# Betriebssysteme

## Kapitel 7: Files

*7.1: Konzepte*

**Stand: WS 07/08**


Prof. Dr. Wolfgang Küchlin

*Dipl.-Inform., Dr. sc. techn. (ETH)*

**Arbeitsbereich Symbolisches Rechnen**
**Wilhelm-Schickard-Institut für Informatik**
**Fakultät für Informations- und Kognitionswissenschaften**

**Universität Tübingen**

**Steinbeis Transferzentrum**
**Objekt- und Internet-Technologien (OIT)**

Wolfgang.Kuechlin@uni-tuebingen.de
http://www-sr.informatik.uni-tuebingen.de

SR 


## Dateien (Files)



- Eine **Datei** besteht aus einer linearen Folge von bytes.
- Zugriff durch
  1. Positionieren
  2. Sequentielles read/write von Bytes
- Eine Datei ist immer auf einem nicht-flüchtigen Speicher „zuhause“, kann aber in den Hauptspeicher abgebildet (mmap) werden.
- Abstraktion „Datei“ verbirgt Eigenheiten von Platten oder Bändern (Adressierung, verstreute Speicherung in endlich großen Blöcken etc.)



## Dateien (Files)



- Einfache Verbund-Struktur
  - eine Zeilenstruktur (Textdateien; sehr wichtig für UNIX)
  - Datensätze fester Länge
  - Datensätze variabler Länge
- Komplexe Strukturen (z.B. ein formatiertes Dokument eines Textverarbeitungssystems; ein ausführbares Programmfile) durch Einfügen von Kontrollzeichen
- Interpretation der Kontrollzeichen entweder durch Betriebssystem oder durch Benutzerprogramm.



## Attribute von Dateien



- Dateien besitzen Attribute.
- Welche Attribute vorhanden sind, hängt vom Betriebssystem ab.
  - Unterschiede zwischen DOS,UNIX,Windows NT
- Häufig vorhandene Attribute
  - Dateiname
    - Enthält oftmals eine **Extension**, die die Interpretation des Inhalts unterstützt.
  - Typ
    - wird bei Systemen gebraucht, die verschiedene Dateitypen unterstützen
      - z.B. reguläres File, symbolischer Link, ...

## Attribute von Dateien (häufig vorhandene Attribute)

- Ort
  - Ein Zeiger auf den Ort des Files auf dem physischen Speichermedium.
- Größe in Bytes
  - der verbrauchte Platz ist i.a. größer, da immer ganze Blöcke alloziert werden.
  - Differenz = „interner Verschnitt“
- Schutzinformation
  - z.B. Information über Lese-, Schreib-, Ausführrechte
- Zeit, Datum, Benutzeridentifikation
  - evtl. unterteilt in Erschaffungszeit, letzter Schreibzugriff, letzter Lesezugriff





## Operationen auf Files

- Anlegen: create
- Öffnen: open
- Schließen: close
- Schreiben: write
- Lesen: read
- Löschen: delete
- Abschneiden: truncate
- Positionieren in der Datei: seek (UNIX: lseek)





## Operationen auf Files

- Zum einen geben Files die Abstraktion einer potentiell unendlichen Datenstruktur (vgl. auch Ströme [streams] ) und den eines persistenten Datenspeichers.
- Zum anderen erlauben viele Files auch einen blockweisen Zugriff (da sie oftmals über Blöcke auf einer Platte realisiert sind).
- Z.B. ist **lseek** nicht auf allen File-Arten sinnvoll (bzw. definiert). Auf der anderen Seite erlauben viele Files nicht nur ein read/write hinter der **letzten Position**, sondern auch Vor-/Rücksprung um Blöcke.





## Dateien und Dateikataloge / Verzeichnisse in UNIX

- Dateien sind Sequenzen von 0 oder mehr Bytes.
- Interpretation der Dateien ist Benutzersache.
- Dateinamen sind bis 255 Zeichen lang.
- Der Dateiteil, der nach einem (dem letzten) „.“ Im Dateinamen kommt, wird aber von vielen Programmen speziell interpretiert.
- Konvention:
  - *.c C-Programme
  - *.s Assembler
  - *.f Fortran

## Zugriffsrechte

- *Zugriffsrechte* sind gruppiert:


- Beispiel: chmod o+r *.c
- Standardrechte: via umask
  - Die Bits in der umask werden von den Bits der Maske abgezogen. (Die Ziffern der umask werden octal interpretiert).
- Beispiel: umask 026 → rwx.r-x.--x





## Verzeichnisse (Directories) in UNIX

**Verzeichnis**
selbst Datei mit Auflistung der gruppiertenDateien.

- Beispiel: cat -v <*dir*>
- **Erzeugen:** mkdir <*name*>
- **Listen:** ls <*name*>
- **Zugriffsrechte:** Wie bisher, aber x Recht bezieht sich auf Zugriffsrecht auf bekannte Datei im Verzeichnis.
- **Verzeichnisse in Verzeichnissen:**
  /usr/bin
  /usr/include/stdio.h





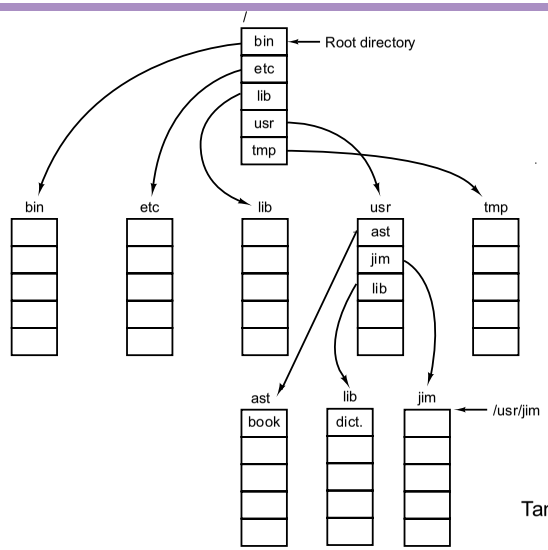
## Dateibäume in UNIX

- Die Dateien sind also (logisch) in einem **Dateibaum** organisiert. Zugriff über **absoluten** Pfadnamen (usr/ast/book) oder **relativ** zur augenblicklichen Position (z.B. book bei position im Katalog /usr/ast/). Es können Dateien eingerichtet werden, die auf andere verweisen (**links**). Dadurch kann auf einer bestehenden Organisation eine andere logische Organisation überlagert werden.
- Durch Links wird das Dateisystem von einem Baum zu einem allgemeineren Graphen, der auch Zyklen enthalten kann.





## Dateibaum




Tanenbaum, Abb.6-10

## Beliebte Dateiverzeichnisse in UNIX

| | |
|---|---|
| /bin | System Binärdateien (ausführbar) |
| /dev | Dateien, die für Geräte stehen |
| /etc | Systemverwaltung |
| /home/name | Home-Verzeichnis für Benutzer |
| /lib | Bibliotheksfunktionen |
| /tmp | Temporäre Dateien |
| /usr | Benutzerdateien |
| /usr/adm | Systemverwaltung |
| /usr/bin | Weitere System-Binärdateien |
| /usr/include/ | System Vorsann Dateien |
| /usr/lib | Compiler etc. |
| /usr/man | On-Line Manual |
| /usr/src | System Quellcode |
| /usr/tnp | Temporärdateien |
| /var/spool | Zwischenspeicher für Drucker etc. |







## Mounting

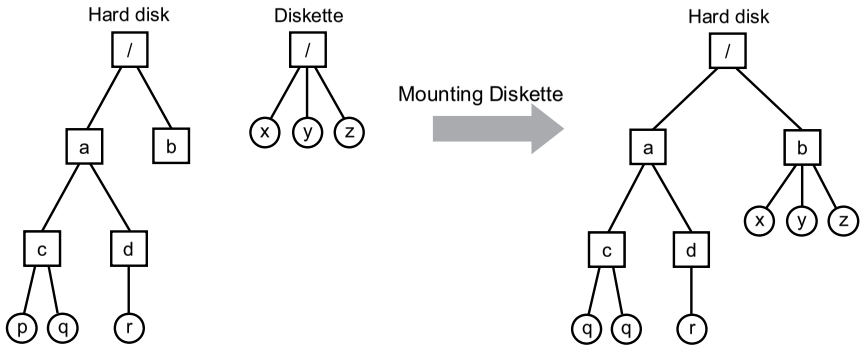
Unter UNIX befindet sich das **root directory „/“** auf einer Festplatte.
Nicht alle subdirectories müssen auf der gleichen **Partition** sein. (Partitionen sind abstrahiert spezielle „Geräte“; andere Arten von Geräten werden wir später behandeln)

Ein Dateibaum, der sich auf einem anderen Gerät befindet, kann mittels eines **mount** Befehls an eine Stelle des Dateibaums eigefügt werden, das durch ein directory gegeben ist, und wird dann als Unterbaum behandelt.
Falls das ursprüngliche directory nicht leer war, so wird dessen Inhalt durch den **mount** Befehl verdeckt (bis er nach einem evtl. **umount** wieder sichtbar wird).





## Mounting: Beispiel




Tanenbaum, Abb.10-26

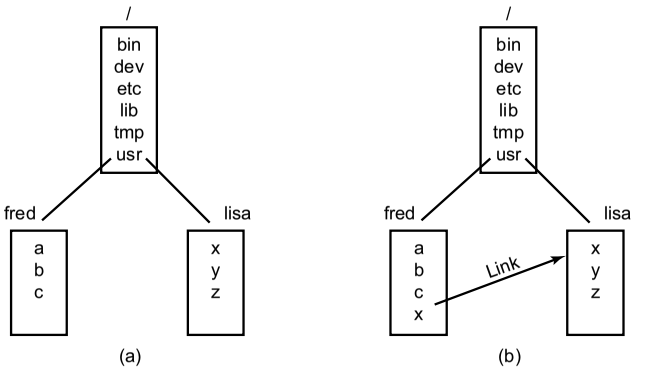




## Links

- In UNIX gibt es hard - und soft (symbolic) links.
  - hard links: innerhalb eines Gerätes (i.a. Plattenpartition)
  - soft links: über Gerätegrenzen hinweg
    - Beispiel: `ln -s /usr/ast/book OS_book`
- Es ist möglich, einen Dateibaum von einem Gerät in den Dateibaum eines anderen Geräts einzutragen (mounting) und somit logisch einzugliedern. Der Benutzer merkt das Überschreiten der Gerätegrenzen dann nicht und arbeitet logisch mit einem einzigen Dateibaum.

## Beispiel: Links




Tanenbaum, Abb. 10-25

F. und L. arbeiten gemeinsam an Datei x im Verzeichnis `/usr/lisa`.

(a) F. kann relativ (`../lisa/x`) oder absolut (`/usr/lisa/x`) auf x zugreifen.

(b) F. kann einen Link auf x erzeugen und somit wie auf eine lokale Datei zugreifen (x).







## Hard Links vs. Symbolic Links

| Soft (Symbolic) Links | Hard Links |
|---|---|
| eigener Dateityp | werden direkt durch die Implementierung des UNIX-Datei-systems realisiert |
| können auf nicht vorhandene Dateien verweisen (z.b. wenn ein directory verschoben wird) | Dies kann bei hard links nicht passieren. |
| Keine Probleme mit physischen Grenzen | Probleme mit physischen Grenzen |
| Bei soft links besteht dieses Problem nicht. | Manche Tools löschen Dateieinträge und kreieren Dateien neu. Dies führt bei hard links zu mehreren unabhängigen Dateien, was oftmals nicht sofort entdeckt wird. |







## Sticky Bit, setuid-Bit, setgid-Bit

- Neben den Bits für r, w, x für jeweils u, g oder o gibt es im Feld der Zugriffsrechte noch einige weitere bits, die gesetzt werden können.
- Mit dem **sticky bit** können Directories mit einem Lock versehen werden, das das Löschen von Files durch group Mitglieder verhindert (die rwx auf dem Directory haben, um z.B. neue Files anlegen zu können oder bestehende ändern dürfen sollen.)







## Sticky Bit, setuid-Bit, setgid-Bit

- Das setuid-Bit kann sinnvollerweise nur bei ausführbaren Dateien gesetzt sein (solche, die also x Permission für ugo besitzen). Ist das setuid-Bit gesetzt, wird normalerweise ein s statt ein x angezeigt.
- Ein Programm, dessen setuid-Bit gesetzt ist, wird unter der User-id ausgeführt, der das File gehört, und nicht unter der User-id des Aufrufers, wie dies im Normalfall geschieht. Hierdurch kann einem normalen Benutzer auf kontrollierte Art Zugriff auf privilegierte Resourcen gewährt werden.

## Der setuid Mechanismus

- Beispiel: Ein Benutzer soll in der `/etc/passwd`-Datei sein eigenes Password ändern dürfen, das dort in DES- verschlüsselter Form steht, aber keine anderen Änderungen vornehmen dürfen. Das File `/etc/passwd` gehört `root`, und normale Benutzer haben keinen Schreibzugriff:

```
-rw-r--r-- 1 root   sys     524   Apr   30   13:54 /etc/passwd
```

- Beim Programmfile `/usr/bin/passwd`, das `root` gehört, ist das setuid-Bit gesetzt.





## Der setuid Mechanismus

```
-r-sr-sr-x 3 root sys 15688 Oct 25 1995 /usr/bin/passwd
```

- Wenn das Programmfile `/usr/bin/passwd` von einem normalen Benutzer aufgerufen wird, läuft es unter der User Id von `root` ab, und kann daher das file `/etc/passwd` ändern. Es ist Aufgabe des Programms `/usr/bin/passwd` zu gewährleisten, daß nur solche Änderungen in `/etc/passwd` vorgenommen werden, die dem Benutzer erlaubt sind.
- Der setuid-Mechanismus stellt ein größeres Sicherheitsproblem in UNIX dar und wird in neueren UNIX-Varianten eingeschränkt. Z.B. dürfen keine shell-scripts mit dem setuid-Bit versehen werden.





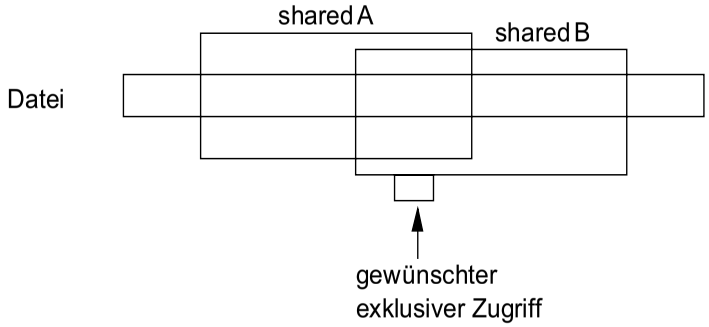
## Locking

- Locking: Dateien können für Exklusivzugriffe reserviert werden (POSIX).
  - Wichtig für Synchronisation paralleler Dateizugriffe.
- 2 Arten von *locks*:
  - **shared** - Reservierung für Lesezugriff (vgl. readers/writers locks)
  - **exclusive** - Reservierung für Schreibzugriff
- Es kann jede zusammenhängende Bytesequenz reserviert werden
- *shared locks* können sich überlappen.
- Exklusivzugriff nur möglich, nachdem alle anderen Reservierungen aufgehoben wurden.





## Datei locking nach POSIX




## Probleme ohne locking

- Ohne Reservierungen sind bei Parallelzugriff Inkonsistenzen und andere semantische Fehler möglich.
- Fehlertyp: Ausfall
  Gegeben Datei *D* und zwei Prozesse $P_1$ und $P_2$:

| | | | |
|---|---|---|---|
| (1) | $P_1$ | liest Byte | $B = C_.$ |
| (2) | $P_2$ | liest Byte | $B = C_.$ |
| (3) | $P_1$ | schreibt | $B = C_1.$ |
| (4) | $P_2$ | schreibt | $B = C_2.$ |

- 2 Probleme:
  - Die Aktion von P1 "fällt unter den Tisch".
  - Falls (3) und (4) vertauscht ausgeführt werden, fällt Aktion von $P_2$ weg.







## Probleme ohne locking

- Fehlertyp: Inkonsistenz
  Gegeben Wort $W = B_1B_0 = C_{11}C_{10}$

| | | | |
|---|---|---|---|
| (1) | $P_1$ | liest Byte | $B_0 = C_{1_0}$ |
| (2) | $P_2$ | schreibt | $B_0 = C_{2_0}$ |
| (3) | $P_2$ | schreibt | $B_1 = C_{2_1}$ |
| (4) | $P_1$ | liest | $B_1 = C_{2_1}$ |

- Problem:
  - $P_1$ liest inkonsistentes Datum $C_{21}C_{10}$ statt entweder $C_{11}C_{10}$ *oder* $C_{21}C_{20}$
- Grundregel: Bei Parallelverarbeitung müssen für Schreibzugriff gemeinsame Daten *immer* reserviert werden.





## Files und Prozesse

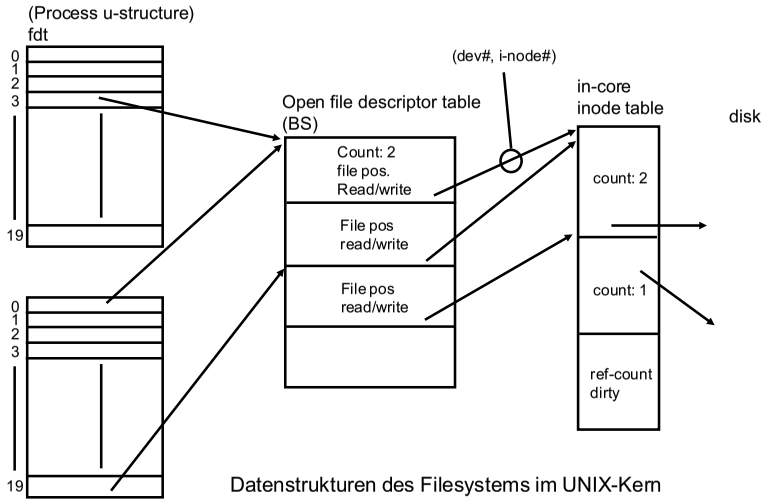
- Die Prozesse selbst greifen auf Dateien über Deskriptoren (file descriptor) zu.
- Ein Deskriptor ist ein Index in die private File-Deskriptor-Tabelle des Prozesses, die in der u-structure angelegt ist. Dort stehen Verweise auf die Einträge in der File-Tabelle.
- Die ersten drei Einträge sind per default
  - **stdin** (==0)
  - **stdout** (==1)
  - **stderr** (==2)
- Die Zwei-Schichten-Struktur aus **fdt** und **ft** erlaubt es verschiedenen Prozessen, sich einen Zugang zu einer Datei mit einer gemeinsamen Lese/Schreib-position zu teilen.







## Files und Prozesse




Datenstrukturen des Filesystems im UNIX-Kern

## Öffnen und Schließen einer Datei

- Vor der ersten Benutzung muss eine Datei **geöffnet** werden.
- OO-Interpretation: Das neue Objekt der Klasse Datei muss initialisiert werden.
- System-Aufruf:

```
fd = open(pathname,flags,modes)
```

| | |
|---|---|
| `pathname` | Dateiname als String. |
| `flags` | Art des Öffnens, z.B. zum Lesen oder Schreiben. |
| `modes` | Zugriffsrechte, falls die Datei neu geschaffen (kreiert) wird. |
| `fd` | **File-Deskriptor**. `fd == -1` zeigt einen Fehler an. |





## Öffnen und Schließen einer Datei

- Es ist durchaus möglich, eine Datei zweimal zu öffnen, z.B. einmal zum Lesen und einmal zum Schreiben.
- Folgender Systemaufruf dupliziert einen Deskriptor:

```
newfd = dup(fd)
```

- Eine Kopie des Eintrags mit Index `fd` wird im ersten freien Platz der Deskriptorstelle untergebracht, sein Index in `newfd` übergeben.





## Öffnen und Schließen einer Datei

- System-Aufruf zum Schließen einer Datei mit Deskriptor fd:

```
close(fd)
```

- `close` ist implizit bei Programmende.
- Beispiel:
  `close(stdin); newfd = dup(3);`
  → File # 3 nimmt den Platz von `stdin` ein.
- `close(fd)` löscht den Eintrag mit Index `fd` in der fdt und schafft so einen freien Eintrag.
- `close` dekrementiert Referenzzähler **count** im
  - entsprechenden ft-Eintrag
  - im in-core inode
- `close` gibt Datenstrukturen frei, wenn **count** = 0 wird.





## Öffnen und Schließen einer Datei

- `open` muss zunächst zum Dateinamen (Pfadnamen) den zugehörigen I-node finden. Hierzu werden Kataloge (*directories*) durchsucht.
  - Ein Katalog ist eine Datei mit einer Tabelle aus (Name, I-node-index) Einträgen (Abbildungs-funktion).
- Um Datei zu finden, wird die Tabelle durchsucht und der I-node Index extrahiert, dann können via I-node die Blöcke gefunden werden.
- Bei Angabe eines absoluten Pfads `/usr/ast/book` beginnt die Suche in I-node #2, dem `root` Katalog.

## Öffnen und Schließen einer Datei

- Ist der gefundene I-node noch nicht im Kern, so wird er von der Platte in die I-node-Tabelle des Kerns geladen.
- Nach Prüfen der Zugriffsrechte wird ein Eintrag in der File-Tabelle angelegt und die Lese/Schreib-position (**offset**) initialisiert.
  - im normalen Lese/Schreibmodus: offset = Dateianfang 0
  - im Schreibanhangmodus **write-append**: offset = Größe der Datei
- Danach wird ein Verweis auf diesen File-Table-Eintrag im privaten file-descriptor table des Prozesses eingetragen und zwar an der ersten freien Stelle. Das Ergebnis fd ist dann der Index dieses letzten Eintrags.
- Es ist durchaus möglich, eine Datei zweimal zu öffnen, z.B. einmal zum Lesen und einmal zum Schreiben.





## Duplizieren von Deskriptoren

- Systemaufruf für das Lesen einer Datei:

```
read(fd, buffer, nbytes)
```

- Man muss von fd zum I-node kommen.
  - fd ist ein Index in die File-descriptor Tabelle (**file descriptor table** - fdt) in der Benutzerstruktur (u structure).
- Man könnte in der fdt direkt den I-node Index eintragen, tut dies aber aus folgendem Grund nicht:
  - Grundverschiedene Prozesse sollen verschiedene Dateipositionen halten können.
  - Verwandte Prozesse (z.B. shell mit 2 Kindern) sollen sich eine Dateiposition teilen können. (shell kann z.B. positionieren und Kind kann dann von dort aus lesen.)
  - Dazu dient die Beschreibungstabelle offener Dateien (**open file descriptor table** - ofdt oder auch **file table** - ft). )





## Lesen einer Datei

- Zwei verschiedene Prozesse können somit dieselbe Datei an verschiedenen Stellen bearbeiten.
- Da bei `fork()` mit der u-structure auch der Prozeß fdt kopiert wird, haben Vater und Kind zunächst gleiche fdt-Einträge.
- Wenn der Vater die Lese/Schreibposition in der Datei verändert, sieht auch das Kind diese Veränderung (und umgekehrt). Dadurch können z.B. 2 Prozesse fortlaufende Einträge in einer Auftragsdatei abwechselnd lesen (und verarbeiten).





## Verschieben der Position des File-Pointers

- Files sind normalerweise auf Platte realisiert und bestehen aus Blöcken. Repositionierung des FilePointers ist deshalb oftmals effzienter möglich, als dies das rein sequentielle File Modell impliziert.
- In UNIX gibt es hierzu den System-Call **lseek**.

```
off_t lseek(int fildes, off_t offset, int whence);
```

- Der File-Pointer kommt an die Position `offset`, die in Abhängigkeit des `whence` Flags vom File-Anfang, der jetzigen Position oder dem File-Ende aus gerechnet wird.
  - `off_t` wird in `sys/types.h` als `long` defniert.

## Implementierung von Dateisystemen

- Einfachste Möglichkeit:
  **Kontinuierliche Speicherzuweisung.**
- Jede Datei besteht aus einer Menge von benachbarten Plattenblöcken.
- Vorteile:
  - Einfache Organisation (nur Startblock und Länge des Files muss bekannt sein).
  - Einfach Abbildung von logischer auf physische Adresse möglich.
  - Mehr oder weniger wahlfreier Zugriff möglich.
- Nachteile:
  - Datei kann nicht wachsen!
  - Verschwendung von Platz (zwischen den Dateien = „externe Fragmentierung“). (Problem dynamischer Platzzuweisung.)





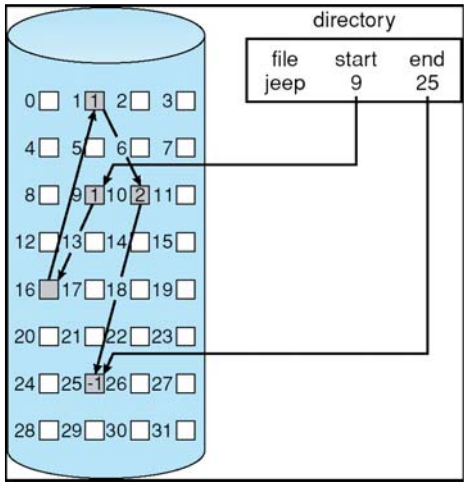
## Speicherzuweisung via verketteter Liste

- Jedes File besteht aus einer verketteten Liste von Blöcken auf der Platte.
- Blöcke können beliebig auf Platte verteilt sein.
- Jeder Block enthält Zeiger auf den nächsten Block.
- Dieses Schema wird beim FAT (**file allocation table**) in MS-DOS, OS/2 und Windows 95 benutzt.





## Das Dateisystem




Allocated as needed, linked together; e.g. file starts at block 9

Silberschatz, Galvin and Gagne, Abb.11.6



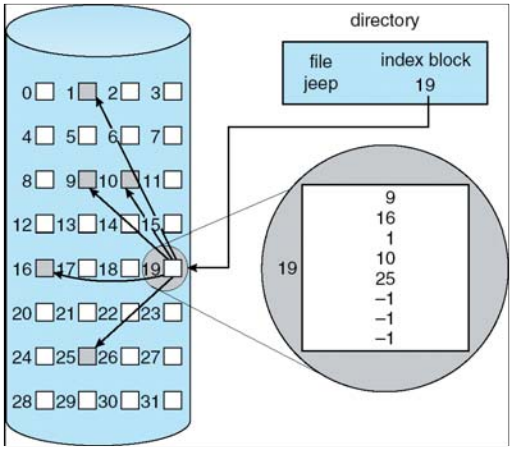


## Indizierte Allozierung

- Bei indizierter Allozierung werden alle Pointer in einem **Index Block** zusammengebracht.
- Indizierte Allozierung kann auch auf mehrere Level verteilt werden, und es kann auch eine Kombination geben.
  - (Schema der direkten Indizierung, indirekte, doppelt indirekte, . . . Indizierung vereinigt, wie z.B. bei UNIX.)



# Beispiel indizierte Allocation

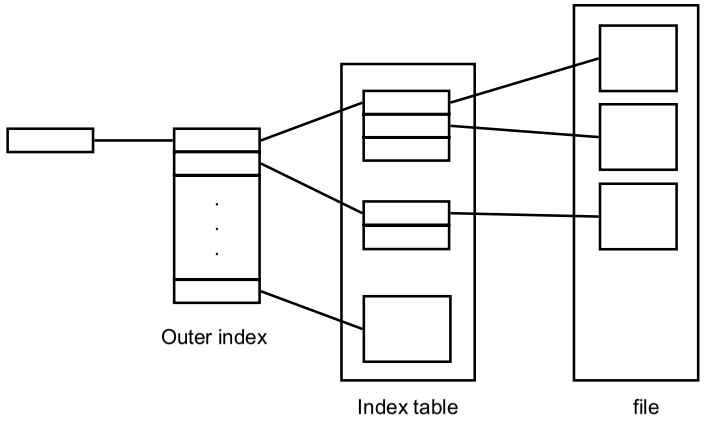



Silberschatz, Galvin and Gagne, Abb.11.8





# Multi-level Indexed Allocation (Prinzip)

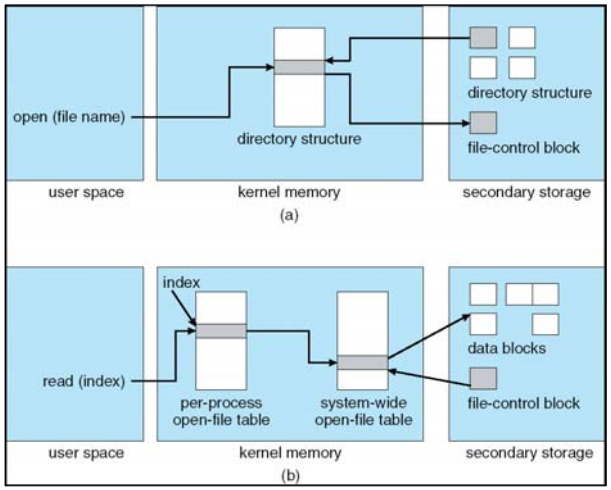



Operating System Concepts, 11.2, Silberschatz and Galvin © 1998





# Indexed Allocation - Mapping




open („filename“):
• durchsuche directory structure auf disk
• kopiere gefundenen FCB in den Kern mit Verweis auf Disk-FCB

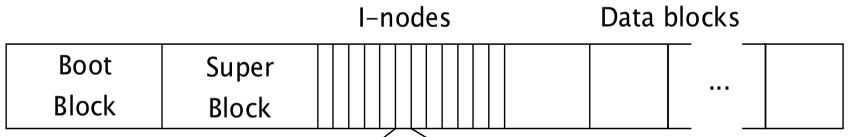
read (index):
• vom Index aus finde FCB im Kern
• Aus read-position berechne gesuchten data-block
•hole data block

Silberschatz, Galvin and Gagne, Abb.11.3



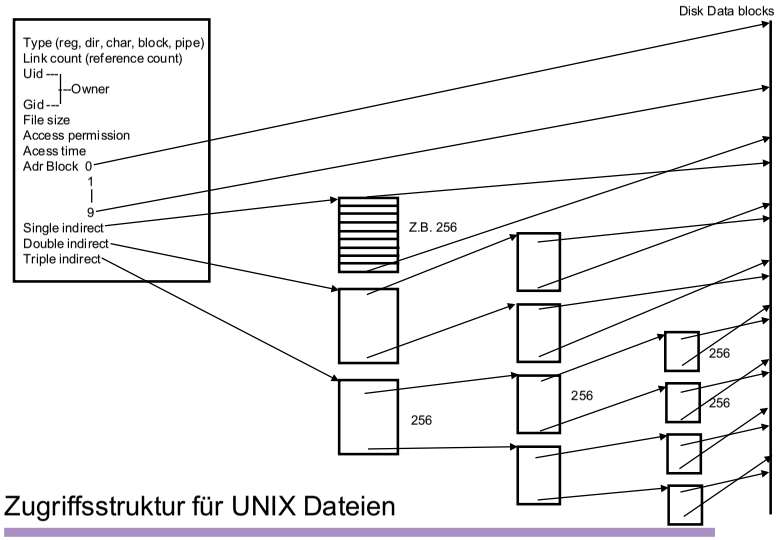


# UNIX Filesystem: Plattenorganisation




| # 0 | # 1 | I-nodes |
|---|---|---|
| Von Unix nicht benutzt | Verwaltung System:<br>Wie viele I-nodes?<br>Wie viele data blocks?<br>Freiliste der blocks.<br>Freiliste der I-nodes. | Verwaltung Datei<br>Zugriffsrechte<br>Eigentümer<br>Adressen Dateiblöcke<br>#2: root dir.<br>#1: bad block handling |

# UNIX: Organisation von I-node und Datei

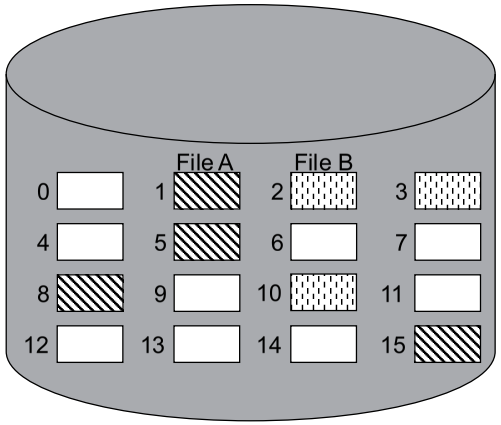



Zugriffsstruktur für UNIX Dateien





# Organisation von I-node und Datei




| I-node File A | |
|---|---|
| … | |
| Adr Block0 | 1 |
| Adr Block1 | 8 |
| Adr Block2 | 15 |
| Adr Block3 | 5 |
| … | |





# File Datenstrukturen im Kern

- I-nodes existieren sowohl auf Disk als auch im HSP, wenn die Datei offen ist. Oben gezeigt sind die disk I-nodes.
- HSP I-nodes haben zusätzliche Felder, u.a.
  - die Position des zugehörigen disk Inode Platzes
  - ein „dirty bit“
  - sowie Zeiger für die Liste freier I-nodes und
  - Zeiger für Verkettung in der Hash-Tabelle der HSP-Inodes
- Eine Schloß- Variable schützt den I-node während eines Systemaufrufs vor dem Zugriff durch andere Prozesse.
- Das Dateisystem errechnet den gefragten Block aus der Dateiposition (in Byte), dem Dateianfang und der Blockgröße.
  - Mit 32 bit Wort können 4 GB adressiert werden.
  - Mit 1 KB Blöcken und 32 bit Blockadressen können im obigen Schema über 16 GB gespeichert werden.





# File Datenstrukturen im Kern

- Zusätzlich zur Repräsentation auf der Platte existieren zu jeder benutzten Datei noch weitere Datenstrukturen im Kern, die den Zugriff regeln und erleichtern. Der inode auf der Platte wird in einen Eintrag der Inode-Tabelle im Kern (**in-core inode table**) geladen (**caching**).
- Um Daten auf Platten mit denen im Hauptspeicher zu synchronisieren, gibt es die Möglichkeit zur expliziten Synchronisation: **fsync**. Auch ohne explizite Synchronisation wird regelmäßig der Eintrag auf der Platte mit dem im HS synchronisiert
- Jeder Zugang zur Datei (vgl. **open**) wird durch einen Eintrag in der File-Tabelle des Kerns vermerkt mit Zugangsart (read/write/read-write) und momentaner Lese/Schreibposition (**position**) (vgl. **lseek**).

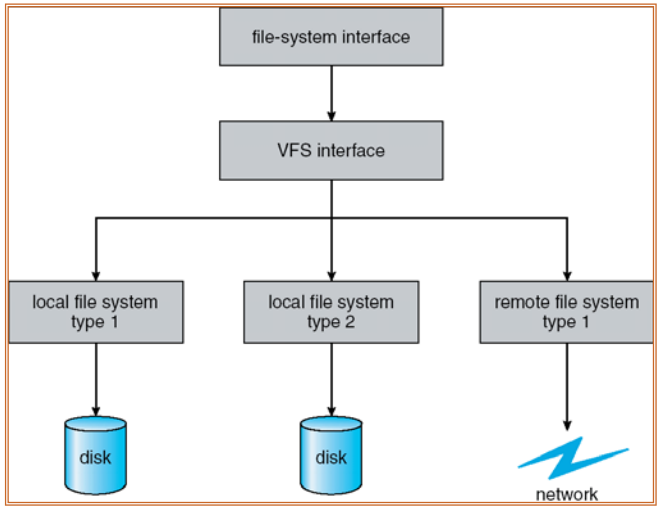
## Virtual File Systems

- Virtual File Systems (VFS) provide an object-oriented way of implementing file systems.
  - VFS provides an interface, which is implemented differently by each file system.
  - VFS provides globally unique file handles across different file systems.
  - the appropriate implementation of **read(filehandle)** is selected according to the type of filehandle
- VFS allows the same system call interface (the API) to be used for different types of file systems.
- The API is to the VFS interface, rather than any specific type of file system.





## Schematic View of Virtual File System




Silberschatz, Galvin and Gagne, Abb.11.4