Bezeichnung von Dateien

- „von aussen“ ist eine Datei über eine **symbolische Adresse** bekannt/erreichbar
  - ein *benutzerdefinierter Name* von beliebiger aber maximaler Länge
  - dieser *Dateiname* ist für das Betriebssystem allgemein „Schall und Rauch“
  - je nach System (☞ Windows) erfolgt jedoch eine Interpretation des *Suffix*
- „nach innen“ besitzt die Datei eine logische bzw. **numerische Adresse**
  - diese Adresse identifiziert den die Datei beschreibenden „*Dateikopf*“
  - zwischen Dateiname und -kopf besteht eine *1:1*- oder *N:1*-Beziehung
  - je nach System (☞ UNIX) hat ein Dateikopf daher ggf. mehrere Namen
- symbolische und numerische Dateiadresse werden als Paar „verzeichnet“

Dateierweiterung

Dateinamensuffix oder *Extension*: eine üblicherweise durch einen Punkt vom Dateinamen abgegrenzte ***symbolische Erweiterung*** des Dateinamens

- liefert einen Hinweis auf das Dateiformat bzw. den Dateitypen

	Suffix	Art	Werkzeug	
☞	`.doc`	Textdokumente	MS-Word	
	`.fm`		Framemaker	`maker(1)`
	`.tex`		LaTeX	`latex(1)`
☞	`.h`	Programme	Präprozessor	`cpp(1)`
	`.c`		Kompilierer	`cc(1)`
	`.s`		Assemblierer	`as(1)`
	`.o`		Binder	`ld(1)`

- ist anwendungsspezifisch ausgelegt und ggf. dem Betriebssystem bekannt

Dateikopf

Indexknoten (*index node*, *inode*) im UNIX-Jargon, enthält *Dateiattribute*:

☞ Eigentümer (*user ID*)
☞ Gruppenzugehörigkeit (*group ID*)
☞ Typ (reguläre/spezielle Datei)
☞ Zugriffsrechte (lesen, schreiben, ausführen; Eigentümer, Gruppe, „Welt“)
☞ Zeitstempel (letzter Zugriff, letzte [*mode*-][26] Änderung)
☞ Anzahl der Verweise („*hard links*“)
☞ Größe (in Bytes)
☞ Adresse(n) der Daten auf dem Speichermedium

Indexknotennummer (*inode number*): die **numerische Adresse** der Datei

[26] Typ und Zugriffsrechte.

Dateityp

reguläre Datei (*regular file, ordinary file*) eindimensionales Bytefeld

Vom System vordefinierte, spezielle Dateien (*special files*):

Verzeichnis (*directory*) Katalog von regulären und/oder speziellen Dateien

Gerätedatei (*device file*) Zugang zu zeichen-/blockorientierten Geräte(treiber)n

„benanntes Rohr“ (*named pipe*) zwischen unverwandten lokalen Prozessen

„Sockel“ (*socket*) Kanal zwischen lokalen/entfernten Prozessen

Dateiverzeichnis

das **Verzeichnis** (*directory*)[27] dient der Gruppierung von Dateinamen

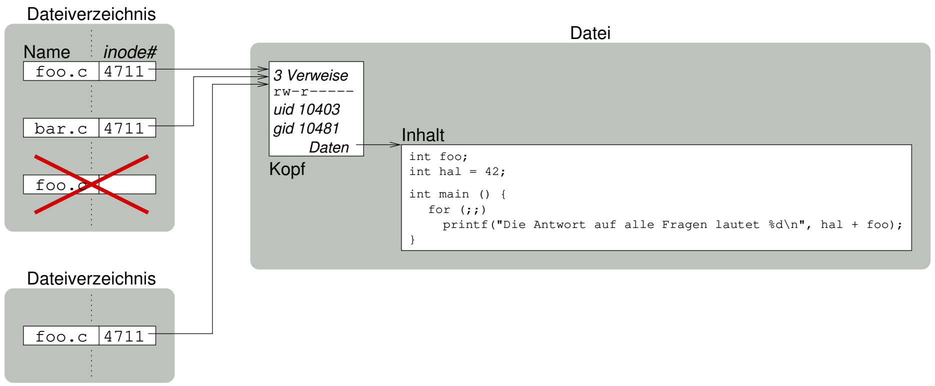
- definiert einen gemeinsamen *Kontext* für die symbolischen Adressen
  ☞ symbolische Adressen sind nur innerhalb ihrer Kontexte eindeutig
- implementiert eine „Umsetzungstabelle“ für symbolische Adressen:

$$\left\{\begin{array}{c}\text{symbolische Adresse}\\ \text{Dateiname}\\ \texttt{foo}\end{array}\right\} \leadsto \left\{\begin{array}{c}\text{numerische Adresse}\\ \text{Dateikopf}\\ \texttt{4711}\end{array}\right\}$$

- führt Buch über die Beziehung zwischen Dateinamen und Dateikopf

[27] Auch als „Katalog“ (*catalogue*) bezeichnet.

Dateiname vs. Dateikopf vs. Datei



Blockorientierte Datenspeicherung

- Dateiinhalte sind permanent gespeichert in Form von Blöcken fester Größe
  - typische Blockgrößen: 512 Bytes (B) – 8 Kilobytes (KB) [14]
  - in Abhängigkeit von $\left\{\begin{array}{l}\text{den physikalischen Parametern des Massenspeichers}\\ \text{der durchschnittlichen Dateilänge}\\ \text{dem Anwendungsprofil}\end{array}\right.$
- der Dateikopf führt Buch über die „Nummern“ der Datenblöcke der Datei
  - die Blocknummer entspricht einer physikalischen Adresse im Massenspeicher
  - die max. Dateilänge hängt ab von Anzahl/Wertebereich der Blocknummern
- Art und Weise der Blocknummernverwaltung ist abhängig von Zugriffsmustern

Zugriffsmuster

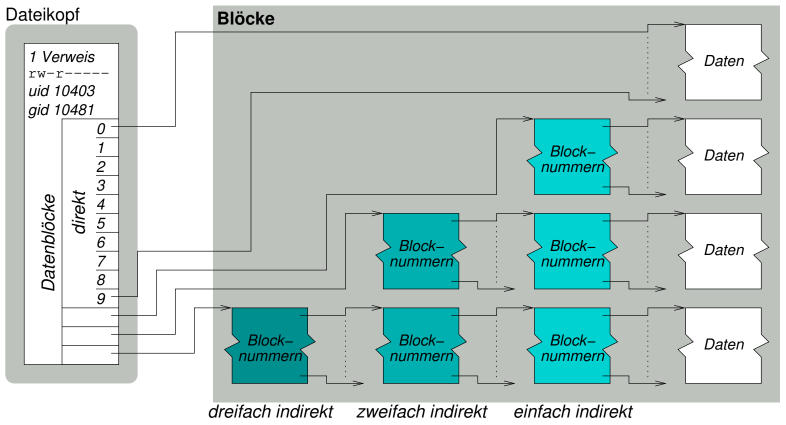
- Blockzugriffe seitens der Anwendung verlaufen sequentiell oder wahlfrei:

  sequentieller Zugriff Dateizugriffe erfolgen nach wohlgeordnetem Muster
  - die Datei wird von vorne nach hinten gelesen/beschrieben
  - geeignet für sequentiell organisierte Massenspeicher (z. B. Magnetbänder)

  wahlfreier Zugriff Dateizugriffe erfolgen nach beliebigen Muster
  - ein wohlgeordnetes Zugriffsmuster ist nicht erkennbar
  - geeignet für wahlfrei organisierte Massenspeicher (z. B. Festplatte)
- entsprechend verfolgt die Blockverwaltung verschiedene Optimierungskriterien
  - Kompromisslösungen bilden jedoch das „Tagesgeschäft“ (☞ UNIX)

Datenblockadressierung

$$sizeof(file) \leq sizeof(block) \times (10 + N + N^2 + N^3)$$



Namensraum — *Namespace*

- bildet einen *Kontext*, in dem jeder Name eine eindeutige Bedeutung besitzt

„Java“ bedeutet im Kontext	{	„Geographie“	eine Insel
		„Genussmittel“	ein Heissgetränk
		„Informatik“	eine Programmiersprache
„C“ bedeutet im Kontext	{	„Sprache“	einen Buchstaben
		„Musik“	eine Note
		„Informatik“	eine Programmiersprache

- die Struktur/Organisation ist flach oder (meist) hierarchisch ausgelegt

Aufbau von Namensräumen

flache Struktur definiert nur einen einzigen Kontext

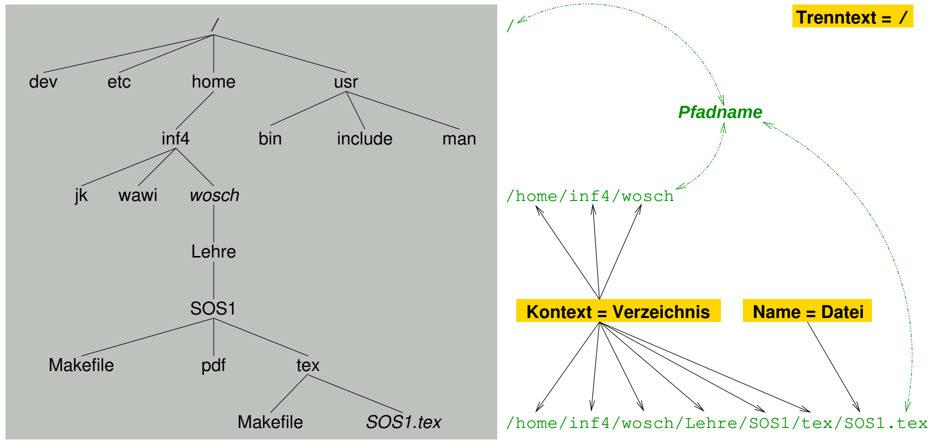
- Eindeutigkeit muss mit der Namenswahl selbst gewährleistet werden

hierarchische Struktur definiert mehrere Kontexte

- Eindeutigkeit wird über einen *Kontextnamen* als Präfix erreicht
- als *Separatoren* werden meist Sonderzeichen („Trenntext“) verwendet:

/	Schrägstrich (*slash*)	UNIX	☞ `foo/bar`
\	zurückgelehnter Schrägstrich (*backslash*)	Windows	☞ `foo\bar`

Hierarchischer Namensraum — Dateibaum (*file tree*)



Sonderverzeichnisse

Wurzelverzeichnis (*root directory*)

- die Wurzel (bei UNIX bezeichnet mit '/') im Dateibaum
- wird vom System gesetzt (privilegierte Operation ☞ `chroot(2)`)

Arbeitsverzeichnis (*working directory*)

- die gegenwärtige Position eines Programms/Prozesses im Dateibaum
- ändert sich beim „Durchklettern“ (☞ `chdir(2)`) des Dateibaums

Heimatverzeichnis (*home directory*)

- das initiale Arbeitsverzeichnis eines Benutzers/Prozesses
- wird vom System gesetzt bei Sitzungsbeginn (*login session* ☞ `login(1)`)

„Spitznamen“ für Verzeichnisse

`.` (*dot*) das aktuelle *Arbeitsverzeichnis*

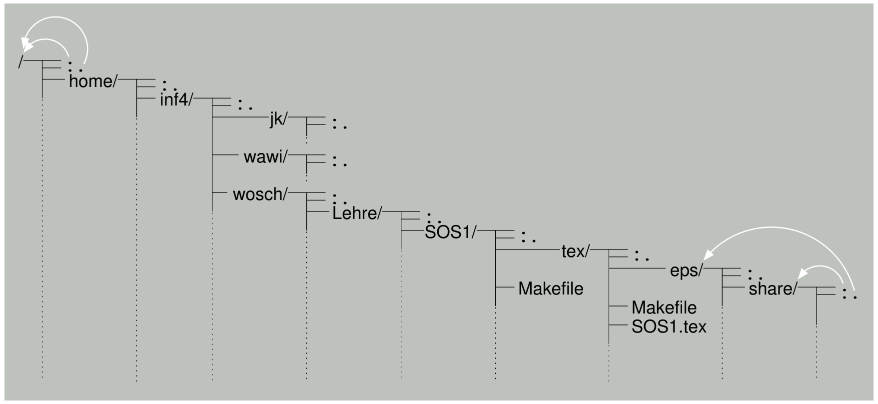
- zur relativen Adressierung des jeweiligen Arbeitsverzeichnisses[28]
- 1. Eintrag in jedem Verzeichnis (UNIX ☞ `mkdir(2)`)

`..` (*dot dot*) das aktuelle *Elternverzeichnis*

- zur relativen Adressierung des jeweils übergeordneten Verzeichnisses
  - entspricht '.', falls es kein Elternverzeichnis gibt (Wurzelverzeichnis)
- 2. Eintrag in jedem Verzeichnis (UNIX ☞ `mkdir(2)`)

[28] Ermöglicht beispielsweise die Implementierung von `ls(1)`, um die Einträge eines Arbeitsverzeichnisses auflisten zu können, ohne dessen wirklichen Namen kennen zu müssen. Ferner kann durch den Präfix „`./`“ zum Namen einer ausführbaren Datei der Aufruf lokaler Kommandos/Programme durchgesetzt werden.

Dynamische Datenstruktur „Dateibaum“



„Navigation“ im Namensraum

- im hierarchischen Namensraum beschreiben „Pfade“ die Wege zu Datei(nam)en
  - ein **Pfadname** (*pathname*) ist ein vollständiger Dateiname
- formaler Aufbau eines (UNIX) Datei- bzw. Pfadnamens in EBNF[15]:

```
pathname      = resolver | [resolver], {name, resolver}, name;
resolver      = {separator}-;
separator     = "/";
name          = {character}-;
character     = character set - separator;
character set = ASCII;
```

- Beispiele: `/`, `.`, `..`, `foo`, `foo/bar`, `/foo`, `bar/`, `./bar/..`, `../foo/.//bar//`

Pfadname

relativer ∼ vom Arbeitsverzeichnis (*working directory*) ausgehend,
z. B. von `/home/inf4/wosch` aus:

☞ `Lehre/SOS1/SOS1.tex`
☞ `./Lehre/SOS1/SOS1.tex`
☞ `../wosch/Lehre/SOS1/SOS1.tex`

oder von `/home/inf4/jk` aus:

☞ `../wosch/Lehre/SOS1/SOS1.tex`

absoluter ∼ vom Wurzelverzeichnis (*root directory*) ausgehend:

☞ `/home/inf4/wosch/Lehre/SOS1/SOS1.tex`

Dateisystem

- Komplex von Datenstrukturen zur Verwaltung von Dateien und Verzeichnissen

  Dateisystemkopf (*super block*) zur Beschreibung des Dateisystems
  Dateikopftabelle (*inode table*) zur Beschreibung von Dateien/Verzeichnisse
  Datenblöcke zur Speicherung der Inhalte der Dateien/Verzeichnisse

- eine (heterogene) dynamische Datenstruktur beschränkten Ausmaßes
  - der Dateisystemkopf legt die jeweiligen „Grenzwerte“ fest

- wird auf genau eine **Partition**[29] (z. B. einer Festplatte) abgebildet

[29] Partitionierung: die logische Unterteilung des Hintergrundspeichers in einen Satz zusammenhängender Sektoren.

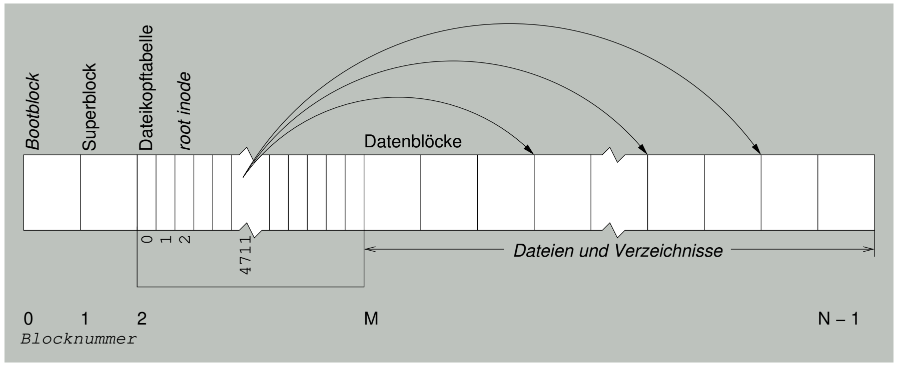
Dateisystemkopf

Superblock (*UNIX File System*, S5FS) logische Blocknummer 1

- enthält Informationen, die das Dateisystem beschreiben und dimensionieren:
  - Name des Wurzelverzeichnisses des Dateisystems
  - Anzahl der dem Dateisystem zugeordneten Blöcke
  - Größe der Dateikopftabelle (d. h., die Anzahl von *inodes*)
  - Anzahl und Liste freier Dateiköpfe bzw. Datenblöcke
- zum „Montieren“ von Dateisystemen benötigte (statische) Systemparameter
  - dynamisches Einbinden von Dateisystemen in (ein) bereits bestehende(s)

☞ Zerstörung des Superblocks z. B. durch einen Absturz (*crash*) der Festplatte hat „unangenehme“ Auswirkungen auf das zugehörige Dateisystem

Dateisystemformat — Partition



Namensauflösung

Abbildung (symb. Addresse ⇝ num. Adresse) .... `creat(2)`/`link(2)`

☞ Bindungsfunktion: den Dateinamen $\left\{ \begin{array}{l} \text{mit einem Dateikopf verknüpfen} \\ \text{in ein Dateiverzeichnis eintragen} \end{array} \right.$

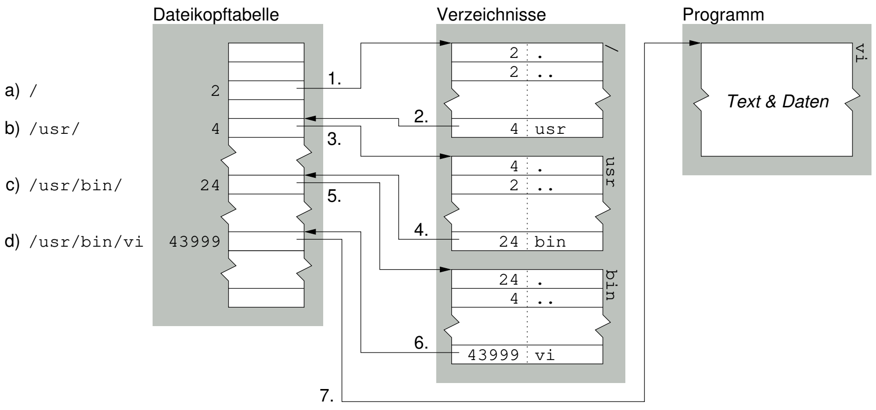
☞ einen Pfadnamen mit einem Dateikopf (*inode*) assoziieren

Umsetzung (symb. Addresse ⇝ num. Adresse) .... `open(2)`

☞ Auflösungsfunktion: einen Pfadnamen interpretieren
☞ Dateiverzeichnisse nach Namenseinträgen durchsuchen

✓ schrittweise für jeden einzelnen Verzeichnisnamen im Pfad
✓ schließlich für den Dateinamen

Auflösung von `/usr/bin/vi`



S5FS „*Considered Harmful*“

- Dateiköpfe liegen vorne auf der Platte, Dateien (inkl. Verzeichnisse) hinten
  - zeitaufwändige Namensauflösung, einhergehend mit „kostspieliger“ E/A
  - nicht jede Dateiänderung ist durchgängig in Bezug auf Hintergrundspeicher
- mehrere „*single point of failure*“[30] wirken Zuverlässigkeit/Robustheit entgegen
  - der Dateikopf (*inode*) — enthält alle Attribute einer Datei
  - die Dateikopftabelle (*inode table*) — enthält alle Dateiköpfe
  - der Superblock — enthält die Parameter des gesamten Dateisystems
- Stromausfall oder „*head crash*“ können sich äußerst unangenehm auswirken

[30] Eine beliebige Komponente eines Systems, die im Fehlerfall den Ausfall des Gesamtsystems nach sich zieht.

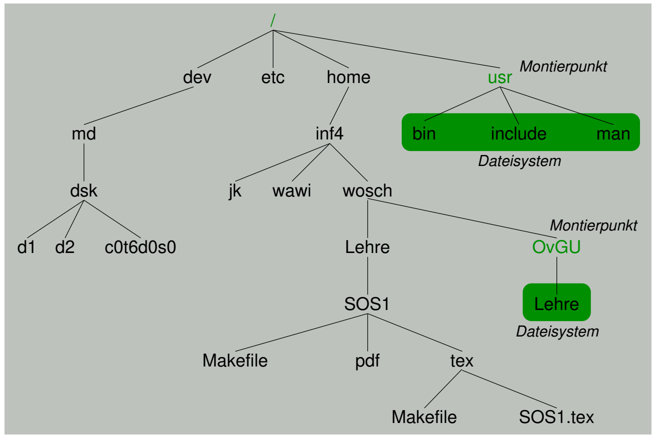
Dateisystemmontage

Montierpunkt (*mount point*) eine Stelle im „Wirtsdateisystem“, an der die Einbindung eines anderen Dateisystems möglich ist

☞ ein Verzeichnis wird mit der Wurzel eines anderen Dateisystems überlagert

- in Bezug auf das Wirtsdateisystem sind montierbare Dateisysteme . . .
  - von gleicher/verschiedener Struktur (z. B. S5FS, UFS, FFS, EXT2, NTFS)
  - auf demselben/einem anderen Gerät (ggf. einem anderen Rechner) resident
- sie bilden eine eigene *Partition*, definiert über eigene Systemparameter

Dateisystemmontage ~wosch/OvGU, /usr



`/dev/md/dsk/d1`

`/dev/md/dsk/d2`

`mount /dev/md/dsk/d2 /usr`

`/dev/md/dsk/c0t6d0s0`

Dateisystemadressraum

- zwei Sorten von Adressen kommen im (S5FS) Dateisystem zur Verwendung:

  Blockadressen (Blocknummern) verweisen auf die für die Speicherung der Informationen zur Verfügung stehenden Datenblöcke
  ☞ Dateisysteme sind auf *blockorientierte Massenspeicher* ausgerichtet
  Dateikopfadressen (Indexknotennummern, *inode numbers*) identifizieren die Strukturen zur Verwaltung von Dateien/Verzeichnissen

- Adressen sind nur gültig (bzw. eindeutig) innerhalb ihres Adressraums
  - Dateiköpfe montierter Dateisysteme sind nicht direkt adressierbar
  - nur über Pfadnamen können Dateien/Verzeichnisse global erreicht werden

- Dateikopfadressen sind kontextabhängig zu betrachten und zu behandeln

Dateikopfreferenzen

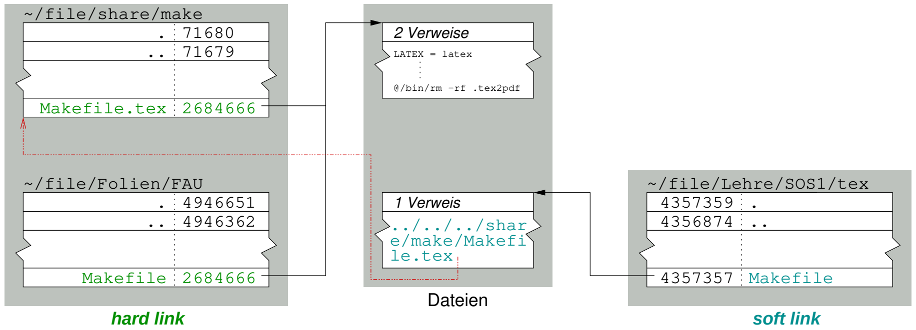
harte **Referenz** (*hard link*, *link*)

- eine Verknüpfung zwischen Dateiname und Dateikopf (*inode*)
  - ist nur lokal (innerhalb eines Dateisystems) definiert
  - als Paar gespeichert im Verzeichnis (Namenseintrag)
- eine „direkte Adresse" auf eine Datei

symbolische Referenz (*symbolic link*, *soft link*)

- eine Verknüpfung zwischen Dateiname und Pfadname
  - kann global (d. h, Dateisystem übergreifend) definiert sein
  - gespeichert im Verzeichnis oder als (vom System definierte) Datei
- eine „indirekte Adresse" auf eine Datei

„*Hard Link*" vs. „*Soft Link*"



Symbolische Referenz „*Considered Harmful*“

```
wosch@lorien 1$ mkdir -p Laptop/faui43w; cd Laptop
wosch@lorien 2$ ln -s faui43w lorien
wosch@lorien 3$ ls -l
total 8
drwxr-xr-x  2 wosch  wosch  68 29 Apr 13:01 faui43w
lrwxr-xr-x  1 wosch  wosch   7 29 Apr 13:02 lorien -> faui43w
wosch@lorien 4$ cd lorien
wosch@lorien 5$ cd ..; rmdir faui43w; cd lorien
-bash: cd: lorien: No such file or directory
wosch@lorien 6$ ls -l
total 8
lrwxr-xr-x  1 wosch  wosch  7 29 Apr 13:02 lorien -> faui43w
wosch@lorien 7$ mkdir faui43w; cd lorien
wosch@lorien 8$ ln -s "gibt es nicht" jippdit
```


Datei — UNIX Systemaufrufe (3)

`fd = open (path, flags, mode)` erzeugt einen Deskriptor für die mit dem Pfadnamen bezeichnete Datei (öffnet die Datei)[31]

- der *Dateizeiger* (*file pointer*) wird auf den Anfang (0) der Datei gesetzt

`ok = close (fd)` invalidiert einen Dateideskriptor (schließt die Datei)

`newfd = dup (oldfd)` dupliziert einen Dateideskriptor (`oldfd`)

[31] Der `mode` Parameter ist erforderlich beim Erzeugen einer Datei, d. h., wenn der Status „`O_CREAT`“ vorgibt.

Datei — UNIX Systemaufrufe (4)

`ok = link (path1, path2)` erzeugt eine „harte Referenz“ (*hard link*)

- der Referenzzähler im (direkt adressierten) *Inode* wird um 1 erhöht
  - `path1` ist der „ursprüngliche“ Pfadname dieser Datei
  - `path2` ist der „alternative“ Pfadname dazu
- ursprünglicher/alternativer Pfadname sind (danach) <u>ununterscheidbar</u>

`ok = unlink (path)` löscht eine „harte Referenz“ (*hard link*)

- der Referenzzähler im (direkt adressierten) *Inode* wird um 1 erniedrigt
- ist der Zählerwert 0, wird die Datei endgültig gelöscht

Datei — UNIX Systemaufrufe (5)

`ok = symlink (path1, path2)` erzeugt eine „weiche Referenz“ (*soft link*)

- der Referenzzähler im (indirekt adressierten) *Inode* bleibt unverändert
  - vielmehr wird ein neuer *Inode* (für `path2`) angelegt
  - der Referenzzähler dieses *Inodes* wird mit 1 initialisiert
- auf eine (bekannte) Datei wird eine *symbolische Referenz* angelegt
  - löschen einer solchen Referenz, löscht nicht die Datei
  - löschen der Datei, erzeugt eine “verwaiste symbolische Referenz“
- erlaubt das Setzen einer „indirekten Adresse“ auf *alle* Arten von Dateien
  - wie z. B. Verzeichnisse oder etwa Einträge anderer Dateisysteme
  - eine solche Adresse ist zusätzlich „hart“ referenzierbar (`link(2)`)

Datei — UNIX Systemaufrufe (6)

`nr = read (fd, buf, nbytes)` Datei ⇒ Puffer

- der Dateizeiger wird um die Anzahl der gelesenen Zeichen weiter gesetzt

`nw = write (fd, buf, nbytes)` Puffer ⇒ Datei

- der Dateizeiger wird um die Anzahl der geschriebenen Zeichen weiter gesetzt

`rwp = lseek (fd, offset, whence)` positioniert den Dateizeiger

Datei — UNIX Systemaufrufe (7)

`ok = fcntl (fd, cmd, arg)` verwaltet eine (geöffnete) Datei

- der optionale Parameter `arg` hängt ab vom Kontrollkommando `cmd`
  - duplizieren von Dateideskriptoren, manipulieren von Dateiattributen
  - freiwilliges Sperrverfahren (*advisory record locking*) zur Koordination
- die geöffnete Datei wird durch einen Dateideskriptor (`fd`) identifiziert

`ok = ioctl (fd, request, argp)` verwaltet ein Gerät oder einen Strom

- die abgesetzten Kommandos lösen gerätespezifische Steuerfunktionen aus
- der Gerätetreiber interpretiert den Befehl samt optionale Parameter `argp`

Prozess . . .

. . . wird durch ein Programm kontrolliert und benötigt zur Ausführung dieses Programms einen Prozessor.

Habermann, *Introduction to Operating System Design*

. . . P ist ein Tripel (S, f, s), wobei S einen Zustandsraum, f eine Aktionsfunktion und $s \subset S$ die Anfangszustände des Prozesses P bezeichnen. Ein Prozess erzeugt Abläufe, die durch die Aktionsfunktion generiert werden können.

Horning/Randell, *Process Structuring*

. . . ist das Aktivitätszentrum innerhalb einer Folge von Elementaroperationen. Damit wird ein Prozess zu einer abstrakten Einheit, die sich durch die Instruktionen eines abstrakten Programms bewegt, wenn dieses auf einem Rechner ausgeführt wird.

Dennis/van Horn, *Programming Semantics for Multiprogrammed Computations*

. . . ist ein Programm in Ausführung.

unbekannte Referenz, „Mundart“

Prozess $\neq$ Programm (1)

- ein Prozess kann die Ausführung mehrerer Programme zur Folge haben:

  Systemaufruf *Anwendungsprogramm* ruft ein *Betriebssystemprogramm* auf
  - ☞ der Prozess ist der *Aktivitätsträger* von mindestens zwei Programmen

- umgekehrt kann ein Programm von mehreren Prozessen ausgeführt werden

  nicht-sequentielles Programm im Falle von Uniprozessorsystemen
  - ☞ „präemptive Programmverarbeitung“[32]
  - ☞ Faden (*thread*), Unterbrechung (*interrupt*)

  paralleles Programm im Falle von Multiprozessorsystemen

[32] Die Ausführung einer Aufgabe kann jederzeit von einer höheren Instanz unterbrochen werden.

Prozess ≠ Programm (2)

☞ Wissen über das gegenwärtig ausgeführte Programm sagt nicht viel aus über die zu dem Zeitpunkt im System stattfindende Aktivität.

- Welche bzw. wieviel $\left\{\begin{array}{c}\text{Fäden führen das Programm zur Zeit aus}\\ \vdots \\ \text{Programmunterbrechungen sind zur Zeit aktiv}\end{array}\right\}$?

☞ Im Betriebssystemkontext ist das Konzept „Prozess“ daher nützlicher als das Konzept „Programm“, um Vorgänge zu beschreiben und zu verwalten.

Prozess ≠ Prozessinkarnation

Prozess ist ein abstraktes Gebilde

- ein „Programm in Ausführung“$\smile$, ein asynchroner Programmablauf$\smile$
- ein „Ablauf“$\smile$, der eine Verwaltungseinheit$\frown$ ist **?**

Prozessinkarnation ist ein konkretes Gebilde

- die „physische Instanz“ des abstrakten Gebildes „Prozess“
  - gebunden an (Software-) Betriebsmittel, verbunden mit einer *Identität*
- die *Verwaltungseinheit*, die einen Prozess beschreibt und repräsentiert

☞ im weitesten Sinne synonyme Begriffe, jedoch nicht in allen Fällen

Prozessmodelle

schwergewichtiger Prozess (*heavyweight process*, „klassischer“ UNIX Prozess)

- Prozessinkarnation und Benutzeradressraum bilden eine Einheit
- Prozesswechsel bedeutet zwei Adressraumwechsel $AR_x \Rightarrow BS \Rightarrow AR_y$

leichtgewichtiger Prozess (*lightweight process*, *kernel-level thread*)

- Prozessinkarnation und Adressraum sind voneinander entkoppelt
- Prozesswechsel bedeutet einen Adressraumwechsel $AR_x \Rightarrow BS \Rightarrow AR_x$

federgewichtiger Prozess (*featherweight process*, *user-level thread*)

- Prozessinkarnation<u>en</u> und Adressraum bilden eine Einheit
- Prozesswechsel entspricht einem *Koroutinenwechsel* (✗ Kap. 6)

Prozessbenutzthierarchie

schwergewichtiger Prozess

$\Downarrow$

leichtgewichtiger Prozess

$\Downarrow$

federgewichtiger Prozess

Einplanung

Auch „Ablaufplanung“, ist erforderlich, um die um den Prozessor (allgemein: die Betriebsmittel) konkurrierenden Prozesse geordnet ablaufen lassen zu können.

Scheduling stellt sich allgemein zwei grundsätzlichen Fragestellungen:

1. Zu welchem *Zeitpunkt* sollen Prozesse ins System eingespeist werden?
2. In welcher *Reihenfolge* sollen Prozesse ablaufen?

Ein ***Scheduling*-Algorithmus** verfolgt das Ziel, den von einem Rechnersystem zu leistenden Arbeitsplan so aufzustellen (und zu aktualisieren), dass ein gewisses Maß an Benutzerzufriedenheit maximiert wird.

Ablaufplanung — *Process Scheduling*

$$\text{Prozessen das Betriebsmittel} \left\{ \begin{array}{l} \textit{Prozessor} \\ \textit{Speicher}^{\dagger} \\ \textit{Gerät}^{\ddagger} \end{array} \right\} \text{zuteilen}$$

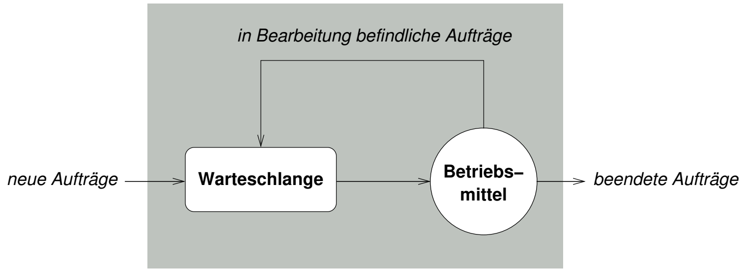
† Das Hardware-Betriebsmittel ({Vorder,Hinter}grund-) *Speicher* steht (insbesondere im Falle von RAM) auch stellvertretend für (wiederverwendbare/konsumierbare) Software-Betriebsmittel wie z. B. Puffer, Nachrichten, Signale.

‡ *Gerät* steht stellvertretend für Drucker, Netzwerk, Platte, . . .

Ablaufplan — *Process Schedule*

- *Fahrplan* zur Belegung der {Hard,Soft}ware-Betriebsmittel durch Prozesse
  - geordnet nach Ankunft, Zeit, Termin, Dringlichkeit, Gewicht, . . .
  - die Ordnung ist eine Funktion der Scheduling-Strategie (bzw. -Algorithmen)
- technisch (zumeist) realisiert auf Basis dynamischer Datenstrukturen
  - eine oder mehrere Queues bzw. Schlangen: *Warteschlangen*
  - die Elemente der Datenstruktur sind die *Prozessdeskriptoren*
- die gewählte Scheduling-Strategie bestimmt u. a. die Rechnerbetriebsart

Scheduling-Modell



Ein einzelner Scheduling-Algorithmus charakterisiert sich durch die Reihenfolge von Prozessen in der Warteschlange und die Bedingungen, unter denen die Prozesse der Warteschlange zugeführt werden.

Warteschlangentheorie

- Betriebssysteme durch die „theoretische/mathematische Brille“ gesehen:
  - ☞ R. W. Conway, L. W. Maxwell, L. W. Millner. *Theory of Scheduling*.
  - ☞ E. G. Coffmann, P. J. Denning. *Operating System Theory*.
  - ☞ L. Kleinrock. *Queuing Theory*.
- die Verfahren stehen und fallen mit den Vorgaben der jeweiligen *Zieldomäne*
  - eine „Eier-legende Wollmilchsau“ des Scheduling wird es nie geben
  - Kompromisslösungen sind geläufig, aber nicht in allen Fällen tragfähig
- *Scheduling* ist als „Querschnittsbelang“ von Betriebssystemen zu verstehen
  - diese Belange zu behandeln, ist eine der (praktischen) Herausforderungen

Speichermodell — UNIX Prozess (1)

UNIX Prozesse sind schwergewichtig: Prozess und Adressraum bilden eine Einheit.

```
int foo;
int hal = 42;

int main () {
    for (;;)
        printf("Die Antwort auf alle Fragen lautet %d\n", hal + foo);
}
```

Wie ist der Adressraum bzw. Speicher des ausführenden Prozesses organisiert?

Speichermodell — UNIX Prozess (2)

```
wosch@faui40 40$ gcc -O6 -static -o hal hal.c; ./hal
Die Antwort auf alle Fragen lautet 42 ... ^Z
wosch@faui40 41$ ps
   PID TTY      TIME CMD
 28426 pts/4    0:00 hal
   205 pts/4    0:00 ps
 25965 pts/4    0:00 tcsh-6.0
wosch@faui40 42$ pmap -x 28426
28426:  ./hal
 Address  Kbytes     RSS    Anon  Locked Mode   Mapped File
00010000     216     216       -       - r-x--  hal           ⇐ Textsegment
00054000      16      16       8       - rwx--  hal           } ⇐ Datensegment
00058000       8       8       8       - rwx--    [ heap ]    }
FFBFE000       8       8       8       - rw---    [ stack ]   ⇐ Stapelsegment
-------- ------- ------- ------- -------
total Kb     248     248      24       -
```

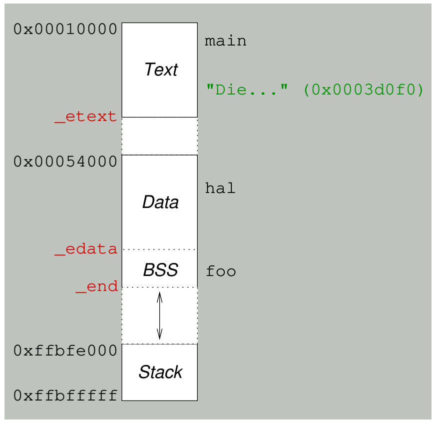

Speichermodell — UNIX Prozess (3)

```
    .file     "hal.c"
    .global   hal
    .section  ".data"
    .align    4
    .type     hal,#object
    .size     hal,4
hal:
    .uaword   42
    .common   foo,4,4
    .section  ".rodata"
    .align    8
.LLC0:
    .asciz    "Die Antwort auf
           alle Fragen lautet %d\n"
    .section  ".text"
    .align    4
    .global   main
    .type     main,#function
    .proc     04
main:
    !#PROLOGUE# 0
    save      %sp, -112, %sp
    !#PROLOGUE# 1
    sethi     %hi(hal), %l2
    sethi     %hi(foo), %l1
    sethi     %hi(.LLC0), %l0
    ld        [%l2+%lo(hal)], %g1
.LL5:
    or        %l0, %lo(.LLC0), %o0
    ld        [%l1+%lo(foo)], %o3
    call      printf, 0
    add       %g1, %o3, %o1
    b         .LL5
    ld        [%l2+%lo(hal)], %g1
.LLfe1:
    .size     main,.LLfe1-main
    .ident    "GCC: (GNU) 3.0.4"
```

gcc -O6 -S hal.c

Speichermodell — UNIX Prozess (4)

```
wosch@faui40 43> nm -p -g hal
 ⋮
 0000066112 T main      ☞ 0x00010240
 0000352140 D hal       ☞ 0x00055f8c
 0000360336 B foo       ☞ 0x00057f90
 ⋮
 0000286461 D _etext    ☞ 0x00045efd
 0000358433 D _edata    ☞ 0x00057821
 0000361444 D _end      ☞ 0x000583e4
 ⋮
```



Speichermodell — UNIX Prozess (5)

Vom Binder definierte Symbole:

`extern etext` ist die erste Adresse nach dem Programmtext

`extern edata` ist die erste Adresse nach dem initialisierten Datenbereich

`extern end` ist die erste Adresse nach dem uninitialisierten Datenbereich[33]

- mit Ausführungsbeginn entspricht `end` der „Bruchstelle" des Programms
  - kann zur Ausführungszeit mit `brk(2)`/`sbrk(2)` verschoben werden
- `sbrk((intptr_t*)0)` liefert den für den Prozess aktuell gültigen Wert

[33] Dem „*Block Started by Symbol* (BSS)" Segment, das vom Lader mit 0 vorinitialisiert wird.

Prozess — UNIX Systemaufrufe (8)

`pid = fork ()` erzeugt einen Kindprozess (exakte Kopie des Elternprozesses)

- zwei Prozesse kehren zurück: Rückgabewert $\left\{ \begin{array}{l} = 0 \\ > 0 \end{array} \right\}$ $\Rightarrow$ ~~Eltern~~prozess Kind / $\Rightarrow$ ~~Kind~~prozess Eltern

`pid = wait (status)` wartet auf Stillstand/Termination eines Kindprozesses

- ein terminierter Kindprozess („Zombie") wird erst jetzt entsorgt
- der Aufrufer erfährt (über `status`) die Stillstands-/Terminationsursache

`exit (status)` terminiert den aufrufenden Prozess

- der Prozess wird zum „Zombie" und vermerkt den Grund der Termination

Prozess — UNIX Systemaufrufe (9)

`nv = nice (incr)` ändert die Priorität des aufrufenden Prozesses

- der angegebene Wert wird zum „*nice value*" (nv) des Prozesses addiert
  - eine positive Zahl: je höher ihr Wert, desto niedriger die Priorität
- für die vom Benutzer einstellbare Priorität gilt: $0 \leq nv \leq (2 * NZERO) - 1$

`pid = getpid ()` liefert die Identifikation des aufrufenden Prozesses

`pid = getppid ()` liefert die Identifikation des Elternprozesses

UNIX Systemaufrufe (10)

`ok = execv (path, argv)` führt eine Datei aus

- das Speicherabbild des laufenden Prozesses wird ersetzt durch ein Programm
  - die angegebene „ausführbare Datei" legt das neue Speicherabbild fest
  - diese Datei wird ausgeführt (CPU) oder interpretiert (z. B. von `sh(1)`)
- das auszuführende Programm erhält einen Argumentenvektor (`argv[]`)
  - die Größe des Vektors wird bestimmt und als `argc` weitergegeben

`ok = execve (path, argv, envp)` dito

- das auszuführende Programm erhält zusätzlich Umgebungsparameter

Zusammenfassung

- Betriebssysteme bieten eine „Hand voll" nützlicher Abstraktionen an
  - Adressräume, Speicher, Dateien, Prozesse
- von zentraler Bedeutung ist die Abbildung von Namen auf Adressen
  - symbolische ⇝ numerische ⇝ logische ⇝ virtuelle ⇝ physikalische Adresse
  - ein Konzept, das im Kontext von Rechnernetzen seine Fortführung findet[34]
- je nach Betriebssystemart sind die Abstraktionen unterschiedlich ausgelegt
  - manche Abstraktionen müssen auch überhaupt nicht angeboten werden

[34] So sind z. B. Email-Adresse und URL nichts anderes als symbolische Adressen, die von „Namensdiensten" auf korrespondierende Rechneradressen (Internet-Hosts) abgebildet werden.