt an, dass nun ein realer Punkt und nicht irgendein Zeichen erwartet wird.

Reguläre Ausdrücke$_4$

\(\)	Gruppieren von Zeichen zu einem Ausdruck
\+	Ein oder mehrere Vorkommnisse des vorhergehenden Ausdrucks
\?	0 oder ein Vorkommnis des vorhergehenden Ausdrucks
|	Entweder vorderer oder hinterer Ausdruck

testfile

```
hallo unix,
das ist eine Testdatei.
hello user,
this is unix.
helo
helelo
```

- grep 'h\(el\)*o' testfile
  → helo
  → helelo
- grep 'h\(el\)\+o' testfile
  → helo
  → helelo
- grep 'h\(el\)\?o' testfile
  → helo
- grep 'h\(e\|a\)llo' testfile
  → hallo unix,
  → hello user,

Gruppieren von regulären Ausdrücken

Manchmal ist es notwendig, Teile eines Ausdrucks zu gruppieren. Gerade wenn ein Teil wiederholt oder speziell behandelt werden soll, benötigt man Gruppen. Gruppen können durch die Angabe von ‚\(' bis zu ‚\)' angegeben werden. Alles was zwischen diesen Klammern steht wird zu einer Gruppe zusammengefasst.

Der Ausdruck ‚h\(el\)*o' gruppiert zuerst die Zeichen ‚el'. Die Angabe des ‚*' ermöglicht nun eine beliebige Wiederholung von ‚el' innerhalb des Ausdruckes. Würde keine Gruppe angegeben werden, so würde sich der ‚*' nur auf das letzte vor Ihm stehende Zeichen beziehen, also auf das ‚l'.

Vorkommnisse von Ausdrücken wiederholen

Gruppen können benutzt werden, um Ausdrücke zu wiederholen. Dabei gibt es verschiedene Varianten, die Ausdrücke verschieden oft wiederholen. Die beliebige Wiederholung (0 oder mehrere Male) durch das Zeichen ‚*' wurde bereits vorgestellt. ‚\+' wiederholt den vorhergehenden Ausdruck nur ein oder mehrere Male. ‚\?' ist hingegen die optionale Wiederholung (0 oder nur 1 Mal).

Der Ausdruck ‚h\(el\)\+o' gibt an, dass ‚el' ein oder mehrmals wiederholt werden kann. In unserem Beispiel werden die Zeilen mit ‚helo' (1x Wiederholung) und ‚helelo' (2x Wiederholung) ausgegeben.

Der Ausdruck ‚h\(el\)\?o' gibt an, dass ‚el' entweder vorkommt oder nicht. In unserem Beispiel wird nur mehr die Zeile ‚helo' (1x Wiederholung) ausgegeben, eine 0fache Wiederholung wäre das Wort ‚ho'.

Alternativen in Ausdrücken verwenden

Alternativen innerhalb von Ausdrücken können durch ‚\|' angegeben werden. Bei der Auswertung muss entweder das linke Zeichen (oder die linke Zeichengruppe) passen oder das rechte Zeichen (oder Zeichengruppe).



Der Ausdruck ‚h\(e\|a\)llo' gibt an, dass das erste Zeichen ein ‚h' sein muss. An zweiter Position kann nun entweder ein ‚e' oder ein ‚a' kommen. Danach muss die Zeichenkette ‚llo' (beginnend mit 3. Position) folgen. Auf dieses Muster passen die Zeilen die ‚hallo' und ‚hello' enthalten.

Reguläre Ausdrücke$_5$

\{n,m\}	Wiederholung des vorangehenden Musters zwischen *n* und *m* mal.
\<	Muster am Wortanfang suchen
\>	Muster am Wortende suchen

testfile

```
hallo unix,
das ist eine Testdatei.
hello user,
this is unix.
helo
helelo
```

- grep 'h\(el\)\{2,3\}' testfile
  → helelo user,
- grep '\<is' testfile
  → das ist eine Textdatei.
  → this is unix.
- grep '\<is\>' testfile
  → this is unix.
- grep 'is\>' testfile
  → this is unix.

Wiederholung von Ausdrücken

Eine weitere Art um Wiederholungen anzugeben ist die Intervall-Wiederholung. In geschwungenen Klammern (‚\{ min, max \}') wird dabei angegeben, wie oft die Wiederholung mindestens erfolgen soll (min) und wie oft die Wiederholung maximal erfolgend soll (max). Natürlich sind alle Wiederholungsanzahlen zwischen min und max auch erlaubt und gültige Übereinstimmungen!

Der Ausdruck ‚h\(el\)\{2,3\}' gibt an, dass zuerst ein ‚h' stehen muss, danach kann die Zeichenkette ‚el' zwei- oder dreimal wiederholt werden. Es würden also alle Zeilen ausgegeben werden, die ‚helel' (2x Wiederholung) oder ‚hellelel' (3x Wiederholung) enthalten. ‚hellelelel' (4x Wiederholung) wäre jedoch keine gültige Übereinstimmung!

Muster am Wortanfang suchen

Ein Muster passt in der Regel auf irgendein Vorkommen innerhalb der Zeile, d.h. wenn das Muster innerhalb eines Wortes gefunden wird, wird die Zeile genauso ausgegeben. Bei regulären Ausdrücken wird – sofern nicht explizit angegeben – nicht zwischen Wörtern (also Wortanfang und Wortende) unterschieden. Der Wortanfang und das Wortende sind durch spezielle Metazeichen markiert, die auch gesondert angegeben werden müssen. Der Wortanfang wird durch ‚\<' spezifiziert.

Der Ausdruck ‚\<is' gibt alle Zeilen aus, in denen sich ein Wort befindet, das mit ‚is' anfängt. Die ausgegebenen Zeilen passen deswegen, weil die Wörter ‚ist' und ‚is' selbst mit einem ‚is' anfangen.

Muster am Wortende suchen

Das Wortende wird durch ‚\>' spezifiziert. Das Wortende bedeutet aber nur, dass das Wort an dieser Stelle aufhört. Ein Wortende alleine spezifiziert noch kein ganzes Wort.



Der Ausdruck ‚is\>' alleine würde zum Beispiel die Zeile ‚this is unix.' liefern. Jedoch wird die Zeile deswegen ausgegeben, weil das Wort ‚this' mit einem ‚is' aufhört. Das eigentliche Wort ‚is' wird gar nicht mehr betrachtet, da bereits eine Übereinstimmung gefunden wurde.

Will man nun ein ganzes Wort suchen, so muss dieses von Wortanfang und Wortende umschlossen werden.



Um nach dem Wort ‚is' zu suchen, muss der Ausdruck wie folgt lauten: ‚\<is\>'. Nur dann wird wirklich nach dem Wort ‚is' gesucht!

Kapitel

9

Prozesse

Dieses Kapitel erläutert das Konzept der Prozesse und erklärt deren Aufbau und Struktur. Darüber hinaus werden Kommandos zur Verwaltung von Prozessen vorgestellt, aber es wird auch das Multitasking von UNIX beschrieben.

Alle Aktivitäten in einem UNIX-Betriebssystem sind durch *Prozesse* (auch Tasks genannt) realisiert. Dabei ist es egal, ob es sich um Aktivitäten der Benutzer, der Druckerverwaltung, der Kommunikation mit anderen Systemen oder um interne Aktivitäten wie der Plattenverwaltung handelt. In diesem Kapitel soll das Prozess-Konzept von UNIX beschrieben werden.

Lehrinhalte und -ziele

Am Ende dieses Kapitels sollten Sie verstanden haben, wie ein Prozess aufgebaut ist und welche Eigenschaften ein Prozess hat. Sie sollten die Möglichkeiten zur Duplizierung und zur Überlagerung von Prozessen kennen. Darüber hinaus sollten Sie die Funktionalität des ausgezeichneten Prozesses ‚init' kennen und erklären können, welche Aufgaben dieser im System hat.

Sie sollten über die verschiedenen Prozesstypen bescheid wissen und auch die wichtigsten Werkzeuge (Kommandos) kennen um Prozesse verwalten zu können. Zum Abschluss sollten Sie noch die einzelnen Prozesszustände und deren Eigenschaften, aber auch Mittel zur Prozesskommunikation, kennen.

Was ist ein Prozess?

Was ist ein Prozess

- Prozess besteht aus zwei Teilen
  - Programm, das im Hauptspeicher geladen ist (d.h. ausgeführt wird)
  - Prozessumgebung, in der Programm korrektes Ablaufen ermöglicht wird (Umgebung wird vom Betriebssystem zur Verfügung gestellt)
- Prozessumgebung liefert Kenndaten zu Prozess
  - Kenndaten werden in Prozesstabelle verwaltet
  - PID, PPID, User ID, Tabelle der geöffneten Dateien, Verweis auf aktuellen Arbeitskatalog, u.a.

Unter einem Prozess versteht man die Ausführung eines Programms. Dabei liegt das Programm in Form einer Datei vor und wird vom Betriebssystem in den Hauptspeicher geladen. Jedes mal wenn der Benutzer ein Programm startet, entsteht ein Prozess. Im Hauptspeicher stehen dann einzelne Programmanweisungen, die von der CPU ausgeführt werden. Nach der Abarbeitung der Programmanweisungen wird der Prozess beendet.

Zur gleichen Zeit stellt das Betriebssystem jedoch eine Prozessumgebung zur Verfügung. Diese ermöglicht das korrekte Ablaufen des Prozesses und sorgt dafür, dass dem Prozess genug Speicher und Ressourcen zur Verfügung gestellt werden. Außerdem wird vom Betriebssystem dafür gesorgt, dass kein anderer Prozess in diese Umgebung eindringen kann und schützt somit den Prozess vor ungewollten externen Manipulationen.

Wird dasselbe Programm gleichzeitig mehrmals gestartet, entstehen mehrere Prozesse, die dasselbe Programm ausführen. Dabei hat jeder Prozess wieder eine eigene Prozessumgebung. Die Prozessumgebung selbst liefert Kenndaten zu dem zugeordneten Prozess. Diese Kenndaten werden in der Prozesstabelle vom Betriebssystem verwaltet. Somit ist das Betriebssystem ständig über den aktuellen Zustand der einzelnen Prozesse informiert.

Die Kenndaten eines Prozesses umfassen folgende Daten:

- PID: Jedem Prozess wird beim Start durch das UNIX-Betriebssystem eine eindeutige Nummer, die Prozessidentifikation (Process Identifier oder PID), zugeordnet. Solange ein Prozess existiert, ist er durch diese Nummer systemweit eindeutig gekennzeichnet.

- PPID: Prozesse sind gewissen Vererbungsregeln unterworfen. Jeder Prozess kann einen oder mehrere neue Prozesse erzeugen, die man Kindprozesse nennt und die eine eigene Prozessnummer haben. Daraus ergibt sich, dass jeder Kindprozess auch einen Vaterprozess haben muss. Die Prozessnummer des Vaterprozesses wird dabei in der PPID (Parent Process ID) gespeichert. Über PID und PPID lassen sich Verwandtschaftsverhältnisse zwischen Prozessen aufzeigen.

- UserID: Ein UNIX-Prozess ist immer einem Benutzer zugeordnet. In der Regel ist dies der Benutzer, der den Prozess gestartet hat. Greift der Prozess auf Ressourcen zu (z.B. Dateien), dann erfolgt die Rechteauswertung auf die Ressource aufgrund der UserID. Für einen Prozess gelten somit die gleichen Einschränkungen wie für den Benutzer, der den Prozess aufgerufen hat. Es kann jedoch manchmal auch von Nöten sein, dass ein Prozess unter einer speziellen Kennung läuft. Als Beispiel dient hier das Programm ‚passwd', dass die Datei ‚/etc/shadow' lesen und schreiben muss um seine Aufgabe zu erfüllen. Im Kapitel 5 wurde bereits das SUID-Bit vorgestellt. Ist dieses Bit für das Programm gesetzt, so steht im Feld UserID anstatt der Kennung des Aufrufers die Kennung des Besitzers der Programmdatei. Somit hat der Prozess die Rechte, die der Besitzer der Programmdatei hat.

- Tabelle der geöffneten Dateien: Wenn Prozesse mit Dateien operieren, ist es meistens unerwünscht, dass ein anderer Prozess zur selben Zeit die gleiche Datei manipuliert. Darum führt das System in einer Tabelle Buch darüber, welche Datei von welchem Prozess gerade geöffnet ist. Will ein anderer Prozess auf die gleiche Datei zugreifen, sorgt das Betriebssystem dafür, dass die beiden Prozesse sich nicht gegenseitig behindern.

- Verweis auf aktuellen Arbeitskatalog: Wie schon des öfteren zuvor erwähnt, unterscheidet UNIX zwischen absoluten und relativen Pfadangaben. Damit UNIX diese relativen Pfadangaben überhaupt bilden kann, muss jeder Prozess den aktuellen Katalog (Verzeichnis) mitführen. Der Arbeitskatalog speichert somit Informationen darüber, wo (von welchem Verzeichnis aus) der Prozess aufgerufen wird und andere Betriebssystemdaten.

Prozesse in UNIX

Prozesse in UNIX

- Prozesse sind in UNIX baumartig angeordnet
- Es gibt einen Vater aller Prozesse → `init` Prozess, dieser wird direkt vom Kernel gestartet (PID von `init` ist 1)
- Alle anderen Prozesse sind Kindprozesse (werden mit dem C-Systemaufruf `fork` erzeugt)

Ähnlich dem Dateisystem ist auch die Prozessstruktur hierarchisch organisiert. Diese Hierarchie ist in der folgenden Abbildung dargestellt.

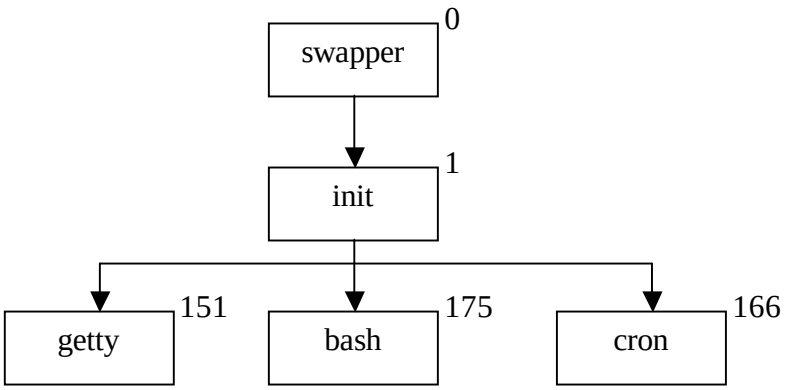


Abbildung 9-4: Prozesshierarchie in UNIX

Die Wurzel der Hierarchie bildet der init-Prozess, der mit der Prozessnummer (PID) 1 beim Booten des Systems gestartet wird. Die Prozessnummer 0 hat der Swapper, ein Prozess, der für die Prozessabwicklung notwendig ist. Alle weiteren Prozesse sind Kindprozesse des init-Prozesses und werden mit Hilfe des Systemaufruf fork erzeugt.

Ab System V Release 4 wird vom init-Prozess für eine Gruppe von Terminalports (virtuelle Konsolen) ein ttymon-Prozess gestartet. Wenn der Anwender sich anmelden möchte, wird für den entsprechenden Port ein Login gestartet. Nach dem Login wird eine Shell gestartet, die als Wurzelprozess aller weiteren Prozesse des Benutzers angesehen wird.

Prozesse duplizieren

Was passiert bei fork

- Prozess, in dem fork aufgerufen wird, wird dupliziert
  - Datensegment
  - Codesegment
  - Stacksegment
  - Prozessumgebung
- Duplizierter Prozess unterscheidet sich in
  - PID (freier Eintrag in Prozesstabelle)
  - PPID (PID des Prozesses, in dem fork aufgerufen wurde)
  - Verbrauchte CPU Zeit wird auf 0 gesetzt

Die Ausführung des Systemaufrufs fork bewirkt die Aufteilung des laufenden Prozesses in zwei unabhängig voneinander ablaufenden, identischen Prozessen. Der neue Prozess (Kindprozess) wird erzeugt, indem vom Vaterprozess eine exakte Kopie erzeugt wird. Dabei wird der Datenbereich, der Codebereich und die Prozessumgebung kopiert. Jeder Prozess kann ein oder mehrere Kindprozesse erzeugen, wobei jeder Kindprozess wiederum eigene Kindprozesse erzeugen kann.

Wird mit fork ein neuer Prozess generiert, so wird zunächst ein freier Eintrag in der Prozesstabelle ermittelt. Dieser Eintrag entspricht dann dem PID des neuen Kindprozesses. Sämtliche Informationen des Vaterprozesses werden an den Kindprozess vererbt. Die nun unabhängig voneinander laufenden Prozesse unterscheiden sich lediglich darin, dass nun der Kindprozess eine neue PID hat und in seiner PPID die PID des Vaterprozesses steht. Nach einem erfolgreichen fork-Aufruf wird der Kindprozess wie jeder normale Prozess behandelt und die Ausführung des Programms wird auch im Kindprozess fortgesetzt.



Da der Systemaufruf fork lediglich zwei identische Prozesse generiert, erbt der Kindprozess vom Vaterprozess folgende Prozessdaten:

- Code-, Daten- und Stacksegment des Programmes
- die Dateideskriptortabelle (die Dateizeigerpositionen bleiben gleich und derselbe Eintrag in der systemweiten Dateitabelle wird von beiden Prozessen genutzt)
- der aktuelle Katalog und der aktuelle Wurzelkatalog

- reale und effektive Benutzernummer und Gruppennummer
- die Priorität des Prozesses
- eventuell gesetzte Signalbehandlungen.

Gegenüber dem Vaterprozess ändern sich folgende Daten:

- Die PIDs und PPIDs sind unterschiedlich (der Kindprozess hat in seiner PPID die PID des Vaterprozesses stehen)
- Die verbrauchte CPU-Zeit des Kindprozesses wird auf 0 gesetzt

Das Systemkommando fork liefert einen Integerwert zurück, der Aufschluss darüber gibt, um welchen Prozess es sich handelt (Vater oder Kind). Liefert der Aufruf 0, so handelt es sich um den neu erzeugten Kindprozess, alle anderen Werte deuten auf den Vaterprozess hin.

! fork liegt nur als Systemaufruf innerhalb eines Systems vor. Somit kann die Duplizierung von Prozessen nur durch Systemprogrammierung (in der Regel durch ein C-Programm) erfolgen. Es gibt kein zu fork äquivalentes Kommando, dass von der Shell aus aufgerufen werden könnte. Programmbeispiel 9-1 zeigt ein unter UNIX lauffähiges C-Programm, das die Systemaufrufe fork und exec nutzt.

Prozesse überlagern

fork und exec

- Nach fork existieren zwei gleiche Prozesse
- Überlagerung des Kind-Prozesses mit neuem Prozess (C Systemaufruf exec)
  - Prozess wird durch neuen Prozess überlagert
    - Code- und Datensegment wird überlagert
    - Neues Stacksegment wird erzeugt
    - Prozessumgebung bleibt bestehen

Die Prozessduplizierung mit fork macht wenig Sinn, solange nicht erwünscht ist, dass jeder neue Prozess genau das gleiche macht wie der alte. Üblicherweise wird fork aufgerufen, um den neu erzeugten Kindprozess durch einen anderen Prozess zu überlagern. Dies geschieht durch den Systemaufruf ‚exec'. Hierbei wird kein neuer Prozess erzeugt, sondern der bestehende Prozess wird durch den neuen Prozess überlagert. Der gesamte Code- und Datenbereich wird überlagert, ein neues Stacksegement wird für den Prozess angelegt aber die alte Umgebung bleibt bestehen. Der überlagerte Prozess geht jedoch beim exec-Aufruf verloren und kann nicht mehr hergestellt werden.



Im Folgenden sind die Prozessdaten verzeichnet, die beim Überlagern unverändert bleiben:

- PID und PPID
- Reale UID und GID
- die Dateideskriptortabelle
- der aktuelle Arbeitskatalog und der Wurzelkatalog
- die Priorität des Prozesses
- das Kontrollterminal
- die verbrauchte CPU-Zeit.

Geändert werden beim Aufruf folgende Prozessdaten:

- Die programmspezifischen Signalbehandlungsroutinen gehen verloren und werden auf die voreingestellten Werte zurückgesetzt, jedoch werden bisher ignorierte Signale weiterhin ignoriert.

- Die effektive UID und GID wird aufgrund der Zugriffsrechte des geladenen Programms gesetzt. Ist weder das SUID noch das SGID-Bit gesetzt, so bleibt die effektive Benutzer- und Gruppenidentifikation erhalten. Ist jedoch eines diese Bits gesetzt, so wird die UID oder GID überschrieben.

Der Systembefehl exec kann nicht direkt aufgerufen werden. Stattdessen muss – je nach Parameter und Pfadauflösung – einer der folgenden Spezialversionen von exec aufgerufen werden:

Funktion	**Argumente**	**Umgebung**	**Pfadsuche**
execl	Liste	automatisch	Nein
execv	Vektor	automatisch	Nein
execle	Liste	manuell	Nein
execve	Vektor	manuell	Nein
execlp	Liste	automatisch	Ja
execvp	Vektor	automatisch	Ja

Funktionen mit einer Liste als Argumente erwarten eine offene Liste von Übergabeparametern, d.h. es können beliebig viele Parameter (je nachdem wie viele gebraucht werden) übergeben werden. Das Ende dieser Liste wird mit ‚`(char*)0`' gekennzeichnet.

Funktionen mit einem Vektor als Argument erwarten ein Array von char*-Werten. Darüber hinaus kann bestimmt werden, ob die Systemumgebung manuell oder automatisch übergeben werden soll, da es manchmal nötig ist, eine geänderte Systemumgebung dem neuen Prozess zu übergeben. Als letzte Option kann noch gewählt werden, ob das Programm von der Shell per Pfadsuche lokalisiert werden soll (Pfadsuche = ja). Ist dies nicht der Fall, so muss die zu startende Datei mit dem vollständigen, absoluten Pfad angegeben werden.

Beispiel

Im Folgenden ist ein C-Programm abgedruckt, dass die Verwendung von fork und exec veranschaulicht. In Kommentaren werden die speziellen Codeteile beschrieben deren Ausführung vom fork- und exec-Kommando abhängig sind.

```
int main(int argc, char* argv[]) {
  int PID;
  PID = fork();
  if (PID==0) {
     /* Dieser Code wird nur vom neu erzeugten Kind ausgeführt */
     execlp("ls", (char*)0);   /* Prozess überlagern */
     /* Dieses Codesegment sollte nie erreicht werden */
  }
  else if (PID > 0) {
     /* Dieser Code wird nur vom Vater ausgeführt */
  }
  return 0;
}
```

Programmbeispiel 9-1: C-Programm zu fork und exec

Der init-Prozess

Der init-Prozess

- init Prozess ist Vater aller Prozesse (wird beim Starten des Systems erzeugt)
- PID des init Prozesses ist 1
- init-Prozess startet Systemprozesse
- init-Prozess erzeugt für jedes Terminal login-Prompt
- Nach login eines Benutzer geht Kontrolle des Terminals zurück an init-Prozess
- UNIX-System kann ohne init-Prozess nicht existieren (startet neu / Kernel Panic)

Wie schon zuvor erwähnt, gibt es einen Prozess, der Vater aller anderen Prozesse ist. Dieser Prozess wird auch init-Prozess genannt. Der init-Prozess wird im Rahmen des Bootvorgangs gestartet und startet zunächst verschiedene Systemprozesse die im Hintergrund laufen. Danach aktiviert er für jedes angeschlossene Terminal einen Prozess getty. Der getty-Prozess initialisiert die Terminal Ports und präsentiert einen Login Prompt. Versucht ein Benutzer ein Login, dann ruft der getty-Prozess einen login-Prozess auf, der dann nach Benutzername und Passwort fragt und die Shell startet. getty und login verwenden kein fork sondern überlagern sich direkt mit exec. Wenn sich der Benutzer dann ausloggt, stirbt die Login-Shell und kommt somit automatisch wieder zum init-Prozess zurück. Dieser startet dann erneut einen getty-Prozess.

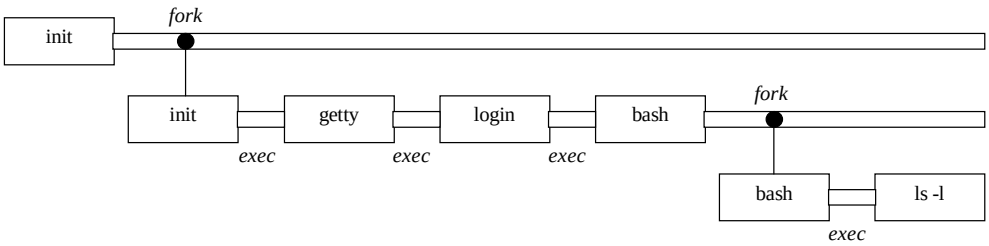


Abbildung 9-5: fork und exec beim Login und bei Ausführung eines ls-Kommandos

Der init-Prozess selbst muss immer laufen. Stirbt dieser so gibt das System entweder eine Fehlermeldung aus (‚kernel panic') oder bootet automatisch neu.

Prozesstypen

Prozesstypen

- Vordergrundprozess
  - Kommunikation über Shell (Standardeingabe und Standardausgabe); es kann immer nur 1 Vordergrundprozess gleichzeitig geben
- Hintergrundprozess
  - Mehrere Prozesse können im Hintergrund laufen (‚&' nach Kommando startet Prozess im Hintergrund)
- Dämonprozess
  - Serverprozesse, die im Hintergrund auf Anfrage warten (werden üblicherweise bei Systemstart erzeugt)
- Batchprozess
  - Vom Benutzer in Queue gereiht, wo sie der Reihe nach abgearbeitet werden

UNIX unterscheidet zwischen verschiedenen Prozesstypen. Dabei werden Prozesse meistens von einer Shell initiiert und kontrolliert. Bereits beim Starten kann man angeben, ob Prozesse im Vordergrund oder im Hintergrund ablaufen sollen.

ⓘ Gibt man nach dem Kommando ein ‚&' an, so wird der Prozess als Hintergrundprozess gestartet. Mit der aktuellen Shell kann ganz normal weitergearbeitet werden. Wird der Prozess nicht im Hintergrund gestartet, so blockiert er die Eingabeaufforderung bis er beendet worden ist.

Vordergrundprozesse

Vordergrundprozesse sind einem Terminal zugeordnet und können von diesem kontrolliert werden. Ein Vordergrundprozess belegt ein Terminal, es kann nur für diesen Prozess benutzt werden. Erst wenn der Prozess beendet ist, kann ein neues Kommando eingegeben werden. An einem Terminal ist während einer Sitzung die Shell aktiv. Startet der Anwender ein Kommando, so erzeugt die Shell einen Kindprozess. Dieser wird abgearbeitet (im Vordergrund) und liefert am Ende ein Signal an seinen Vaterprozess zurück. Der Vaterprozess dagegen wird schlafen gelegt und wartet auf das Ende des Kindprozesses.

Hintergrundprozesse

Ein Hintergrundprozess läuft nicht unter der Kontrolle eines Terminals ab, er ist davon gelöst. Das Terminal ist durch diesen Prozess nicht blockiert. Die Shell meldet sich mit einem Prompt, und der Benutzer kann sofort weitere Kommandos starten. Kindprozess und Vaterprozess laufen unabhängig voneinander, sie laufen asynchron ab. Liefert der Prozess Daten in die Standardausgabe, so tut er dies

weiterhin. Dabei kann es zu Kollisionen mit anderen Prozessen, die ebenfalls nach Standardausgabe schreiben, kommen. Die Ausgabe von Hintergrundprozessen sollte daher mit einer Ausgabeumleitung oder einer Pipe umgelenkt werden.

Prozesse können beim Starten von der Shell automatisch in den Hintergrund geschickt werden, indem hinter das Kommando ein ‚&'-Zeichen gesetzt wird. Nach dem Start des Kommandos wird die Prozessnummer ausgegeben und anschließend meldet sich die Shell wieder mit dem Prompt.

Dämonenprozesse

In einem laufenden UNIX-System gibt es eine Reihe von Prozessen, die systemweite Aufgaben wahrnehmen. Diese Prozesse werden zumeist beim Hochfahren des Systems gestartet. Sie sind keinem Terminal (oder Benutzer) zugeordnet und werden daemon-Prozesse (oder Dämonenprozesse) genannt. Dämonenprozesse sind Serverprozesse und warten im Hintergrund darauf, dass ein anderer Prozess ihr Service anfordert. Bekannte Beispiele sind der Netzwerkdämon inetd, der Druckerdämon lpd oder der Scheduler (Ein- und Auslagern von Prozessen).

Batchprozesse

Batchprozesse automatisieren Aufgaben. Sie werden dabei vom Benutzer in eine Queue gereiht, wo sie sequentiell abgearbeitet werden. Sie haben keine Verbindung zum Terminal. Typische Beispiele für solche Prozesse sind Datenbankabgleiche oder zeitintensive Rechenoperationen, die aus mehreren einzelnen Prozessen bestehen.

Prozessverwaltung

Prozessverwaltung$_1$

- ps
  - Listet alle aktiven Prozesse auf

a	Listet alle Prozesse
e	Zeigt Umgebung des Prozesses
l	Listet zusätzliche Information
u	Zeigt Benutzername und Startzeit
O	Sortiert Ausgabe (siehe man-Pages)

- Angezeigte Informationen
  - Priorität, PID, PPID, Status, Zeit, User ID, Group ID

In einem UNIX-System werden beim Systemstart üblicherweise viele Dämonprozesse gestartet. Benutzer starten oftmals – gerade in der X-Window Umgebung – sehr viele Hintergrundprozesse. Durch diese Prozessflut verliert man schnell den Überblick über die gestarteten Prozesse. Das Kommando ‚ps' liefert Informationen über den Status der aktiven Prozesse in einem UNIX-System. Der Umfang der Informationen (welche Prozesse und welche Daten) kann durch Optionen gesteuert werden. Die einzelnen Optionen sind in den entsprechenden Online Hilfeseiten (manual pages) erklärt. Die eigenen Prozesse erhält man durch die einfache Eingabe von „ps".

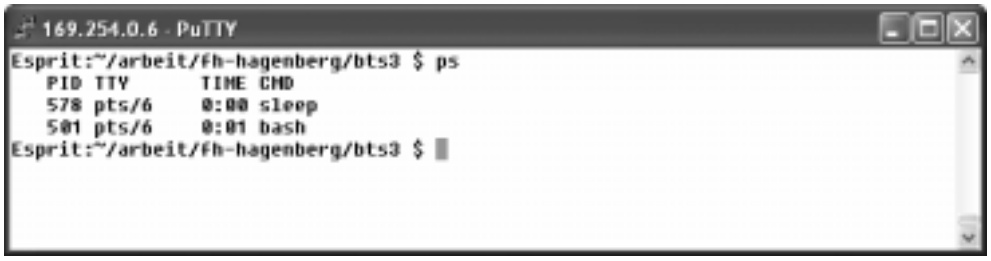


Abbildung 9-6: Prozesshierarchie in UNIX

Dabei wird dem Benutzer eine Liste der eigenen Prozesse geliefert, zusammen mit der dazugehörigen Prozessnummer (PID), das Terminal an dem der Prozess arbeitet, die verbrauchte Systemzeit und der Name des aktiven Kommandos.

Sind umfangreiche Informationen erforderlich, so ist die Option –f nützlich. Hier wird zusätzlich noch zu jedem Prozess die PID des Vaterprozesses (PPID), der Besitzer und die Prozessorzeit angegeben.

Prozessverwaltung$_2$

- `kill`
  - Sendet Signal an einen Prozess
  - Syntax: `kill [optionen] PIDs`

l	Listet alle Signale
-s *signal*	Angabe des zu sendenden Signals (auch Name des Signals ist möglich)
-signal	Wie -s

- Signal Nummer 9 (SIGKILL) kann nicht ignoriert werden (Prozess wird auf jeden Fall beendet)

Mit dem Kommando ‚kill' können Signale an einen UNIX-Prozess geschickt werden. Diese Möglichkeit wurde bereits in Kapitel 7 „Shellprogrammierung" vorgestellt, zusammen mit einer Liste von Signalen die es üblicherweise auf einen UNIX-System gibt.

Die Übermittlung von Signalen erfolgt vom Betriebssystem. Dabei reagiert der angesprochene Prozess auf das übermittelte Signal und ruft – sofern implementiert – die entsprechende Signalroutine auf, oder terminiert. Die Übermittlung erfolgt mit dem Kommando kill.

```
kill –Signalnummer PID
```

Das Kill-Kommando selbst erwartet beim Aufruf eine Option und die Prozessnummer, an die das Signal übermittelt werden soll. Die Option ist in der Regel das Signal, dass übermittelt werden soll. Dabei kann das Signal direkt als Option angegeben werden (z.B. –9) oder als Parameter der Option –s (z.B. –s 9). Das Signal kann dabei als Nummer oder aber als symbolischer Name angegeben werden. UNIX kennt dabei eine Vielzahl von Signalen mit unterschiedlicher Bedeutung, die aber weitgehend standardisiert sind.

„kill –l" liefert eine Liste mit allen im System implementierten Signalen, deren Nummer und symbolischer Name.

Signale eigen sich sehr gut dazu, Prozesse – die vielleicht nicht mehr reagieren – von außen zu beenden. Wird zum Beispiel keine Signalnummer angegeben, so wird das Signal 15 (SIGTERM) an den Prozess gesendet, der den Prozess terminiert.

Wird als PID eine 0 angegeben, so wird das Signal an alle Prozesse eines Anwenders, inklusive der Login-Shell, gesendet.

! Wie im Kapitel 7 angedeutet, besteht die Möglichkeit, auf ankommende Signale zu reagieren bzw. diese zu ignorieren. Dies ist für alle Signale möglich, lediglich das Signal 9 (SIGKILL) kann nicht ignoriert werden. Erhält ein Prozess das Signal 9, beendet er auf jeden Fall.

Hilfreich ist das kill-Kommando, wenn sich ein Hintergrundprozess aufgehängt hat oder das Terminal keine Eingabe mehr annimmt. In diesem Fall kann der Prozess (z.B. von einem Nachbarterminal) beendet werden.

! Ein Benutzer kann aber nur diejenigen Prozesse abbrechen, die er selbst gestartet hat. Lediglich der Superuser kann beliebige Prozesse beenden. Eine Ausnahme bildet hier der init-Prozess (PID 1), der auch vom Superuser nicht abgebrochen werden kann.

Prozesse überleben „logout“

Wie schon oben eingehend erwähnt, sterben Kindprozesse automatisch, wenn der Vaterprozess terminiert. Bei einem „logout“ wird üblicherweise der Vater aller in einer Session gestarteten Benutzerprozesse beendet. Das nohup (No Hang Up) Kommando ermöglicht es, einen Prozess sogar nach dem Abmelden eines Benutzers vom System weiterlaufen zu lassen.

nohup *Kommando* &

Wenn die Ausgaben nicht explizit umgeleitet werden, so wird die Standardausgabe und der Standard-Fehlerkanal in die Datei ‚nohup.out' im zuletzt aktuellen Verzeichnis geschrieben. Sollte dies auf Grund fehlender Zugriffsrechte nicht möglich sein, wird diese Datei im Home-Verzeichnis des Benutzers angelegt.

Falls ein solcher Prozess vom Terminal lesen möchte, wird er automatisch unterbrochen und schlafen gelegt. Um die Eingabe zu tätigen, muss er dann vorher wieder explizit in den Vordergrund geholt werden (Kommando ‚fg').

Die Shell meldet jedoch eine Warnung, falls ein Benutzer sich vom System abmelden will und im Hintergrund noch Prozesse laufen. Man kann dann entsprechend reagieren oder das logout abermals versuchen.

Prozesszustände

Prozesszustände

- UNIX Systeme kennen im Wesentlichen drei Prozesszustände
  - Running: Das dem Prozess zu Grunde liegenden Programm wird von der CPU ausgeführt
  - Ready: Prozess kann ausgeführt werden, CPU ist aber einem anderen Prozess zugeordnet
  - Blocked: Prozess benötigt Daten von externen Quellen, um weiterlaufen zu können

UNIX ist ein Multitasking Betriebssystem. Das bedeutet, dass mehrere Prozesse scheinbar zur gleichen Zeit ablaufen können, da die CPU-Zeit auf die verschiedenen Prozesse aufgeteilt wird. Um eine solche Aufteilung realisieren zu können, sind verschiedene Prozesszustände notwendig. Die einzelnen Prozesszustände sollen anhand der Bedeutung von Multitasking erläutert werden. Um detaillierte Informationen zu Multitasking / Multithreading zu bekommen, wird an dieser Stelle auf einschlägige Fachliteratur verwiesen.

Multitasking über Zeitscheibenverfahren

UNIX, als ein Multiuser- und Multitasking-System, erlaubt es, dass mehrere Benutzer gleichzeitig auf ein und demselben System arbeiten. Daher muss es auch möglich sein, dass mehrere Prozesse parallel auf dem Rechner laufen. Dazu sind verschiedene Mechanismen notwendig, die für die Koordination und Verwaltung der Prozesse zuständig sind.

Die Koordination und Verwaltung wird vom sogenannten Scheduler übernommen, ein zentraler Prozess der einzelnen Prozessen CPU-Zeit zuteilt. Dazu verwendet er ein Zeitscheibenverfahren wo den einzelnen Prozessen in regelmäßigen Zeitintervallen eine gewisse Rechenzeit (auch Zeitscheibe genannt) zugeordnet wird. Nur während dieser Zeitscheibe wird der Prozess von der CPU bearbeitet.

Nachdem die Zeitscheibe abgelaufen ist, wird der nächste Prozess abgearbeitet. Ziel ist es, allen Anwendern und laufenden Prozessen möglichst gleich viel Prozessorzeit zukommen zu lassen.

Damit ein Prozess ausgeführt werden kann, müssen alle für die Ausführung notwendigen Ressourcen (CPU, Plattenzugriffe, etc.) zur Verfügung stehen. Die CPU selbst erweist sich als „teuerste“ Einheit, da sie zu einem Zeitpunkt immer nur einen Prozess bearbeiten kann. Um die CPU nun nicht unnötig zu belasten, werden Prozesse in verschiedene Zustände gebracht, die sich auch auf die Zuteilung von Zeitscheiben zu dem Prozess auswirken.

Prozesszustände

Wird ein Prozess gerade bearbeitet, d.h. die dem Prozess zugeordnete Zeitscheibe ist gerade aktiv und die CPU bearbeitet den Prozess, dann befindet er sich im Zustand „*running*“. Dies ist der einzige Zustand, in dem der Prozess wirklich aktiv arbeitet.

Läuft die Zeitscheibe für den Prozess ab, dann wird dem Prozess die CPU vom Scheduler entzogen. Der Prozess wird nun nicht mehr aktiv bearbeitet, sondern wartet auf die nächste, ihm zugeteilte Zeitscheibe. Diesen Zustand nennt man „*ready*“ (oder auch „*suspended*“), da der Prozess arbeiten könnte, die CPU jedoch gerade einem anderen Prozess zugeordnet ist. Prozesse die im ready-Zustand sind, werden vom Scheduler bei der Zeitscheibenvergabe berücksichtigt, d.h. der Scheduler reiht diese Prozesse und teilt diesen in ihrer Reihenfolge Zeitscheiben zu.

Zeitweise wartet ein Prozess jedoch auf eine Eingabe oder die Freigabe einer benötigten Ressource. Solange diese Eingabe oder Ressource nicht verfügbar ist, kann der Prozess nicht weiterarbeiten. In solchen Fällen macht es wenig Sinn, dem Prozess Zeitscheiben zuzuordnen und ihn aktiv zu bearbeiten. Darum wird der Prozess in den Zustand „*blocked*“ (oder auch „*wait*“) geschickt und wartet dort auf die Eingabe oder die Freigabe der Ressource. Prozesse die in diesem Zustand sind, werden vom Scheduler bei der Zeitscheibenvergabe nicht berücksichtigt, d.h. sie bekommen keine CPU-Zeit zugeteilt. Erst wenn die Ressource oder Eingabe vorhanden ist, wird der Prozess wieder in den „ready“-Zustand versetzt und wartet dort auf die Zuweisung einer Zeitscheibe durch den Scheduler.

Wird ein Prozess beendet, so wird er nicht gleich aus der systemweiten Prozesstabelle entfernt, da er erst noch seinen Vater über die Terminierung informieren muss. Bis dies geschehen ist, befindet er sich im „*zombie*“-Zustand. Hier wird dem Prozess vom Scheduler in keiner Weise mehr Beachtung geschenkt, da dieser keine aktive Arbeit mehr verrichten wird.

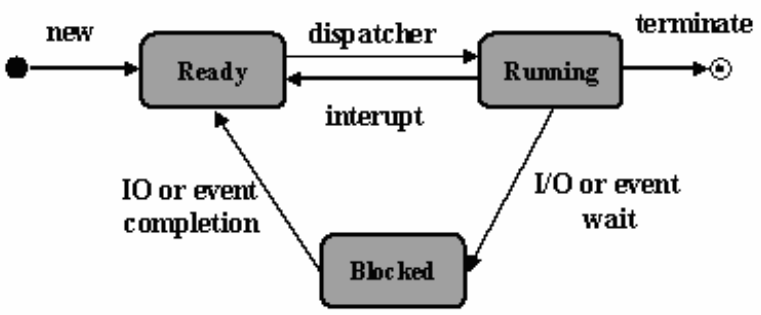


Abbildung 9-7: Prozesszustände

Wird ein Prozess erzeugt, so wird er automatisch in den „ready“-Zustand versetzt. Da der Prozess zuvor noch nie gestartet wurde, sorgt eine hohe Priorität dafür, dass der Prozess unmittelbar nach dem Start erstmals ausgeführt wird. Der Scheduler bezieht den Prozess bei der nächsten Zeitscheibenvergabe somit in Betracht. Wird dem Prozess eine Zeitscheibe zugeteilt und diese gerade bearbeitet, so wird der Prozess in den Zustand „running“ versetzt. Sobald die Zeitscheibe abgelaufen ist, befindet er sich wieder im Zustand „ready“. Dies geschieht so lange, bis der Prozess terminiert, oder auf eine Ressource oder Eingabe von außen wartet. Prozesse, die auf ein externes Ereignis warten, werden in den „blocked“-Zustand versetzt. Der Scheduler zieht sie gar nicht mehr in Betracht, wenn Zeitscheiben vergeben werden sollen. Erfolgt die Eingabe oder wird die Ressource freigegeben, so wird der Prozess in den Zustand „ready“ versetzt. Aus dem „blocked“-Zustand führt kein Weg mehr zur direkten Ausführung („running“).

Echtzeitverarbeitung durch Prozessprioritäten

Ziel des Zeitscheibenverfahrens ist es, alle Anwender und Prozesse möglichst gleichmäßig zu bedienen. Dabei werden interaktive Prozesse meist bevorzugt, damit für die Benutzer keine Wartezeiten entstehen. Der Scheduler benötigt daher Informationen, wann welcher Prozess aktiv werden soll. Hierzu wird die Priorität eines Prozesses hergenommen. Jedem Prozess wird beim Start eine Priorität zugeordnet, die Aufschluss über die Wichtigkeit des Prozesses gibt. Der Scheduler beachtet die Priorität aller in Frage kommenden Prozesse und berücksichtigt diese bei der Zeitscheibenvergabe. Dabei wird dem Prozess mit der höchsten Priorität die nächste Zeitscheibe zugeordnet. Nachdem ein Prozess bearbeitet wurde, wird seine Priorität verkleinert. Sobald ein anderer Prozess nun eine größere Priorität hat, wird diesem die nächste Zeitscheibe zugeordnet.

Die Priorität eines Prozesses ist daher kein fester Wert sondern wird vom Betriebssystem ständig neu berechnet. Prozesse, die eine starke CPU-Belastung verursachen, bekommen eine niedrige Priorität, damit andere Prozesse nicht blockiert werden. Für die Berechnung werden aber auch andere Daten wie die bisherige Rechenzeit, die Wartezeiten, die Größe des Programms und die bisherige Laufzeit berücksichtigt.

Auch die Länge der zugeordneten Zeitscheibe ist variabel. Prozesse mit einer niedrigen Priorität bekommen eine längere Zeitscheibe als welche mit einer hohen.

Prioritätsklassen

ⓘ Prioritäten alleine erlauben zwar eine „faire“ Zuteilung von Zeitscheiben, erlauben aber keine Echtzeitbearbeitung. Darum wurden ab System V Release 4 Prioritätsklassen eingeführt, die den UNIX-Kernel echtzeitfähig machen.

Im System werden grundsätzlich drei Prioritätsklassen unterschieden:

- Time-Sharing: In dieser Klasse liegen normale Benutzerprozesse, denen vom Scheduler in zeitlichen Abständen (abhängig von deren Priorität) Zeitscheiben zugeordnet werden. Die Priorität der Prozesse wird dabei ständig neu berechnet und alle Prozesse werden als gleichwertig betrachtet.

- Real-Time: Diese Klasse ermöglicht die Echtzeitbearbeitung von Prozessen. Dabei wird jedem Prozess dieser Klasse eine Priorität zugewiesen, die während der Laufzeit nicht mehr verändert wird. Diese Prozesse fallen nicht unter das Time-Sharing und dadurch kann die genaue Ablaufreihenfolge der Prozesse festgelegt werden. Nur ein privilegierter Benutzer hat die Möglichkeit die Priorität und somit die Ablaufreihenfolge der Echtzeitprozesse zu beeinflussen.

- System: Prozesse in dieser Klasse sind unbedingt für den reibungslosen Betrieb des Systems erforderlich. Hier liegen nur die wichtigsten Prozesse und diese haben die höchste Priorität im System. Diese Ablaufreihenfolge und Priorität dieser Prozesse kann nicht beeinflusst werden.

Die Prioritätsintervalle der einzelnen Klassen können im Kernel konfiguriert werden. Dabei muss aber immer darauf geachtet werden, dass Prozesse in der Systemklasse die höchste Priorität haben und Echtzeitprozesse mehr Priorität haben sollten als Time-Sharing Prozesse.



Es sollte auch beachtet werden, dass Echtzeitprozesse starke Auswirkung auf das Verhalten der anderen Prozesse und damit auf das Gesamtsystem haben. Prozesse sollten nur dann als Echtzeitprozesse gestartet werden, wenn sie wirklich zeitkritische Operationen durchführen müssen.

Prioritätsvergabe

Die Priorität eines Prozesses kann mittels des Kommandos ‚nice' beim Start des Prozesses festgelegt werden.

```
nice [-increment] Kommando &
```

Wird der nice-Befehl in der oben angegebenen Form gestartet, so wird das angegebene *Kommando* als Hintergrundprozess (durch die Angabe von ‚&') gestartet. Der Wert im Feld *increment* gibt an, um wie viel die Priorität des Prozesses herabgesetzt werden soll. Einem normalen Benutzer ist es nur möglich, die Priorität eines Prozesses zu vermindern. Der Superuser hat jedoch die Möglichkeit, einen negativen increment-Wert anzugeben um somit die Priorität des Prozesses zu erhöhen.

Prozesskommunikation

Prozesskommunikation

- Verschiedene Möglichkeiten wie zwei Prozesse miteinander kommunizieren können
  - Signale: Ein Prozess sendet ein Signal an einen anderen Prozess; dieser reagiert entsprechend auf das Signal
  - Pipes: realisieren Queues; ein Prozess schreibt Ergebnis in eine Pipe, ein zweiter Prozess liest von dieser Pipe
    In UNIX existieren anonyme und benannte Pipes
  - Sockets: ermöglichen Kommunikation zwischen Prozessen, die auf verschiedenen physikalischen Rechnern laufen (Kommunikation über Netzwerk)

UNIX bietet verschiedene Möglichkeiten der Prozesskommunikation. Einige Möglichkeiten (z.B. „shared memory“) sind jedoch eher für die Systemprogrammierung relevant, können aber aus der Shell nicht verwendet werden. In weiterer Folge werden lediglich die Möglichkeiten betrachtet, die auch für die Shellprogrammierung relevant sind. Die folgende Auflistung der verschiedenen Prozesskommunikationsmöglichkeiten ist somit nur ein Auszug aus dem realen Angebot.

Signale

Signale sind die einfachste Möglichkeit, um einem Prozess eine Nachricht zukommen zu lassen. Dabei sendet ein Prozess ein Signal an einen anderen Prozess, der dann entsprechend auf das Signal reagiert. Signale und die Möglichkeit zum Implementieren von eigenen Signalroutinen wurden bereits in diesem Kapitel und im Kapitel 7 „Shellprogrammierung“ eingehender behandelt.

Pipes

Eine Pipe ist eine temporäre Datei, die vom System verwaltet wird. Jede Pipe hat eine Lese- und eine Schreibseite. Dabei schreibt ein oder mehrere Prozesse in die Datei und ein anderer Prozess liest die Daten. Die Pipe geht nach dem FIFO (First In, First Out) Prinzip vor. Dabei werden zuerst die Daten gelesen, die als erstes in die Pipe reingeschrieben wurden (Queue). Die Möglichkeit zur Nutzung von Pipes wurde bereits im Kapitel 6 „Die Shell“ behandelt.

Sockets

Sockets sind im Prinzip wie Pipes, jedoch ermöglichen sie die Kommunikation über ein Netzwerk. Auch bei Sockets schreibt ein Prozess Daten in einen Zwischenspeicher und ein zweiter Prozess liest diese Daten. Der Unterschied zu Pipes ist jedoch, dass die beiden Enden verteilt liegen und die Daten zwischen den beiden Enden über ein Netzwerk übertragen werden.

Kapitel

10

Der Bootvorgang

Dieses Kapitel erläutert, was nach dem Einschalten des Rechners bis zum Linux-Login passiert. Dabei wird der Bootmanager LILO behandelt, aber auch der Bootvorgang selbst.

Bis zum endgültigen Login laufen im Hintergrund sehr viele Dinge ab. Das reicht vom Testen und Initialisieren der Hardware bis hin zum Laden von Treibern und Hintergrundprozessen – abhängig davon welche Konfiguration gestartet werden soll. Dieses Kapitel erläutert, was genau nach dem Einschalten des Rechners bis hin zum endgültigen UNIX-Login passiert. Im Speziellen wird kurz der Linux Bootmanager ‚LILO' vorgestellt, aber auch die Funktion und der Aufbau der Datei ‚/etc/inittab' zusammen mit dem init-Prozess näher erläutert.

Lehrinhalte und -ziele

Sie sollten nach diesem Kapitel in der Lage sein, zu verstehen, was beim Hochstarten eines UNIX-Systems genau passiert. Darüber hinaus sollten Sie die Funktion des init-Prozesses kennen und wissen, welche Funktion die Datei ‚/etc/inittab' hat. Letztendlich sollten Sie den Begriff ‚Runlevel' kennen und wissen, welchen Zweck die verschiedenen Runlevels haben.

Der BIOS-Selbsttest

BIOS Selbsttest

- BIOS = Basic Input/Output System
- Power On Self Test (POST) 2 Schritte
  - Test und Initialisierung zentraler Hardware
    - CPU, BIOS-ROM, DMA (Direct Memory Access) – Controller, Tastatur, ersten 64k des RAMs, Interupt- und Cache-Controller
  - Test und Initialisierung von Systemerweiterungen
    - RAM über 64k, Schnittstellen, Disketten- und Festplattenkontroller, Video-ROM
- Falls Test erfolgreich wird nach gültiger Boot-Sequenz gesucht

Direkt nach dem Einschalten – also noch bevor irgendein Betriebssystem geladen wurde – beginnt der Rechner mit der Programmausführung an einer festgelegten Adresse (F000:FFF0h). Dabei liegt das Programm nicht auf einer Festplatte, sondern ist fest auf ein Chip – dem sogenannten BIOS-Chip – gespeichert. Das BIOS (Basic Input Output System) steuert dabei die im Rechner integrierte Hardware. Ein Teil dieses Programms hat die Funktion, die im System integrierte Hardware zu testen. Dieser Test ist auch unter dem Namen ‚Power On Self Test' (POST) bekannt. Diese Tests werden in der Regel sehr schnell durchgeführt und wird normalerweise vom Benutzer kaum wahrgenommen. Da viele Tests bereits abgewickelt werden, bevor die Grafikhardware initialisiert wurde, werden Fehler entweder bis zur Initialisierung der Grafikhardware gemerkt und dann ausgegeben – oder wenn diese fehlschlägt als eine Sequenz von Piepstönen am PC-Lautsprecher ausgegeben.

ⓘ [Die genaue Reihenfolge der einzelnen Tests ist vom Hersteller des BIOS abhängig. Die Reihenfolge der nun folgenden Auflistung gilt für das AMI Hi-Flex BIOS Chip. Jedoch werden die hier aufgeführten Grundkomponenten immer getestet – lediglich die Reihenfolge des Testablaufs obliegt dem Hersteller.

Power On Self Test

Zuerst wird der Prozessor selbst – im Detail die Prozessorregister – auf Fehlerfreiheit überprüft, da ohne Prozessor nichts geschehen kann. Ist dieser fehlerhaft, so werden keine weiteren Tests mehr durchgeführt und der Fehler anhand von Signaltönen über den PC-Lautsprecher signalisiert.

Im zweiten Schritt wird zumeist das BIOS-ROM, also das Chip welches das BIOS-Programm enthält, getestet. Hier erfolgt ein einfacher Checksummenvergleich – weicht der Soll-Wert der Checksumme vom Ist-Wert ab, so liegt ein Fehler im BIOS-Chip vor.

In weiterer Folge wird der Keyboard Kontroller getestet. Schlägt dieser Test fehl, so wird im Normalfall keine Tonfolge ausgegeben sondern der Fehler zu einem späteren Zeitpunkt – nämlich dann wenn die Grafikkarte initialisiert wurde – am Bildschirm ausgegeben.

In einem nächsten Schritt wird das CMOS getestet. Das CMOS ist ein batteriegepufferter Speicher, der die variablen Einstellungen des BIOS zwischenspeichert. Hier wird z.B. die aktuelle Uhrzeit, die Festplatten-geometriedaten und sonstige BIOS-Einstellungen gespeichert. Durch die Batteriepufferung sind die Daten selbst nach längerem Ausschalten des PCs immer noch verfügbar.

Nun erfolgen weitere Tests von Hardware, die als eigenständige Chips am Mainboard integriert sind. Zuerst wird der Systemzeitgeber geprüft, dann erfolgt eine Analyse der verschiedenen Speicherchips die im System integriert sind. Hier wird zwischen dem herkömmlichen Arbeitsspeicher (RAM) und dem Cache-Speicher unterschieden. Der Cache-Speicher ist sehr viel kleiner als der Arbeitsspeicher, jedoch sehr viel schneller und wird nur vom Prozessor selbst für kurzfristige Zwischenspeicherungen verwendet. Dies erlaubt dem Prozessor schneller Daten zu verarbeiten. Der Arbeitsspeicher steht hingegen Anwendungen zur Verfügung und wird vom Betriebssystem verwaltet. Da zum Zeitpunkt des Tests der zentralen Hardware jedoch nur die ersten 64 KByte des Arbeitsspeichers notwendig sind, werden vorerst nur diese getestet. Abgeschlossen wird der Speichertest mit dem Prüfen der Checksumme des CMOS. Dadurch dass dieser Speicher batteriegepuffert ist, muss geprüft werden, ob die Daten durch einen eventuellen Ausfall der Batterie an Gültigkeit verloren haben. Treten bei diesen Tests Fehler auf, werden diese nach Initialisierung der Grafikkarte am Bildschirm ausgegeben.

Nun wird die Grafikkarte des Systems getestet und initialisiert. Falls zuvor bereits Fehler aufgetreten sind, werden diese nach der Initialisierung am Bildschirm ausgegeben. Alle Fehler die nun in weiterer Folge auftreten, werden auch am Bildschirm ausgegeben.

Nach dem Initialisieren der Grafikkarte wird der gesamte, im System verfügbare Speicher getestet. Zuerst werden nur die ersten 640 KByte des Systems getestet, da sich der Prozessor zu diesem Zeitpunkt noch in einem Modus (‚Real Mode) befindet, in dem er nicht mehr als 1 MB des Systemspeichers ansprechen kann. Dann wird der sogenannte ‚Protected Mode' des Prozessors getestet. In diesem Modus kann der Prozessor den ganzen, im System befindlichen Speicher ansprechen. Bei dieser Gelegenheit wird zusätzlich noch der restliche Speicherbereich (also alles was jenseits der 640 KByte-Grenze liegt) getestet.

Zum Abschluss werden noch weitere Kontroller (z.B. DMA-Kontroller, Disketten- und Festplattenlaufwerkskontroller) getestet und initialisiert.

Start des Bootvorgangs

Wenn alle Tests positiv verlaufen sind, sucht das BIOS nach einer angegebenen Bootsequenz. Die Bootsequenz definiert dabei, welche Massenspeichermedien für einen Bootvorgang in Frage kommen. Unter Booten wird im Allgemeinen das Laden eines Betriebssystems in den Hauptspeicher und das Anstoßen der Ausführung jenes Betriebssystems verstanden. Das BIOS beginnt nun mit dem ersten, in der Bootsequenz angegebenen Speichermedium. Beinhaltet dieses Speichermedium *keine* Bootinformation, wird versucht, vom nächsten in der Sequenz angegebenen Speichermedium zu suchen. Moderne Computersysteme erlauben das Booten von Disketten, Festplatten, CD-ROM-Laufwerken und sogar Netzwerkkarten.

Der Bootloader

Bootloader

- Nötig wenn mehrere Systeme auf einem Rechner installiert sind (ansonsten könnte Initialcode des BS in MBR (Master Boot Record) stehen)
- Bootloader kann z.B.
  - Parameter an Betriebssystem übergeben
  - Betriebssystem mit Passwort schützen
- Bootloader zweistufig
  - Erste Stufe im MBR (darf nur 446 Byte groß sein)
  - Zweite Stufe auf Platte
- Bootloader kann dann eines der von ihm verwalteten Betriebssysteme starten (in Hauptspeicher laden)

Funktion von Bootloadern

Nachdem das BIOS die Tests abgeschlossen hat, sucht jenes nach einem Bootcode im sogenannten ‚Master Boot Record' (MBR) der angegebenen Bootmedien, der dann geladen und gestartet wird. Meist steht in diesem MBR ein Bootloader, der dafür sorgt, dass das zugrundeliegende Betriebssystem korrekt gestartet wird.

ⓘ Genau genommen benötigt man keinen Bootloader, um ein Betriebssystem zu starten. In den alten MS-DOS Zeiten war es üblich, dass der initiale Teil des Betriebssystems selbst im Master Boot Record platziert wurde. Dieser Code hatte die Aufgabe, das entsprechende Bootmedium zu initialisieren und dann die restlichen Teile des Betriebssystem zu laden.

Bootloader sind für verschiedene Aufgaben vorgesehen. Das größte Anwendungsspektrum ist das Koordinieren und Laden eines Betriebssystems wenn mehrere Systeme auf dem selben Rechner installiert sind. Sekundäre Aufgaben sind aber auch das Übergeben eventueller Parameter an das Betriebssystem bereits beim ‚Hochfahren' oder das Schützen eines solchen durch die Eingabe eines Passworts.

ⓘ Der MBR selbst ist nur 446 Bytes groß. Die mittlerweile sehr umfangreich gewordenen Bootloader passen meist nicht mehr in diesen kleinen Bereich der Festplatte. Darum sind moderne Bootloader in zwei Teile geteilt. Der erste Teil liegt im MBR und ist dafür zuständig, dass die Festplatte, die den zweiten Teil des Bootloaders enthält, initialisiert und benutzbar gemacht wird. Nachdem die Festplatte für den Bootloader benutzbar ist, kann dieser auch auf den zweiten Teil des Bootloaders zugreifen und bietet dann erst die volle Funktionalität.

Nach dem Wählen eines Betriebssystems wird der Bootloder dann das gewählte in den Hauptspeicher laden und initialisieren.

Der Linux Bootloader LILO

LILO ist der Linux Boot-Loader und hat sich in der letzten Zeit zu einem sehr komfortablen Tool entwickelt, dass das Wählen eines Betriebssystem in einem graphischen Modus anbietet. LILO selbst ist in der Lage, verschiedene Betriebsysteme zu starten. Das Programm wird so konfiguriert, dass ihm alle Informationen zum Start vorliegen. Wenn zum Beispiel eine Version von Windows gestartet werden soll, muss LILO wissen, in welchem Bootsektor oder welcher Partition die Startdateien jenes Betriebssystems abgelegt sind. Im Falle von Linux enthält LILO einen Verweis auf den zu ladenden Kernel. Dies ermöglicht die Verwendung mehrerer, unterschiedlicher Kernel auf ein und dem selben Linux-System. Dabei ist LILO in der Lage, dem Kernel beliebige Parameter mitzugeben. Parameter, die vom Kernel nicht verstanden werden, werden automatisch an den während des Bootprozesses gestarteten ‚init'-Prozess weitergeleitet. Dieser Prozess wird noch später in diesem Kapitel detaillierter behandelt.

Initialisierung des Kernels

Kernel-Initialisierung

- Der vom Bootloader ausgewählte Kernel wird in Hauptspeicher geladen und dekomprimiert (Prozessor beim Laden noch im Real-Mode → Speicheradressen unter 1MB)
- Nach Dekomprimieren wird BS geladen
  - Interne Tabellen initialisieren (IRQs, Fehlerinterrupts, Speicherseitenverwaltung)
  - Weiterer Hardware-Test (Auto-Detect) – werden hier Fehlermeldungen am Bildschirm ausgegeben, ist Gerät später nicht verfügbar
  - Swapper-Prozess wird gestartet (PID = 0)
  - init-Prozess wird gestartet (PID = 1)

Nach Auswahl des zu startenden Kernels im Bootloader sorgt dieser dafür, dass der gewünschte Kernel im Dateisystem gefunden, geladen und ausgeführt wird. Da aktuelle Kernelversionen mittlerweile sehr groß geworden sind und der Prozessor zu diesem Zeitpunkt nur ca. 1 MB an Arbeitsspeicher verwalten kann, müssen hier spezielle Vorkehrungen getroffen werden.



Jeder Intel-basierte Prozessor versteht im Prinzip zwei Operationsmodi. Im sogenannten ‚Real-Mode' kann der Prozessor nur auf ca. 1 MB (real 640 KB) des gesamten Arbeitsspeichers zugreifen. Der Prozessor versteht jedoch auch den sogenannten ‚Protected-Mode', in dem der gesamt verfügbare Arbeitsspeicher adressiert und verwendet werden kann.

Da der Protected Mode zuerst aktiviert werden muss, damit dieser verwendet werden kann, geht das Laden des Kernels in zwei Schritten vonstatten. Der Kernel selbst liegt in komprimierter Form vor, sodass er kleiner als 640 KB ist. Die Idee ist, dass der Kernel so in den Hauptspeicher passt. Dann wird ein Teil des Kernels – der in der Lage ist, den Prozessor in den Protected Mode umzuschalten – entpackt und gestartet. Nachdem das Codesegment ausgeführt wurde, befindet sich der Prozessor im Protected Mode und kann den gesamten verfügbaren Arbeitsspeicher nutzen. Nun wird der restliche Teil des Kernels entpackt, geladen und das eigentliche Betriebssystem gestartet.

Start des Betriebssystems

Beim Start des Betriebssystems werden zuerst werden interne Funktionen und Tabellen des Systems initialisiert. Dies sind die Tabellen, die für eine spätere Prozessverwaltung notwendig sind (z.B. Speicherseitenverwaltung, Fehlerinterrupt-

Tabelle, Prozesstabelle), aber auch Tabellen die Auskunft über Hardware geben (z.B. IRQ-Tabelle).

Nun wird auch der Scheduler gestartet und initialisiert. Der Scheduler ist für das Aufteilen von Prozessorzeit auf die einzelnen, scheinbar gleichzeitig laufenden Prozesse verantwortlich und muss dadurch schon in einer sehr frühen Phase gestartet werden. Dann erfolgt neben dem Initialisieren des Bildschirms und der Grafikkarte ein weiterer Hardwaretest. Die Hardware wird nun mittels verschiedener Tests genauer untersucht als dies bereits beim POST erfolgt ist. Hier wird insbesondere das Ansprechen der Hardware über die installierten Treiber getestet. Tritt beim Testen eines Gerätes ein Fehler auf, so ist dieses Gerät später nicht mehr verfügbar.

Zu diesem Zeitpunkt wird der Prozessor auch vom privilegierten Modus (Kernel Modus) in den nicht privilegierten Modus (User Modus) geschaltet. Diese Modi spiegeln sich in der Ringstruktur des Prozessors wieder. Prozesse im User Modus (normalerweise Ring 3 bei einem Intel-Prozessor) dürfen nicht mehr direkt auf die Hardware zugreifen, sondern müssen den Zugriff über Prozesse die im Kernel Modus (normalerweise Ring 0 bei einem Intel-Prozessor) laufen regeln (Treiber).

Nach dem Umschalten in den User Modus wird der erste Prozess mit der PID 0 gestartet – der sogenannte Swapper-Prozess. Dieser ist für Multitasking-Funktionalität unentbehrlich und muss deswegen vor allen anderen Prozessen gestartet werden. Dann wird der Prozess ‚init' (PID 1) erzeugt und gestartet.



Init wird bei UNIX-Systemen oft als Vater aller Prozesse bezeichnet, da alle von nun an erzeugten Prozesse von init abgeleitet werden. Anders als bei regulären UNIX-Systemen wird in Linux jedoch ‚init' durch einen Aufruf der Systemfunktion ‚fork()' vom Swapper-Prozess (in Linux auch idle-Prozess genannt) abgeleitet. Korrekt gesehen ist in einem Linux System der idle-Prozess der Vater aller Prozesse.

Wenn der Prozess init mit seiner Arbeit beginnt, sucht er zunächst nach seiner Konfigurationsdatei mit dem Namen ‚/etc/inittab'. In dieser Datei werden Optionen und Einstellungen definiert, die dem init-Prozess mitteilen, welche Aktionen beim Starten ausgeführt werden sollen. Der Aufbau und die Struktur dieser Datei wird nun in weiterer Folge näher beschrieben.

Die Datei ‚/etc/inittab'

Die Datei `/etc/inittab`

- Wird vom init-Prozess als erstes abgearbeitet
- Aufbau der Datei
  - `ID:Runlevel:Aktion:Prozess` wobei
    - ID: Vierstellige eindeutige Zeichenfolge
    - Runlevel: für welchen Runlevel die Aktion auszuführen ist
    - Aktion: Wie wird der Prozess gestartet
    - Prozess: Das zu startenden Programm oder Skript

Die Datei ‚/etc/inittab' enthält die Konfigurationsdaten des init-Prozesses und wird somit als erstes abgearbeitet. Der Aufbau der Datei gestaltet sich so, dass gleichartige Zeilen die beim Starten auszuführenden Aktionen beschreiben. Jede Zeile besteht aus 4 Feldern, getrennt durch einen Doppelpunkt, und besitzt dabei folgenden Aufbau:

```
ID:Runlevel:Aktion:Prozess
```

Das Feld ‚ID' beinhaltet eine fast beliebige, maximal vierstellige Zeichenfolge. Jede ID darf aber nur einmal vergeben werden und sollte bei Login-Prozessen (‚getty'-Kommando) der Nummer des Terminals auf dem dieser gestartet wird entsprechen.

Das Feld ‚Runlevel' spezifiziert die Konfiguration, in der die angegebene Aktion auszuführen ist. Ein Runlevel bestimmt dabei einen speziellen Zustand (oder Konfiguration) des UNIX-Systems. Da die Bezeichner eines Runlevels aus nur einem Zeichen bestehen, werden diese einfach hintereinander geschrieben, wenn die gleiche Aktion für mehrere verschiedene Runlevels ausgeführt werden soll. Jedoch wird das Feld bei bestimmten Aktionen (‚sysinit', ‚boot' oder ‚bootwait') ignoriert.

Im Feld ‚Aktion' wird angegeben, wie der Prozessstart zu erfolgen hat. Die hier möglichen Optionen werden noch später in diesem Kapitel ausführlicher behandelt.

Das Feld ‚Prozess' enthält letztendlich das zu startende Skript oder Programm.

Als erstes durchsucht der init-Prozess die Tabelle ‚/etc/inittab' nach einem Eintrag, in dessen Aktionsfeld sysinit, boot oder bootwait steht. Solche Einträge kennzeichnen Prozesse, die bereits in frühen Phasen des Systemstarts geladen werden müssen (z.B. Überprüfen und Einbinden der Dateisysteme, Setzen der Systemzeit, Initialisierung des Swap-Speichers).



Das Verhalten des init-Prozesses und die Interpretation der Datei ‚/etc/inittab' ist stark distributionsabhängig. Genauere Informationen entnehmen Sie bitte den jeweiligen Handbüchern oder manual pages.

Ein wichtiger Eintrag ist die Zeile, die als Aktion den Wert ‚initdefault' enthält. Diese Zeile kommt nur einmal vor und hat eine spezielle Bedeutung. Diese Zeile UNIX an, in welches Runlevel (also welche Konfiguration) beim Starten automatisch gewechselt werden soll. Die angegebene ID und ein eventuelles Kommando haben keinen Einfluss auf den Bootvorgang. Im folgenden wird eine initdefault-Zeile gezeigt, die standardmäßig im Runlevel 5 (graphischer Modus) startet. Entgegen der anderen Zeilen darf hier nur ein einziger Runlevel angegeben werden.

```
id:5:initdefault:
```

Für jeden definierten Runlevel muss ein Eintrag in der Datei ‚/etc/inittab' vorhanden sein. Dabei ist die Definition der einzelnen Runlevel stark distributionsabhängig. In letzter Zeit wird jedoch zunehmend der Trend beobachtet, dass die Distributionen die gleichen Runlevel für die gleichen Betriebssystemzustände hernehmen.

Funktion von Runlevels

Was ist ein Runlevel

- Runlevel definiert bestimmten Zustand eines UNIX-Systems (aktive Prozesse, die beim Bootvorgang initialisiert wurden)
  - z.B. in Netzwerk-Runlevel zahlreiche Dämonen (httpd), die in Runlevel ohne Netzwerk nicht notwendig sind
- Prinzipiell Runlevel 0-9, a,b,c,s verfügbar
  - 0-6 mit bestimmten Funktionalitäten versehen
  - Rest stark Distributionsabhängig
    - Meist 7-9 und a-c nicht verwendet, s Single-User-Modus für Superuser
- Runlevel-Konstellation ist konfigurierbar

Runlevel definieren einen bestimmten Zustand oder eine bestimmte Konfiguration innerhalb eines UNIX-Systems. Prinzipiell unterscheiden sich Runlevel voneinander durch die An- oder Abwesenheit bestimmter Hintergrund- und Dämonenprozesse. Runlevel können zur Laufzeit beliebig geändert werden. Dieses Konzept erlaubt es, das System ohne großen Aufwand von einer Konfiguration in eine andere zu bringen, ohne das System selbst neu starten zu müssen.

Prinzipiell stehen die Runlevel ‚0' bis ‚9', ‚a', ‚b', ‚c' und ‚s' zur Verfügung. Welche Konfiguration in welchem Runlevel abgebildet ist, ist dabei stark Distributionsabhängig. In letzter Zeit ist jedoch zunehmend der Trend erkennbar, dass verschiedene Distributionen Runlevel angleichen. So sind Runlevel 0-6 mit bestimmten Funktionalitäten versehen, die Runlevel 7-9 und die Spezialrunlevel a, b und c werden nicht verwendet. Der Spezialrunlevel s wird häufig als ‚Single User Mode' verwendet und dient dem Superuser zur Durchführung von Administrationsarbeiten und Systemwiederherstellungsarbeiten.



Runlevel sind hochgradig konfigurierbar und es steht in der Verantwortung des Systemadministrators zu bestimmen, welche Dienste in welchem Runlevel aktiv oder inaktiv sein sollen.

Vordefinierte Runlevel
(SuSE)

- 0 Halt
- s Single User Mode (nicht direkt benutzt)
- 1 Single User Mode
- 2 Multi User ohne Netzwerk
- 3 Multi User mit Netzwerk
- 4 frei
- 5 Multi User mit Netzwerk und graphischer Oberfläche
- 6 Reboot

Die oben abgebildete Folie zeigt die von vielen Distributionen (z.B. SuSE, RedHat) angenommene Standardkonfiguration von Runlevels.

Der Runlevel 0 beendet alle aktiven Prozesse in einem System und kommt somit dem Niederfahren des Rechners gleich.

Im Runlevel 1 – dem sogenannte Single User Mode – wird UNIX in einer minimalen Form gestartet (es werden nur die notwendigsten Hintergrundprozesse gestartet) für etwaige Reparaturen. In diesem Runlevel wird der login-Prozess nie gestartet, sondern direkt die Shell mit root-Rechten. Dieser Modus bietet sich für Bootdisketten für etwaige Reparaturen als Standardrunlevel an. Den Runlevel selbst darf nur der Superuser (‚root') ändern, jedoch sollte eine Bootdiskette, die in diesen Modus startet sicher verstaut sein.

Runlevel 2 bietet bereits die Möglichkeit, dass sich mehrere Benutzer an einem System anmelden können, jedoch ist keine Netzwerkunterstützung vorgesehen. Benutzer können sich nur lokal am Rechner anmelden. Die Netzwerkunterstützung wird im Runlevel 3 hinzugefügt. Runlevel 5 ergänzt Runlevel 2 zusätzlich noch durch die graphische Benutzeroberfläche X-Window.

Runlevel 6 realisiert einen Neustart des Systems. Zuerst werden alle aktiven Prozesse und Dämonen beendet, dann wird jedoch – im Gegensatz zum Runlevel 0 – der Rechner angewiesen, neu zu starten.

Der Spezialrunlevel s ist gleich definiert wie Runlevel 1.

Aktionen für Runlevels

Aktionen für Runlevels$_1$

- respawn: Wenn Prozess terminiert, wird selbiger nochmals neu gestartet
- wait: bevor ‚init' fortsetzt, muss Prozess terminieren
- once: wird ausgeführt, wenn in Runlevel gewechselt wird
- boot: beim Booten ausgeführt
- bootwait: gleich wie boot, jedoch wartet ‚init' auf Terminierung des Prozesses
- off: Ignorieren der Zeile

In der Datei ‚/etc/inittab' kann neben den anderen Feldern auch noch eine Aktion angegeben werden, die Aufschluss darüber gibt, wie das angegebene Programm in einem Runlevel gestartet werden soll:

- respawn: Diese Aktion bewirkt, dass ein Prozess sofort neu gestartet wird, wenn dieser terminiert.

- wait: Diese Aktion weist den init-Prozess an, den nächsten Prozess erst zu starten, wenn dieser fertig ist. Diese Art von Aktion wird dann gebraucht, wenn ein Prozess erst vollständig abgearbeitet werden muss (z.B. Dateisystemprüfung) bevor der Nächste starten soll.

- once: Hier wird der angegebene Prozess ganz normal gestartet und nicht auf dessen Beendigung gewartet. Stattdessen wird der nächste, in Frage kommende Prozess gesucht und gestartet.

- boot: Hier definierte Prozesse werden unabhängig von einem Runlevel bereits beim Hochfahren des Systems ausgeführt. Ein eventueller Wert im Feld Runlevel wird bei dieser Aktion ignoriert.

- bootwait: Diese Aktion vollzieht das gleiche wie die ‚boot'-Aktion, jedoch wartet der init-Prozess auf die Beendigung des angegebenen Prozesses bevor der nächste in Frage kommende Prozess gestartet wird.

- off: Diese Aktion bewirkt das vollständige ignorieren der Zeile. Der angegebene Prozess wird nicht ausgeführt.

Aktionen für Runlevels$_2$

- ondemand: Prozess wird ausgeführt, wenn einer der „ondemand runlevels“ aufgerufen wird
- initdefault: Runlevel, in welchem nach Booten gewechselt werden soll.
- sysinit: Wird beim Booten ausgeführt, jedoch vor ‚boot‘ oder ‚bootwait‘
- powerwait: Bei Stromausfall (USV übernimmt), ‚init‘ wartet bis Kommando abgearbeitet

- ondemand: Der angegebene Prozess wird nur dann ausgeführt, wenn es sich bei dem Runlevel um ein ‚ondemand runlevel’ (Spezialrunlevel) handelt. Diese Runlevel tragen die Bezeichnung ‚a-c’ oder ‚s’.

- initdefault: Diese Aktion kennzeichnet eine spezielle Zeile innerhalb der Datei ‚/etc/inittab’. Hier wird sowohl die angegebene ID als auch ein eventuell angegebenes Kommando ignoriert. Ist das System mit dem Bootvorgang fertig, so wird nach einer Zeile gesucht, die als Aktion den Wert ‚initdefault’ trägt. Der Runlevel, der hier spezifiziert ist, wird als Standard-Runlevel im System eingetragen. Nach dem Booten wechselt das System automatisch in diesen Standard-Runlevel.

- sysinit: Hier definierte Prozesse werden unabhängig von einem Runlevel bereits beim Hochfahren des Systems ausgeführt. Diese Prozesse werden aber *VOR* boot oder bootwait-Aktionsprozessen ausgeführt.

- powerwait: Prozesse dieser Aktion werden unabhängig von einem Runlevel ausgeführt, wenn eine Unterbrechungsfreie Stromversorgung (USV) die Spannungsversorgung des Rechners übernimmt (z.B. bei Stromausfall). Der init-Prozess wartet, bis das Kommando abgearbeitet wurde, bevor das nächste in Frage kommende Programm gestartet wird.

Aktionen für Runlevels$_3$

- powerfail: Bei Stromausfall (USV übernimmt),
- powerokwait: Wenn Strom wieder da ist, ‚init' wartet!
- powerfailnow: Wenn Batterie der USV aufgebraucht ist
- ctrlaltdel: Wenn die Tastenkombination <*Ctrl*>+<*Alt*>+<*Entf*> gedrückt wurde
- kbrequest: Beim drücken einer speziellen Tastenkombination (SysReq)

- Prinzipiell: „`man inittab`" liefert distributions-abhängige Information

- powerfail: Diese Aktion ähnelt der Aktion ‚powerwait', jedoch wartet der init-Prozess nicht auf Beendigung des angegebenen Prozesses sondern beginnt sofort mit der Ausführung des nächsten in Frage kommenden Programms.

- powerokwait: Prozesse dieser Aktion werden unabhängig von einem Runlevel ausgeführt, wenn nach einem Übernehmen der USV die normale Spannungsversorgung wiederhergestellt worden ist und die USV nicht mehr notwendig ist. Jedoch wartet der init-Prozess auf Beendigung des Prozesses, bevor der nächste in Frage kommende Prozess gestartet wird.

- powerfailnow: Prozesse dieser Aktion werden unabhängig von einem Runlevel ausgeführt und zwar dann, wenn bereits die USV die Spannungsversorgung übernommen hat, jedoch am Ende ihrer Kapazität ist. Normalerweise wird in dieser Zeile das System ordnungsgemäß runtergefahren (im Prinzip wird das System angewiesen, in den Runlevel 0 zu wechseln).

- ctrlaltdel: Prozesse dieser Aktion werden unabhängig von einem Runlevel ausgeführt und zwar dann, wenn die Tastenkombination <STRG>+<ALT>+<ENTF> gedrückt wurde.

- kbrequest: Hier können spezielle Tastenkombinationen behandelt werden.

Prinzipiell ist der Umfang der angebotenen Funktionen von der Distribution und dem Derivat abhängig. Informationen erhält man über die Manual-Pages von ‚inittab' („man inittab").

Konfiguration der Runlevel

Runlevel-Konfiguration

- Jeder Runlevel in Dateisystem eigenes Verzeichnis (Namensgebung rc*x*.d, wobei *x* der jeweilige Runlevel ist)
  - Verzeichnis bei SuSE: `/sbin/init.d/rc`*x*`.d`
- Jedes Skript, das in Runlevel zu starten ist, erscheint als Link im Runlevel-Verzeichnis
- Namenskonvention für Links (3 Komponenten)
  - K/S
    - K Skript ist ein Stop-Skript (Kill)
    - S Skript ist ein Start-Skript (Start)
  - Priorität
    - 00-99 00 wird vor 99 abgearbeitet
  - Eigentlicher Name des Links
    - Frei wählbar, sprechend

Wenn in einen Runlevel gewechselt wird, werden eine Reihe von Prozessen gestartet, andere beendet. Dies erfolgt im Normalfall durch den Aufruf von zahlreichen Skripten. Um diese Skripte nicht für jeden Runlevel neu schreiben zu müssen, existiert für jeden Runlevel ein eigenes Verzeichnis. In dieses Verzeichnis werden dann Links zu den zu startenden Skripts abgelegt. Die Skripte selbst werden meist in ein gemeinsames Verzeichnis gespeichert, dass sich eine Ebene über den einzelnen Runlevel-Verzeichnissen befindet. Wird dann in den Runlevel gewechselt, so scheint es, als wären die notwendigen Skripte in dem Runlevel-Verzeichnis enthalten.

Wo sich die Runlevel-Verzeichnisse und das Skript-Verzeichnis befindet, ist wieder von der eingesetzten Distribution abhängig. Aktuelle Versionen verwenden größtenteils die selben Verzeichnisse. Die gemeinsamen Skripte werden dabei in das Verzeichnis ‚/etc/rc.d' gespeichert. Im selben Verzeichnis existiert dann für jeden Runlevel ein eigenes Unterverzeichnis, dass der Namenskonvention ‚/etc/rc.d/rc*x*.d' (*x* steht für die Nummer oder den Buchstaben des Runlevels) folgt.

Wie schon zuvor erwähnt, werden beim Wechseln in einen Runlevel gewisse, darin befindliche Skripts ausgeführt. Damit das System weiß, welche Skripts auszuführen sind, müssen mehrere Voraussetzungen erfüllt sein. Zum einen müssen überhaupt Skripte im Verzeichnis existieren. Da oftmals in verschiedenen Runlevel die gleichen Prozesse betroffen sind, befinden sich in den Runlevel nur mehr Links auf die korrespondierenden Dateien im Skriptverzeichnis. Zum anderen müssen die Dateinamen einer bestimmten Konvention folgen, damit das System weiß, welche Dateien in welcher Reihenfolge auszuführen sind. Immerhin setzen manche Dienste auf die Funktionalität anderer Dienste aus und das System muss dafür

sorgen, dass eine gewisse Reihenfolge eingehalten wird. Deswegen besteht der Dateiname in den Runlevel-Verzeichnissen aus drei Komponenten:

1. Der erste Buchstabe beschreibt, ob es sich beim Skript um ein Startskript (ein Prozess wird gestartet) oder um ein Stopskript (ein Prozess wird beendet) handelt. Der Buchstabe ‚S' steht für ein Startskript, der Buchstabe ‚K' für ein Stopskript (Kill-Script).

2. Dann folgt eine zweistellige Nummer, die eine Priorität des Skripts angibt. Dabei sind Werte von ‚00' bis ‚99' möglich. Niedrige Werte werden früher ausgeführt als hohe Werte. Somit kann mit der Nummer die Ablaufreihenfolge der Skripts beeinflusst werden. Wichtig ist, dass für die Ablaufbestimmung ein Zeichenkettenvergleich hergenommen wird und deswegen die Konvention einer zweistelligen Nummer unbedingt eingehalten werden muss (z.B. ist die Zeichenkette ‚1' bei einem Zeichenkettenvergleich größer als ‚02', jedoch ist ‚01' kleiner als ‚02').

3. Der Rest ist eine beliebige Zeichenkette, die im Normalfall den Namen des Skripts oder dessen Funktion widerspiegelt. Dieser sollte treffend und aussagekräftig gewählt werden, da dies die Administration erheblich erleichtert.



In vielen Fällen ist es nicht mehr notwendig, solche Start- und Stopskripts selbst zu erstellen, da die heutzutage gebräuchlichen Installationswerkzeuge diese Aufgabe dem Benutzer abnehmen. Yast (SuSE) und linuxconf (RedHat) sind in der Lage, entsprechende Skripts selbst zu erstellen und auch wieder zu entfernen, wenn der entsprechende Dienst vom System deinstalliert wird.

Arbeitsweise des Skriptes rc

Arbeitsweise rc-Script

- Resource Control (rc) Skript ist für Wechsel der Runlevel zuständig
- rc-Skript bekommt als Parameter neuen Runlevel
- Zwei Schritte
  - Zuerst im alten Runlevel alle K-Skripts ausführen (diese Prozesse werden gestoppt)
  - In neuen Runlevel wechseln und alle S-Skripts ausführen (notwendige Prozesse werden gestartet)
- Wechsel des Runlevels
  - z.B. „`telinit 3`“ oder „`init 3`“ wechselt in den Runlevel 3

Das Resource Control (rc) Skript ist für das Ausführen der sich in einem Runlevel-Verzeichnis befindlichen Skripte verantwortlich. Dabei nimmt das rc-Skript als Parameter den zu startenden, neuen Runlevel. Bevor die Dienste des neuen Runlevel gestartet werden können, müssen zuerst die des Alten gestoppt werden. Dazu bestimmt das Skript zunächst einmal den aktuellen Runlevel und wechselt in dessen Verzeichnis. Dort werden nun alle Stopskripte (Skripte, die mit dem Buchstaben ‚K' beginnen) der Reihe nach ausgeführt. Diese Stopskripte bekommen als Parameter das Argument ‚stop' mit. Danach wird in das Verzeichnis des neuen Runlevels gewechselt und die dort befindlichen Startskripte (Skripte, die mit dem Buchstaben ‚S' beginnen) ausgeführt. Diese bekommen als Parameter das Argument ‚start'.

ⓘ Im Prinzip sind ja die in einem Runlevel-Verzeichnis liegenden Skripte Links auf allgemeine Skripte, die eine Ebene höher liegen. Diese kennen die Parameter ‚start' zum Starten des Prozesses, ‚stop' zum Beenden des Prozesses, ‚restart' zum Neustart des Prozesses und ‚status' zum Anzeigen von Statusinformationen. Je nach Implementierung können die Skripte auch mehrere Parameter verstehen.

Ein Wechsel des Runlevels ist immer möglich. Denkbar ist, dass der Systemadministrator das System im Single User Modus startet, irgendwelche Änderungen an der Konfiguration vornimmt und dann in den normalen Modus wechselt. Ein solcher Wechsel ist wesentlich schneller vollzogen als ein Neustart. Der Runlevel kann mit dem ‚telinit'-Kommando gewechselt werden oder es wird ‚init' mit einem Parameter aufgerufen. Beide Prozesse erwarten das zu startende Runlevel als Argument („telinit 3“ bzw. „init 3“ wechselt in Runlevel 3). Den Runlevel darf jedoch nur der Benutzer ‚root' ändern!