

Betriebssysteme I

Bachelor-/Diplomstudiengang Wirtschaftsinformatik
Wintersemester 2015/16

- ▶ 2 SWS Vorlesung + 2 SWS Praktikum
(Praktikumsbetreuung:
Prof. Fritzsche/Herr T. Schubert/Herr R. Ringel)
- ▶ 1 Beleg als Voraussetzung zur Zulassung zur Prüfung
- ▶ Prüfung: Klausur (90 min, ohne Unterlagen)

Prof. Dr. Hartmut Fritzsche
Fakultät Informatik/Mathematik
Büro: Z 342
Sprechzeit: Mittwoch 15.10 – 16.40 Uhr

Tel. (0351) 462 2606
fritzsche@informatik.htw-dresden.de
www.informatik.htw-dresden.de/~fritzschi

Betriebssysteme I

Studiengang Wirtschaftsinformatik (042)

Wintersemester 2015/16

Inhalt

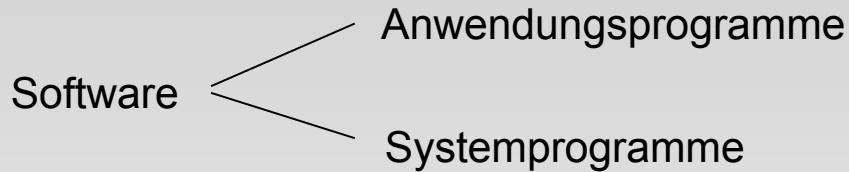
- | | | |
|------------------------------|---|---|
| 1. Einführung |  | |
| 2. Grundlagen |  |  |
| 3. Spezielle Werkzeuge |  | |
| 4. Betriebssystemkonzepte |  | |
| 5. Verteilte Betriebssysteme |  | |

Inhalt

- Einführung
 - Was ist ein Betriebssystem ?
 - Klassifizierung und Standardisierung
- Grundlagen für die Arbeit mit UNIX (LINUX)
 - Zugang zum UNIX-System
 - Kommandosprache und Kommandointerpretation
 - Das LINUX-Dateisystem
 - Arbeit mit Textdateien
 - Dateischutz
 - Programme und Prozesse
 - Kommandoprozeduren
- Spezielle Werkzeuge unter UNIX
 - awk
 - find
 - sort
 - join
 - tar
 - Bildschirmeditor vi
 - Die Programmiersprache C
- Betriebssystem-Konzepte
 - Starten eines UNIX-Systems und Anmelden eines Benutzers
 - Prozeßverwaltung
 - Dateisysteme
 - Das Ein-/Ausgabesystem
 - Das X-Window-System
- Verteilte Betriebssysteme
 - Grundkonzepte
 - Kommunikation in verteilten Systemen

1. Einführung

1.1 Was ist ein Betriebssystem?

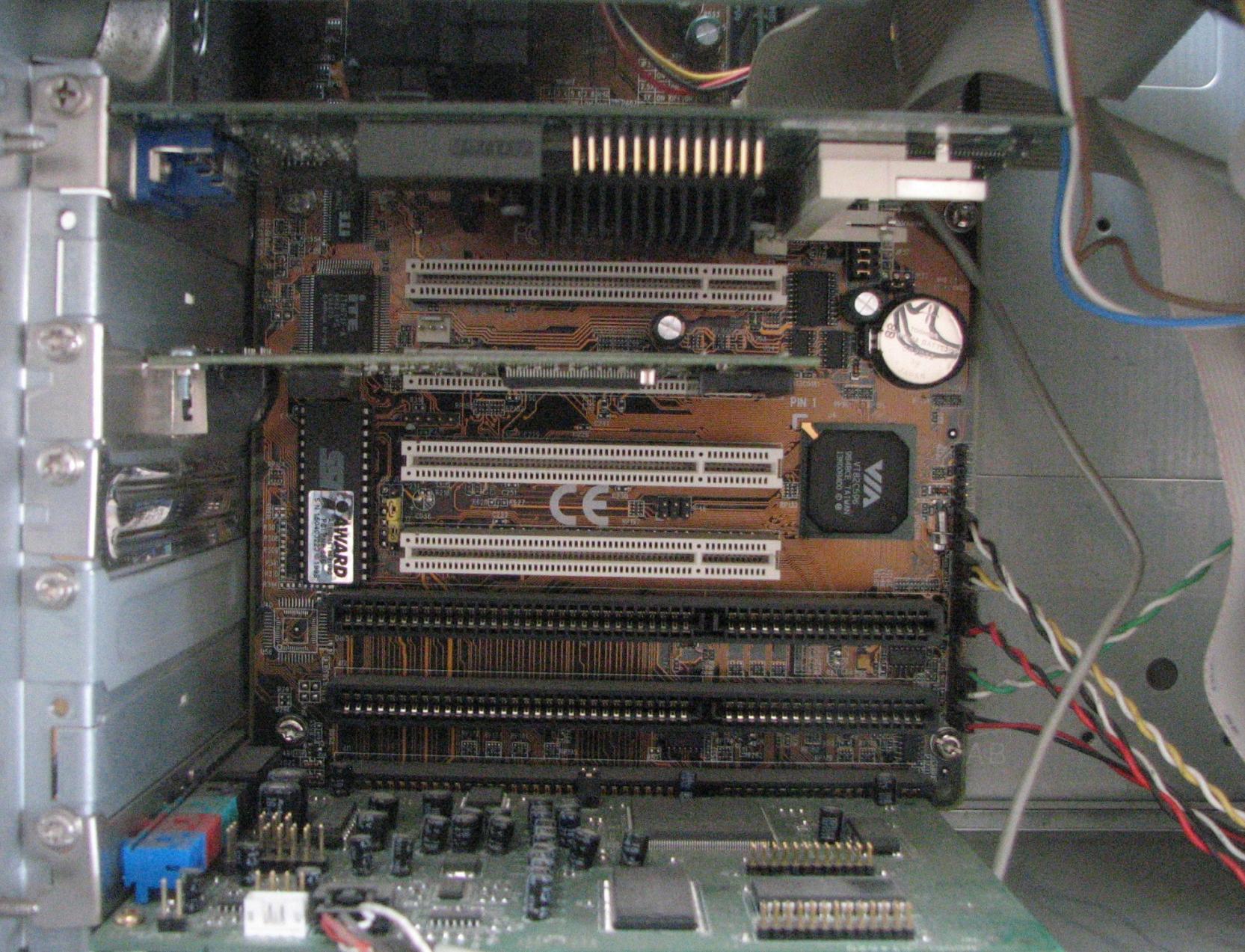


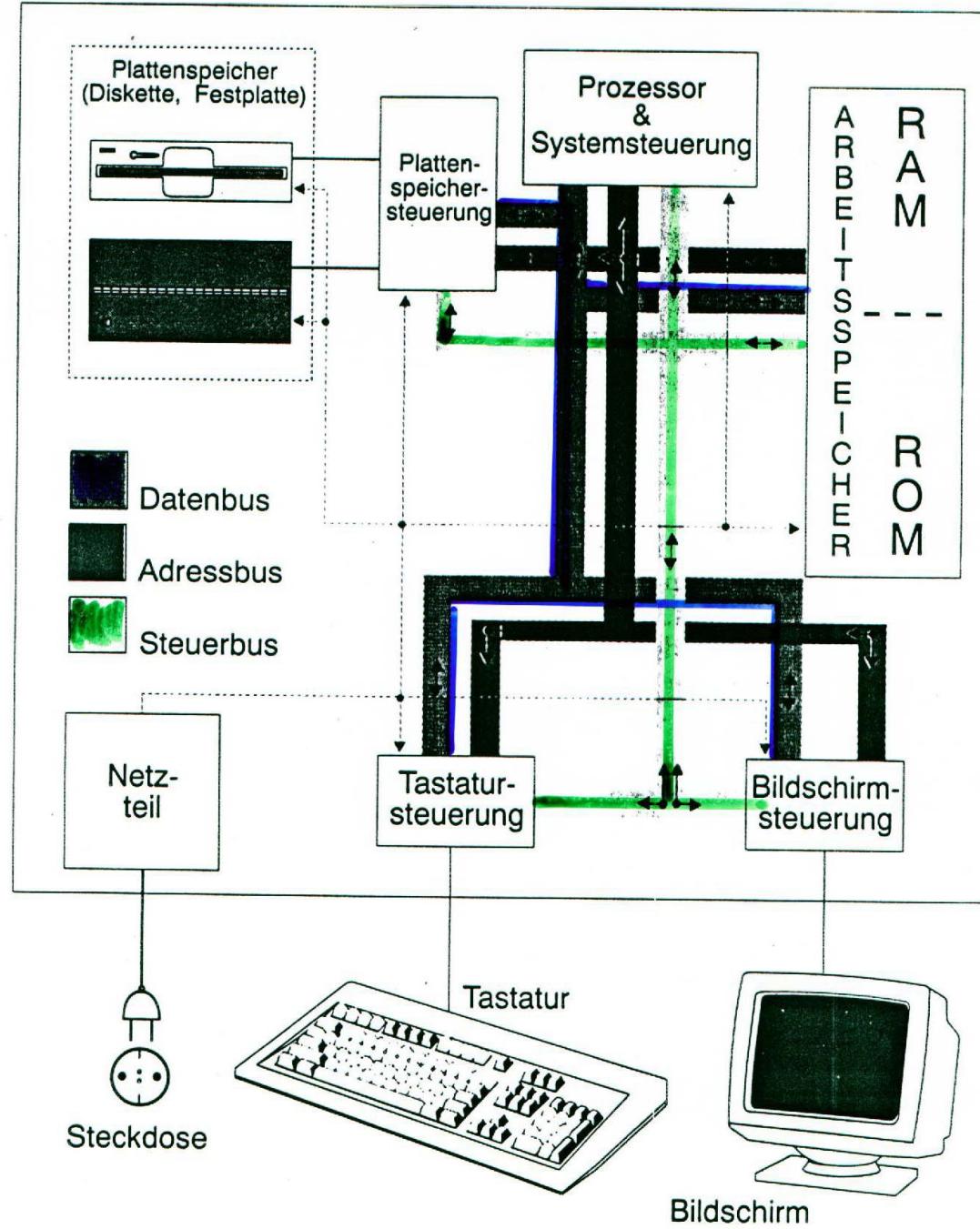
Betriebssystem: grundlegendstes Systemprogramm

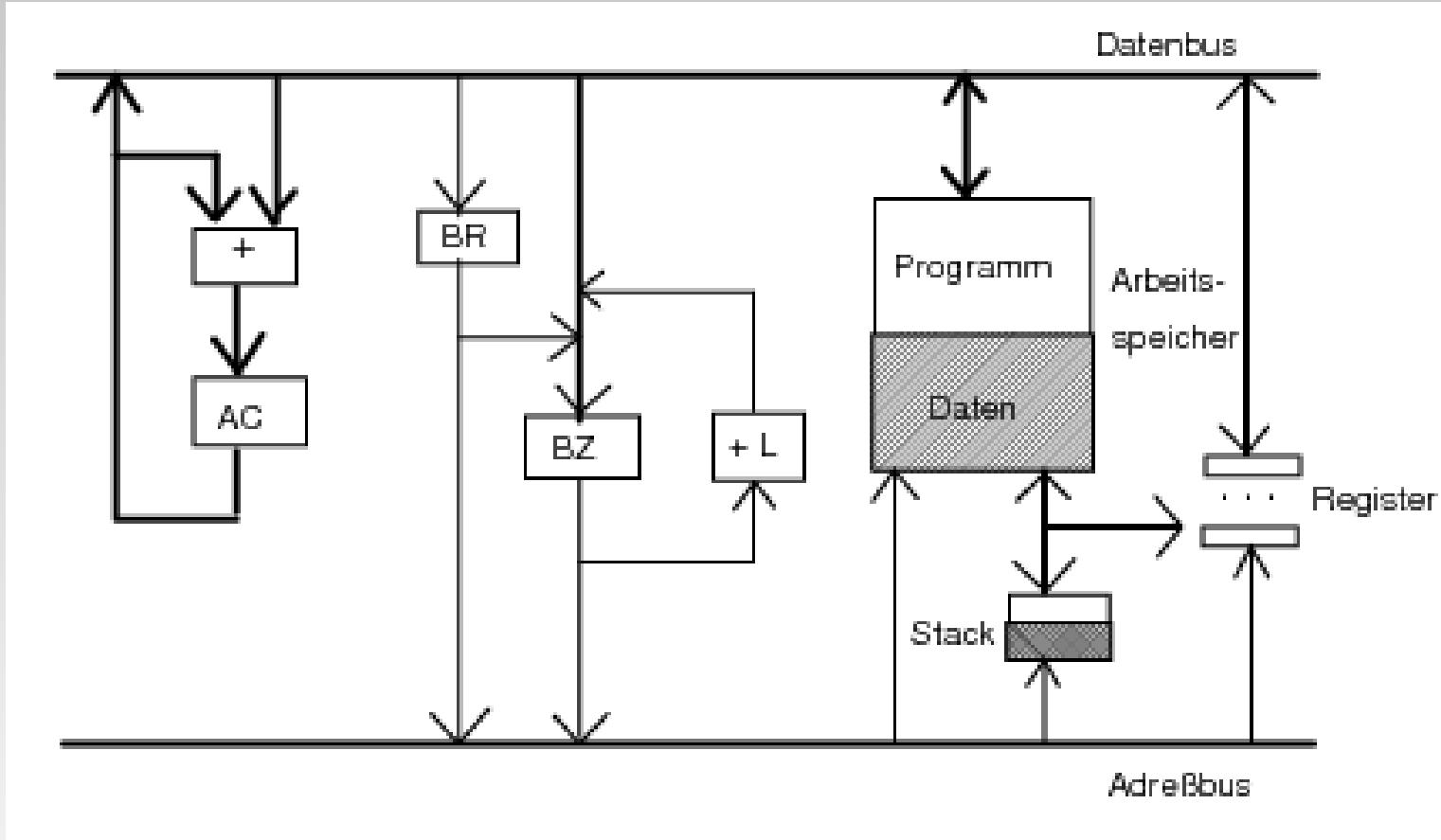
$$\text{Rechnersystem (RS)} = \text{Rechenanlage (RA)} + \text{Betriebssystem (BS)}$$

Der Nutzer muss von der Komplexität der Hardware abgeschirmt werden.

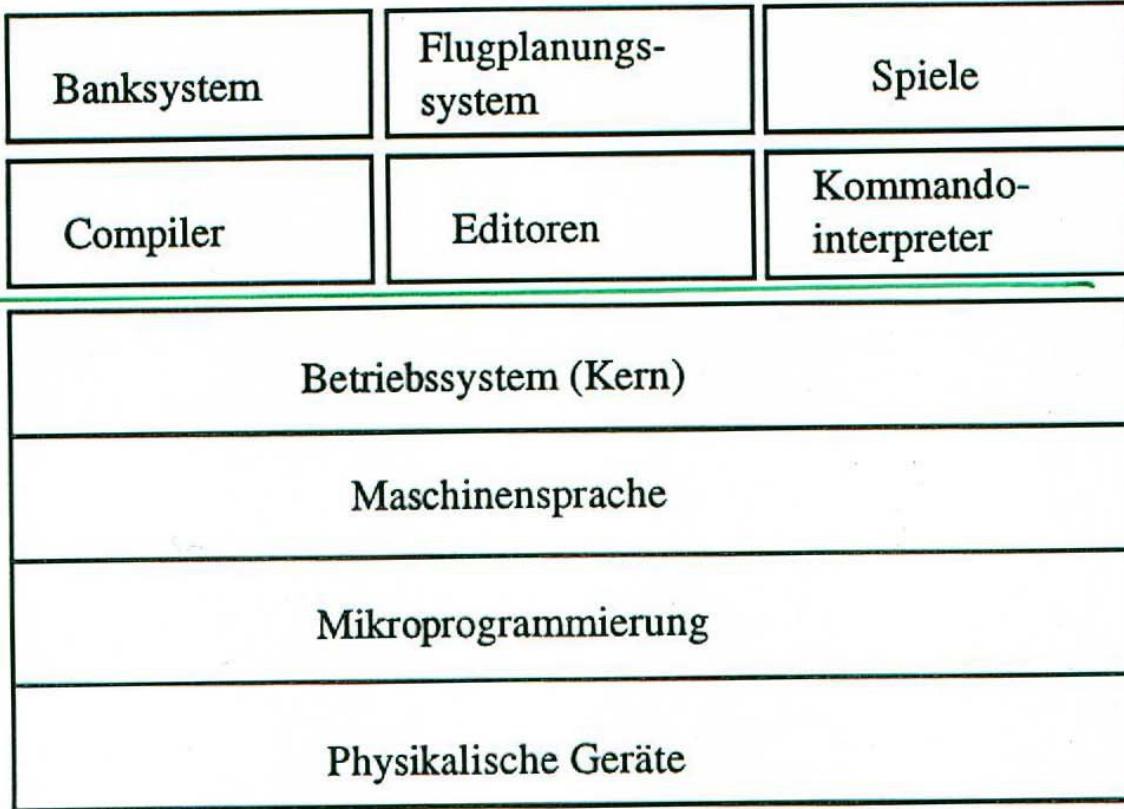
Dem Nutzer wird eine **virtuelle Maschine** (= BS) angeboten, die einfacher zu verstehen und zu programmieren ist







Anwendungs-
programme



System-
programme

Hardware

Kommando -
← Spreche

← C

Supervisor - Modus

Sichten bei der Betrachtung von BS:

Top-down : Nutzung der **virtuellen Maschine** durch Benutzer

Bottom-up : untersuchen, wie ein BS eine virtuelle Maschine realisiert

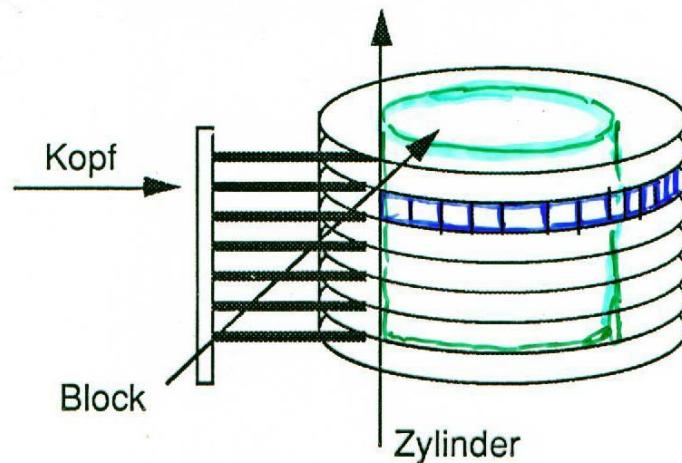
(BS muss alle Bestandteile eines komplexen Systems verwalten:
Zuteilung von Prozessoren, Speichern, I/O-Geräten für
konkurrierende Programme – „**Betriebsmittelverwaltung**“
registrieren – realisieren – vermitteln)

Benutzer kommunizieren mit einem BS über

- ➡ eine grafische Benutzeroberfläche (GUI)
- ➡ Kommandos in einer einfachen Programmiersprache (Shell, Kommando-Interpreter)
- ➡ eine programmiersprachliche Schnittstelle:
Systemaufrufe in C, BS selbst in C programmiert

Dateien auf Plattspeichern

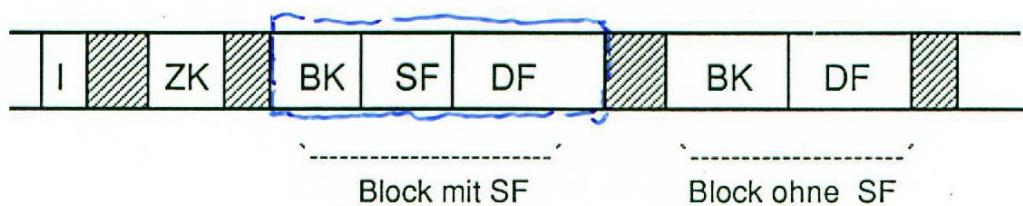
- Direktzugriff auf Böcke gewährleisten
- Blockadresse wird im Block mit abgespeichert



`n = read(DD, adr, anz);`

anz viele Zeichen werden aus der durch **DD** identifizierten Datei in den an der Adresse **adr** beginnenden Speicherbereich gelesen.

Spureinteilung :



I : Indexmarke (Spuranfang)

ZK : Spuradresse (Zylinder- u. Kopfnummer)

BK : Blockkennung einschließlich Längenfeld

SF : Schlüsselfeld

DF : Datenfeld

1.2 Klassifizierung und Standardisierung von BS

1. Generation (vor 1955) : kein BS ! J. von Neumann, K. Zuse, N. J. Lehmann



2. Generation (1955 – 65) : jeweils 1 Programm zur Abarbeitung im HS, „Job“,
CPU wartet die meiste Zeit auf E/A, Stapelverarbeitung



3. Generation (1965 – 80) : HS in Partitionen geteilt, in jeder Partition kann
ein Programm abgearbeitet werden
(Multiprogramming)



„Spooling“-Technik

(Simultaneous Peripheral Operation on Line)



Timesharing, Prozesskonzept,

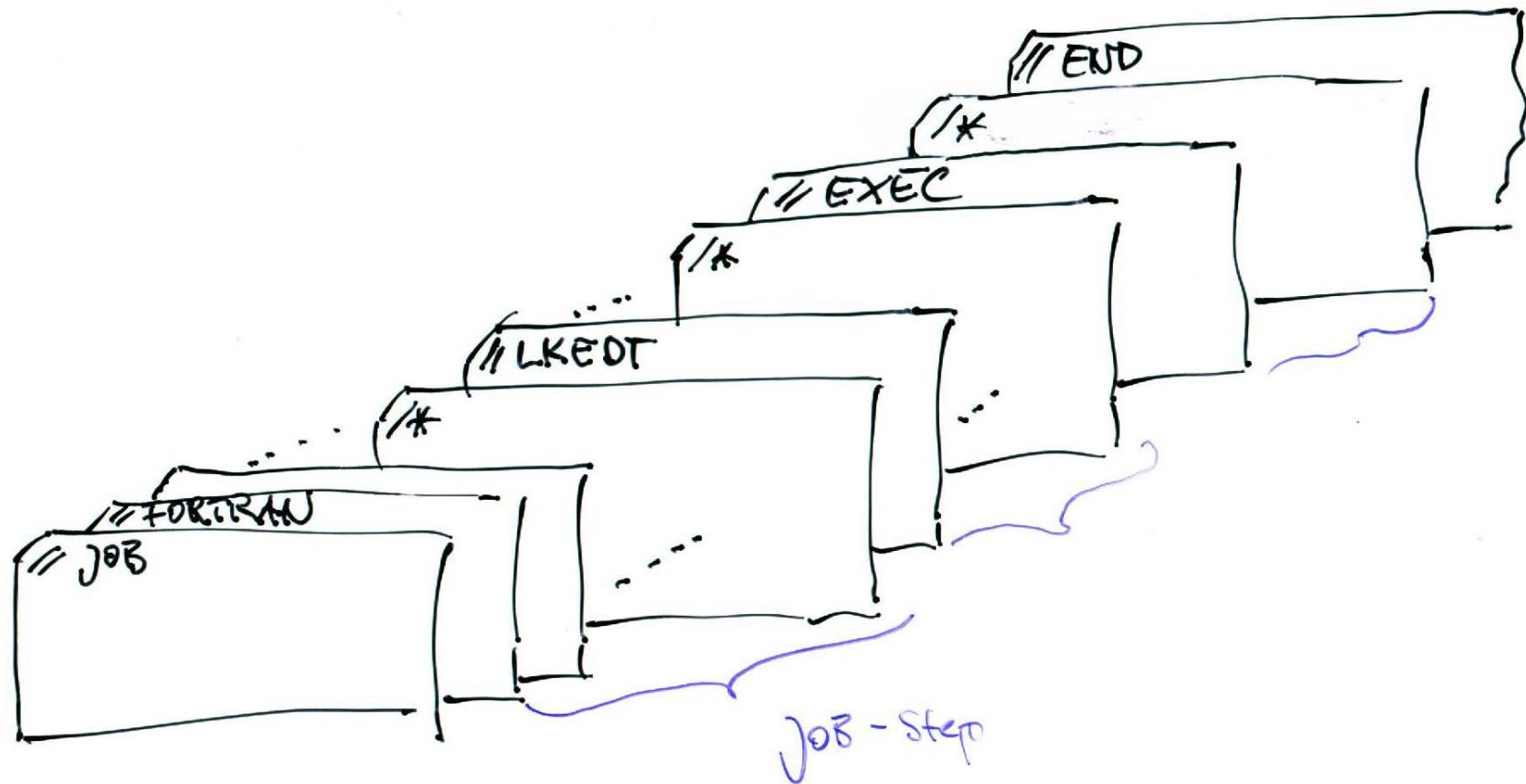
schneller online-Zugang für Terminals +
Stapeljobs im Hintergrund

4. Generation (1980 -90) : Netzwerk-BS, verteilte BS





Job: Folge mehrerer Programme zusammen abarbeitbar:



Netzwerkbetriebssystem:

- ▶ auf jedem Rechner läuft ein eigenes Betriebssystem, die Nutzer sind sich der Existenz vieler Rechner bewusst, sie können
 - Zugang auf entfernte Rechner haben
 - Dateien von einem Rechner zu anderem kopieren

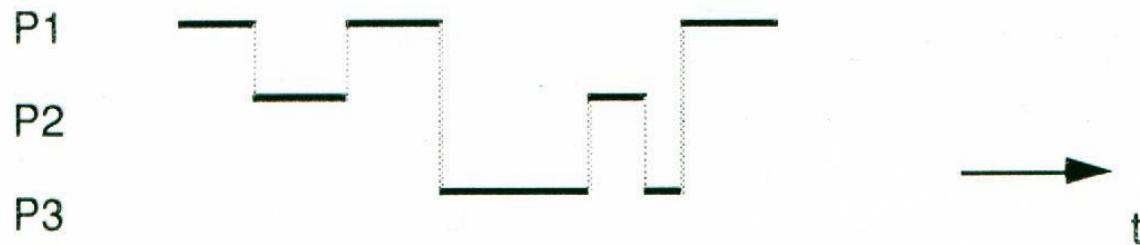
„Echte“ Verteilte (Betriebs-) Systeme

- ▶ Das BS sieht für den Benutzer aus wie ein traditionelles Einprozessorsystem, es besteht aber aus vielen Prozessoren. Nutzer haben keine Kenntnis, wo Programme ausgeführt werden.

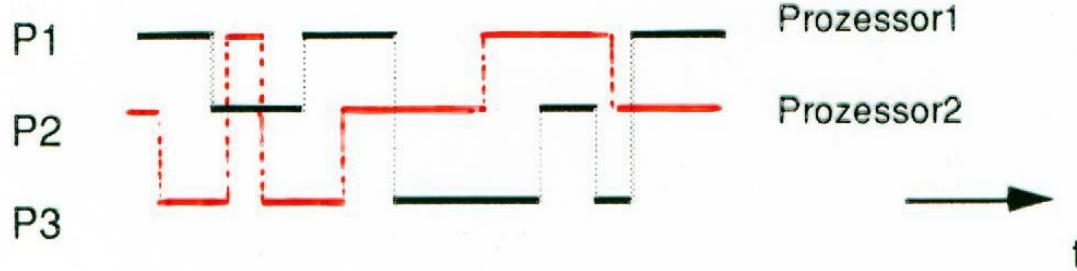


Prozeßumschaltung :

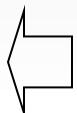
neue Situation: Mehr Prozesse als Prozessoren



Mehrprozeßbetrieb/
Zeitmultiplexbetrieb
(*multiprogramming*)



Mehrprozessorbetrieb



2.1 Zugang zum UNIX-System

Die Zugangskontrolle für einen Benutzer erfolgt durch das Programm `login`.

- Eingabe Benutzer-account: **Identifizierung** am System
- Eingabe Passwort: **Authentifizierung** am System

- Die Eingaben sind "Case-sensitiv", d.h. Groß- und Kleinbuchstaben werden unterschieden!
- Vorsicht vor „Spoofing“ – Programmen!

2.2 Kommandosprache und Kommandointerpretation

```
fritzsch@ilux150:~> date
Mi 17. Okt 16:39:54 CEST 2007
fritzsch@ilux150:~> Date
-bash: Date: command not found
fritzsch@ilux150:~>
```

Behandlung weiterer Kommandos im Praktikum:

```
who
clear
pwd
ls
cd
help
man
```

→ Besprechen von **Syntax** und **Semantik** von Kommandos

```
fritzs@ilux150:~> help
GNU bash, version 3.00.16(1)-release (i586-suse-linux)
These shell commands are defined internally. Type 'help' to see this list.
Type 'help name' to find out more about the function 'name'.
Use 'info bash' to find out more about the shell in general.
Use 'man -k' or 'info' to find out more about commands not in this list.
```

A star (*) next to a name means that the command is disabled.

```
*[DIGITS | WORD] [&]                                (( expression ))
. filename [arguments]                            :
[ arg... ]                                         [[ expression ]]
alias [-p] [name[=value] ... ]      bg [job_spec]
bind [-lpvsPVS] [-m keymap] [-f fi break [n]
builtin [shell-builtin [arg ...]]  caller [EXPR]
case WORD in [PATTERN [| PATTERN]. cd [-L|-P] [dir]
command [-pVv] command [arg ...]  compgen [-abcdefgjksuv] [-o option
complete [-abcdefgjksuv] [-pr] [-o continue [n]
```

fritzs@ilux150:~> cal

Oktober 2007

So	Mo	Di	Mi	Do	Fr	Sa
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

fritzs@ilux150:~>

```
fritzsch@ilux150:~> cal
```

Oktober 2007

So	Mo	Di	Mi	Do	Fr	Sa
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

```
fritzsch@ilux150:~> cal 11 2007
```

November 2007

So	Mo	Di	Mi	Do	Fr	Sa
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	

```
fritzsch@ilux150:~>
```

```
fritzsch@ilux150:~> cal
```

Oktober 2007

So	Mo	Di	Mi	Do	Fr	Sa
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

```
fritzsch@ilux150:~> cal 11 2007
```

November 2007

So	Mo	Di	Mi	Do	Fr	Sa
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	

```
fritzsch@ilux150:~> cal 13 2007
```

```
cal: illegaler Wert für Monat: benutzen Sie 1-12
```

```
fritzsch@ilux150:~>
```

```
fritzsch@ilux150:~> cal
```

Oktober 2007

So	Mo	Di	Mi	Do	Fr	Sa
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

```
fritzsch@ilux150:~> cal 11 2007
```

November 2007

So	Mo	Di	Mi	Do	Fr	Sa
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	

```
fritzsch@ilux150:~> cal 13 2007
```

```
cal: illegaler Wert für Monat: benutzen Sie 1-12
```

```
fritzsch@ilux150:~> cal 1 2 3
```

```
Aufruf: cal [-13smjyV] [ Monat ] Jahr
```

```
fritzsch@ilux150:~>
```

Sprache zur Beschreibung einer Sprache : **Metasprache**
Erweiterte Backus-Naur Form (EBNF)

```
fritzsch@ilux150:~> cal 1 2 3
Aufruf: cal [-13smjyV] [[Monat] Jahr]
fritzsch@ilux150:~>
```

Bedeutungstragende Einheiten „**Token**“ (Worte)

Metasprachliche Notation (Ausdrücke):

- Kommandobestandteile, die für sich selbst stehen: Teletype-Schrift
- metasprachliche Variablen: Monat oder *monat* oder <monat>
- wahlweise Angaben: [....]
- beliebig oft zu wiederholende Angaben: ...
- alternative Angaben: |
- Klammerung metasprachlicher Ausdrücke: { }

Syntax:

`cal [[argument1] argument2]`

oder

`cal [argument1 [argument2]]`

?

- „Mnemonische“ Bezeichnungen: *argument1* besser: *monat*
 argument2 *jahr*
- Syntaktische Regeln:

monat → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12
jahr →

Semantik: „Ausgabe eines Kalenders in der Form“

„ist definiert als ...“

Beispiel Kommandosprache:

Syntax

benutzereingabe → kommando [; kommando] ... <↓

| spezialform <↓

kommando → kommandoname [argument ...]

argument → optionsangabe [parameter ...]

Semantik

Die *kommandos* einer *benutzereingabe* werden nacheinander („sequentiell“) abgearbeitet.

Der *kommandoname* ist meist der Name eines Programmes. Der Kommandointerpreter sucht in diesem Fall das Programm mit dem Namen *kommandoname* und führt es aus, falls er es findet (die Shell suspendiert sich, bis das aufgerufene Programm terminiert).

Sogenannte "eingebaute" Kommandos führt die Shell allerdings selbst aus.

Das LINUX-Dateisystem

UNIX unterscheidet folgende Arten von Dateien:

- (gewöhnliche) Datei (*ordinary file*)
- Gerätedatei, auch als Spezialdateien bezeichnet (*special file*)
- Verzeichnis (*directory*)
- Pipe, Röhre (*pipe*)

Datei: Eine Datei ist aus Sicht des Betriebssystems UNIX eine auf einem externen Datenträger gespeicherte Folge von Bytes. Die Interpretation des Inhalts hängt von den Programmen ab, die eine Datei verarbeiten. Die Länge einer Datei kann in Blöcken (zu 512 oder 1024 Byte) oder in Bytes angegeben sein. Praktisch können Dateien nahezu beliebig lang sein (ca. 17 Milliarden Zeichen).

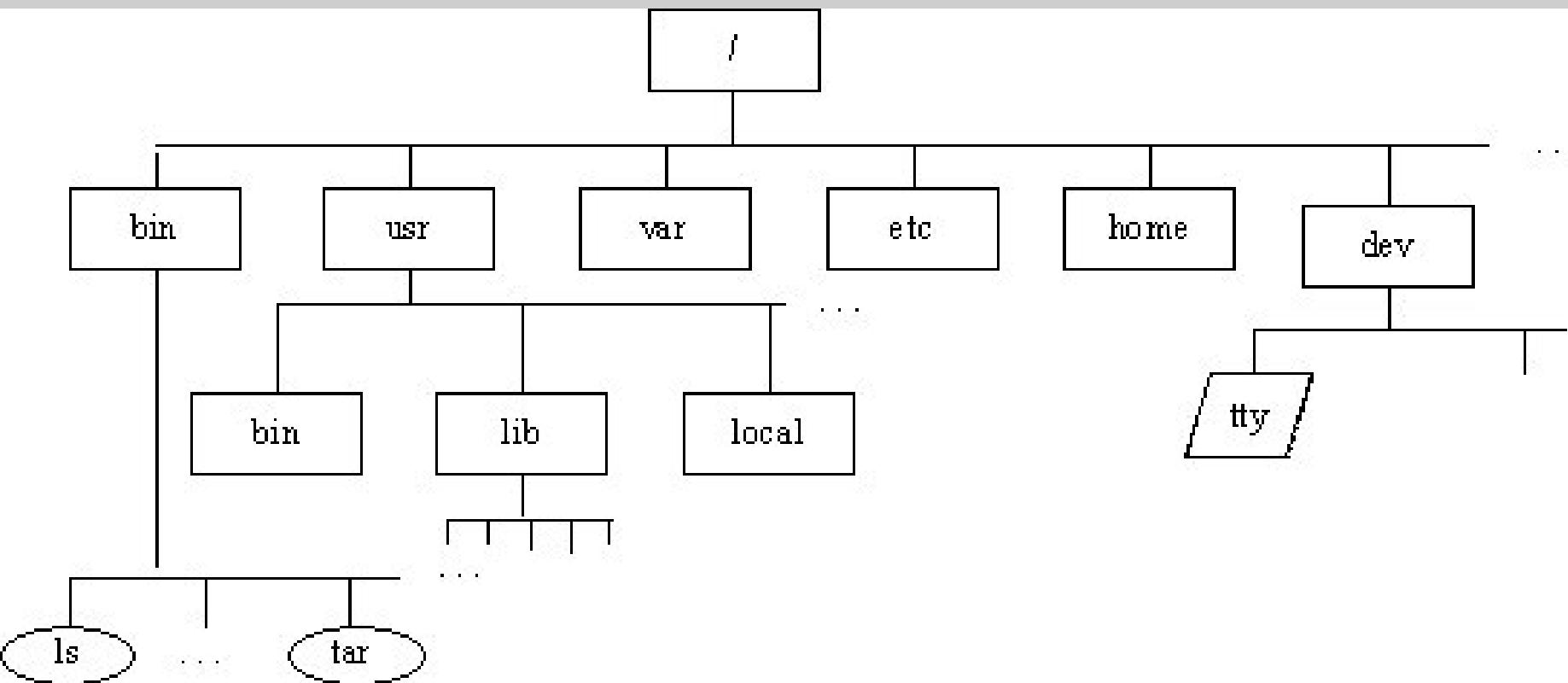
Gerätedatei:

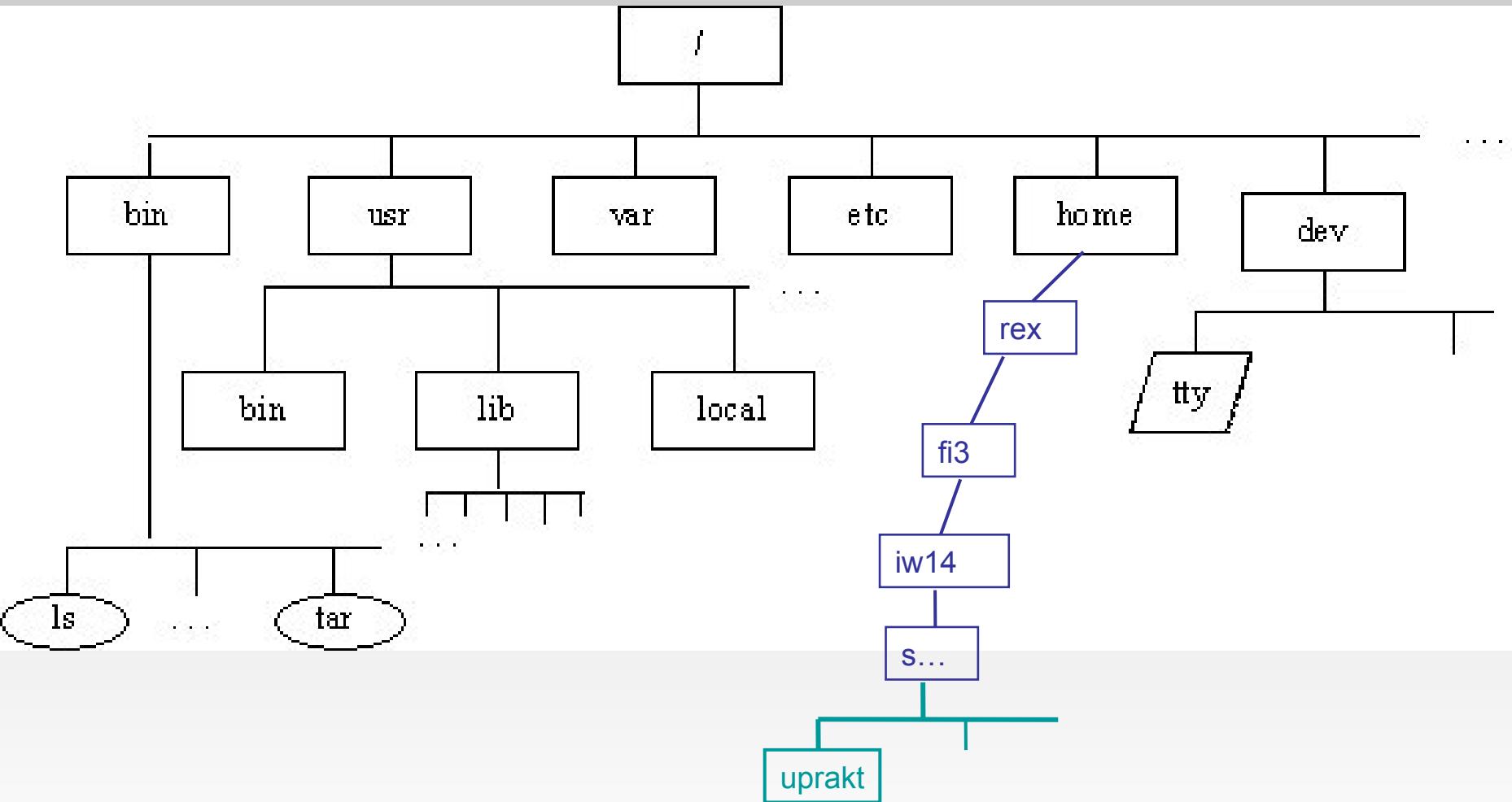
Peripheriegeräte werden als Dateien repräsentiert. Für jedes Peripheriegerät existiert mindestens eine Gerätedatei im Verzeichnis `/dev` (z.B. die Gerätedateien für Drucker, CD-ROM-Laufwerke, Terminals). Das Schreiben oder Lesen einer Gerätedatei bewirkt die Aktivierung des Gerätes.

Verzeichnis: Ist eine Datei, in der logisch zusammengehörige Dateien zusammengefasst werden. Ein Verzeichnis kann Dateien aller Arten (also auch Verzeichnisse selbst) logisch enthalten. Ein Verzeichnis in einem Verzeichnis heißt Unterverzeichnis (*subdirectory*).

Dateisystem:

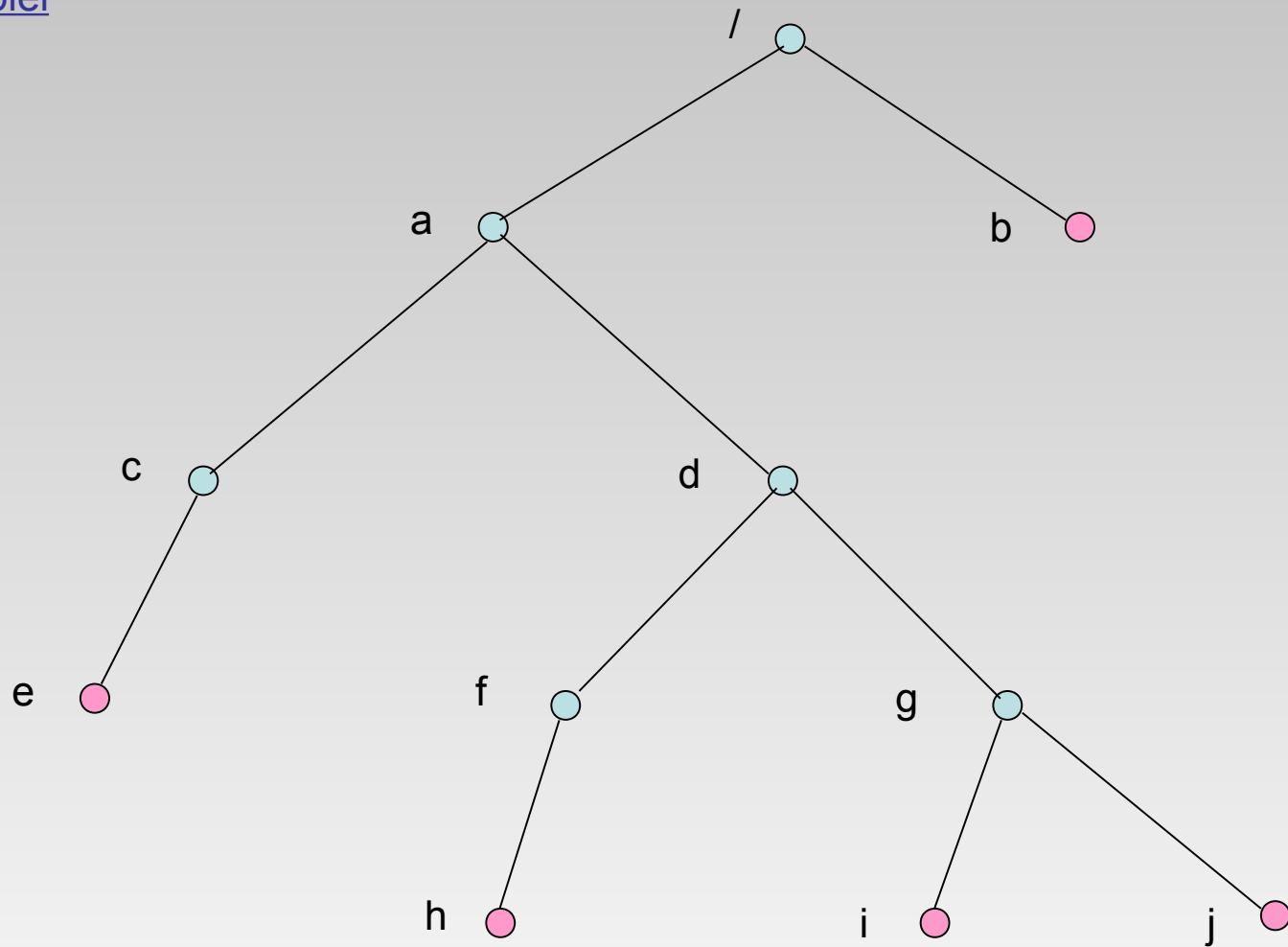
- hierarchische Struktur (Baumstruktur) → Anfangsvorstellung, später modifiziert
Art und Weise, wie Teile eines Ganzen untereinander und zu diesem Ganzen verbunden sind [Wikipedia]
- dargestellt als gerichteter Graph
 - innere Knoten entsprechen Verzeichnissen
 - Blattknoten entsprechen Dateien
 - (Verzeichnissen oder Gerätedateien oder Pipes oder gewöhnlichen Dateien)
- Jede Datei besitzt einen Dateinamen. Der Name allein kennzeichnet die Datei im Baum nicht eindeutig!

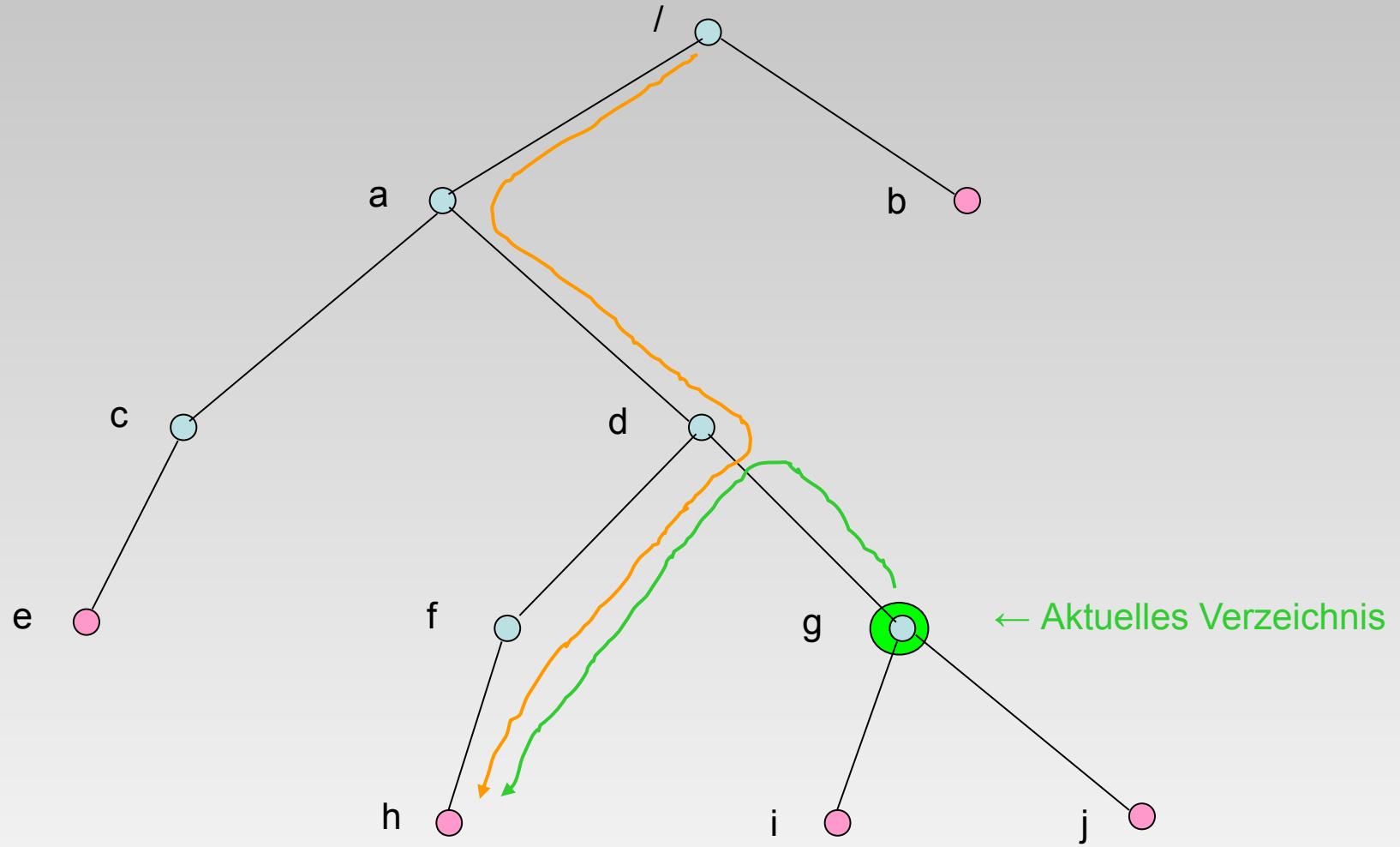




- /bin** Programme der elementaren LINUX-Kommandos, die von allen Benutzern ausgeführt werden können + Programme, die Benutzerkommandos bearbeiten (Shells)
- /dev** alle Gerätedateien
- /etc** Konfigurationsdateien für das System
(Benutzerverwaltung, Netzwerkkonfiguration u.a.)
- /home** Home-Verzeichnisse der Benutzer
(Superuser: **/root**)
- /lib** gemeinsam genutzte Bibliotheken, von Programmen benötigt
- /usr** (UNIX-System Resources)
Anwendungsprogramme, das X-Window-System, die Sourcen von LINUX,
überwiegend statische Daten
- /usr/bin** weitere ausführbare Programme
- /var** überwiegend veränderliche Daten
(oft gleiche Namen wie in **/usr**)

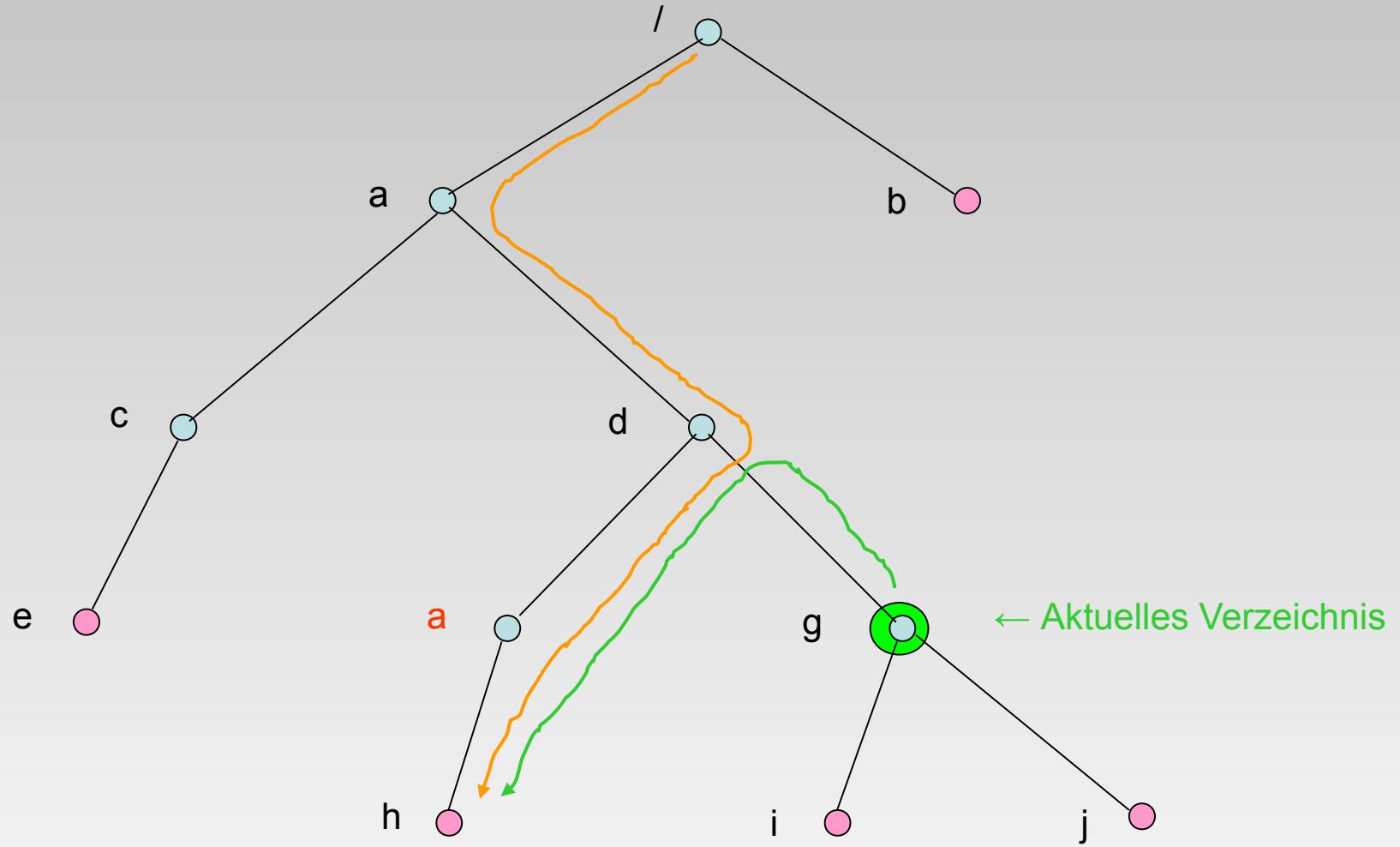
Beispiel





/a/d/f/h

../f/h



/a/d/a/h

.../a/h

Pfadnamen

- Jede Datei im Dateisystem (Dateibaum) ist durch einen eindeutigen Pfadnamen gekennzeichnet.

Absolute Pfadnamen bezeichnen eine Datei eindeutig durch Angabe eines Pfades,

- beginnend beim Wurzelverzeichnis `/`, gefolgt von
- der Auflistung der Namen aller Verzeichnisse auf dem Weg zur Datei durch den Dateibaum, wobei die Namen durch `/` getrennt werden,
- endend mit dem Dateinamen.

Relative Pfadnamen bezeichnen eine Datei eindeutig durch Angabe eines Pfades,

- beginnend mit einem Namen, der im aktuellen Verzeichnis enthalten ist, gefolgt von
- der Auflistung der Namen aller Verzeichnisse auf dem Weg zur Datei durch den Dateibaum, wobei die Namen durch `/` getrennt werden,
- endend mit dem Dateinamen.

Merke: ein relativer Pfadname beginnt nicht mit einem `/`.

Es werden folgende Bezeichnungen für spezielle Unterverzeichnisse verwendet:

- Wurzelverzeichnis (*root directory*) Benennung: /
 - aktuelles Verzeichnis (*working directory*) Benennung: .
 - Login-Verzeichnis (*login directory*)
... ist das erste aktuelle Verzeichnis nach dem Login
 - Home-Verzeichnis (*home directory*)
... Platz des Nutzers im Dateisystem
 - Eltern-Verzeichnis (*parent directory*) eines beliebigen Verzeichnisses
Benennung: ..

→ das Elternverzeichnis zum Wurzelverzeichnis ist das Wurzelverzeichnis selbst.

Kommandos:

Syntax:

`pwd`

durch Angabe eines
absoluten oder relativen
Pfadnamens

Semantik:

Ausgabe des absoluten Pfadnamens zum aktuellen Verzeichnis.

`cd [verzeichnis]`

Wechseln des aktuellen Verzeichnisses: Macht *verzeichnis* zum
aktuellen Verzeichnis. Default: Login-Verzeichnis.

`ls [option ...] [datei ...]`

Ausgabe der Namen von Dateien

- falls *datei* gewöhnliche Datei → Dateiname
 - falls *datei* Verzeichnis → Dateinamen im Verzeichnis
alphabetisch sortiert
- Default: aktuelles Verzeichnis

Optionen: `-l, -i, -s, -a, -R, ...`

Beispiel:

```
$ ls
artikel      hp.f        korr-c      mail_bs     prog.b5      unix.f
auftraege    htw_bitmap  korr-f      mail_k98    rob1_mail   up1.c
ausg         hunix.c     kunden     mail_pgp    s2273       up1.f
bedien       hunix.f     kunst.c    parmueb    saetze     up2.f
hagner       i           kunst.f    prakt      short      uv
hkunst.c    k           lprog      prakt_bs2  stadt      w
hkunst.f   k2273      mail_ai98  prg.b5     unix.c    xinitrc
```

Die Ausgabe in Langform hat folgende Gestalt:

```
$ ls -l
total 448 ← Aus. Blöcke
-rw-r--r--  1 studi    system      617 Jan 28 1998 artikel
-rw-r--r--  1 studi    system      197 Jan 28 1998 auftraege
...

```

Bedeutung:

						Dateiname
						Datum/Uhrzeit letzte Modif.
						Dateigröße in Bytes
						Gruppenname des Dateibesitzers
						login-Kennung des Dateibesitzers
						Anzahl der Links auf diese Datei
	Zugriffsrechte					
Dateityp						

Benennung von Dateien

- UNIX unterstützt mehrteilige Dateinamen, wobei die Bestandteile durch einen Punkt voneinander getrennt sind.

Beispiele: `unix.c`
 `bs_script.html`

ist ein C-Quellprogramm
ist ein Text in der
Dokumentenbeschreibungssprache HTML

Verborgene Dateien

- als verborgene Dateien wird eine Teilmenge der in einem Verzeichnis enthaltenen Dateien bezeichnet.
- Die Dateinamen verborgener Dateien beginnen mit einem Punkt

Beispiel `.bashrc`)

- verborgene Dateien werden mit dem Kommando `ls` nur bei Verwendung der Option `-a` angezeigt

„Physische“ Dateiorganisation

- ▶ Beim Erstellen einer Datei wird zusätzlich zum Datenteil eine kleine Tabelle, ein sog. **i-Node** erzeugt.
Darin Angaben über (Metadaten):
 - Dateityp
 - Größe der Datei
 - Lage der Datei auf dem Datenträger
 - Eigentümer
 - usw.
- ▶ Jeder i-Node besitzt eine **i-Nummer**.
- ▶ In einem Verzeichnis existiert pro verwalteter Datei ein Eintrag mit:
 - dem Namen der Datei
 - i-Nummer
- ▶ Ein und dieselbe Datei kann mehrere Namen (in unterschiedlichen Verzeichnissen) besitzen!
Sie besitzt immer genau eine i-Nummer!

i-Nummer

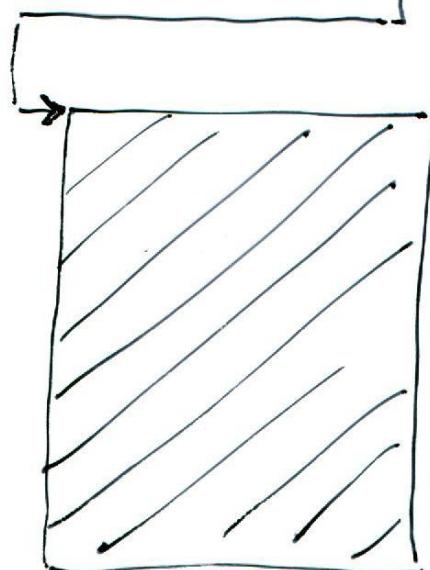
i-Node
137

Eigentümer
Gruppenzugehörigkeit
Zugriffsrechte
Dateigröße
Erstellungsdatum
Datum letz. Modif.
<i>Adressen der Daten</i>
...

137	t.c
276	programme

Directory

Dateiname



- Erzeugen von Verzeichnissen (*make directory*)

```
mkdir verzeichnis ...
```

(Voraussetzung ist die Schreiberlaubnis im Elternverzeichnis)

Bsp.: **mkdir** dir1 dir1/dir2

- Löschen von Verzeichnissen (*remove directory*)

```
rmdir verzeichnis ...
```

(Voraussetzung: Das zu löschenende Verzeichnis ist leer)

Bsp.: **rmdir** dir1/dir2 dir1

► Löschen von Dateien (*remove*)

```
rm [ option ... ] datei ...
```

Entfernt Namen und i-Nummer aus dem Verzeichnis.

Bsp.: **rm** **-r** *verzeichnis*

Löscht Verzeichnis mit allen darin enthaltenen Dateien (auch Verzeichnissen) rekursiv.

Vorsicht vor **rm** **-r** */**

► Kopieren von Dateien (*copy*)

```
cp      quelldatei zieldatei  
| quelldatei ... zielverzeichnis
```

Erzeugt eine Kopie der *quelldatei* (im gleichen Verzeichnis oder in einem anderen Verz.)

- die Dateiinhalte sind identisch
- neue i-Nummer

Im Falle *zielverzeichnis* : Die Quelldateien werden in das Verzeichnis kopiert, die Namen werden übernommen.

Zugriffsrechte werden kopiert, Eigentümer der neuen Datei ist der Benutzer, der das Kommando abgesetzt hat.

Ist eine Datei mit gleichem Namen schon vorhanden, wird diese überschrieben!

► Verschieben von Dateien (*move*)

```
mv [option ...] dateinamealt dateinameneu
```

Der *dateinameneu* wird anstelle von *dateinamealt* mit der i-Nummer im Verzeichnis assoziiert.

Die i-Nummer bleibt dieselbe. Der i-Node und die Datei selbst werden nicht verändert. Auch das Verschieben von Verzeichnissen ist möglich.

→ eine Verschiebung im selben Verzeichnis bedeutet ein Umbenennen.

Optionen:

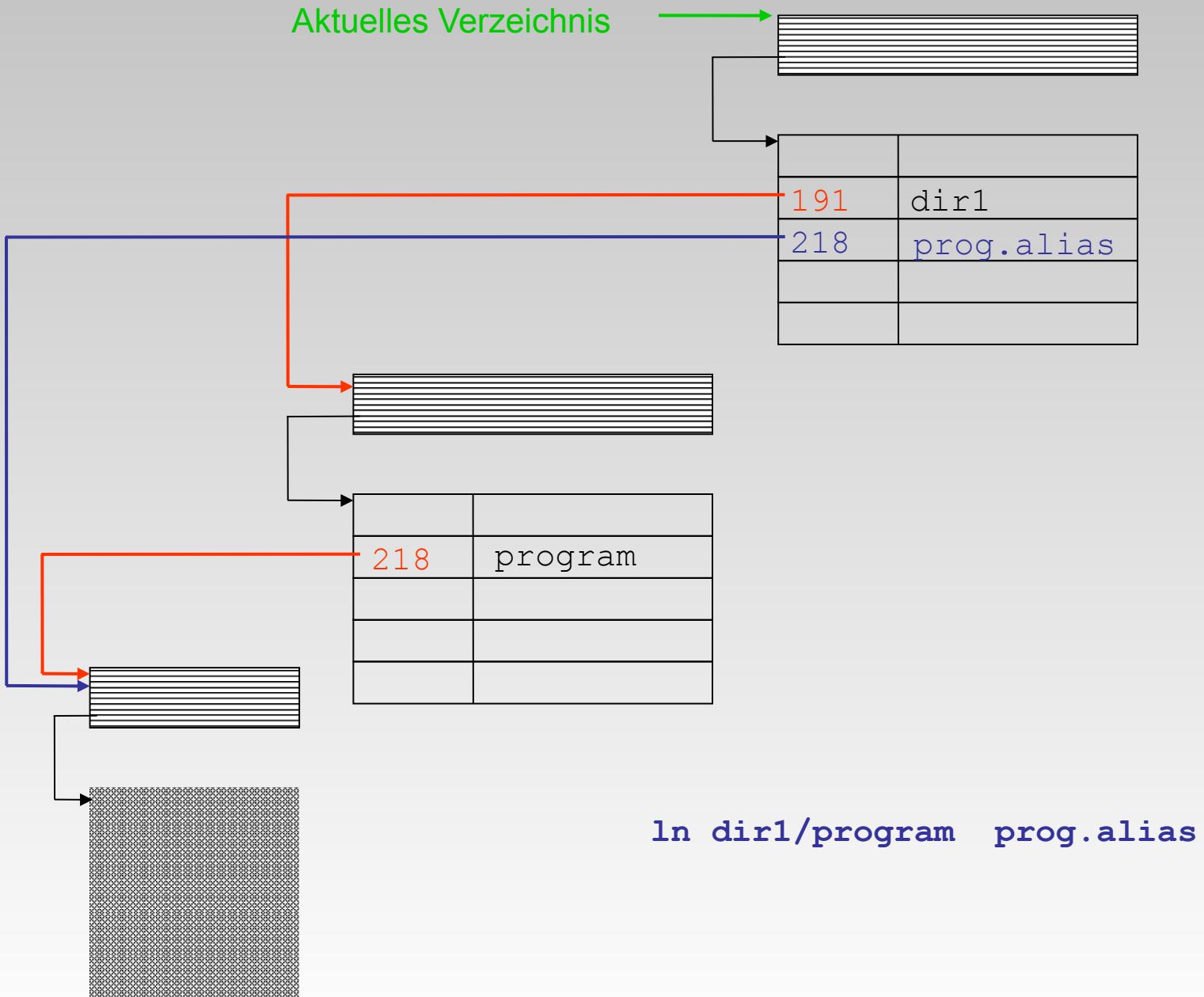
- i Warnung und Bestätigung vor dem Überschreiben.

Mehrere Namen für dieselbe Datei : „Links“ (Hard-Links)

```
ln [option ...] dateiname aliasname
```

Es wird ein Verweis auf eine bereits bestehende Datei (dieselbe i-Nummer) in einem Verzeichnis eingerichtet, die Datei wird dadurch mit *aliasname* benannt. Die Datei kann anschließend auch unter dem Aliasnamen angesprochen werden.

```
Bsp.: ln dir1/program prog.alias
```



Symbolische Links (Soft-Links)

- ▶ Verweise über Dateisystemgrenzen hinweg
In einer Dateihierarchie können Teilhierarchien existieren (Dateisysteme)
- ▶ Mit der Option **-s** werden symbolische Links erzeugt.
- ▶ Name der bereits existierenden Datei muss mit absolutem Pfadnamen angegeben werden.

Beispiel:

```
ln -s /bin/pwd mypwd
```

```
ls -l
```

```
...
```

```
1rwxrwxrwx 1 fritzschi 8 2006-11-02 12:16 mypwd -> /bin/pwd
```

```
...
```

Was tut der Kommandointerpreter (die Shell) ?

- ▶ verarbeitet Kommandos einer Eingabe von links nach rechts
- ▶ expandiert Zeichenmuster
- ▶ sucht das Programm, dessen Name im Kommando angegeben ist, in den ihm bekannten Verzeichnissen
- ▶ sorgt für
 - Laden Programm in den Hauptspeicher
 - Übergabe der Zeichenkette (einschließlich Programm-Name)
 - Ausführung
- ▶ suspendiert sich, bis das aufgerufene Programm terminiert.
- ▶ Das Programm wertet Optionen aus und verarbeitet die angegebenen Argumente.
- ▶ Die Shell hat mit dem Nutzer Geräte vereinbart, über die die Ein-/Ausgabe mit Programm erfolgt: `stdin` - Tastatur, `stdout`, `stderr` - Bildschirm

Zeichenmuster (pattern) für Dateinamen

- ▶ mit einem Zeichenmuster können mehrere Dateien angesprochen werden.
- ▶ **Platzhalterzeichen** in Zeichenmustern
- ▶ Mustervergleich: Zeichenmuster ↔ vorhandene Dateinamen
- ▶ Umformung erfolgt durch den Kommandointerpreter
- ▶ passt kein Dateiname, wird das Zeichenmuster übergeben

?

beliebiges einzelnes Zeichen

*

beliebige Anzahl beliebiger Zeichen

[...]

einzelnes der in Klammern eingeschlossenen Zeichen

[z1–z2]

einzelnes Zeichen, lexikographisch zwischen z1 und z2 (ASCII)

[! ...]

einzelnes Zeichen das nicht in der Aufzählung nach dem Ausrufezeichen vorkommt

Entwerten von Sonderzeichen

Sonderzeichen für die Shell:

| & ; * ? [] < > ! " ' ` \

↑
backquote

„Ausschalten“ der Sonderbedeutung:

- 1) \ vor einzelnen Zeichen
- 2) " ... " double quotes schützen alle Zeichen außer:

\ ` \$ "



Sonderbedeutung nur vor

\$ ` " \ <newline>

- 3) ' ... ' schützen alle Zeichen außer '

Probleme:

- manche Kommandos verwenden eigene Sonderzeichen in Mustern
→ zusätzlicher Entwertungsmechanismus
- Kommandosubstitution:
→ Ausgabe der Ausführung eines Kommandos ist Parameter bei Ausführung eines anderen Kommandos

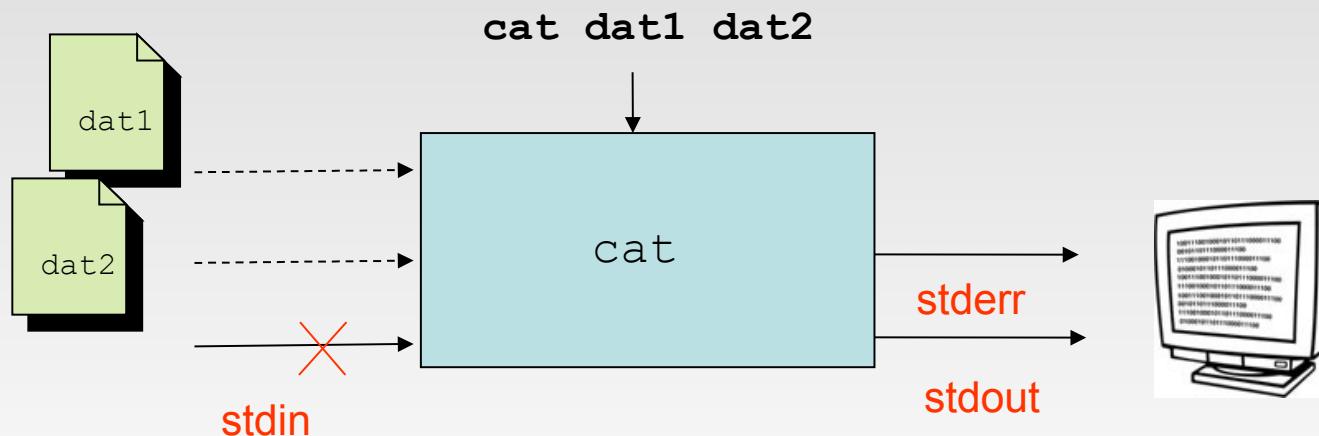
Beispiele:

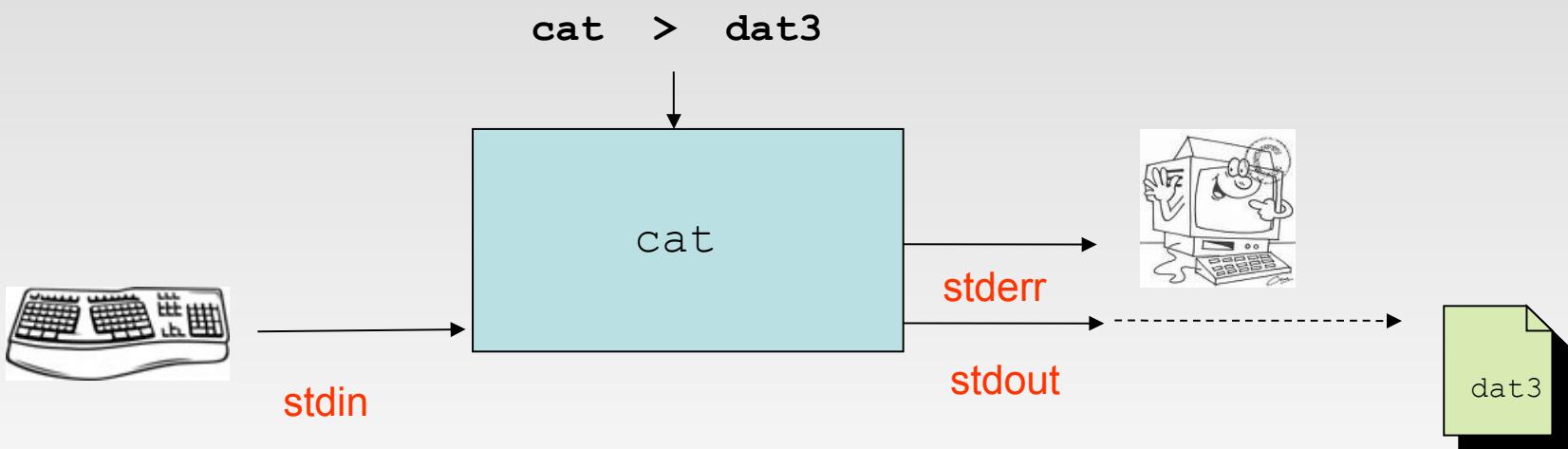
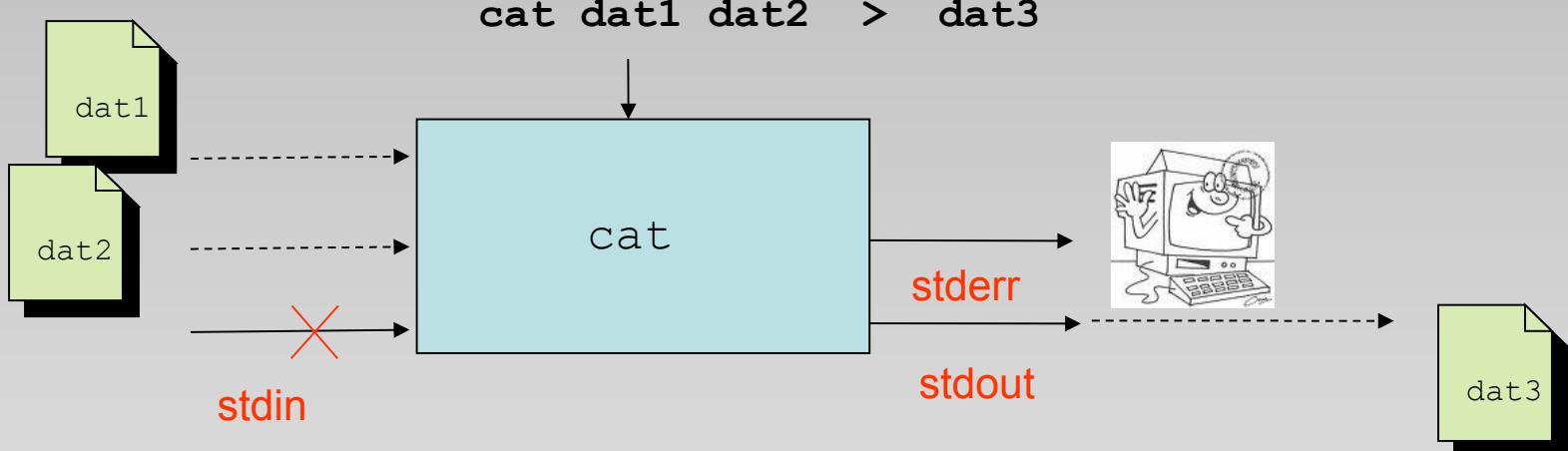
```
$ echo \"\$!?\a\b\1  
''$!?ab1  
  
$ echo "\"$\\x\\'*#\\a?\"  
"$\\x\\'*#\\a?  
  
$ echo '\"$\\x\\'*#\\a?''  
\"$\\x\\'*#a?
```

Verketten von (Text-)Dateien: `cat` – concatenate

```
cat [option ...] [ datei ... ]
```

verkettet die durch *datei* ... spezifizierten Dateien und gibt das Ergebnis auf **stdout** aus





Umlenkung der Ein- und Ausgabe

- Speichern der Kommandoausgabe in eine Datei:

```
kommando [n] > dateiname
```

Die Datei *dateiname* wird neu angelegt.

Beispiele: ls > verzinhalt

 echo „Ein einfacher Text“ > dat1

- Lesen der Kommandoeingabe aus einer Datei:

```
kommando < dateiname
```

- Umlenkung von Systemmeldungen (stderr):

```
kommando 2 > dateiname
```

Programme sprechen Dateien über Dateideskriptoren an:

```
0 - stdin
1 - stdout
2 - stderr
```

Die Dateien mit den Dateideskriptoren 0, 1, 2 werden von jedem Programm geöffnet
(und evtl. noch weitere)

kommando >> dateiname

Die Ausgaben des *kommando* werden an den Inhalt der Datei
dateiname angefügt.

Existiert die Datei noch nicht, wird sie neu angelegt.

Zählen von Zeichen Wörtern und Zeilen: **wc** – word count

```
wc [option ...] [ datei ... ]
```

Optionen:

- l zählt Zeilen (*lines*)
- w zählt Wörter (*words*)
- c zählt Zeichen (*characters*)

Default: -lwc falls keine Optionsangabe
Text von **stdin**, falls keine *datei*

Beispiel:

```
wc <|  
Das ist <|  
ein Mustertext <|  
^D  
2            4            23
```

Pipes (*pipelines*) - eine Nutzersicht

```
ls > liste  
wc -w < liste
```

Die Datei `liste` ist eigentlich überflüssig.

→ stdout von `ls` direkt mit stdin von `wc` durch Röhre verbinden!

```
ls | wc -w
```

Semantik:

Die durch den Pipe-Operator `|` verbundenen Kommandos werden (zeitgleich) ausgeführt. Dabei wird die Ausgabe `stdout` des vor dem Pipe-Operator stehenden Kommandos auf die Eingabe `stdin` des nach dem Pipe-Operators stehenden Kommandos gelegt.

Suche nach Zeichenmustern in Text-Dateien

- fgrep** *fast regular expression pattern* - Suche nach einfachen Strings, schnell
- egrep** lässt reguläre Ausdrücke für die Suche zu, langsam
- grep** nicht alle Möglichkeiten für reguläre Ausdrücke, zwischen **fgrep** und **egrep**

egrep [option ...] *muster* [*datei* ...]

Ausgabe der Zeilen der *datei* (oder mehrerer) auf **stdout**, die eine Zeichenkette enthalten, die auf *muster* „passt“. „passt“ bedeutet, es wird ein Mustervergleich durchgeführt (*pattern matching*).

Bsp.: **egrep M[ea][iy]er Liste**

gibt alle Zeilen einer Datei **Liste** aus, in denen ein Name **Meier**, **Meyer**, **Maier** oder **Mayer** vorkommt.

```
~> cat Liste
```

```
Rolf Meier  
Karl May  
Mike Miller  
Horst Mayer  
Peter Meyer  
Gisela Bayer  
Gerd Maier
```

```
~> egrep M[ea][iy]er Liste
```

```
Rolf Meier  
Horst Mayer  
Peter Meyer  
Gerd Maier
```

```
~> egrep M(u|e|i)ller Liste
```

```
-bash: i)ller: command not found  
grep: Unmatched ( or (^
```

```
~> egrep 'M(u|e|i)ller' Liste
```

```
Mike Miller
```

```
~>
```

Der | wird als „Pipe-Operator“
interpretiert!

Reguläre Ausdrücke

→ beschreiben Mengen von Zeichenketten

Definition:

1. ϵ ist ein regulärer Ausdruck, bezeichnet die leere Zeichenkette
2. ein Zeichen des Alphabets ist ein regulärer Ausdruck
3. sind r und u reguläre Ausdrücke, dann auch
 - r^* wiederholtes, aufeinander folgendes Vorkommen von r
(beliebig oft, einschließlich 0 mal)
 - $r \cdot u$ aufeinander folgendes Vorkommen von r und u (Verkettung der Zeichenketten, manchmal auch als $r \cdot u$ notiert)
 - $r \mid u$ alternatives Vorkommen von r und u
(Vereinigungsmenge der durch r und u beschriebenen Zeichenketten)
 - (r) reguläre Ausdrücke können geklammert werden

Priorität: *

:

|

fallend

Speziell für **egrep** gilt:

c beliebiges Zeichen außer Metazeichen

\c beliebiges Zeichen, hebt Meta-Bedeutung auf

Metazeichen: . + * ? | () [] \^ \\$ { }

.

Steht für beliebiges Zeichen

^

(am Beginn eines Ausdrucks) für Zeilenanfang

\$

(am Ende eines Ausdrucks) für Zeilenende

[*teilmenge*]

einzelnes Zeichen aus der Teilmenge

[^ *teilmenge*]

einzelnes Zeichen, das nicht in der Teilmenge enthalten ist

[z1-z2]

einzelnes Zeichen, dessen Code zwischen z1 und z2 liegt
(einschließlich)

r +

einmaliges oder mehrmaliges Vorkommen von r

r ?

optionales Vorkommen von r

r {n}

n Wiederholungen von r (Verkettung)

r {n1,n2}

Anzahl von Wiederholungen, zwischen n1 und n2 (einschließlich)

Sonderfälle:

- minus (-) als erstes Zeichen einer *teilmenge* steht für sich selbst

- auch Leerzeichen ist ein Zeichen: 'a b' ≠ 'a b'

```
Bsp.: egrep '^(([+-]?[1-9][0-9]{0,4})|0)$' dat1
```

Erlaubte ganze Zahlen:

- pro Zeile eine Zahl am Anfang der Zeile
- bei Zahl 0 kein Vorzeichen möglich
- maximal 5-stellig
- wahlweise Vorzeichen
- keine führenden Nullen

```
~>cat dat1
```

```
+123  
0  
23  
-11 // minus elf  
-0  
-1  
12345  
123456  
1000 100
```

```
~> egrep '^(([+-]?[1-9][0-9]{0,4})|0)$' dat1
```

```
+123  
0  
-1  
12345
```

Nicht durch reguläre Ausdrücke beschreibbar sind

→ beliebig tief (= „rekursiv“) verschachtelte Klammergebirge

Bsp.:

(... (... (...) ...) ...)

Ist

(333 (123 245) 678)

eine gültige Zeichenkette?

→ es gibt keinen regulären Ausdruck als Muster!

Editoren

standardmäßig vorhandene Editoren:

- zeilenorientierter Editor

ed

- Bildschirmeditor

vi

- Editor

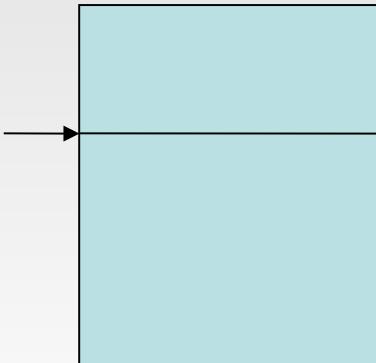
ex

(Erweiterung von ed, kann in vi umgeschaltet werden)

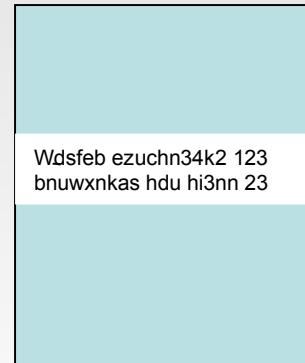
- emacs

...

ed



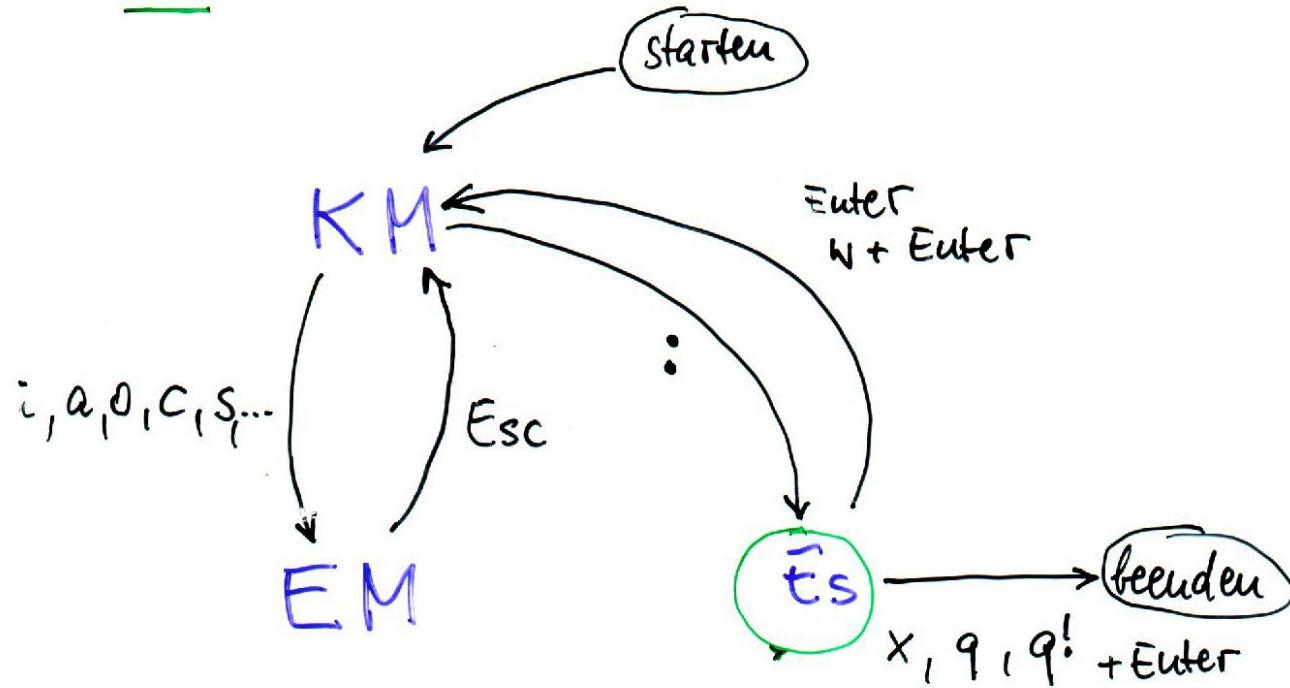
vi



Eine Kopie der zu bearbeitenden Datei wird in einem Puffer abgelegt.

Nach Abschluss aller Änderungen kann der Puffer in die Datei zurückgeschrieben werden.

Vi



KM Kommando modus

EM Eingabemodus

Es Escape-Modus

Ex-Befehlsmodus

Dateischutz

- Benutzer darf nicht ohne weiteres auf Dateien anderer Benutzer zugreifen (sonst haben Passwörter keinen Sinn!)
- Gruppe von Benutzern soll bestimmte Dateien gemeinsam nutzen (z.B. Projekt). Dritten soll der Zugriff verwehrt bleiben.
- Bezuglich der Rechte werden folgende Nutzerklassen unterschieden:
 - der Eigentümer (**user**)
 - Gruppe, der der Eigentümer zugeordnet ist (**group**)
 - alle übrigen Benutzer (**other**)(die Bezeichnung **all** kann als Generalisierung aller 3 Klassen verwendet werden)

Im i-Node jeder Datei sind u.a.

- für jede der Nutzerklassen je eine
 - read (r)
 - write (w)
 - execute (x) Erlaubnis,
- Eigentümer (uid des Benutzers)
- Gruppe (gid des Benutzers)
(dessen Mitgl. der Eigentümer z.Z. der Erstellung war)
gespeichert.

Bsp.:

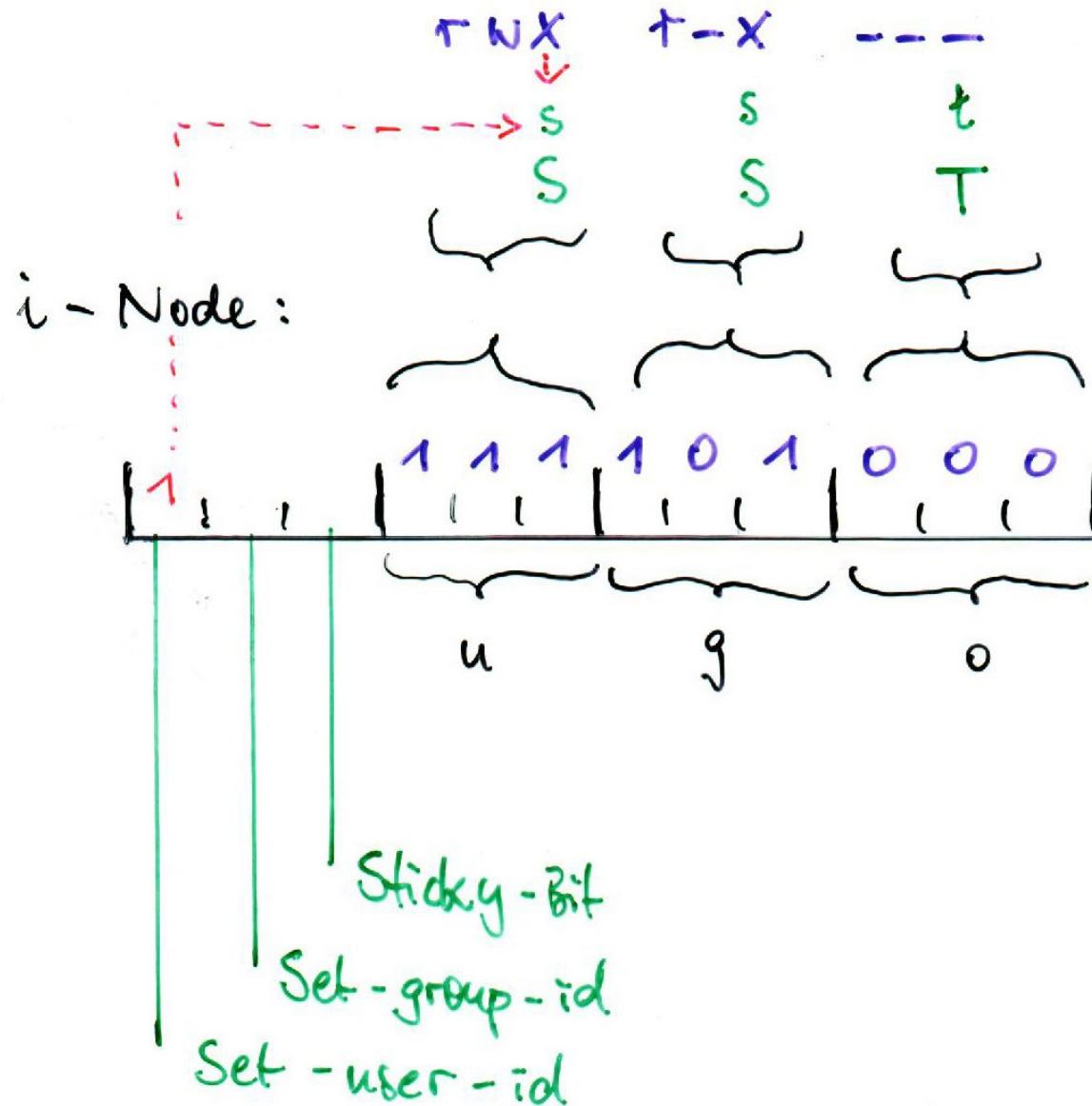
~>id

uid=10085(fritzs) gid=10000(fi) Gruppen=9000(jamus),10000(fi)

~>

- ▶ Dateibesitzer ist der Benutzer, der die Datei erzeugt hat, wem das Verzeichnis gehört, in dem die Datei eingetragen wird, ist gleichgültig, solange der Erzeuger Schreiberlaubnis für dieses Verzeichnis besitzt.
- ▶ Kopieren mittels `cp`
 - Zugriffsrechte und Besitzverhältnisse der Zielfile werden beibehalten, falls die Datei schon existiert.
 - wird die Datei neu angelegt, erhält sie die Zugriffsrechte der Originaldatei, aber einen neuen Besitzer
- ▶ Bei `mv` und `ln` bleiben die Zugriffsrechte der Originaldatei immer erhalten.

Anteile mit ls



Bedeutung unterschiedlich bei unterschiedlichen Dateitypen:

- kein Verzeichnis:

r : Erlaubnis zum Lesen

w : Erlaubnis zum Schreiben

x : Erlaubnis, den in der Datei enthaltenen Code auszuführen

- Verzeichnis:

r : Inhalt darf mit `ls` aufgelistet werden

w : im Verzeichnis dürfen Dateien angelegt, modifiziert, gelöscht werden

x : auf Verzeichniseinträge darf zugegriffen werden,

Verzeichnis darf mittels `cd` zum aktuellen Verzeichnis gemacht werden

Fehlt das Recht, darf der Verzeichnisname nicht als Komponente in einem Pfadnamen verwendet werden!

Ausführung von Binärdateien

- ▶ x-Recht für Binärdatei setzen
- ▶ unterscheiden:
 - UID des aufrufenden Benutzers
 - UID des Besitzers der Datei
- ▶ Programme werden unter Eigentümer-/Gruppenkennung des aufrufenden Benutzers ausgeführt
- ▶ Bestimmte Dienstprogramme laufen unter Autorität des Superusers, müssen aber auch von anderen Nutzern ausführbar sein.

Beispiel: Programm zum Ändern des Passwortes – das Programm greift auf die Passworddatei zu!

- ▶ Trick: Eigentümer-/Gruppenkennung der Datei (die das auszuführende Programm enthält) kann
 - unter bestimmten Voraussetzungen
 - zeitlich begrenztauf den ausführenden Prozess (des Benutzers) übertragen werden (= **effektive UID** des Benutzerprozesses)

set-user-id (gesetzt)	Jeder Benutzer(-prozess), der das Programm ausführt, hat für die Dauer der Programmausführung die gleichen Rechte wie der Eigentümer des Programms. (bei gesetztem x-Recht für den Eigentümer der Datei)
set-group-id (gesetzt)	Bei gesetztem x-Recht für die Gruppe: Jeder Benutzer(-prozess), der das Programm ausführt, hat für die Dauer der Programmausführung die gleichen Rechte wie die Mitglieder der Gruppe. Bei <u>nicht</u> gesetztem x-Recht für die Gruppe: Die Datei wird für alleinigen Lese-/Schreibzugriff zur Verfügung gestellt. Lesen oder Schreiben durch andere Programme wird verwehrt, solange ein Programm auf die Datei zugreift.
Sticky-bit (gesetzt)	Nach Ausführung des Programms wird dieses nicht – wie sonst üblich – aus dem Hauptspeicher entfernt. Änderungsrecht hat nur der Superuser.

Verändern der Zugriffsberechtigungen - `chmod`

```
chmod    absolutmodus  datei ...
```

```
chmod    symbolischer_modus  datei ...
```

absolutmodus : Angabe eines Oktalwertes für je drei Bits, von links nach rechts in der Bedeutung

set-user-id

set-group-id

sticky-bit

rwx für user

rwx für group

rwx für other

Die Rechte werden entsprechend des Oktalwertes gesetzt

symbolischer_modus : [ugo] *operator* [rwxslt]

u für user

g für group

o für others

a für all Default: a

operator + Recht hinzufügen
- Recht entziehen
= Recht als neues Zugriffsrecht vergeben

r Leserecht

w Schreibrecht

x Ausführrecht

s set-user-id (im Zusammenhang mit u)

t set-group-id (im Zusammenhang mit g)

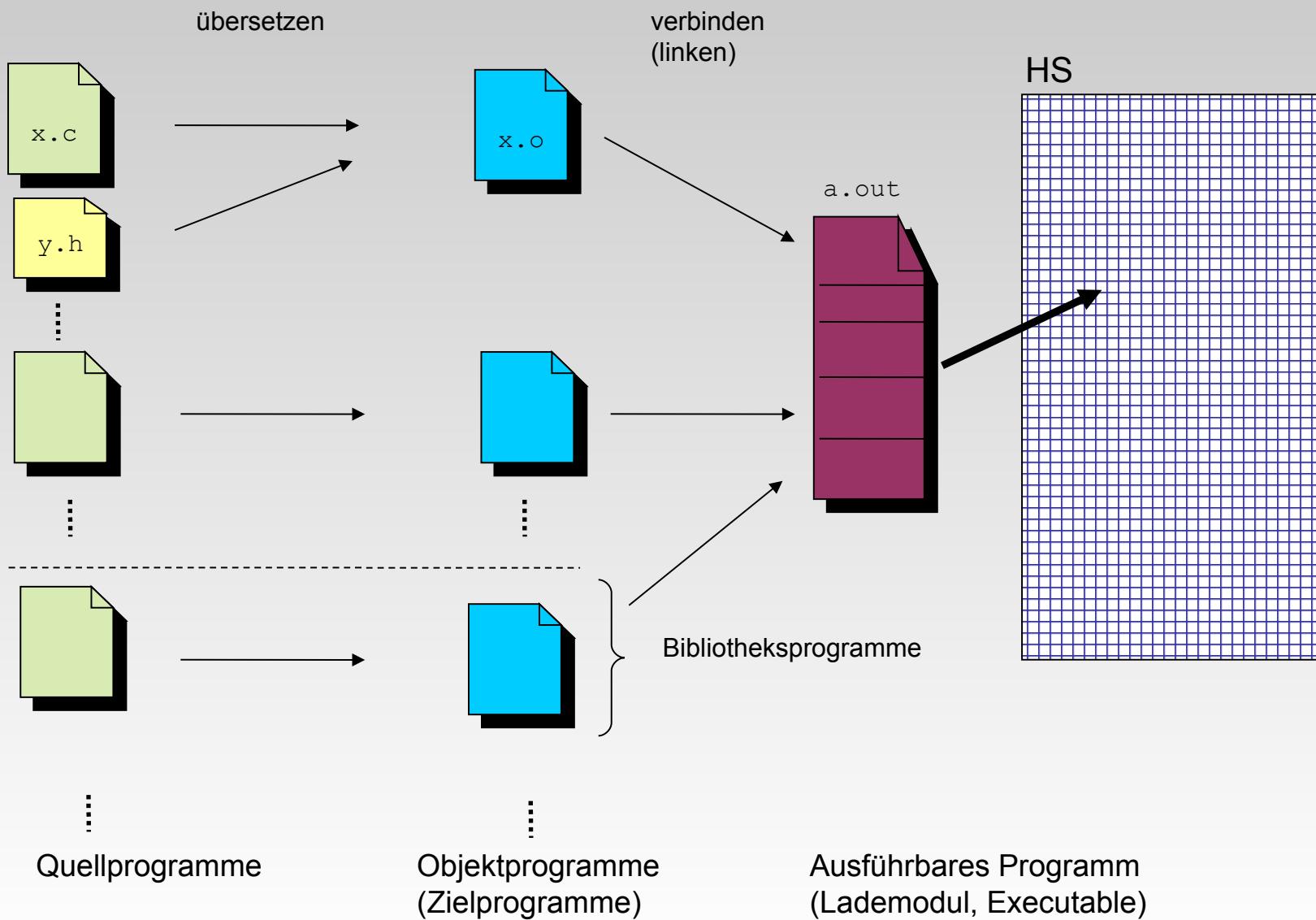
t sticky-bit

l für exklusiven Lese-/Schreibzugriff

Default: Entfernen aller Zugriffsrechte

```
~> pwd
/u/fritzs
~> ls -ld uprakt
drwxr--r-- 5 fritzsch fi 4096 2005-10-27 10:16 uprakt
~> ls -ld uprakt/text
drwxr-xr-x 2 fritzsch fi 4096 2006-11-01 12:27 uprakt/text
~> chmod u-wx uprakt
~> ls -l uprakt
...
/bin/ls: uprakt/text: Keine Berechtigung
/bin/ls: uprakt/a.out: Keine Berechtigung
...
~> chmod u-r uprakt
~> ls -l uprakt
/bin/ls: uprakt: Keine Berechtigung
~> chmod u+rw uprakt
~> ls -l uprakt
...
/bin/ls: uprakt/text: Keine Berechtigung
/bin/ls: uprakt/a.out: Keine Berechtigung
...
~> ls -l
...
drw-r--r-- 5 fritzsch fi 4096 2005-10-27 10:16 uprakt
...
```

Übersetzen, Verbinden und Ausführen von Programmen:



Erstellen eines C-Programmes `simple.c` mit einem Texteditor:

```
#include <stdio.h>
#include <unistd.h>

main () {
    printf("Hier ist das Programm simple\n");
    execl("/bin/ls","ls", "-l", "a.out", 0);
}
```

Übersetzen und Ausführen:

```
~> cc simple.c
~> ./a.out
Hier ist das Programm simple
-rwxr-xr-x  5 fritzsch fi 9537 15. Nov 10:07 a.out
~>
```

Programme und Prozesse

Prozessdefinition: Wird ein Programm aufgerufen (Benutzerprogr., Kommando), wird der Programmcode in den HS geladen u. abgearbeitet.

Das ablaufende Programm heißt Prozess (Task).

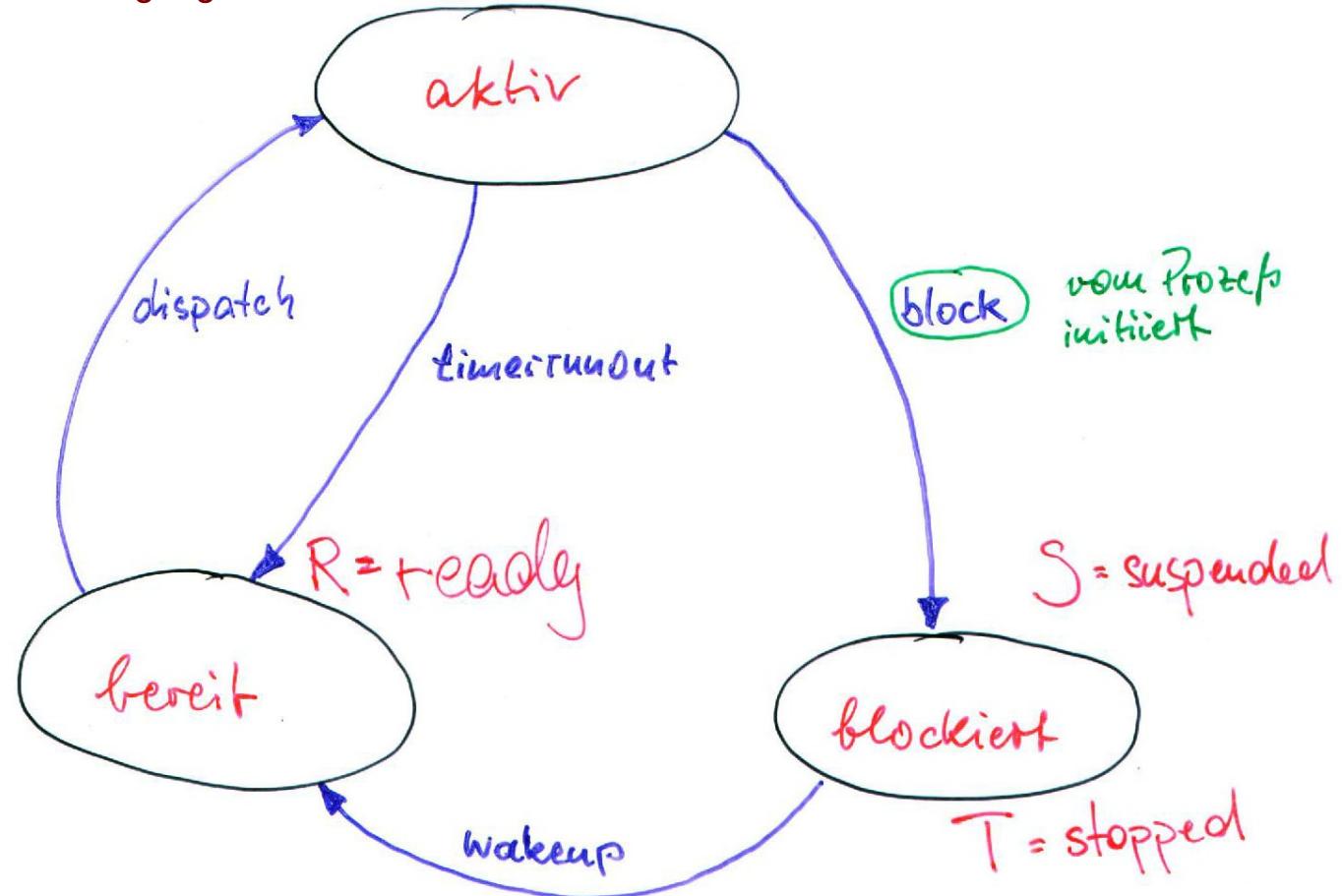
→ *Jeder Prozess besitzt eine eindeutige PID.*

- ▶ In einem Multitasking-BS entscheidet das BS periodisch, einen laufenden Prozess zu unterbrechen und einen anderen Prozess zu starten (wenn die dem Prozess zustehende CPU-Zeit verbraucht ist). -> **Prozessscheduling**
- ▶ Wenn ein Prozess suspendiert wurde, muss er später in genau diesem Zustand wieder gestartet werden.

Zustandsgraph:

- Ovale bezeichnen Zustände
- Pfeile bezeichnen Zustandsübergänge

$R = \text{running}$



$\xrightarrow{\text{dispatch}}$

bereit

<u>T</u>	<u>0</u>	<u>1</u>
----------	----------	----------

$\xleftarrow{\text{timertimeout}}$

$\xleftarrow{\text{wakeup}}$

blockiert

<u>T</u>	<u>0</u>	<u>1</u>
----------	----------	----------

$\xleftarrow{\text{block}}$

Prozesshierarchie: Ein Prozess kann von sich aus einen neuen Prozess erzeugen und starten.

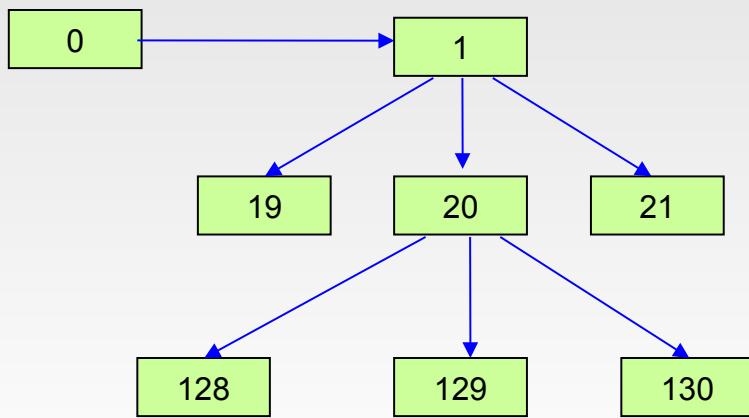


Kindprozess (child process)



Elternprozess (parent process)

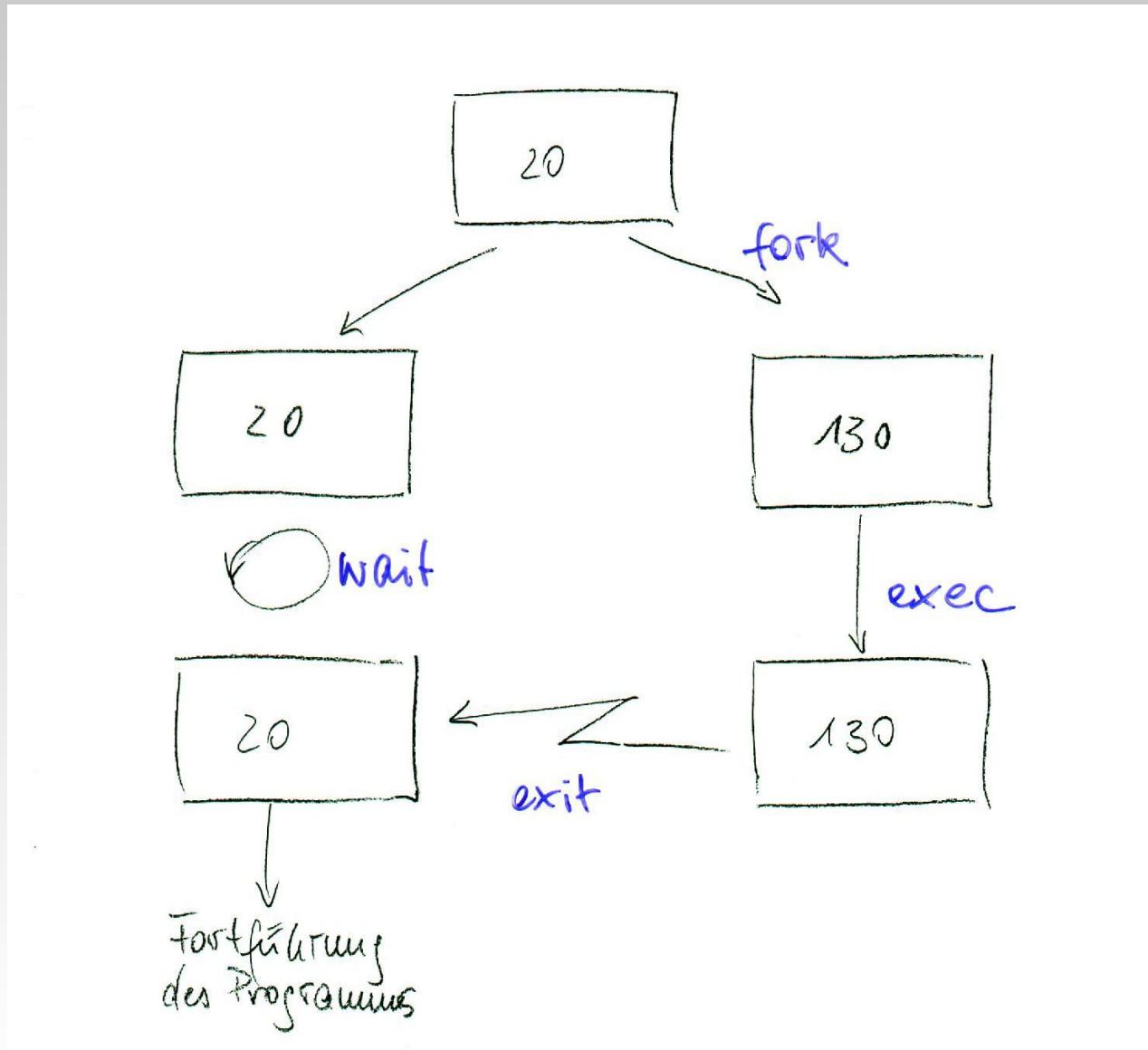
Beim Erzeugen eines Prozesses wird mittels des Systemaufrufs **fork** eine Kopie des erzeugenden Prozesses erzeugt



Prozesskenndaten (sind in einer Prozesstabelle gespeichert)

- Prozessnummer (PID) bzgl. Prozessverwaltung
 - Prozessnummer des Elternprozesses (PPID) bzgl. Speicherverwaltung
 - Benutzerkennung (UID) und Gruppenkennung (GID) bzgl. Dateiverwaltung
 - Prozesspriorität (PRI)
 - Prozesszustand (STAT)
 - Kontrollterminal (TTY)
 - zugeordnete Datenbereiche (Programm + Daten)
 - aktuelles Verzeichnis
 - die geöffneten Dateien
 - Environment (Variablen und Werte)
 - der Stack
 - der Befehlszähler
 - Register
- ...

Entstehung und Enden von Prozessen



pvater

```
main() {
    int w_status;
    int pid_fork;
    if((pid_fork = fork()) == 0) {
        execl("psohn","psohn",0);
        printf("execl gescheitert\n");
        exit(1);
    }
    if(pid_fork == -1) {
        printf("fork gescheitert\n");
        exit(2);
    }
    wait(&w_status);
    printf("Beendigungsgrund: %x\n",w_status & 0xff);
    printf("Exit-Status : %x\n", (w_status>>8) & 0xff);
}
```

Anwendung, wenn kein Programm psohn existiert:

```
~> pvater
execl gescheitert
Beendigungsgrund: 0
Exit-Status : 1
```

Anwendung wenn ein Program **psohn** existiert mit:

```
main() {
    exit(4);
}
```

```
~> pvater
Beendigungsgrund: 0
Exit-Status : 4
```

Pipes (pipelines)

```
ls > liste  
wc -w < liste
```

Die Datei `liste` ist eigentlich überflüssig.

→ stdout von `ls` direkt mit stdin von `wc` durch Röhre verbinden!

```
ls | wc -w
```

- ▶ Die Kommandos laufen in unterschiedlichen Prozessen ab
- ▶ Synchronisation: `ls` hält an, wenn die Pipe voll ist
`wc` wartet, wenn die Pipe leer ist

Abspeichern von Zwischenergebnissen in eine Datei

Kommando tee

Beisp.:

```
cat d1 d2 | tee d12 | wc -w
```

tee schreibt → eine Kopie der Eingabe auf stdout
+
→ eine Kopie auf d12

Named Pipes

können von unabhängigen Prozessen angesprochen werden, nicht nur von „verwandten“ Prozessen

→ fifo-Dateien

first in – first out

Kommandoabarbeitung durch die Shell:

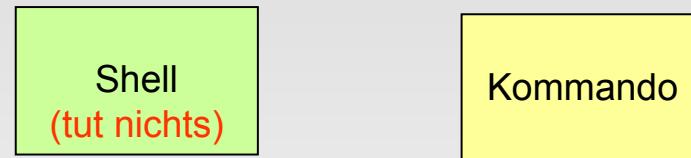
1. Die Shell liest ein Kommando ein



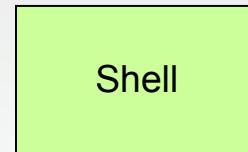
1. Der Shell-Prozess schafft eine Kopie von sich selbst



1. In der Kopie wird das Kommando zur Ausführung gebracht



4. Die Arbeit des Kommandos und der zugehörige Prozess sind beendet. Die Shell nimmt wieder Eingaben entgegen



- ▶ Man kann die Shell veranlassen, sofort nach dem Start des Kindprozesses ihr Prompt-Zeichen auszugeben und neue Kommandos entgegenzunehmen.
- ▶ Dem Kommando wird ein & -Zeichen nachgestellt.

Beisp.:

```
~>cc test.c &  
47  
~>
```

- ▶ Die Übersetzung läuft im „Hintergrund“ → Das wait entfällt
- ▶ Sollen mehrere Prozesse im Hintergrund abgearbeitet werden, ist die Kommandofolge zu klammern

Beisp.:

```
~>(cc test.c ; a.out) &
```

Kommandoprozeduren

- ▶ Textdatei, in der Kommandos gespeichert sind (= *Shell-Script*)
- ▶ Shell-Script repräsentiert selbst ein Kommando
- ▶ Anlegen:

1. echo "date;pwd" > datepwd
2. mittels Texteditor

- ▶ Ausführen:

- a) Starten

1. sh < datepwd
2. sh datepwd
3. chmod u+x datepwd
datepwd
4. . datepwd

Starten einer Subshell

- b) Beenden

- Shell beendet Prozedur nach Ausführung der letzten Anweisung
- exit *n* mit *n* < 256

Variablen („Zeichenvariablen“) → wesentliches Konzept aus höheren Programmiersprachen

- Speicherung von Zeichenketten-Objekten
- Benennung

variablename → *bu* (*bu* | *zi* | [])*

bu → ...

zi → ...

(Empfehlung: Kleinbuchstaben)

- Wertzuweisung

wertzuweisung → *variablename=[ausdruck]*

| *read variablename*

ausdruck → *literal*

| *`kommandofolge`*

- Bezugnahmen auf den Wert

`$variablename`

`${variablename}`

Beispiele:

1) Literal

Zuweisung a=/glb/studi
 cd \$a

Abfrage Wert echo \$a

Verkettung mit Zeichenkette

echo \${a}/short/abk

2) Kommandosubstitution

Der Wert in einer Zuweisung an eine Variable ergibt sich als Ergebnis der Ausführung einer Kommandofolge

```
~>pwd  
/u/fritzschen/uprakt  
~>a=`pwd`  
~>echo $a  
/u/fritzschen/uprakt
```

Erklärung:

```
~>pwd  
/u/fritzschen/uprakt  
~>a=`pwd`          1. Schritt: pwd → /u/fritzschen/uprakt  
                      2. Schritt: a=/u/fritzschen/uprakt  
~>echo $a  
/u/fritzschen/uprakt
```

3) Lesen von `stdin`

```
~>read a
```

liest eine Zeile von `stdin` und weist die Zeichenkette der Variablen `a` als Wert zu.

1+2 ↘

```
~>echo $a  
1+2
```

Sichtbarkeit (Scope) von Variablen

- ▶ Variablen sind nur in der Shell sichtbar, in der sie definiert werden.
- ▶ Variablen erhalten Werte auf eine der folgenden Arten:
 - der Variablen wurde ein Wert vom Anwender zugewiesen
 - durch die Shell (Variablen mit spezieller Bedeutung für die Shell)
 - der Wert wurde über den Umgebungsbereich an die Shell übergeben

Der Umgebungsbereich (**Environment**)

- ▶ Jedem Prozess zugeordnet: Menge von Assoziationen
 - Name – Zeichenkettenwert*
 - ...
- Das im Prozess ausgeführte Programm kann auf die Zeichenkettenwerte im Umgebungsbereich über die Namen zugreifen
- ▶ Die Shell bekommt beim Aufruf einen Umgebungsbereich übergeben und legt lokale Shellvariablen an.

- Beim Start eines externen Kommandos übergibt die Shell eine Kopie des Umgebungsbereichs, den sie selbst bei ihrem Start erhalten hat.

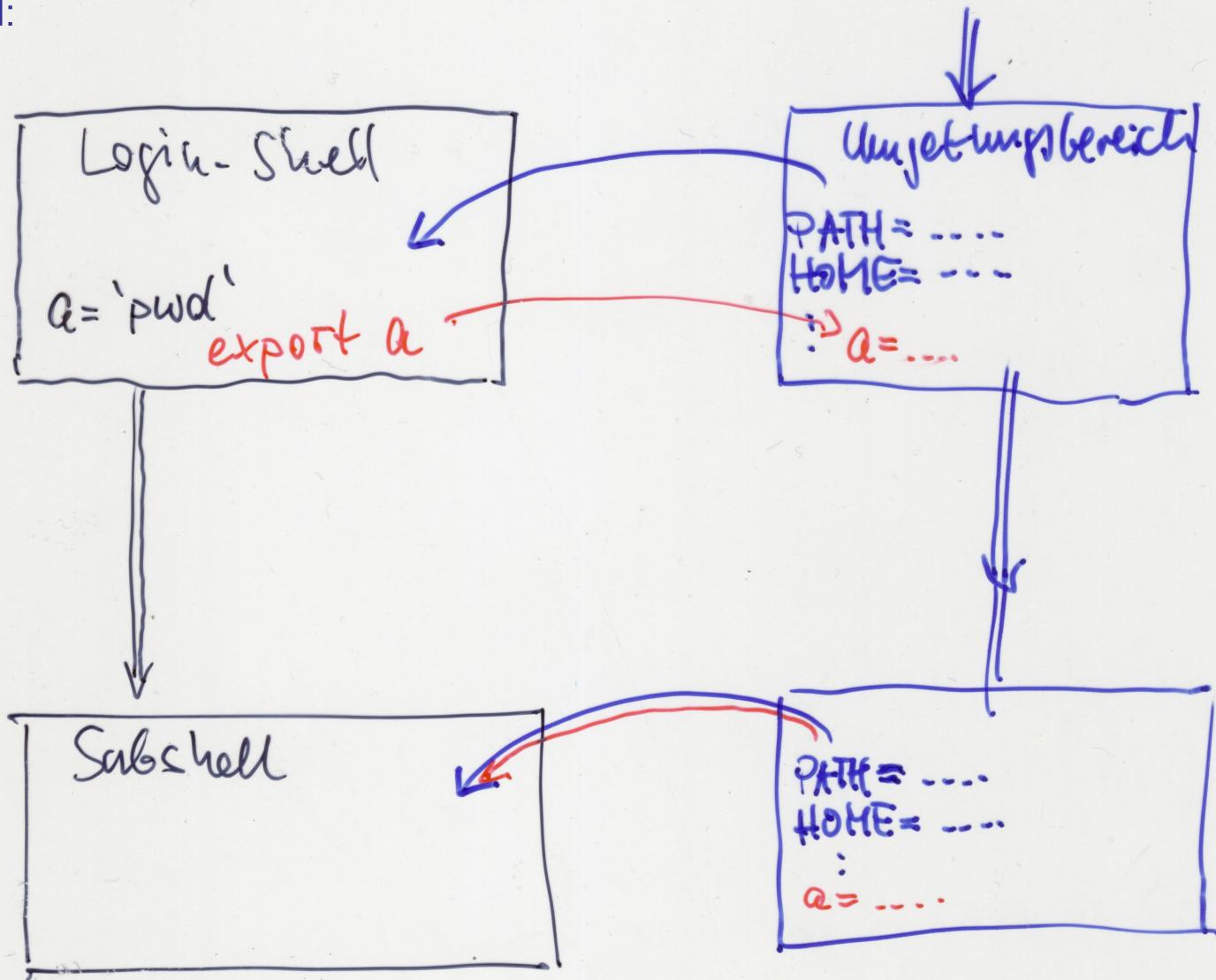
Beispiel für Umgebungsbereich:

```
PATH=/usr/local/bin:/usr/bin  
TERM=vt100  
HOME=/u/fritzs  
PS1=hallo  
PS2=>
```

```
export variablename [= wert] ...
```

Die Variablen *variablename* ... werden in das Environment aufgenommen, das Kindprozessen bei der Ausführung von Kommandos übergeben wird. Ggf. wird den Variablen jeweils *wert* zugewiesen.

Beispiel:



Beispiel:

proz2

```
ls $a
```

~>a=`pwd`

/u/fritzschen/uprakt/dir1

~>sh proz2

u v w proz2

~>cd ..

~>ls

x y z dir1

~>sh dir1/proz2

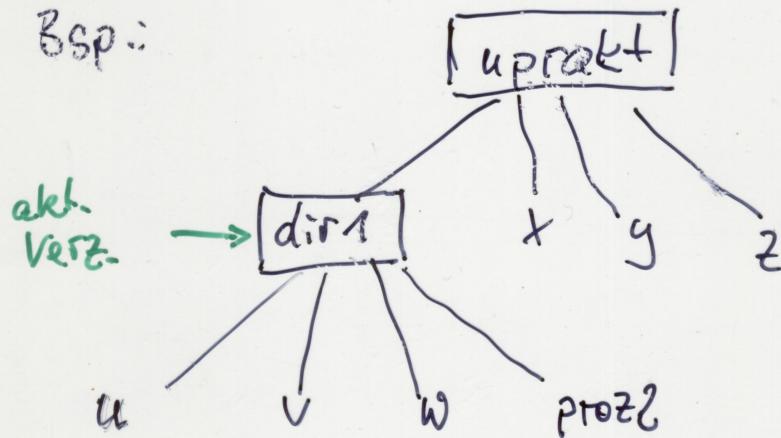
x y z dir1

~>export a

~>sh dir1/proz2

u v w proz2

Bsp.:



Parameterübergabe

- ▶ Positionsparameter werden wie beim Aufruf von Kommandos übergeben (durch Trennzeichen getrennt, hinter dem Namen der Prozedur)
- ▶ Auf Positionsparameter in der Befehlszeile kann in der Kommandoprozedur über Variablen zugegriffen werden.

Spezielle vordefinierte Variablen:

\$0	Name der Kommandoprozedur, die gerade ausgeführt wird
\$1, \$2, ...	1. Positionsparameter, 2. Positionsparameter, ...
\$#	Anzahl der Positionsparameter
\$*	alle Positionsparameter
\$\$	Prozessnummer der Shell
\$?	Rückkehrcode des zuletzt ausgeführten Kommandos
\$!	PID des letzten Background-Kommandos

```
echo "ich heisse $0"
echo "ich wurde mit $# Parametern aufgerufen"
echo "mein erster Parameter ist $1"
echo "mein zweiter Parameter ist $2"
echo "mein dritter Parameter ist $3"
echo "mein vierter Parameter ist $4"
echo "mein fuenfter Parameter ist $5"
echo "meine saemtlichen Parameter: $*"
```

\$5 :
leere
Zeichenkette

```
fritzsch@isys12:~/BS> /glb/studi/parmueb huhu haha $HOME `pwd`  
ich heisse /glb/studi/parmueb  
ich wurde mit 4 Parametern aufgerufen  
mein erster Parameter ist huhu  
mein zweiter Parameter ist haha  
mein dritter Parameter ist /u/fritzsch  
mein vierter Parameter ist /u/fritzsch/BS  
mein fuenfter Parameter ist  
meine saemtlichen Parameter: huhu haha /u/fritzsch /u/fritzsch/BS  
fritzsch@isys12:~/BS>
```

```
fritzsch@isys12:~/BS> echo $?  
0 ←  
fritzsch@isys12:~/BS>
```

► Anzeigen/Wertzuweisung für lokale Shellvariablen

```
set [-optionen] [arg . . .]
```

Ohne Parameter zeigt `set` die Namen und Werte aller lokalen Shell-Variablen an. Die Argumente `arg` werden (falls vorhanden) den Variablen `$1, $2, ...` in der angegebenen Reihenfolge zugewiesen.

Beispiel:

```
~>set `echo a b c`  
~>echo $*  
a b c  
~>set `pwd`  
~>echo $*  
/u/fritzs
```

► Anzeigen des Umgebungsreiches

```
env
```

Ablaufsteuerung in Kommandoprozeduren:

- Sequenz
 - Alternative
 - Iteration
-

Sequenz

kommandofolge

Der RC einer Kommandofolge ist der RC des letzten Kommandos der Folge

Beispiel:

proz3

```
pwd  
date  
ps
```

```
~> ./proz3  
/u/fritzschen/uprakt  
Do Nov 23 08:05:10 CET 2006  
    PID  TTY          TIME CMD  
 25911  pts/0        00:00:00 bash  
 25969  pts/0        00:00:00 bash  
 25971  pts/0        00:00:00 ps  
~> echo $?  
0
```

Alternative

`if`-Anweisung:

```
if  kommandofolge1
then   kommandofolge2
[ else   kommandofolge3 ]
fi
```

Falls der RC von `kommandofolge1` gleich 0 ist,
wird `kommandofolge2` ausgeführt,
andernfalls `kommandofolge3` (wenn vorhanden).

Beispiel:

```
if test -d $1
then
    ls -l $1
else
    echo "$1 existiert nicht oder ist kein Verz."
fi
```

test: RC = 0, falls \$1 eine existierende Datei und ein Verzeichnis ist,
RC ≠ 0 sonst

Schachtelung:

```
if komfolge1
then komfolge2
else if komfolge3
    then komfolge4
    else komfolge5
    fi [ ]
fi
```

```
if komfolge1
then komfolge2
elif komfolge3
then komfolge4
else komfolge5
fi
```

case-Anweisung:

```
case ausdruck in  
    zeichenmuster ) kommandofolge ;;  
    ...  
esac
```

Die Shell testet sukzessive, ob die angegebenen *zeichenmuster* auf den Zeichenkettenwert von *ausdruck* passen.

Wenn ja, wird die zugehörige *kommandofolge* ausgeführt und die Prozedur mit dem auf `esac` logisch folgenden Kommando fortgeführt.

Beispiel:

pcase

```
~> pcase  
weiß  
~> echo $?  
4  
~>
```

```
read x  
case $x in  
    rot) exit 1;;  
    blau) exit 2;;  
    gelb) exit 3;;  
    *)    exit 4;;  
esac
```

Iteration (Schleife)

while-Anweisung:

```
while kommandofolge1
do   kommandofolge2
done
```

kommandofolge1 wird ausgeführt. Der RC des letzten ausgeführten Kommandos der *kommandofolge1* wird getestet.

RC \neq 0 : Ausführung beendet, Fortsetzung mit dem auf *done* logisch folgenden Kommando

RC = 0 : Ausführung von *kommandofolge2*, anschließend erneut *kommandofolge1 usw.*

Beispiel:

```
n=1
while test $1 -ge $n
do
    echo "$n-ter Schleifendurchlauf"
    n=`expr $n + 1`
done
```

Lernzettel nicht vergessen!

Ein Aufruf bsp_while 4 erzeugt folgende Ausgabe auf stdout:

```
1-ter Schleifendurchlauf
2-ter Schleifendurchlauf
3-ter Schleifendurchlauf
4-ter Schleifendurchlauf
```

`until`-Anweisung:

```
until  kommandofolge1
do   kommandofolge2
done
```

kommandofolge1 wird ausgeführt. Der RC von *kommandofolge1* wird das erste Mal getestet, bevor *kommandofolge2* das erste Mal ausgeführt wird.

RC = 0 : die Ausführung der `until`-Konstruktion ist beendet.
Fortsetzung mit dem auf `done` folgenden Kommando

RC ≠ 0 : *kommandofolge2* wird ausgeführt und anschließend wieder *kommandofolge1* usw.

Beispiel:

```
until test -f dat1
do
    echo "bitte jetzt Datei dat1 bereitstellen!"
    sleep 360    # Unterbrechung der Prozedur
                  # für 360 sec.
done
```

Die Ausgabe der Meldung erfolgt alle 6 min., bis eine entsprechende Datei vorhanden ist.

for-Anweisung:

```
for name [in wert ...]  
do kommandofolge  
done
```

Die Variable name nimmt sukzessive die Werte wert ... an. Jeweils im Anschluss wird kommandofolge ausgeführt.

Falls *in wert ...* fehlt, wird als Standardwert
 *in \$** angenommen.

(*kommandofolge* wird damit für jeden Positionsparameter der Prozedur einmal durchlaufen)

Beispiel: erzeuge

```
for i in $*
do
    echo > $i
done
```

Aufruf: erzeuge a b c

Die Prozedur `erzeuge` erzeugt für jeden als Positionsparameter angegebenen Namen eine Datei mit diesem Namen.
Die Datei ist nicht leer!

test-Kommando

```
test      { -status datei }
          | {wert1 -vergleich wert2}
          | { zeichenkette1 = zeichenkette2}
          | { zeichenkette }
```

Das Kommando `test` setzt einen RC in Abhängigkeit davon, ob bestimmte Bedingungen erfüllt sind.
Falls ja, ist der RC = 0, sonst ungleich 0 (meist 1).

Einige mögliche Anwendungen sind:

`test -f dateiname`

RC = 0, falls eine Datei mit dem Namen `dateiname` existiert und kein Verzeichnis ist.

`test -d dateiname`

RC = 0, falls eine Datei mit Namen `dateiname` existiert und ein Verzeichnis ist.

`test -s dateiname`

RC = 0, falls eine Datei mit Namen *dateiname* existiert und nicht leer ist.

`test -r dateiname`

RC = 0, falls eine Datei mit Namen *dateiname* existiert und gelesen werden darf.

`test -w dateiname`

RC = 0, falls eine Datei mit Namen *dateiname* existiert und beschrieben werden darf.

`test string`

RC = 0, wenn *string* nicht die leere Zeichenkette ist.

`test s1 = s2`

RC = 0, wenn Zeichenkette *s1* gleich Zeichenkette *s2* ist.

`test n1 -ge n2`

RC = 0, wenn die ganze Zahl *n1* größer oder gleich der ganzen Zahl *n2* ist.

Es bedeuten (nach den englischen Bezeichnungen)

ge - größer oder gleich

gt - größer als

lt - kleiner als

le - kleiner oder gleich

ne - ungleich

eq - gleich

Hinweis: Es kann

[u ... u]

ausstelle von

test ...

verwendet werden

Bsp.:

if [u-f datu]

then ...

Es ist zu beachten, dass Shell-Variablen (z.B. `$1`) bei Zeichenkettenvergleichen in doppelte Apostrophe (" ... ") eingeschlossen werden müssen. In diesem Fall ergibt sich bei leeren oder undefinierten Variablen die leere Zeichenkette als Wert.

- Angenommen, eine Kommandoprozedur hat keinen Parameter übergeben bekommen. In diesem Fall liefert `test "$1"` den Wert `false`, während `test $1` einen Syntaxfehler ergibt, weil das Argument fehlt.

Beispiel:

`pset`

```
if test "$1" = $2
then
    echo $1 ist gleich $2
else
    echo $1 ist ungleich $2
fi
```

```
~> ./pset a
./pset: line 1: test: a: unary operator expected
a ist ungleich
~> ./pset a b
a ist ungleich b
```

Hinweis: Nennen Sie nie eine eigene Kommandoprozedur `test`, damit es nicht zu einem Konflikt mit dem Kommando `test` kommt!

Auswerten von Ausdrücken - expr

expr *argument* ...

argument → *operand* { *operator* *operand* } ...

Ergebnis der Auswertung des Ausdrucks *argument* auf `stdout`.
Numerische Operationen sind auf ganzzahlige Operationen beschränkt.

Exit-Status (Rückkehrcode):

- 0 - falls Ergebnis weder leere Zeichenkette noch 0
- 2 - falls Ausdruck ungültig (z.B. 2 + + 3)
- 1 - sonst

Es gelten die üblichen Vorrangregeln.

Operatoren: alle Operatoren sind zweistellig

1. Arithmetische Operatoren: +, -, *, /, %
1. Logische Operatoren: |, &
- 

false : leere Zeichenkette oder 0

true : sonst

Ergebnis: | *operand1*, falls dieser nicht *false*, sonst *operand2*
& 0, falls einer der Operanden *false*, sonst *operand1*

1. Vergleichsoperatoren: <, <=, >, >=, =, !=

true : 1 Ausgabe auf `stdout`, aber `RC=0` bei Ausgabe 1

false : 0

- „enthalten in“ - Operator : :

ausdruck1 : *ausdruck2*

Es wird geprüft, ob Zeichenmuster *ausdruck1* in Zeichenmuster *ausdruck2* enthalten ist.

AND-/OR-Verknüpfung von Kommandos

Grundsatz: Die Ausführung von Kommandos ist beendet, wenn der logische Wert feststeht.

```
kommando1 { && kommando2 } ...
```

kommando1 wird ausgeführt. kommando2 wird nur ausgeführt, wenn kommando1 den RC = 0 liefert (RC = 0 entspricht *true*).

```
kommando1 { || kommando2 } ...
```

kommando1 wird ausgeführt. kommando2 wird nur ausgeführt, wenn kommando1 einen RC ≠ 0 liefert (RC ≠ 0 entspricht *false*).

Negation von Kommandos

```
! kommando2
```

Beispiel: prozlog

```
if test -f $1 ||  
    (echo -n "aktueller Parameter bezeichnet "  
     "keine existierende Datei" && false)  
then  
    echo "Datei vorhanden. RC =" $?  
else  
    echo "RC =" $?  
fi
```

Aufruf: ~> prozlog prozlog
Datei vorhanden. RC = 0

~> prozlog proz2
aktueller Parameter bezeichnet keine existierende Datei
RC = 1

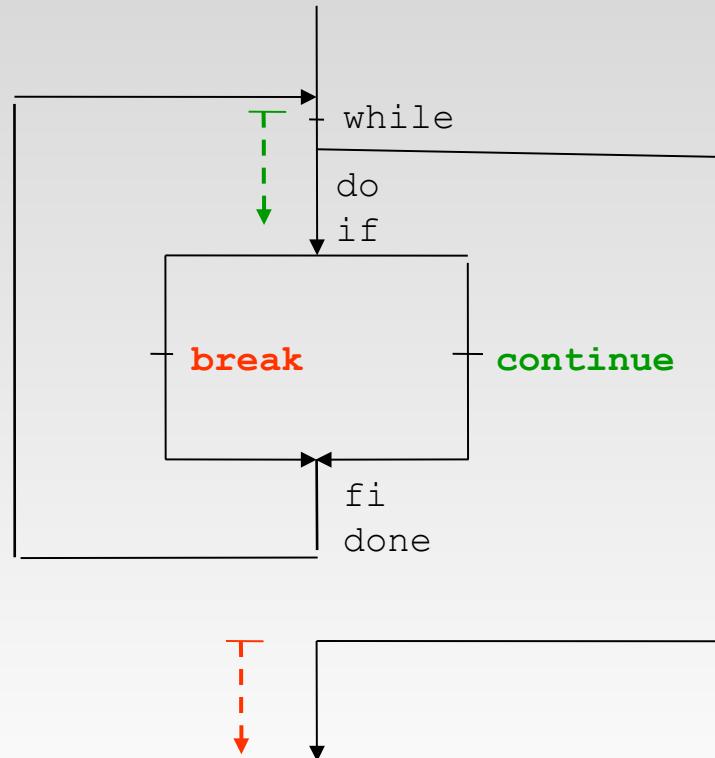
Zusätzliche Shell-interne Kommandos

break

Abbruch einer Schleife

continue

Abbruch eines Schleifendurchlaufs

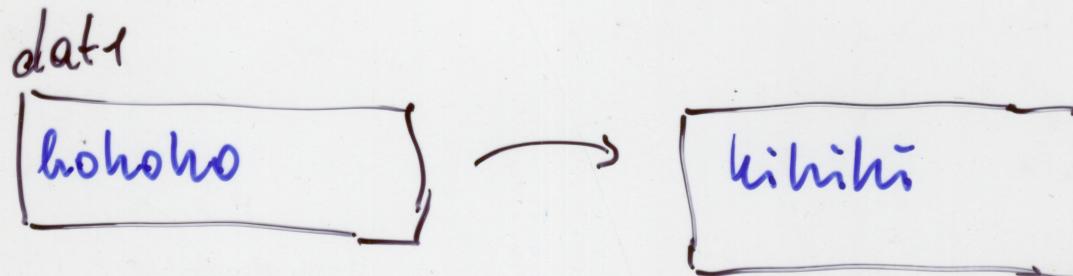


Standardeingabe für Kommandos

im Text einer Prozedur

Bsp:

subst2durch3 dat1 o i



In der Datei (Parameter 1) wird in allen Zeilen jedes Vorkommen von o (Parameter 2) durch i (Parameter 3) ersetzt.

subst durch 3

ed \$1 <<%

g/\$2/S//\$/3/g

w

q

%

Kommando

g /muster/ befehlsliste

Der „Globalbefehl“ g führt für alle Zeilen (im angegebenen Zeilenbereich), die das Zeichenmuster *muster* enthalten, die in der *befehlsliste* enthaltenen Befehle aus.

Befehl s (*substitute*): Ersetzen von Zeichenketten

[*zeilenangabe*] s /*muster/ersetzung/[g][p]*

- *zeilenangabe* gibt an, welche Zeilen im Arbeitsbereich betroffen sind.
Default: aktuelle Zeile
- s sucht das jeweils erste Vorkommen von *muster* in den betreffenden Zeilen und ersetzt es durch *ersetzung*.
- Die Angabe eines nachgestellten *g* bewirkt, dass jedes Vorkommen von *muster* in den betreffenden Zeilen ersetzt wird
- Bei Angabe von *p* wird die geänderte Zeile angezeigt

- Der Befehl *s* merkt sich die zuletzt substituierte Zeichenkette und kennt auch die zuletzt durch Textsuche gesuchte Zeichenkette.
- Folgen das erste und zweite Slash-Zeichen aufeinander, wird die gemerkte Zeichenkette ersetzt.