Universität Bamberg

GRABS: Grundlagen der Rechnerarchitektur und Betriebssysteme

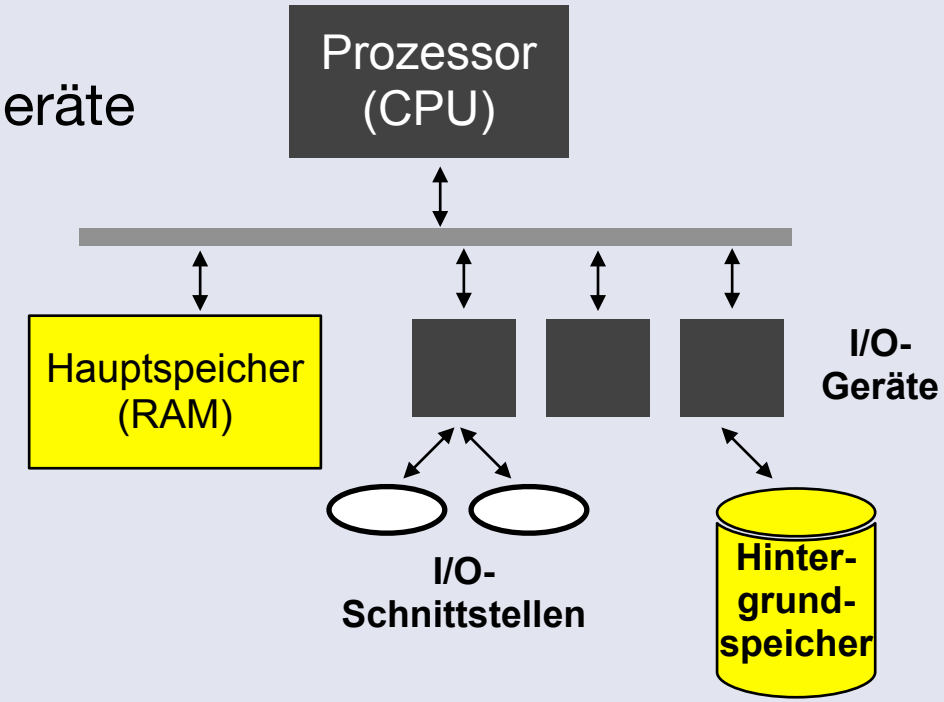
Literatur:
Arpaci-Dusseau [1]
Kap. 39–40

Vorlesung 14:
Persistente Speicherung – Dateisysteme

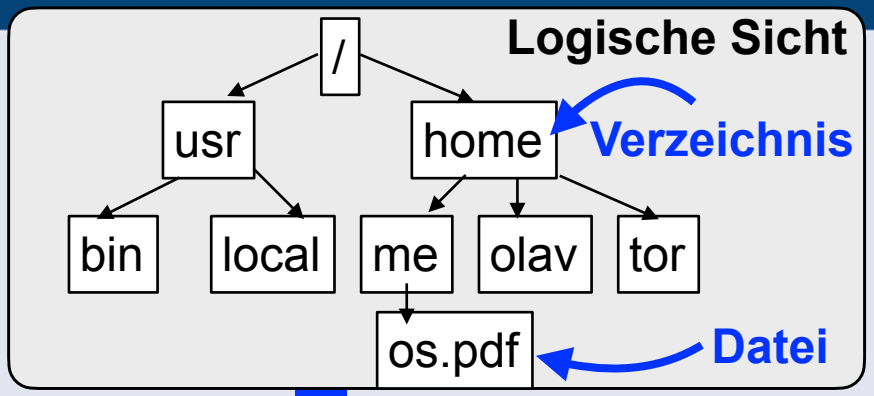
Michael Engel (michael.engel@uni-bamberg.de)
Lehrstuhl für Praktische Informatik, insbes. Systemnahe Programmierung
https://www.uni-bamberg.de/sysnap

Ressourcen, nochmal…

- Bisher haben wir betrachtet:
  - Prozessor (CPU)
  - Hauptspeicher
  - I/O-Geräte
    - Blockorientierte Geräte
- **Heute: "Hintergrundspeicher"**



Hintergrundspeicher

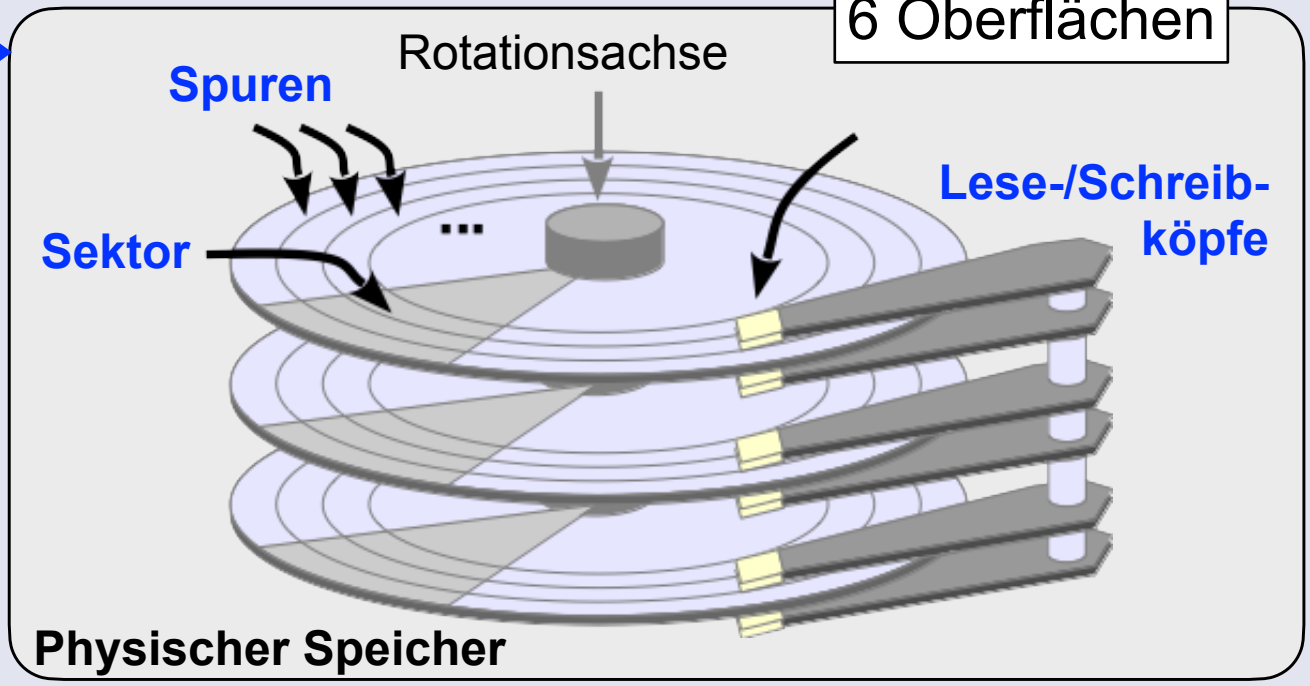


Dateisysteme ermöglichen die **dauerhafte Speicherung** großer Datenmengel

Abbildung

Das Betriebs-system stellt den Anwendungen eine **logische Sicht** bereit und muss diese effizient implementieren

Festplatte mit 6 Oberflächen



Dateiabstraktion

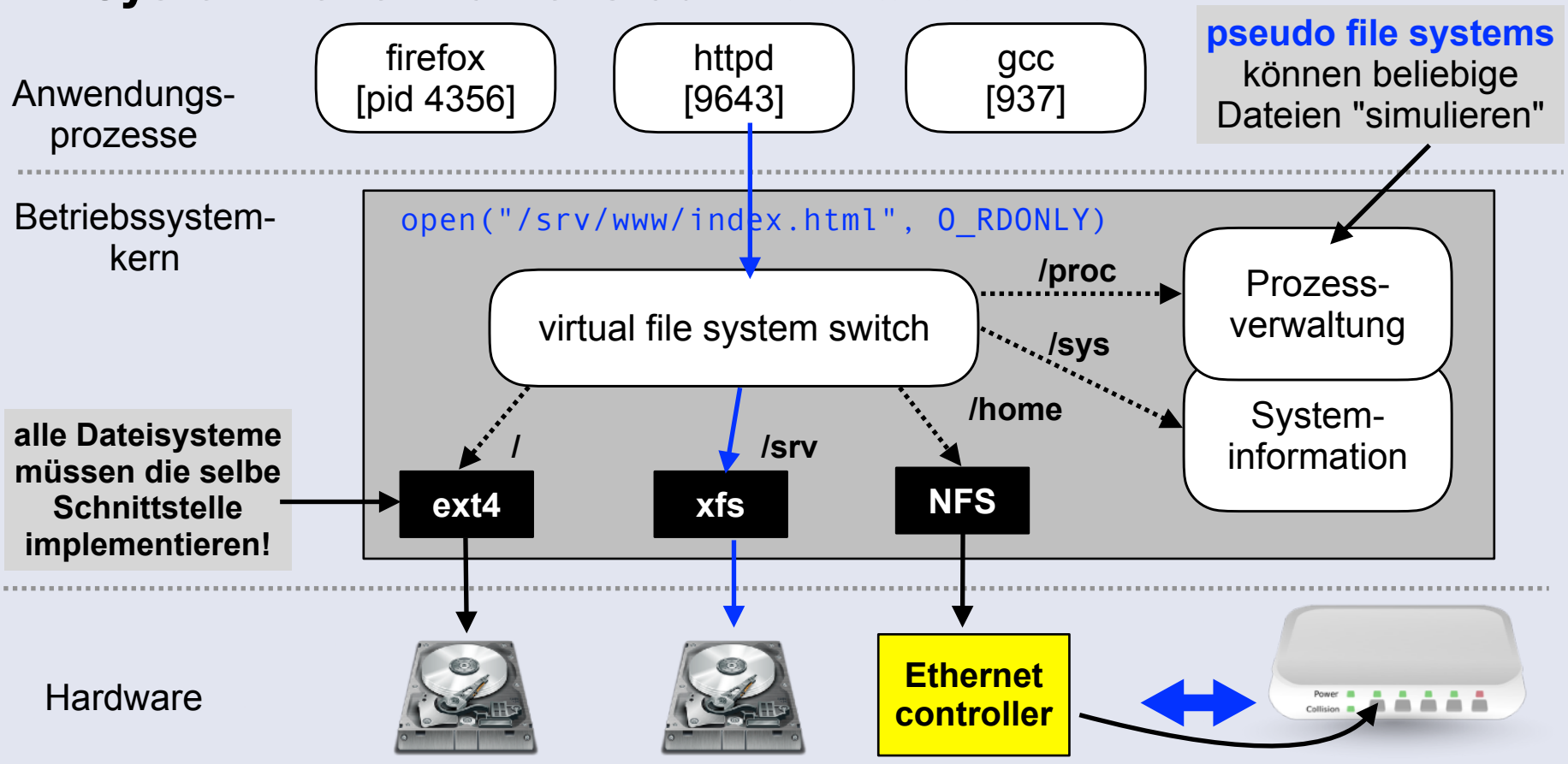
- Unix-Prinzip: "***alles ist eine Datei***"
  - Genaugenommen: jede Ressource im System kann über einen ***Namen*** in einer Verzeichnishierarchie angesprochen werden
  - Zugriffe auf die Ressource erfolgen durch die bekannten Unix-Systemaufrufe für Dateizugriff
  - Zugriffsrechte kontrollieren den Zugriff auf die Ressourcen
- Beispiele:
  - Reguläre Dateien und Verzeichnisse
  - Spezialdateien für Geräte, symbolische Links, *named pipes*
  - Virtuelle Dateien mit Prozess- und Systeminformationen
- In Unix nicht vollständig konsistent, aber z.B. in Plan 9:
  - Netzwerkverbindungen und -protokolle
  - Zugriff auf den Grafikspeicher

Dateizugriff

- Dateien werden über prozesslokale Dateideskriptoren *(file descriptors)* angesprochen
  - positive Ganzzahl, kann neu zugewiesen werden
- Die Unix-API zum Dateizugriff besteht aus vier Systemaufrufen:
- `int open(const char *path, int oflag, ...);`
  - Versucht, die Datei mit dem angegebenen Pfadnamen `path` mit dem Zugriffsmodus `flags` (read only, read/write etc.) zu öffnen
  - Gibt bei Erfolg ***file descriptor*** (fd) zur Verwendung bei Dateizugriffen zurück
- `ssize_t read(int fd, void *buf, size_t nbyte);`
- `ssize_t write(int fd, const void *buf, size_t nbyte);`
  - Liest (schreibt) `nbyte` Bytes von (in) Datei `fd` in (aus) dem Speicher beginnend an Userspace-Speicheradresse `buf`
- `int close(int fildes);`
  - Schließt die Datei: leert alle Puffer und invalidiert Dateideskriptor

Der Unix virtual file system (VFS) switch

- **Systemweiter Namensraum** für Dateien



Virtual file system: mounting

Universität Bamberg

Systemaufruf:

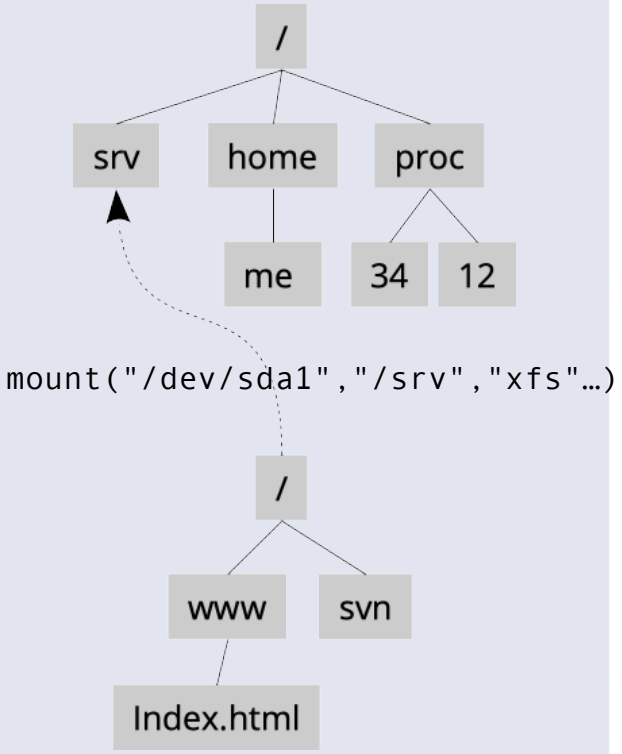
```
int mount(const char *source, const char
  *target, const char *filesystemtype,
  unsigned long mountflags, const void *data);
```

Verbindet ("mounted") ein Dateisystem mit dem angegebenen Verzeichnis in der Verzeichnishierarchie

Systemaufruf:

```
int umount(const char *target);
```

Löst die Verbindung. *Achtung:* **umount**, nicht unmount!

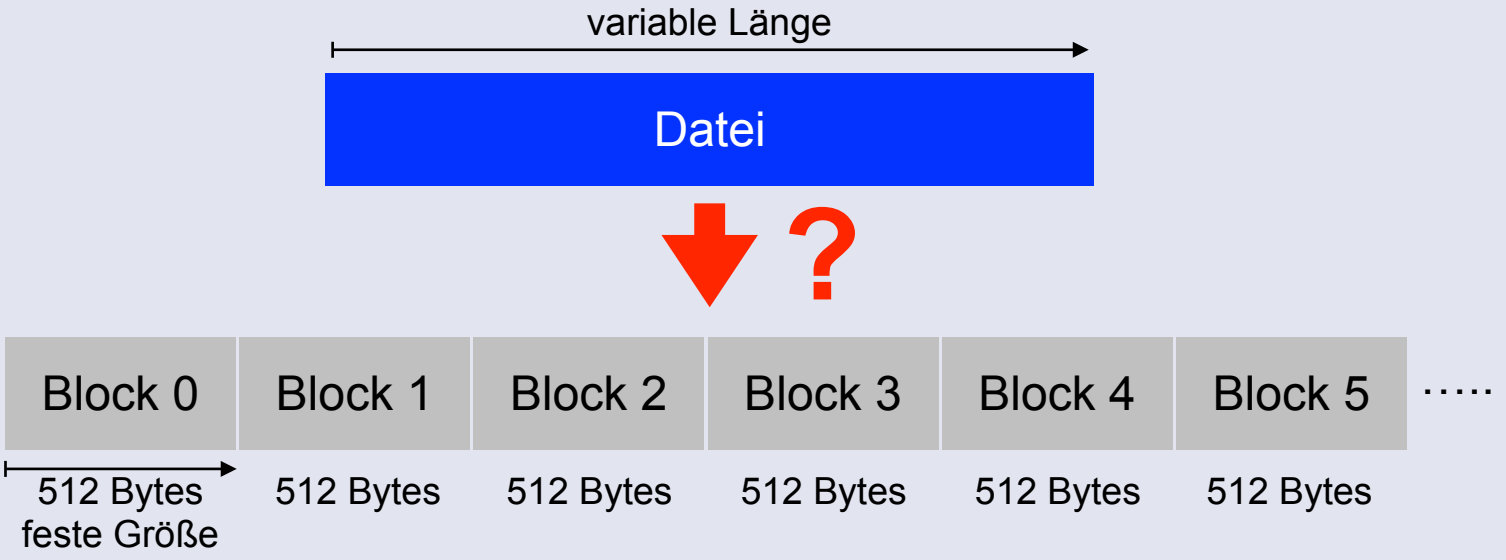


Verwendung dieser Systemaufrufe nur mit **Systemadministrator-Privilegien!**

Beim Booten des Systems werden alle in **/etc/fstab** gelisteten Dateisysteme automatisch gemounted

Dateispeicherung

- Meist benötigen Dateien mehrere Blöcke
  - Eine Platte wird hier als großes Array von Blöcken betrachtet
    - Jeder Block hat identische Größe, z.B. 512 Bytes
    - Dies ist eine *Abstraction* von Köpfen, Spuren und Sektoren einer realen Festplatte
  - In welchen Blöcken soll eine Datei gespeichert werden?



Zusammenhängende Speicherung

- Dateien werden in Blöcken mit aufeinanderfolgenden Blocknummern gespeichert
  - Benötigt Information über den StartBlock und die Anzahl an Folgeblöcken, z.B. Start: Block 2, Länge: 3 Blöcke

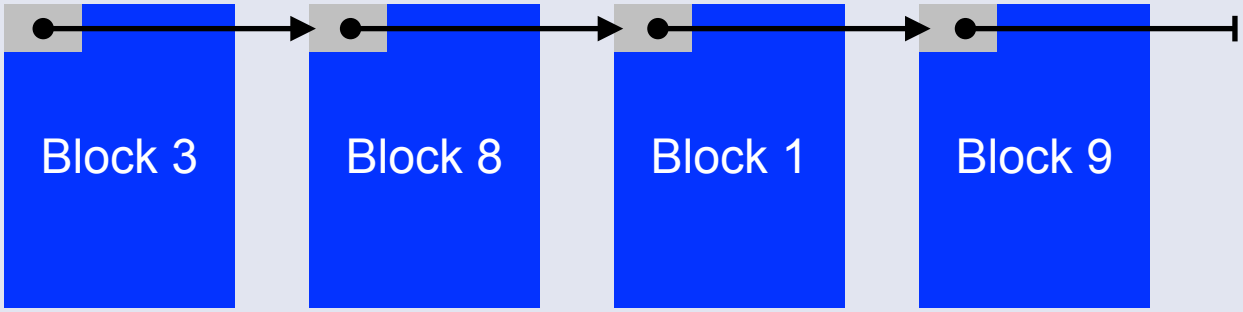
- Vorteil:
  - Zugriff auf alle Blocks mit minimaler Verzögerung durch Positionierung des Festplattenarms
  - Schneller direkter Zugriff auf beliebige Dateiposition
  - Verwendet für nur-lese-Dateisysteme, z.B. CD-ROM/DVD

Probleme zusammenhängender Speicherung

- **Finden von freiem Speicher** auf der Festplatte
  - notwendig: ausreichend große Menge zusammenhängender Blöcke
- **Fragmentierung**
  - Freie Blöcke können nicht genutzt werden, wenn die zusammenhängende Menge zu klein für eine Datei ist
- Die **Größe neuer Dateien** ist meist nicht im Voraus bekannt
  - **Erweitern** (Größe erhöhen) einer Datei ist problematisch, falls die folgenden Blöcke schon belegt sind
  - Erfordert **Umkopieren von Daten**, falls nicht genügend freie Folgeblöcke vorhanden sind

Speicherung mit verketteten Listen

- Blöcke einer Datei sind verlinkt



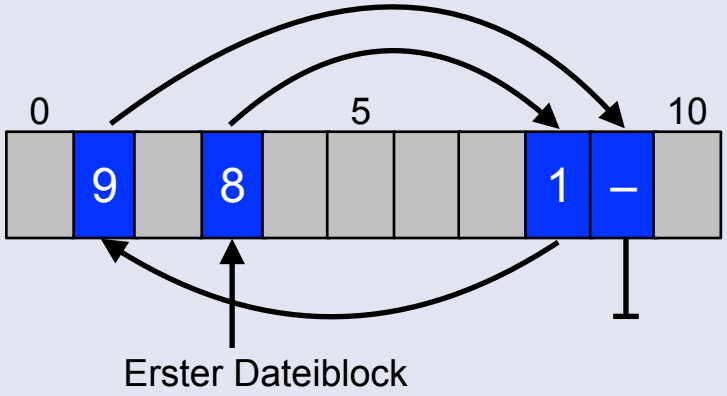
  - z.B. bei Commodore-Diskettenlaufwerken (C64/VC1514 usw.)
    - Blockgröße 256 Bytes
    - Erste zwei Bytes: Spur- und Sektornr. des Folgeblocks
    - Falls Spurnummer = 0 ➛ letzter Block
    - 254 Byte Nutzdaten
- Dateien können vergrößert und verkleinert werden

Probleme mit verketteten Listen

- Verfügbarer Speicherplatz verringert um den **für Zeiger benötigten Speicher**
  - Problematisch bei Verwendung von seitenbasiertem virtuellen Speicher: eine Seite benötigt dann immer Teile zweier Blöcke
- **Fehleranfällig**
  - Eine Datei kann nicht vollständig wiederhergestellt werden, wenn die Zeigerinformation Fehler enthält
- **Direkter Zugriff** auf beliebige Dateipositionen ist schwierig
- Erfordert **dauernde Repositionierung** des Festplattenkopfes, wenn Dateiblöcke über die Platte verteilt sind

Verkettete Listen: FAT

- Links sind in separaten Blöcken gespeichert
  - FAT: *file allocation table* (zuerst in MS-DOS verwendet)



- Vorteile:
  - Alle Bytes eines Datenblocks sind Nutzdaten
  - Redundante Speicherung der FAT ist möglich
    - Nützlich im Fehlerfall

Probleme mit verketteten Listen (2)

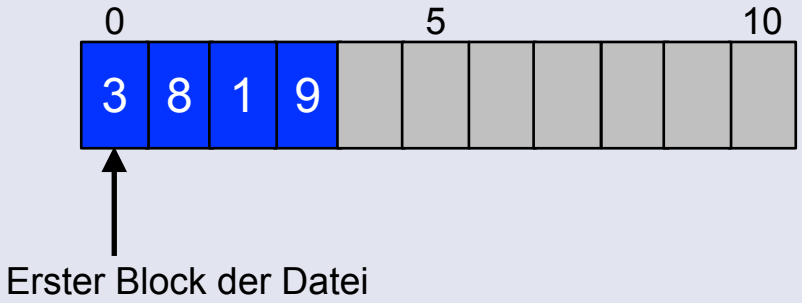
- **Zusätzliches Laden** mindestens eines Blocks notwendig
  - Es ist möglich, die FAT zur Effizienzsteigerung zu cachen
- **Nicht genutzte Information wird geladen**
  - FAT enthält Links für *alle Dateien*
- **Suchoverhead** für den Datenblock innerhalb einer Datei, der gesuchte Information enthält
- **Dauernde Repositionierung** des Festplattenkopfes notwendig, wenn Datenblöcke über die Festplatte verteilt sind

Diskussion: chunks/extents/clusters

- Variation:
  - Aufteilen einer Datei in Folge von Blöcken, die zusammenhängend gespeichert werden (*chunk*, *extent* oder *cluster* genannt)
  - Reduziert die Anzahl an Positionierungsvorgängen
  - Verbessert die Geschwindigkeit für die Suche eines Blocks linear
    - abhängig von der *chunk*-Größe
- Probleme:
  - Zusätzliche Information benötigt, um *chunks* zu verwalten
  - Fragmentierung
    - feste Größe: innerhalb einer **Folge** (**interne Fragmentierung**)
    - variable Größe: außerhalb von **Folgen** (**externe Fragment.**)
- Wird in der Praxis verwendet, hat aber keine bedeutenden Vorteile

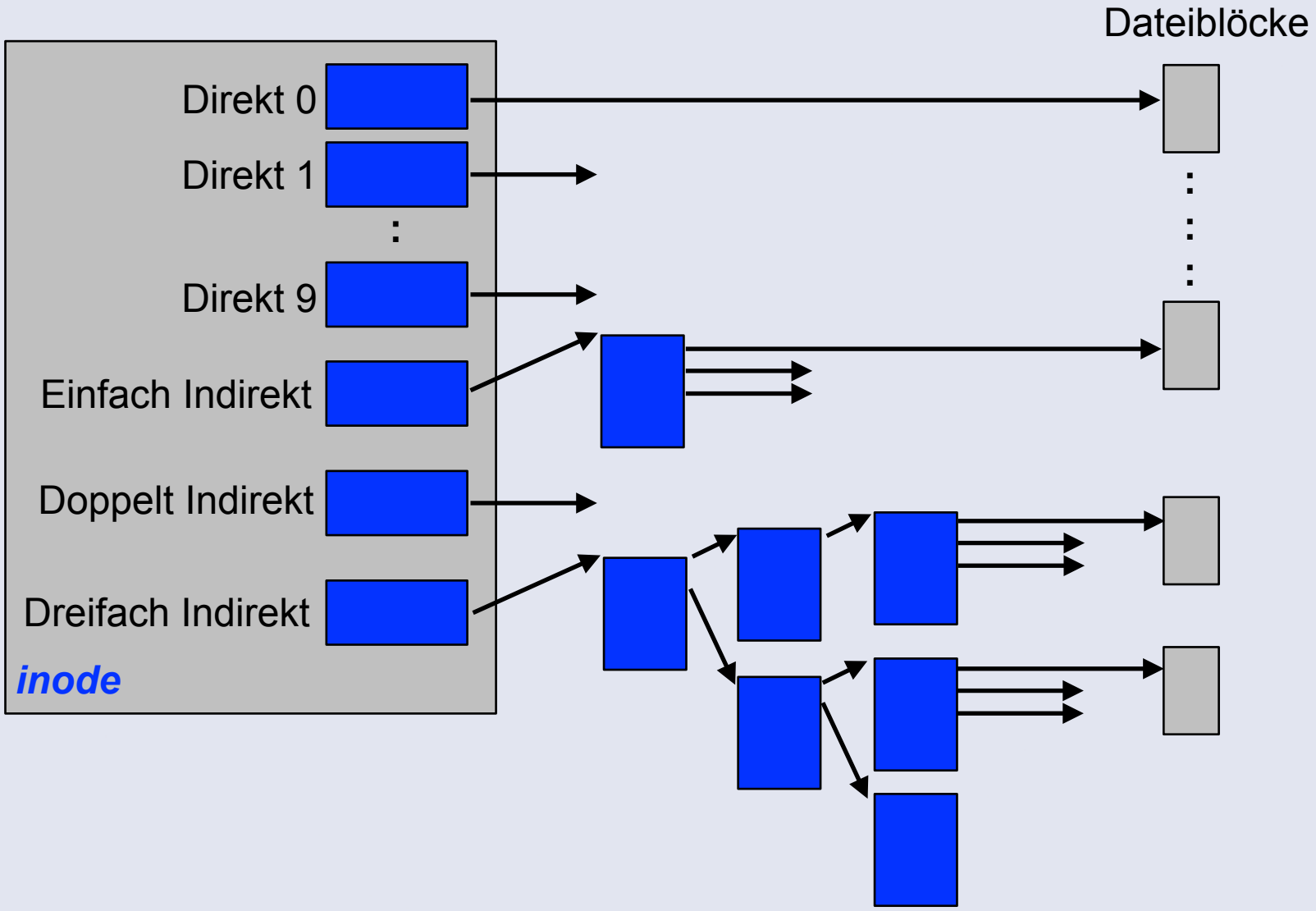
Indexierter Speicher

- Ein spezieller Block enthält Blocknummern der Datenblöcke einer Datei:



- Problem:
  - Feste Anzahl an Blöcken im Index-Block
    - Fragmentierung für kleine Dateien
    - Erweiterungen notwendig für große Dateien

Indexierter Speicher: Unix *inodes*

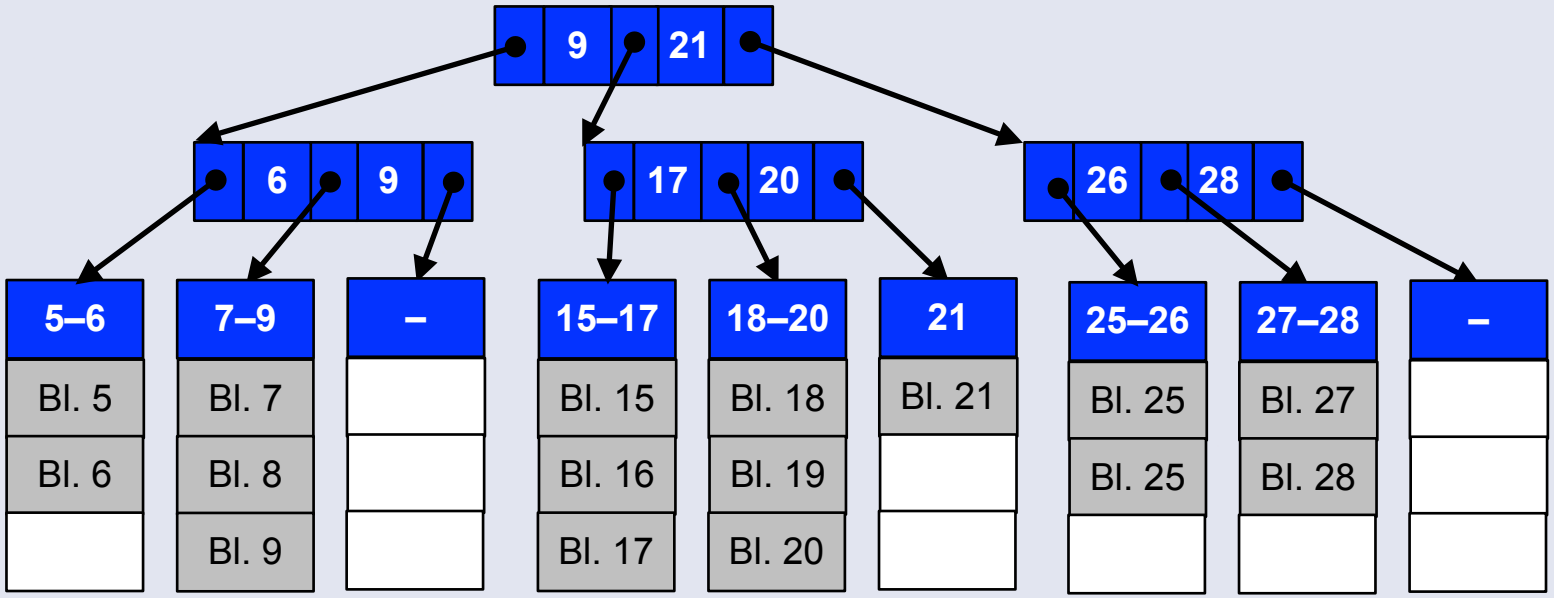


Indexierter Speicher: Diskussion

- Verwendung mehrere Indexebenen:
  - inodes benötigen immer einen Block auf der Festplatte (Fragmentierung ist kein Problem bei kleinen Dateien)
  - Mehrere Index-Ebenen ermöglichen es, große Dateien zu verwalten
- Nachteil:
  - mehrere Blöcke müssen (für große Dateien) geladen werden

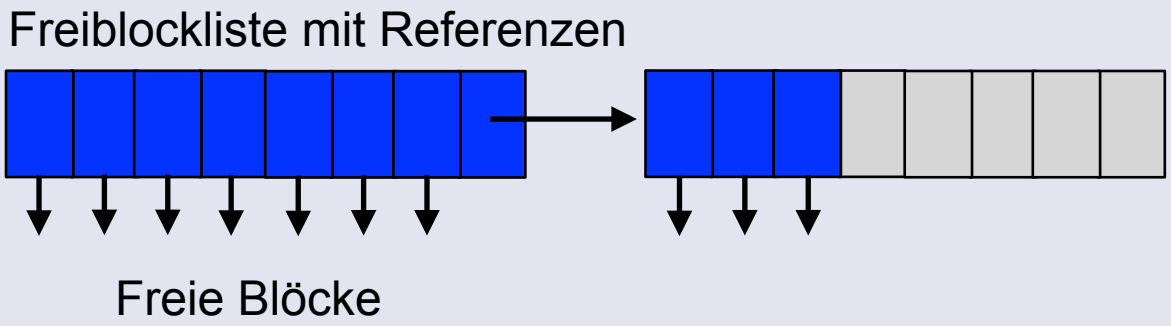
Baumstrukturierter sequentieller Speicher

- In Datenbanken verwendet, um effizient Datensätze zu gegebenem Suchschlüssel zu finden
  - Schlüsselraum kann dünn besetzt sein
- Wird auch verwendet, um Teile *(chunks)* von Dateien mit bestimmter Dateiposition zu finden, z.B. bei NTFS, btrfs, IBM JFS2, Apple HFS+ (B+ tree)



Freispeicherverwaltung

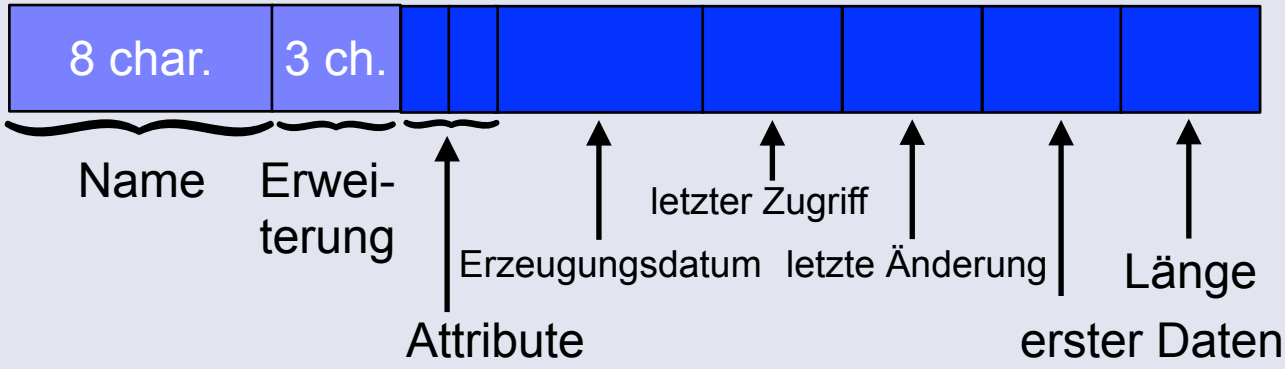
- **Bitvektoren** zeigen für jeden Block an, ob er benutzt ist oder nicht
- **Verkettete Listen** bilden die freien Blöcke ab
  - Verkettungsinformation oft in den freien Blöcken selbst gespeichert
  - Optimierung: Information über freie Block werden an zentraler Stelle gespeichert
  - Optimierung: ein freier Block enthält viele Blocknummern zusätzlicher freier Blöcke und evtl. auch die Blocknummer eines Blocks mit Information zu weiteren freien Blöcken



Verzeichnisverwaltung: Listen

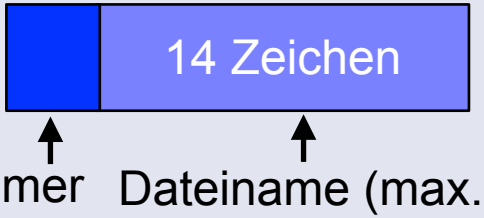
- Einträge identischer Länge nacheinander in Liste gespeichert:

FAT32:



*Lange Dateinamen in **VFAT** verwenden mehrere Verzeichniseinträge*

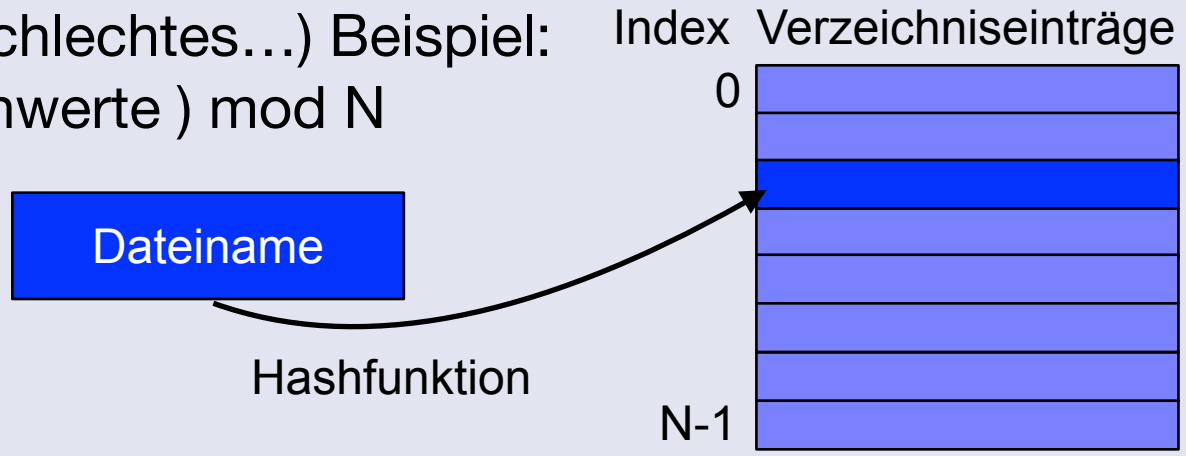
Unix System V 3.2:



- Probleme:
  - Lineare Suche nach gegebenem Eintrag notwendig
  - Wenn Liste sortiert ist: schnelle Suche, aber aufwändiges Einfügen

Verwendung von Hash-Funktionen

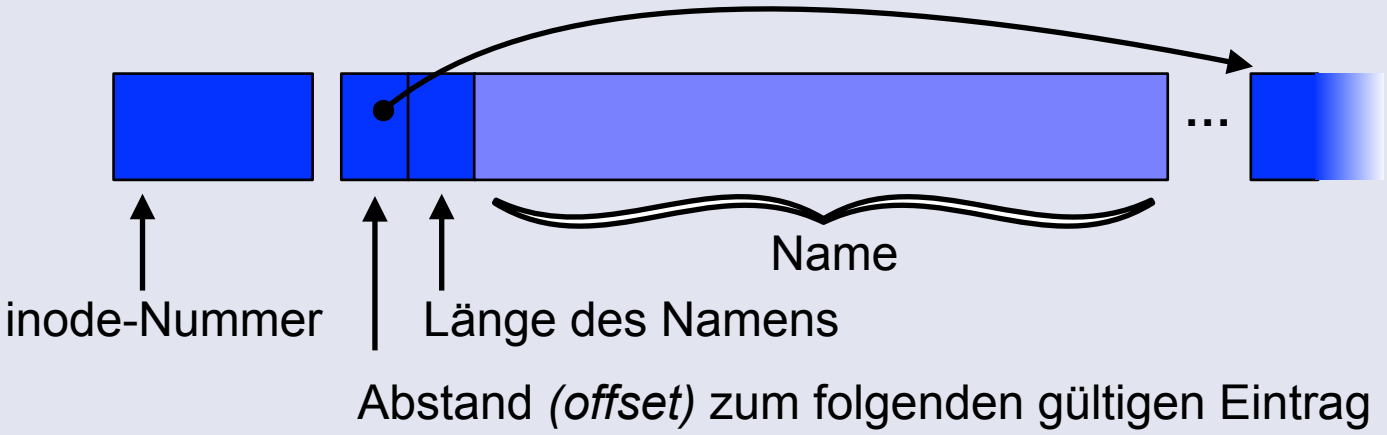
- Funktion bildet Dateinamen auf Index in der Verzeichnisliste ab
- Ermöglicht schnelleren Zugriff auf den Eintrag
  - keine lineare Suche notwendig
- Einfaches (aber schlechtes…) Beispiel:
  (∑ ASCII-Zeichenwerte) mod N



- Probleme:
  - Hash-Kollisionen (mehrere Dateinamen auf selben Eintrag abgebildet)
  - Anpassung der Listengröße notwendig, wenn Liste voll ist

Listenelemente mit variabler Länge

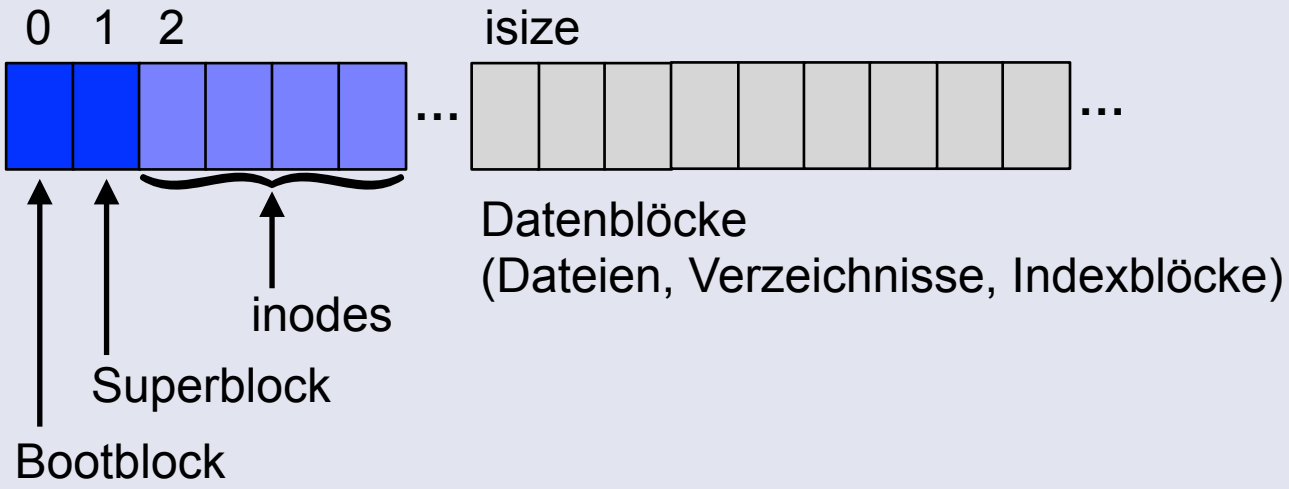
- Beispiel: verwendet in 4.2 BSD, System V Rel. 4 usw.



- Probleme:
  - Verwaltung freier Listeneinträge
  - Fragmentierung (Kompaktierung erforderlich)

Unix-Beispiel: System V-Dateisystem

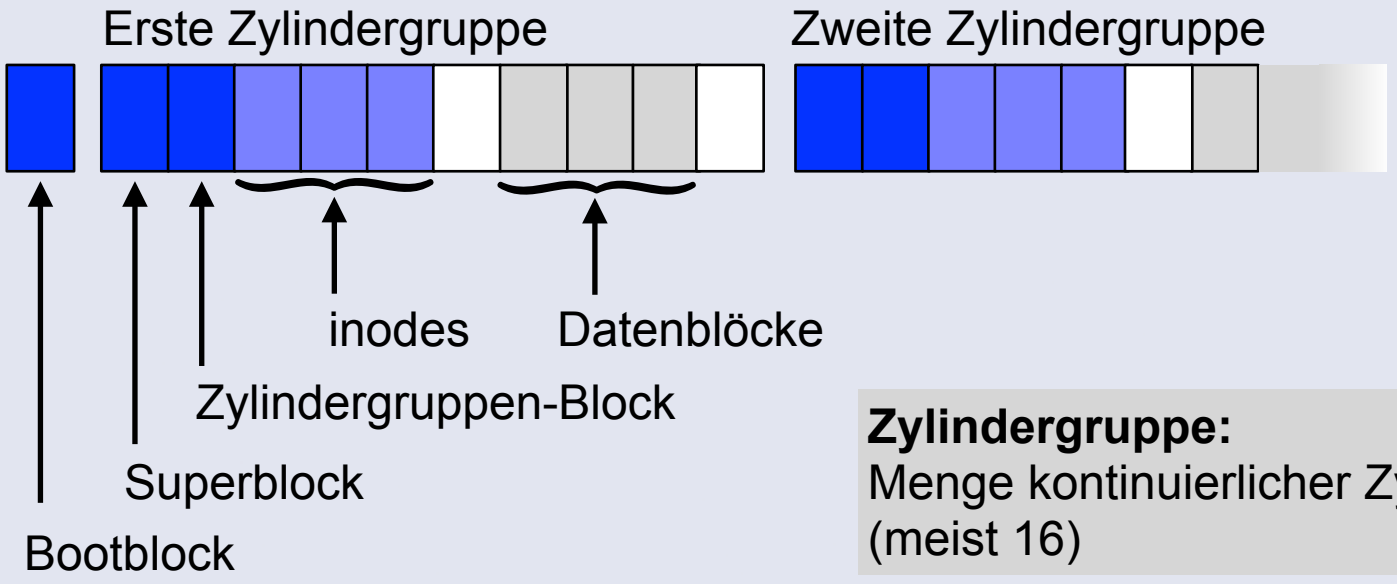
- Blockorganisation



- **Bootblock:** enthält Informationen, um das OS zu laden
- **Superblock:** enthält Verwaltungsdaten des Dateisystems:
  - Anzahl an Blöcken und ***inodes***
  - Anzahl an und Liste von freien Blöcken und inodes
  - Attribute (z.B. Angabe, ob das Dateisystem modifiziert wurde)

Unix-Beispiel: Berkeley Fast File System

- Blockorganisation (verwendet ab 4.2 BSD Unix)

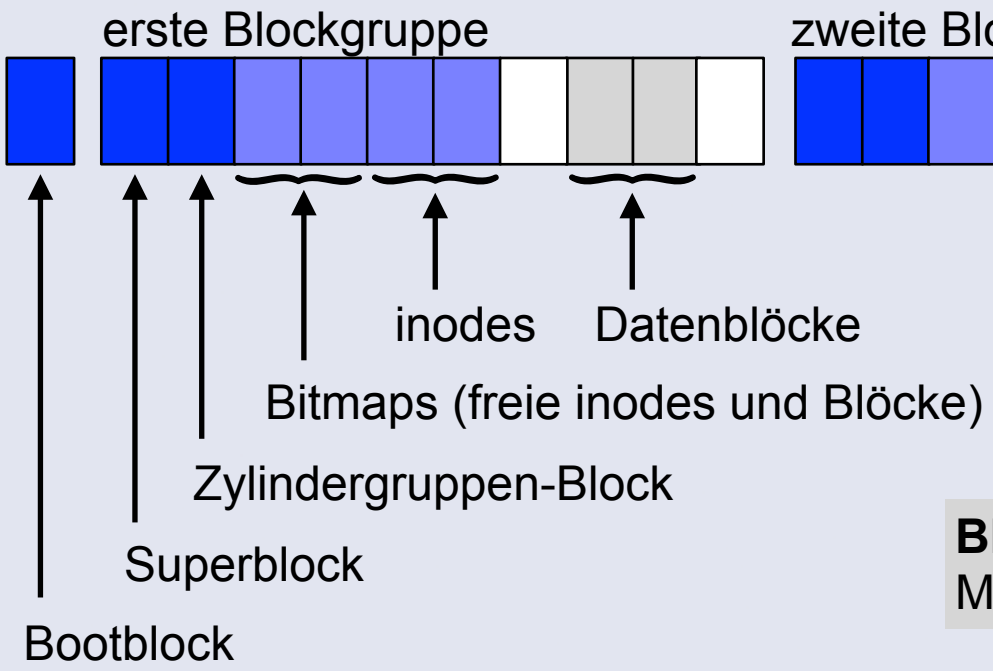


Zylindergruppe:
Menge kontinuierlicher Zylinder (meist 16)

  - Eine Kopie des Superblocks wird in jeder Zylindergruppe gespeichert
  - Eine Datei wird, wenn möglich, innerhalb einer einzelnen Zylindergruppe gespeichert
  - Verzeichnisse sind verteilt, Dateien eines Verzeichnisses werden zusammen gespeichert
- Vorteil: reduzierte Positionierungsdauer

Unix-Beispiel: Linux-Dateisysteme ext2/3/4

- Blockorganisation



Blockgruppe:
Menge kontinuierlicher Blöcke

- Ähnliches Layout wie das BSD FFS
- Blockgruppen sind nicht von Zylindergrenzen abhängig

Zusammenfassung

Dateisysteme sind eine **Betriebssystemabstraktion**

- Logisch zusammenhängende Informationen werden als Datei repräsentiert und gespeichert
- Oft wird eine hierarchische Verzeichnisstruktur zur Organisation von Daten verwendet

... werden **von der Hardware beeinflusst**

- Minimierung der Positionierungsdauer für Festplatten
- *Wear levelling* für Flash-Speicher

... werden **von Anwendungsprofilen beeinflusst**

- Blockgröße
  - zu klein → management data structures can lead to *performance loss*
  - zu groß → Fragmentierung verschwendet Plattenplatz
- Struktur von Verzeichnissen
  - ohne Hashfunktion → lang dauernde Suchen
  - mit Hashfunktion → höherer Verwaltungsaufwand

Referenzen

[1] Karels, M.; M. K. McKusick (September 1986).
Towards a Compatible File System Interface.
Proceedings of the European UNIX Users Group Meeting, Manchester, England: EUUG. pp. 481–496

[2] Apple Inc. **Inside Macintosh Volume 2: Files**.

[3] Maurice J. Bach.
Design of the UNIX Operating System.
Prentice Hall 1986, ISBN 978-0132017992

[4] Marshall Kirk McKusick, William Joy, Samuel Leffler and Robert Fabry.
A Fast File System for UNIX
ACM Transactions on Computer Systems. 2 (3): 181–197, 1984

[5] R. Card, T. Y. Ts'o, and S. Tweedie.
Design and implementation of the second extended filesystem
Proceedings of the 1994 Amsterdam Linux Conference, 1994.