Dateien und Dateisysteme

1. Allgemeines zum Datei-Konzept
2. Verzeichnisstruktur
   - Single-Level-Verzeichnis
   - Two-Level-Verzeichnis
   - Verzeichnisbäume
   - Verzeichnisse mit azyklischen und allgemeinen Graphen
3. Implementierung von Dateisystemen
4. Belegungsstrategien
   - Zusammenhängende Belegung
   - Verkettete Belegung
   - Indizierte Belegung
5. Speicherplatzverwaltung
6. Implementierung von Verzeichnissen

Datei-Konzept / Motivation

Prozesse arbeiten auf dem Hauptspeicher:

- Direkter Zugriff auf die Daten im Hauptspeicher
- Größe vom Hauptspeicher begrenzt
- Daten sind verloren, sobald der Prozess beendet wird

➔ Daten müssen permanent gespeichert werden

- Hintergrundspeicher mit sehr großer Kapazität

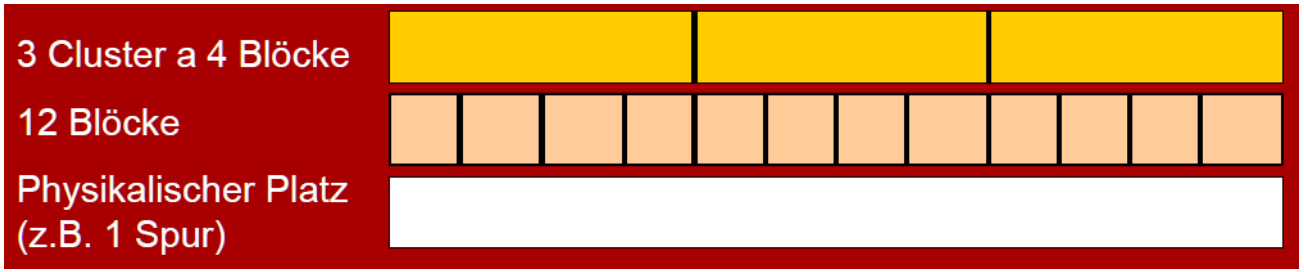
Betriebssystem als auch Benutzer besitzen oft hunderte von Dateien

➔ Flexible Organisation und effizienter Zugriff auf die Dateien nötig

➔ Bei Multiuser Systemen müssen Dateien vor dem Zugriff Dritter geschützt werden

Massenpeicher

- Hintergrundspeicher
  - physikalisches Medium mit Blockstruktur
    - Festplatten 512 bzw. 4096 Bytes
    - Optische Medien 2048 Bytes
  - 4 oder 8 Blöcke = 1 Cluster



- Beispiele
  - Magnetband (lesend/schreibend)
  - Festplatte (lesend/schreibend)
  - CD-ROM (lesend), CD-RW (lesend/schreibend)
  - DVD
  - USB

Datei-Konzept

- Datei
  - Eine mit Namen versehene Sammlung zusammengehöriger Informationen, die auf einem Hintergrundspeicher liegt.
    - ➔ Quellcode, Objektprogramme, Texte, Bilder, Audio, Video, ...

- Dateistruktur, abhängig vom Typ
  - Textdatei: Sequenz von Zeichen, organisiert als Zeilen und Absätze
  - Quellcode: Menge von Funktionen mit eigener Struktur
  - Bitmap: Menge von bits
  - Binärdatei: Sequenz von Bytes
  - Programmdatei: Menge von Code-Abschnitten

Dateiattribute

- Dateiname: Für Menschen lesbare Bezeichnung der Datei
  - Länge und erlaubte Zeichen variieren oft
  - Unix: Maximal 256 Zeichen
  - Windows 2000: Maximal 215 Zeichen, nicht erlaubte Zeichen `\ / : * ? " < > |`
  - Windows 10: 260 Zeichen, seit *Anniversary Update* geht auch länger
- Datei-Identifikator: für Menschen nicht-lesbare Bezeichnung der Datei
- Typinformationen, z.B. program**.c**, filesys**.doc**, brief**.txt**, bild**.gif**
  - Einige Systeme unterscheiden zwischen Dateitypen andere nicht
- Tabelle mit Beschreibung der Dateien (Metadaten)
  - Größe
  - Position im Dateisystem bzw. Hintergrundspeicher
  - Zugriffsrechte: Wer darf was mit der Datei machen?
  - Uhrzeit, Datum der Dateierstellung bzw. -änderung
  - Benutzeridentitäten / Rechte

Datei-Operationen

- Basisoperationen auf Dateien (betriebssystemabhängig)
  - Erzeugen von Dateien
    - Finde genügend freien Speicher und informiere das System
  - Schreiben auf eine Datei
    - Finde Datei im Verzeichnis und verwalte Schreibposition
  - Lesen von einer Datei
    - Finde Datei im Verzeichnis und verwalte Leseposition
  - Löschen einer Datei
    - Finde Datei im Verzeichnis, gib belegten Speicher frei und lösche Eintrag aus Verzeichnis
  - Umpositionierung von Zeigern
- Höhere Operationen auf Dateien
  - ➔ können aus Basisoperationen zusammengesetzt werden
    - Umbenennen von Dateien
    - Kopieren von Dateien
    - Anfügen an eine Datei

Datei-Operationen (2)

- Operationen erfordern das Auffinden einer Datei im Verzeichnis
  - Open-File Table (**OFT**): enthält die Liste aller geöffneten Dateien
  - `open`: fügt Datei in OFT ein
  - `close`: entfernt Datei aus OFT
  - Bei Multiuser/Multiprozess Systemen existiert eine lokale OFT und eine globale OFT
- Informationen zu geöffneten Dateien im OFT
  - Zeiger auf aktuelle Position in der Datei (pro Prozess)
  - Zugreifende Prozesse Zähler: Datei wird aus OFT erst entfernt, wenn dieser Zähler Null ist
  - Position der Datei auf Hintergrundspeicher
  - Zugriffsrechte (im lokalen OFT, pro Prozess)
  - Informationen zu gesperrten Bereichen

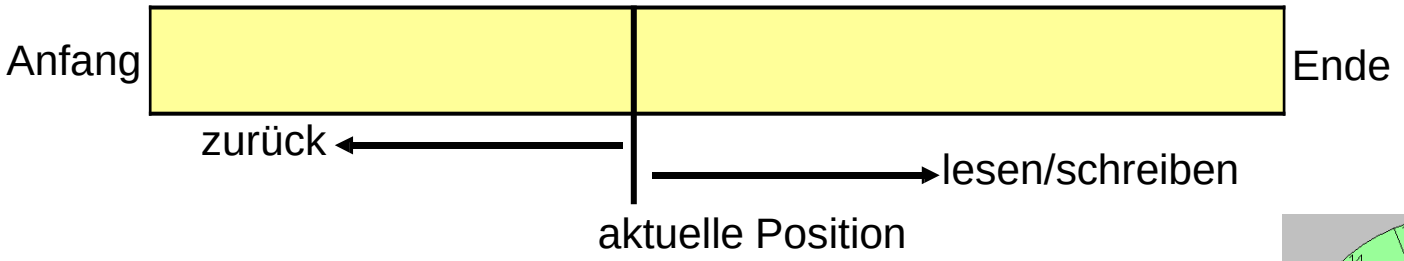
Design Fragen

- Dateitypen
  - Betriebssystem kennt definierte Dateitypen
    - Bessere Unterstützung der Benutzer
    - Einschränkung der Benutzer auf die definierten Typen

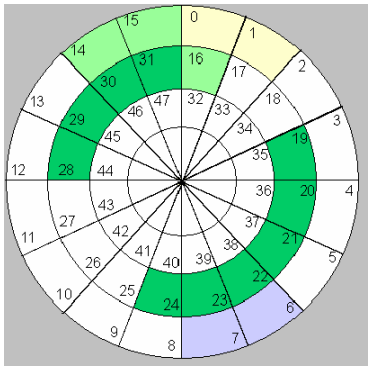
- Dateistruktur
  - Besteht eine Datei aus einer Aneinanderreihung von Bytes oder gibt es eine Strukturierung
  - Zu wenig Strukturen: Programmierung umständlich
  - Zu viele Strukturen: Aufwändige Programmierung

Zugriffsarten

- Zugriffsarten
  - Sequenzieller Zugriff
    - Bandmodell
    - `reset, read next, write next, skip(n)`



  - Direkter Zugriff (Direct Access)
    - Plattenmodell
    - Beliebiger Zugriff auf erforderliche Daten
    - Andere Zugriffsarten können einfach durch DA simuliert werden



- Zugriff ggf. nur auf vorher geöffnete Dateien (**open-file-table**)

Verzeichnisstruktur / Organisation

Auf einem Dateisystem sind oft Millionen von Dateien

➔ Organisation der Dateien durch:

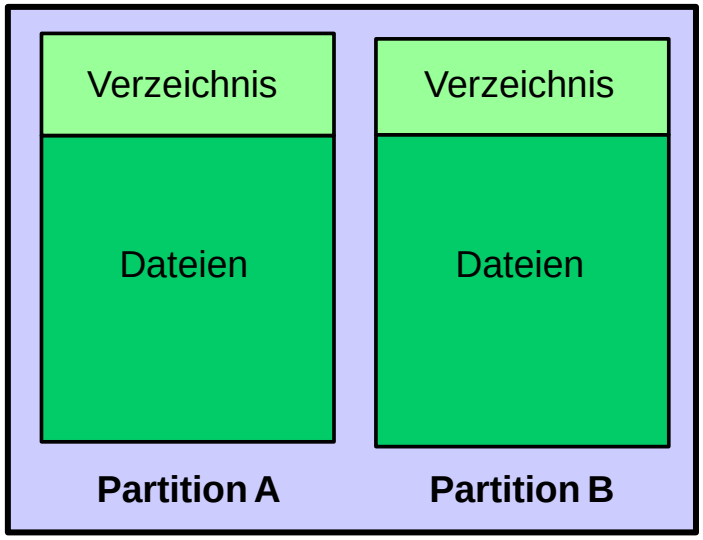
- Partitionen
- Verzeichnisse

Partition
- Virtuelle Platte

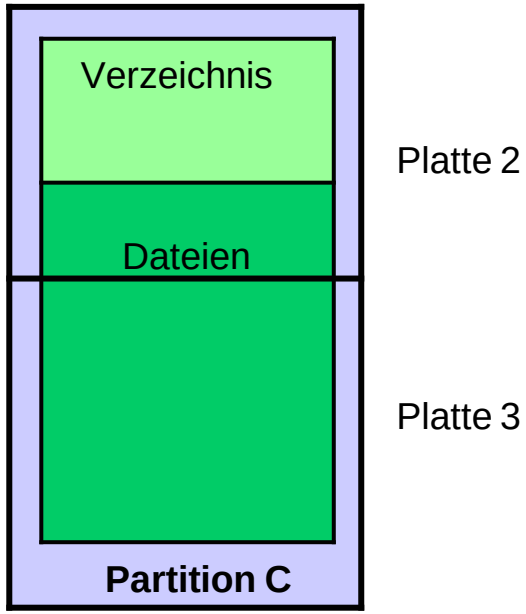
- Eine Platte mit mehreren **Partitionen**

- Eine **Partition** über mehrere Platten

1. Möglichkeit ➔ eine Festplatte mit mehreren Partitionen



2. Möglichkeit ➔ eine Partition über mehrere Festplatten



Verzeichnisstruktur

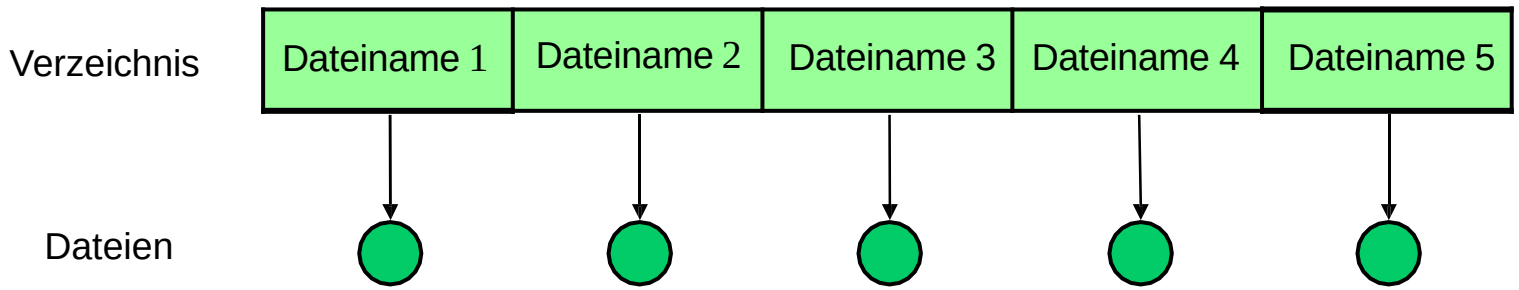
- Verzeichnis (Directory)
  - Ein Verzeichnis umfasst die Namen und die übrigen Attribute aller seiner Dateien.

- Operationen auf Verzeichnissen
  - Suchen nach einer Datei
  - Erzeugen einer Datei
  - Löschen einer Datei
  - Umbenennen einer Datei
  - Auflisten von Dateien
  - Durchlaufen von Verzeichnissen oder des gesamten Verzeichnisses

➔ Aufbau des Verzeichnisses beeinflusst die Effizienz des Dateisystems

Single-Level-Verzeichnis

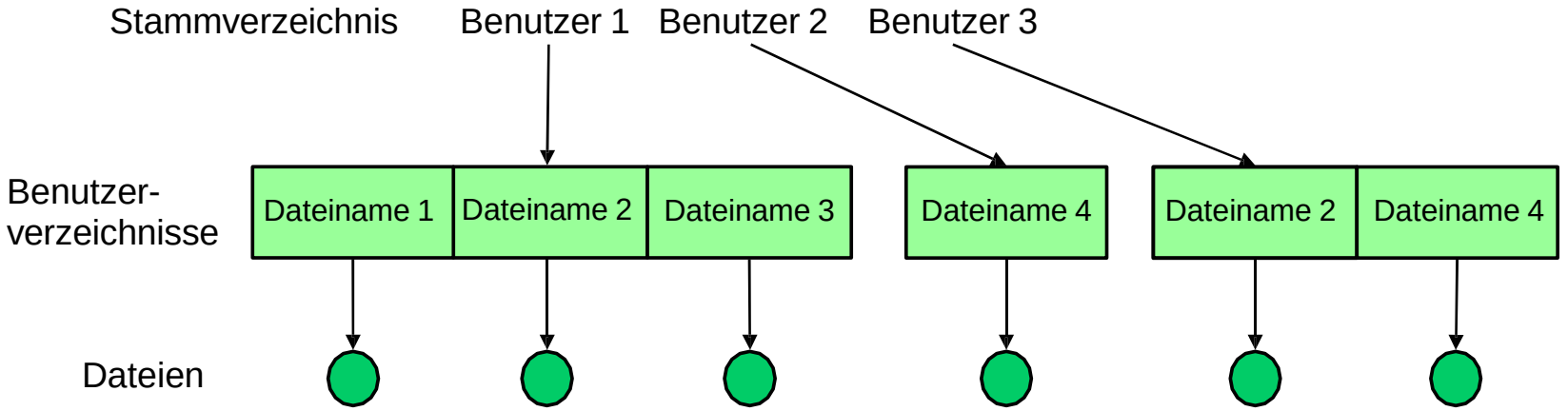
- Alle Dateien in **einem** Verzeichnis



- Eigenschaften:
  - \+ einfache Implementierung
  - – unübersichtlich bei mehreren Benutzern oder vielen Dateien
  - – alle Dateien müssen unterschiedliche Namen besitzen
  - – keine Organisationsmöglichkeit

Two-Level-Verzeichnis

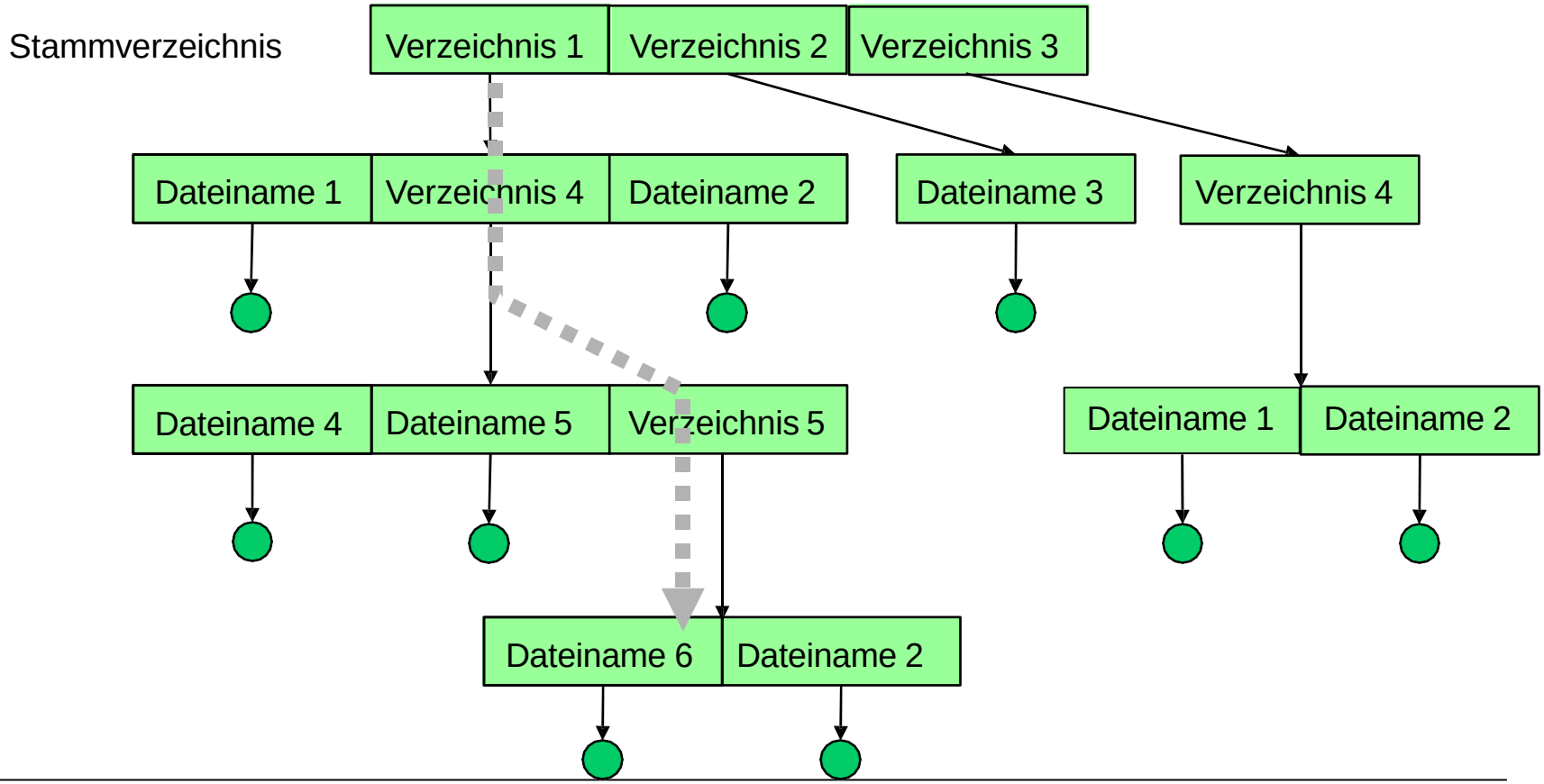
- Verzeichnis mit **zwei Ebenen,** wobei jeder Benutzer ein eigenes Verzeichnis besitzt.



- **Eigenschaften**
  - \+ verschiedene Benutzer können Dateien unter gleichem Namen abspeichern
  - \+ Zugriff auf eigene Dateien über Dateiname,
  - \+ Zugriff auf fremde Dateien über Pfad ➔ Benutzer- und Dateiname, z.B. **`\Benutzer3\Dateiname2`**
  - – Zugriff schwer, wenn Benutzer zusammen arbeiten wollen
  - – Zugriff auf Systemdateien? ➔ Suchpfad, z.B. durch Variable **`PATH`**

Verzeichnisbäume

Verallgemeinerung: **Verzeichnisbäume** mit beliebigen Unterverzeichnissen.
Dateiname entspricht dem eindeutigen Pfad vom Wurzelverzeichnis (root-directory) durch alle Unterverzeichnisse bis zur eigentlichen Datei.



Verzeichnisbäume (2)

- Eigenschaften von Verzeichnisbäumen
  - Einfacher Dateiname wird nur im aktuellen Verzeichnis gesucht.
  - Datei außerhalb des aktuellen Verzeichnisses muss über Pfad angegeben werden.
  - Absoluter Pfad: Von der Wurzel bis zur Datei
  - Relativer Pfad: Von der aktuellen Position zur Datei

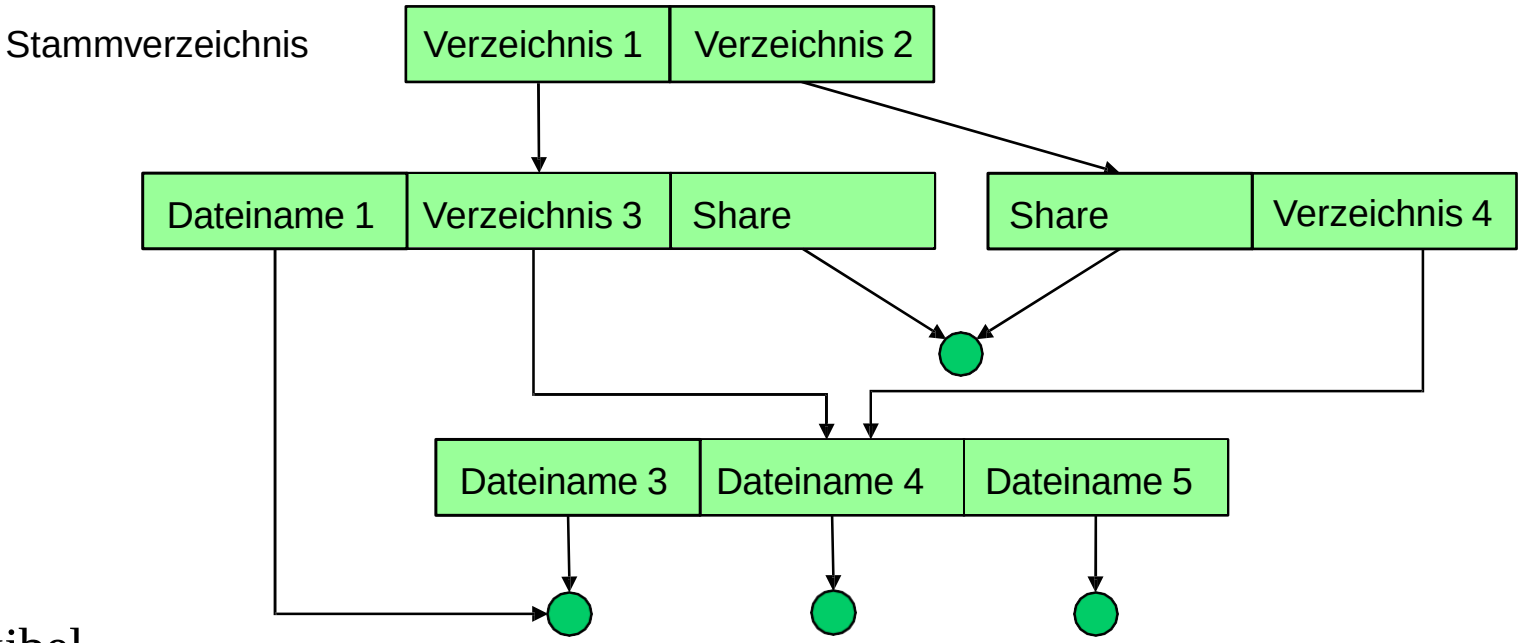
Beispiel:
- aktuelles Verzeichnis: Verzeichnis4
- absoluter Pfad: \Verzeichnis1\Verzeichnis4\Verzeichnis5\Dateiname6
- relativer Pfad: Verzeichnis5\Dateiname6

- Löschen eines Verzeichnisses
  - entweder erst leeren, dann löschen ➔ erheblicher Aufwand
  - oder mit allen Dateien und Unterverzeichnissen löschen ➔ gefährlich

Azyklische und allgemeine Graphen

- Unterverzeichnis oder Datei erscheint im Dateisystem an zwei oder mehr Stellen, ist jedoch physikalisch nur einmal vorhanden.

➔ Änderungen sind an allen Stellen sofort ersichtlich.



\+ flexibel

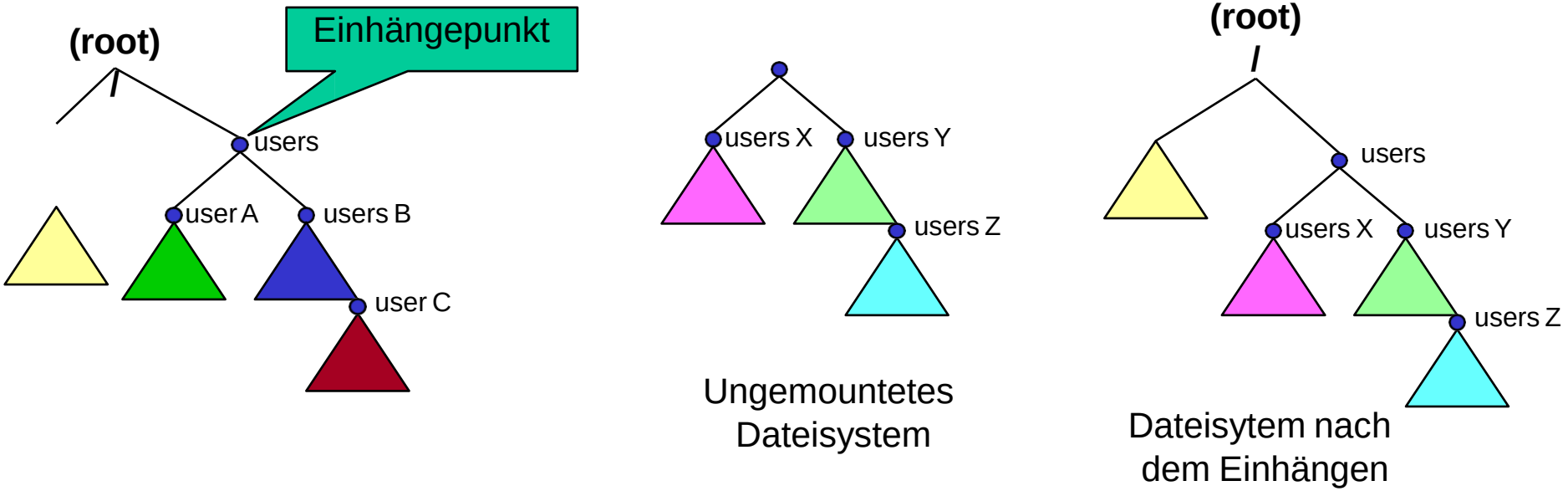
– Löschen von Dateien oder Unterverzeichnissen kompliziert

⇨ Was passiert mit Link nach dem Löschen des Ziels?

– Freiheit von zyklischen Verweisen sicherstellen

Einhängen von Dateisystemen

- Dateisystem muss vor der Verwendung beim System angemeldet werden
  ➔ Einhängen eines Dateisystems (mount)
- Ein Dateisystem wird an ein Einhängepunkt eingehängt
  - Spezielle Einhängepunkte oder beliebig im Verzeichnis
  - Einhängen an leere/volle Verzeichnisse
  - Mehrmaliges Einhängen des gleichen Dateisystems an unterschiedliche Stellen
  - Zeit des Einhängens: beim Hochfahren des Systems / zu beliebigen Zeiten



Implementierung von Dateisystemen

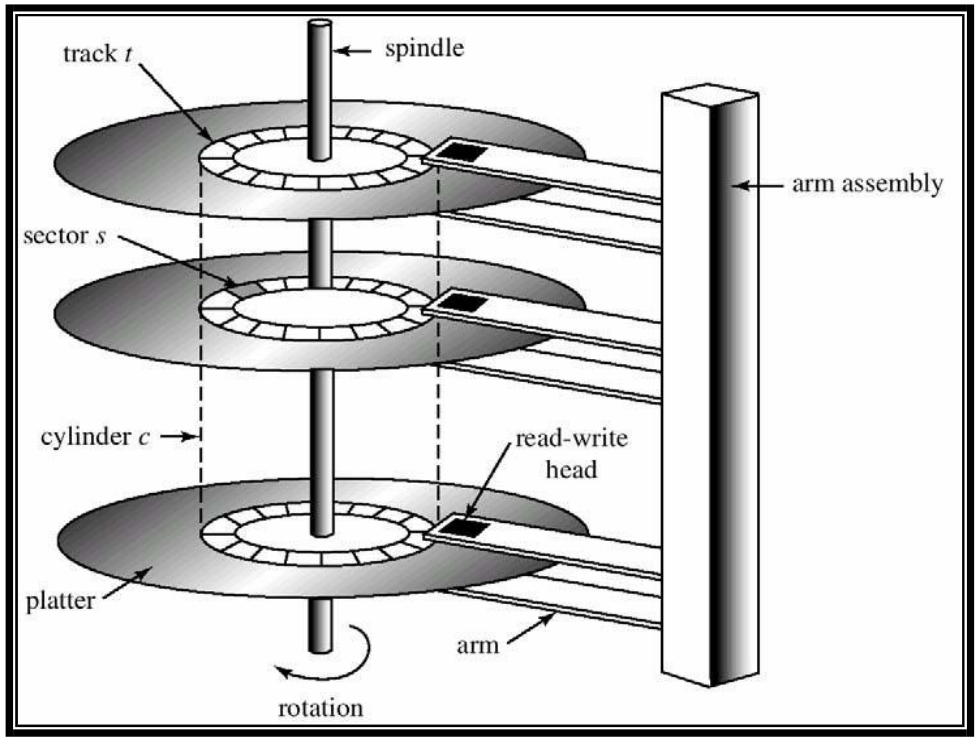
- Dateisysteme werden auf Hintergrundspeicher implementiert
  - Gebräuchlichster Hintergrundspeicher: Festplatte
    - Lesen und Schreiben auf die gleiche Position
    - Direkter Zugriff auf beliebige Daten

- Fragen bei der Implementierung von Dateisystemen
  - Strukturierung von Dateien
  - Verwaltung des Speicherplatzes
  - Implementierung von Verzeichnissen

Implementierung auf Festplatten

Festplatte

- Daten werden magnetisch gespeichert
- Eine Festplatte besteht aus Scheiben und Lese/Schreibköpfen (beidseitig)
- Scheiben bestehen aus Spuren (*Tracks*), die in Sektoren aufgeteilt sind
- Zylinder: Menge aller Spuren, die bei einer bestimmten Position der L/S-köpfe erreicht werden
- Zugriff wird durch Gerätekontroller (*disk controller*) gesteuert
- Um den Durchsatz zu erhöhen, werden die Daten Blockweise übertragen. Ein Block = Mehrere Sektoren
- Aktuelle Festplatten haben Kapazitäten bis mehrere Terabytes



1 GByte = 1000 MByte
= 1000 000 kByte
= 1000 000 000 Byte
= 8000 000 000 bit

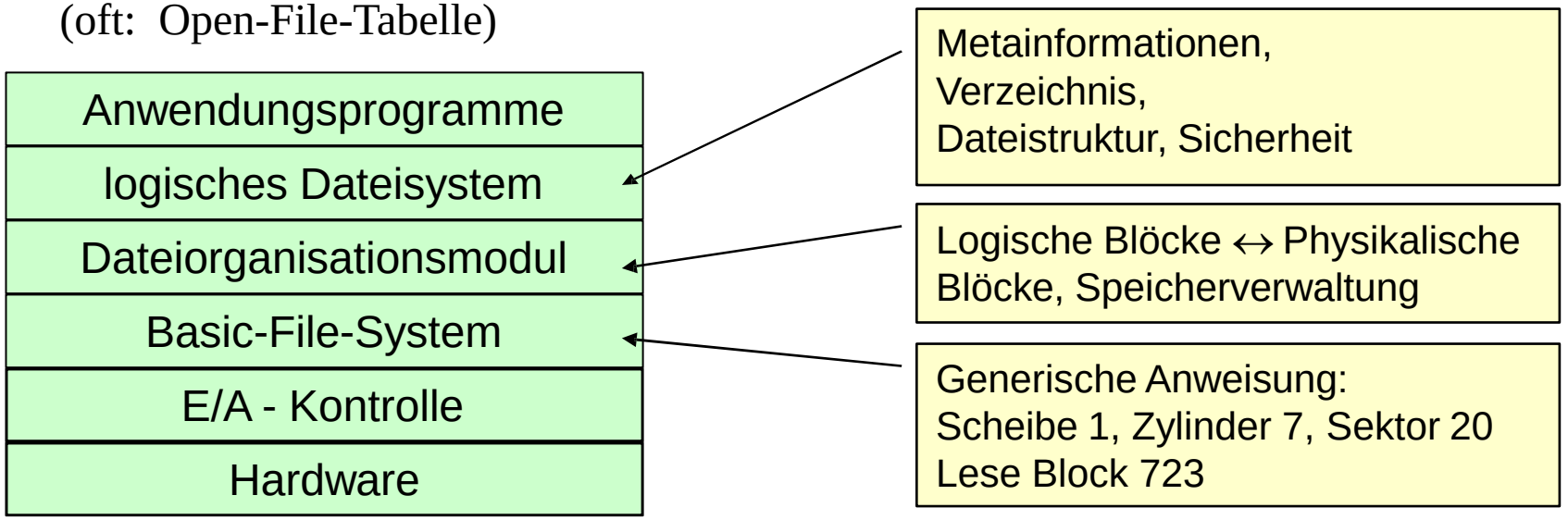
Implementierung: FAT Dateisystem

FAT-Dateisystem (File-Allocation-Table)

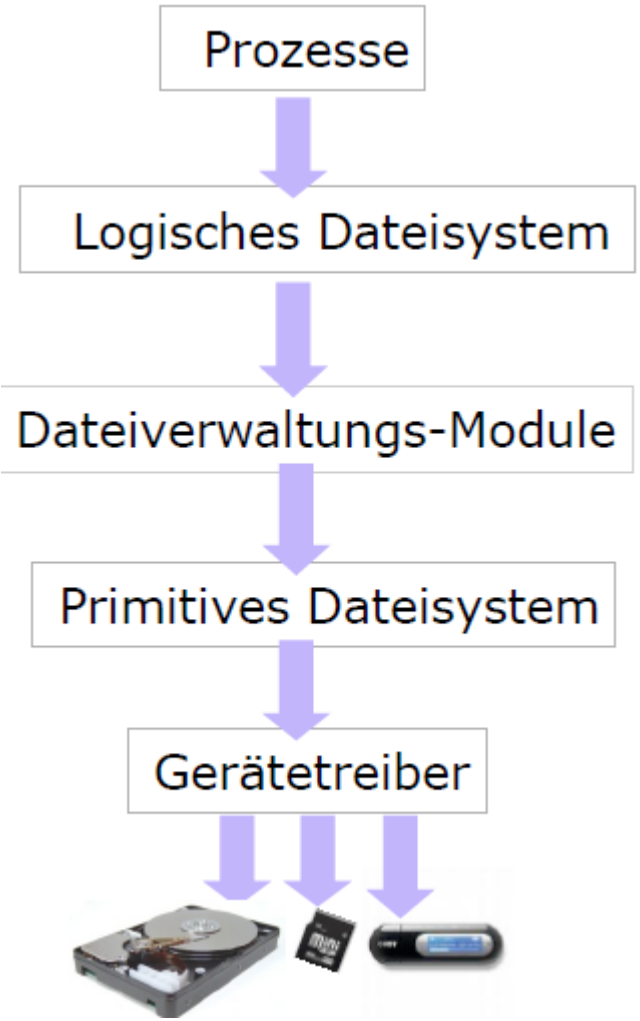
FAT16 (DOS, Windows 95/ NT)		
Partitionsgröße	Clustergröße	Sektoren /Cluster
16-127 MB	2 KB	4
128-255 MB	4 KB	8
256-511 MB	8 KB	16
512-1023 MB	16 KB	32
1024-2047 MB	32 KB	64
2048-4096 MB	64 KB	128 nur Win NT
FAT32 (DOS, Windows 98, 98 SE, 2000)		
Partitionsgröße	Clustergröße	Sektoren /Cluster
256 MB – 8,01 GB	4 KB	8
8,02 GB – 16,02 GB	8 KB	16
16,03 GB– 32,04 GB	16 KB	32
> 32,04 GB	32 KB	64

Implementierung: Schichten

- Um Komplexität zu verringern sind Dateisysteme als Schichten realisiert
  - **E/A-Kontrolle**: Treiber für Dateiübertragung, Hauptspeicher ↔ Plattensystem
  - **Basic-File-System**: veranlasst Treiber, physikalische Blöcke auf Festplatte zu schreiben bzw. von dort zu lesen
  - **Dateiorganisationsmodul**: Abbildung logische ↔ physikalische Blockadressen
  - **Logisches Dateisystem**: erledigt Neueintragung/Löschung einer Datei sowie E/A-Zugriffe auf eine Datei und versorgt das Dateiorganisationsmodul mit Informationen zu einem (symbolischen) Dateinamen unter Verwendung der Verzeichnisstruktur (oft: Open-File-Tabelle)



Schichten eines Dateisystems



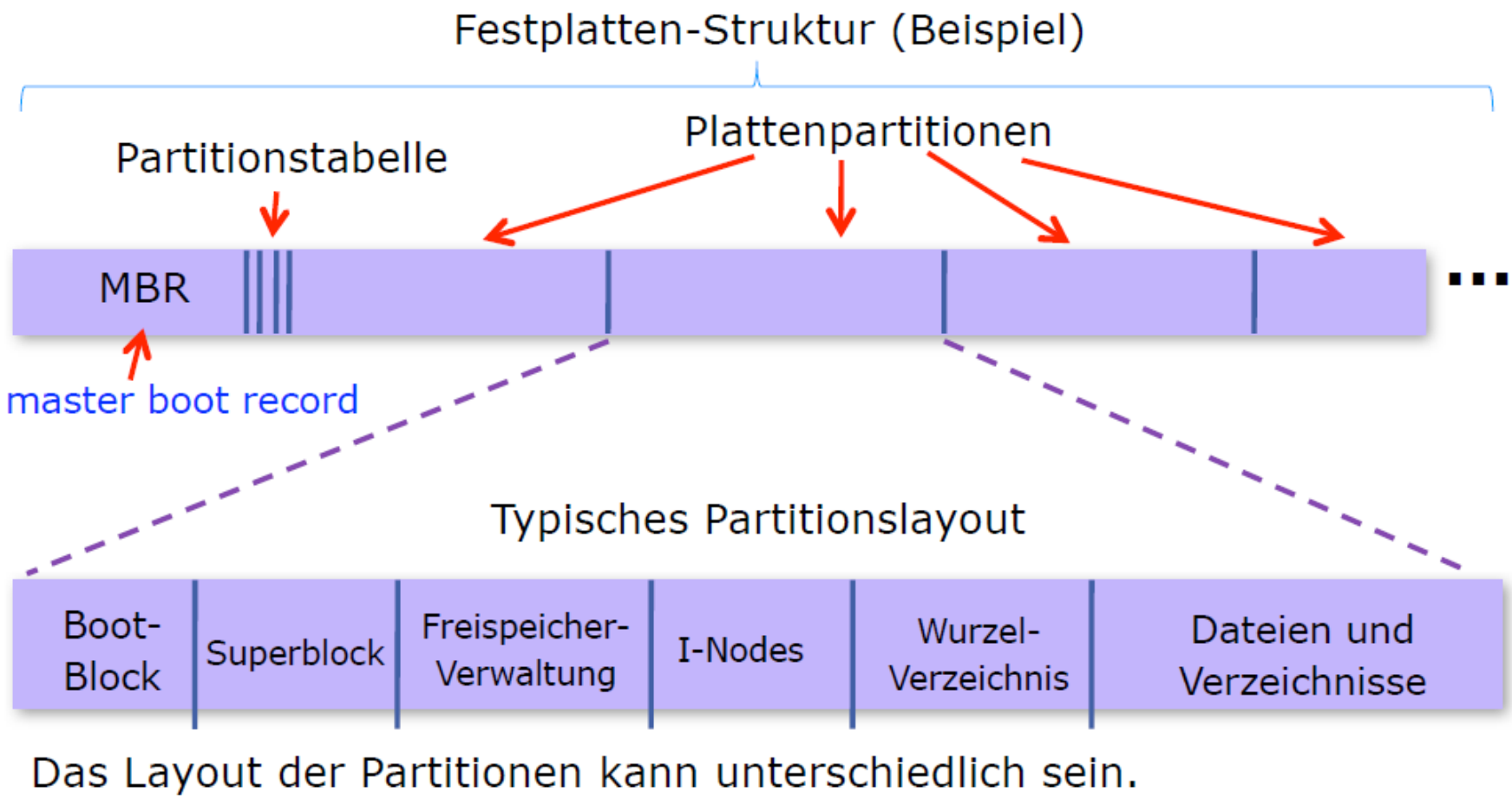
open("Dateiname"); read(pos);

readblock 1 of file 12540
read block at offset 3045

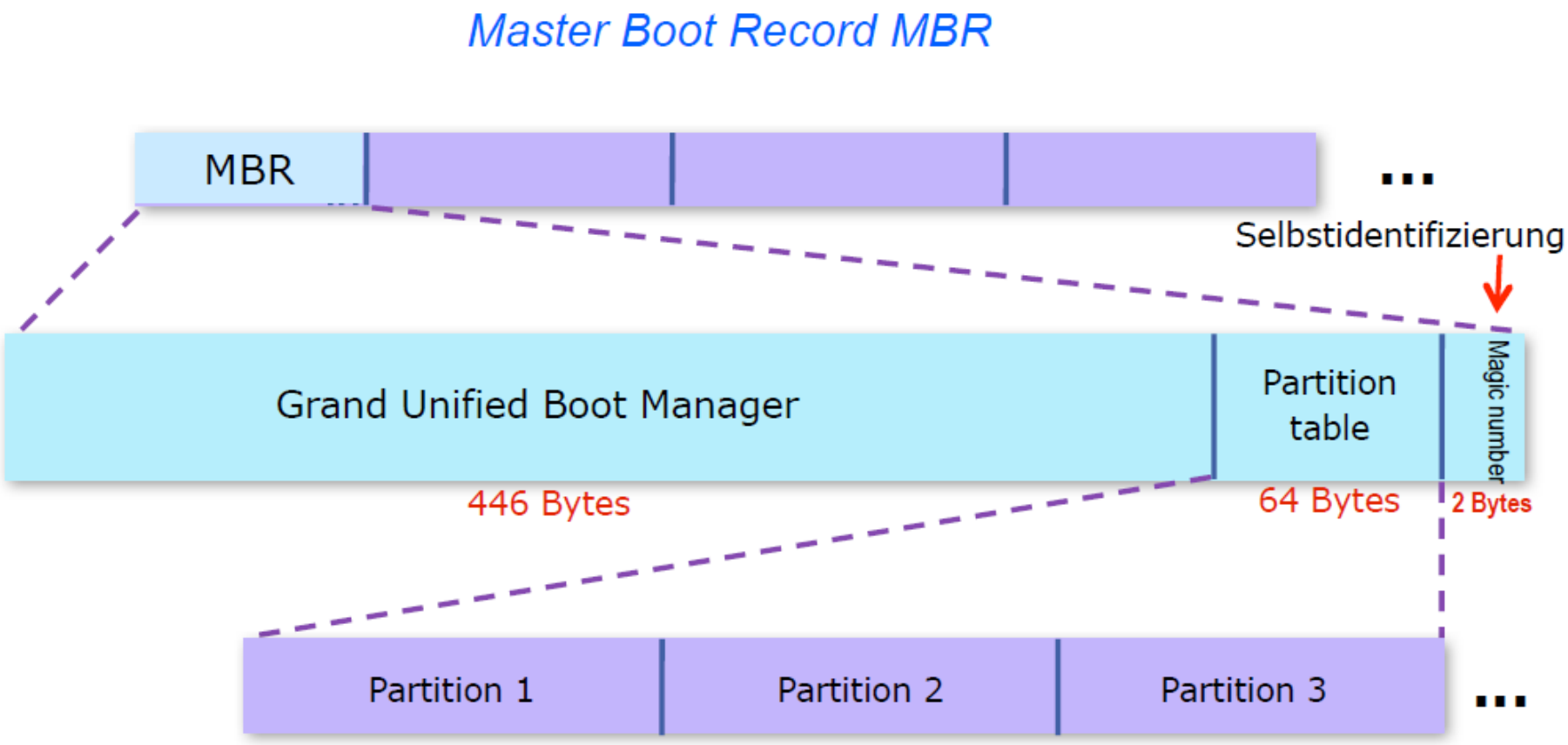
read drive 1, cylinder 20, track 2, sec 2
or read LBA 231230

out 210h, ax

Dateisystem-Implementierung



Festplatten-Struktur (Beispiel)



MBR und Superblock

- Master Boot Record
  - Sektor 0
  - Im MBR ist die Partitionstabelle, die Anfangs- und Endadresse jeder Partition beinhaltet.
  - Eine Partition ist immer als aktiv markiert.
  - Wenn der Computer gestartet wird, liest das BIOS den MBR ein und führt ihn aus.
    - Der erste Block der aktiven Partition (Boot-Block) wird gesucht und ausgeführt.
      - Das Programm im Boot-Block lädt das Betriebssystem.
  - Jede Partition beginnt immer mit einem Boot-Block, der das eigentliche Betriebssystem lädt.
- Superblock oder Volume Control Block
  - Wichtige Parameter des Dateisystems sind hier.
  - Wird im Speicher geladen, wenn nach Starten des Computers zum ersten Mal auf das Dateisystem zugegriffen wird.
  - Hier sind Informationen wie z.B.
    - **Magische Zahl**, um Typ und Größe des Dateisystems sowie andere Verwaltungsinformation zu identifizieren.

Weitere wichtige Information des Dateisystems wie Freispeicher-Verwaltung, I-Nodes, Wurzelverzeichnisse usw. wird je nach Dateisystem unterschiedlich organisiert.

Strukturen des Dateisystems

mount table

- Hier sind alle Dateisysteme, die im Betriebssystem aktiv sind.

in-memory directory-structure cache

- Information über die vor kurzem verwendeten Verzeichnisse.

system-wide open-file table

- Kopie der FCB von jeder Datei, die gerade geöffnet ist.

per-process open-file table

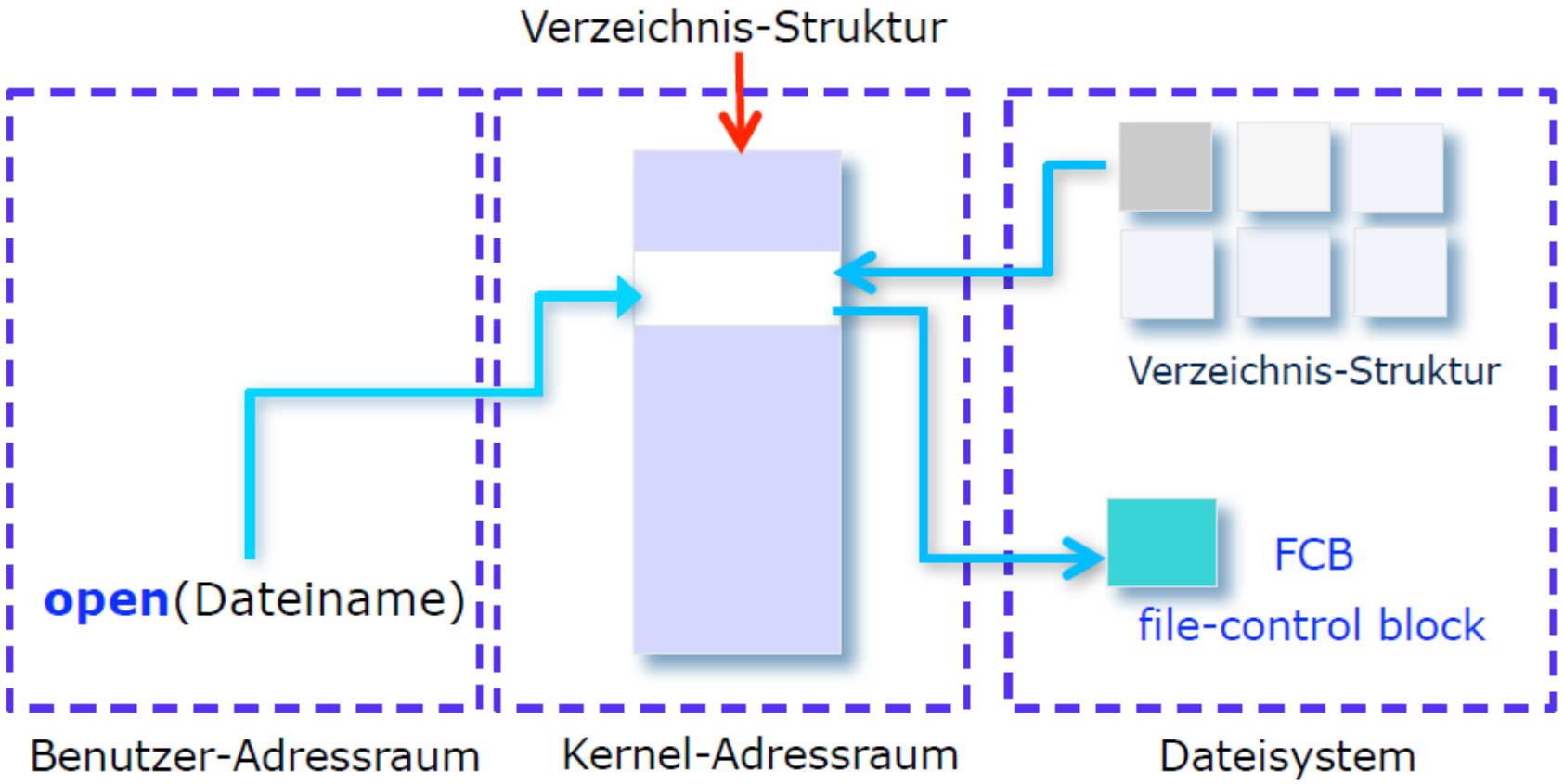
- Zeigen auf Einträge im system-wide open-file table, sowie Prozess spezifische Datei-Information.

Dateisteuerblock (FCB)

- Zugriffsrechte
- Datum (create, access, modify)
- Datei (owner, group) ACL (access control list)
- Dateigroße
- Dateiblöcke oder Zeiger auf Dateiblöcken

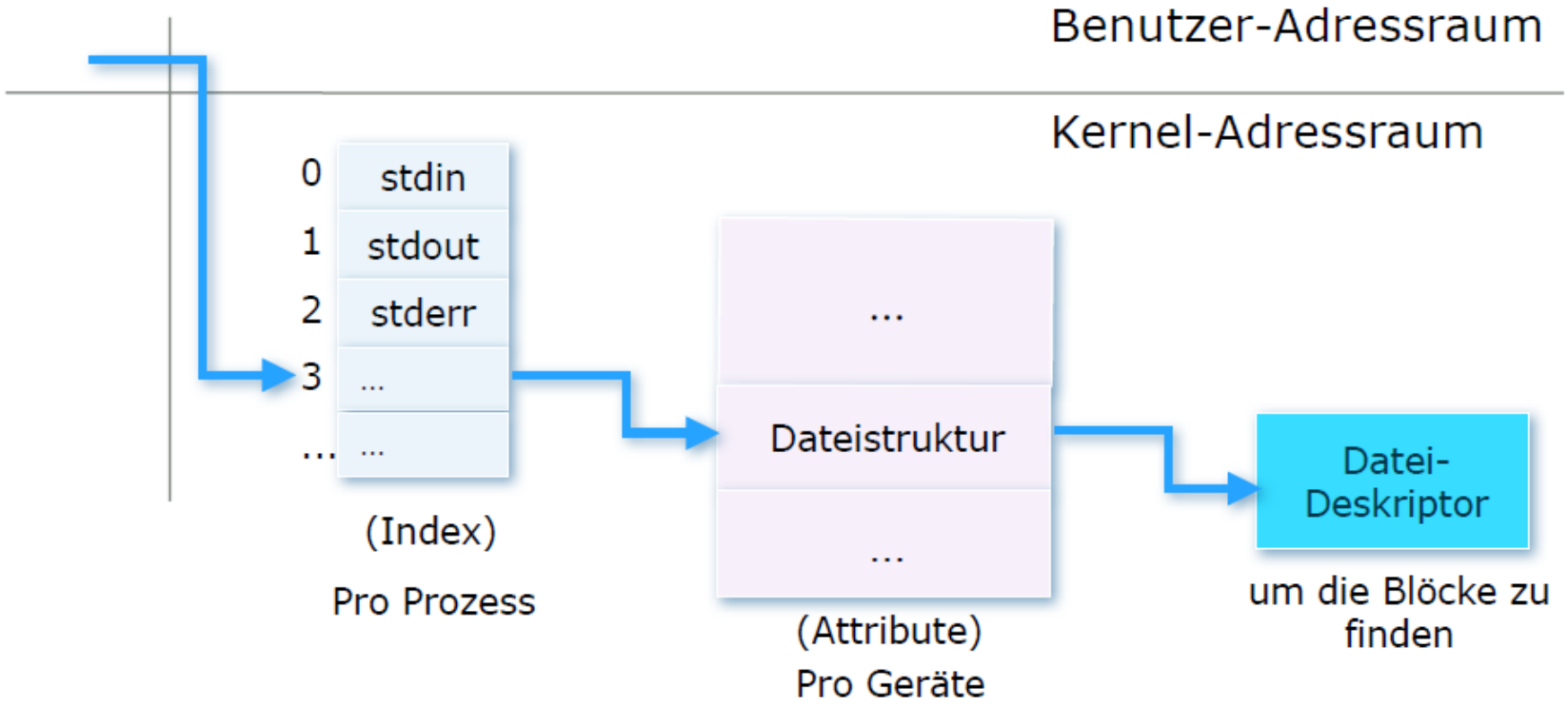
Wenn eine neue Datei erzeugt wird, wird das logische Dateisystem eingeschaltet und eine neue FCB erzeugt. Oft gibt es im Speicher eine Liste freier FCBs, die im voraus erzeugt werden.

Das Öffnen einer Datei

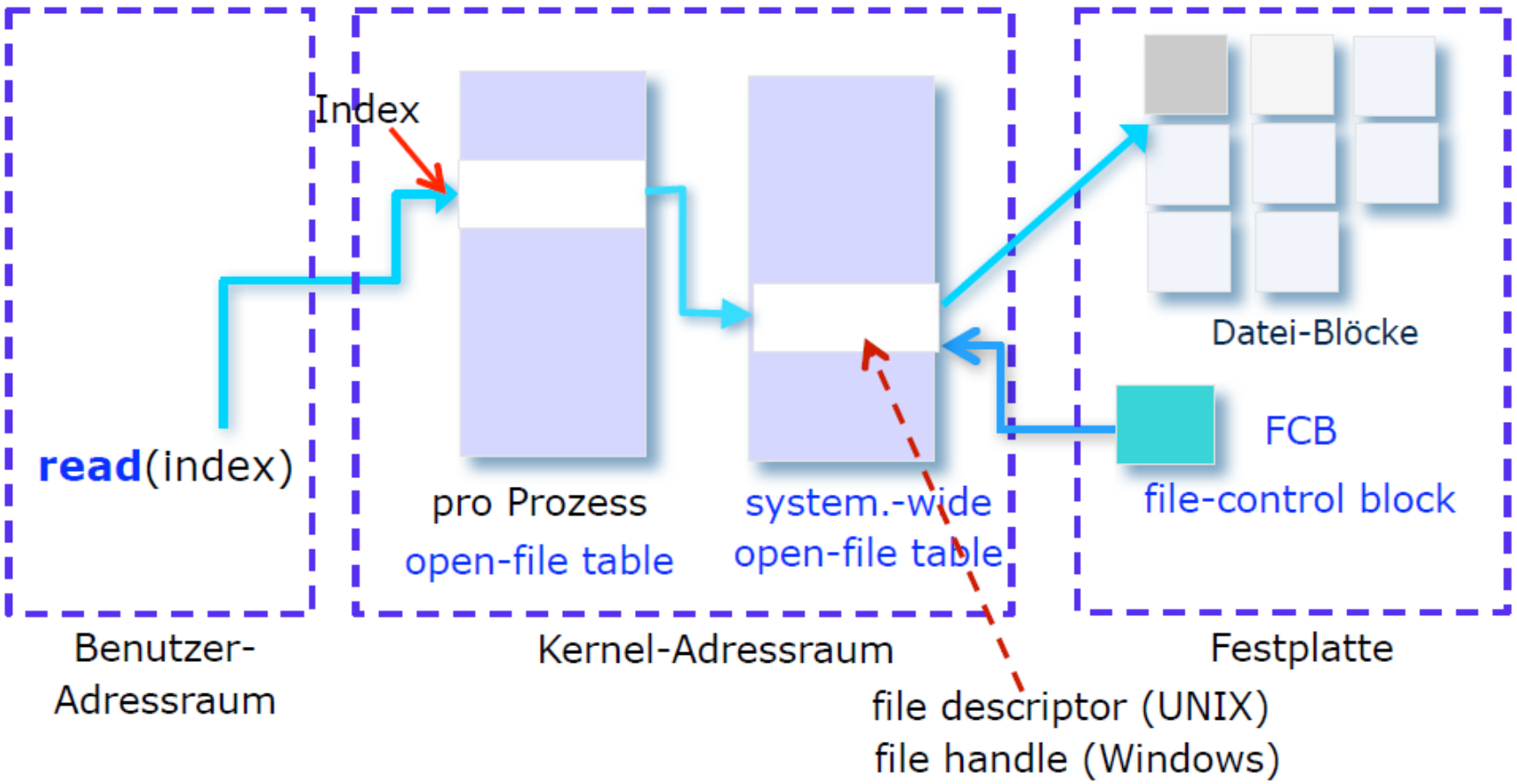


Beispiel: Unix open()

```
int fid = open("Dateiname", flags);
read(fid, ...);
```



Das Lesen in einer Datei

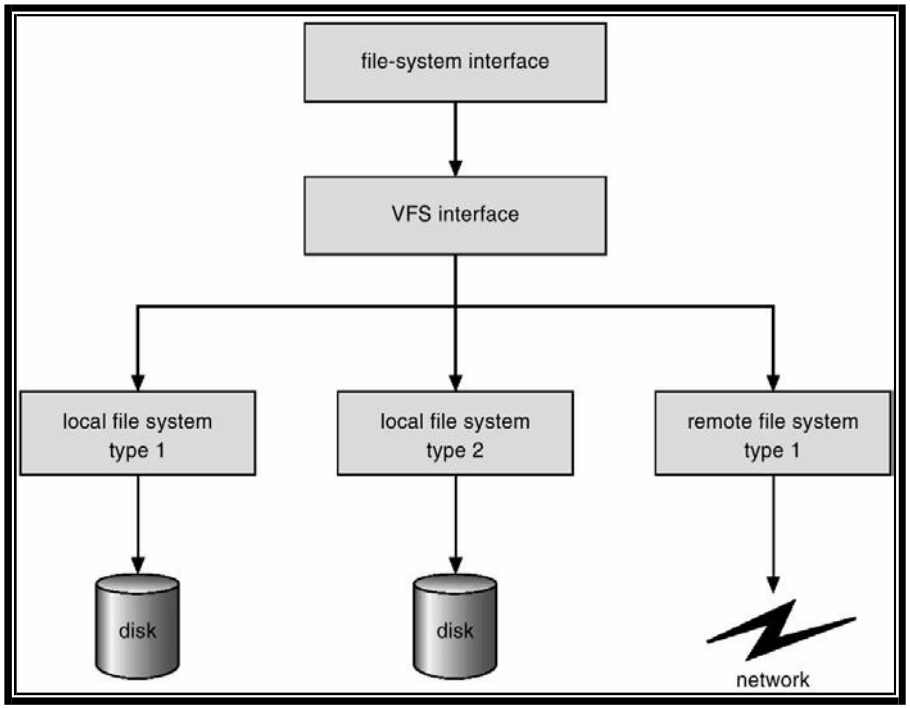


Implementierung: VFS

Wozu ein Virtuelles Dateisystem (Virtual File System)?

- Moderne Rechnersysteme unterstützen mehrere Dateisysteme, z.B. CD-ROM, DVD, FAT16, FAT32, NTFS, ext2fs, HPFS
- Wie können unterschiedliche Dateisysteme in einen Verzeichnis integriert werden?
- Wie können Benutzer zwischen unterschiedlichen Dateisystemen wechseln und navigieren?

- Lösung 1:
  - Implementierung alle Zugriffsroutinen für jedes Dateisystem
- Lösung 2:
  - Virtuelles Dateisystem, welches dem Benutzer eine transparente Sicht erlaubt

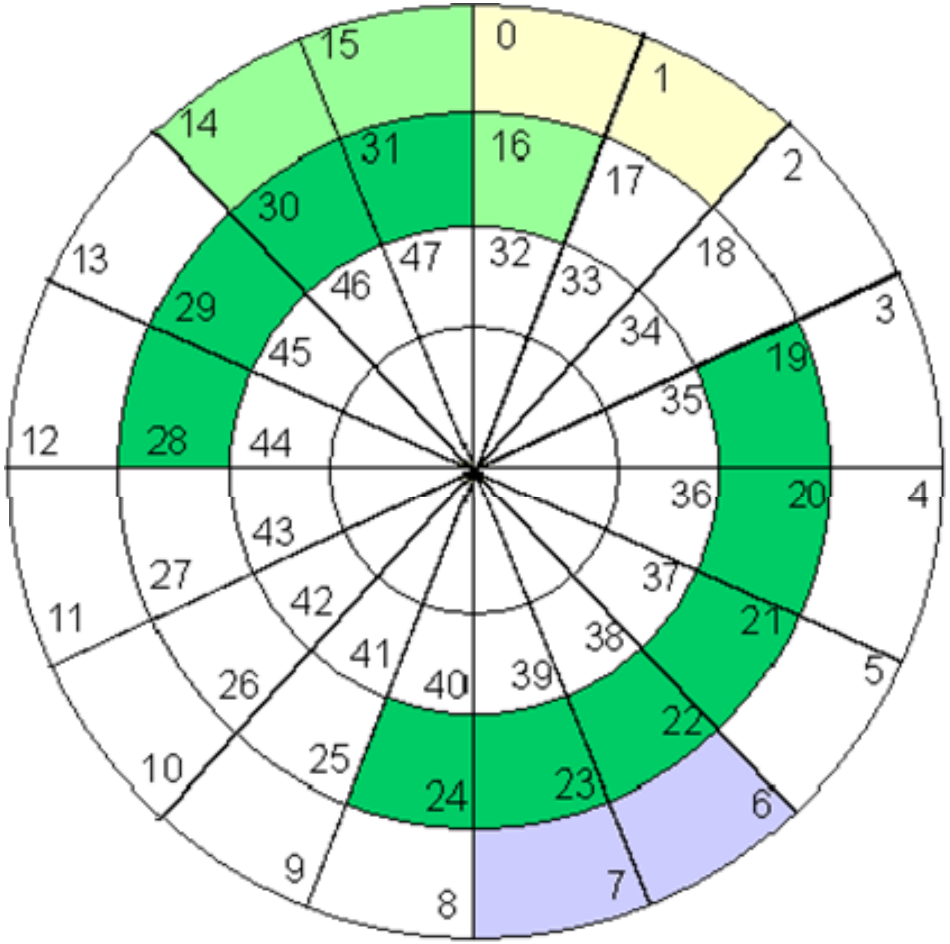


Implementierung von Dateisystemen

- Belegungsstrategien:
  - Wie soll der Plattenspeicher auf die Dateien aufgeteilt werden?
- Ziele:
  - schneller Zugriff auf Dateien
  - effiziente Nutzung des Plattenspeichers
- Allokations-Strategien:
  - zusammenhängende Belegung
  - verkettete Belegung
  - Belegung durch verkettete Listen mit einer Tabelle im Arbeitsspeicher (Indexed allocation)

Zusammenhängende Belegung

Eine Datei belegt eine Reihe zusammenhängender Blöcke auf der Festplatte



Verzeichnis

Datei	Start	Länge
Name1	0	2
Name2	14	3
Name3	19	6
Name4	28	4
Name5	6	2

Ähnlich wie „Segmentierung“
mit allen Vor- und Nachteilen

Zusammenhängende Belegung (2)

- Vorteile
  - \+ Sequenzieller und direkter Zugriff auf Datei
    sequenziell: zuletzt referenzierte Blockadresse + 1
    direkt: Startblock der Datei ist *b* (physikalisch), physikalische Adresse des logischen Blocks *i* der Datei ist $b + i, \forall i = 0,1,...$

- Nachteile
  - Speicherplatz für neue Dateien finden: **first-fit**, **best-fit**, **rotating-first-fit**
    freier Speicherplatz wird allmählich in kleine Stücke zerlegt
    $\Rightarrow$ **externe Fragmentierung**
  - Größe einer Datei muss bei Erzeugung angegeben werden
    - falls nicht bekannt, möglicherweise deutlich überschätzt
    - falls bekannt, aber Datei über lange Zeit anwächst, bleibt viel Speicherplatz lange ungenutzt

    $\Rightarrow$ **interne Fragmentierung**

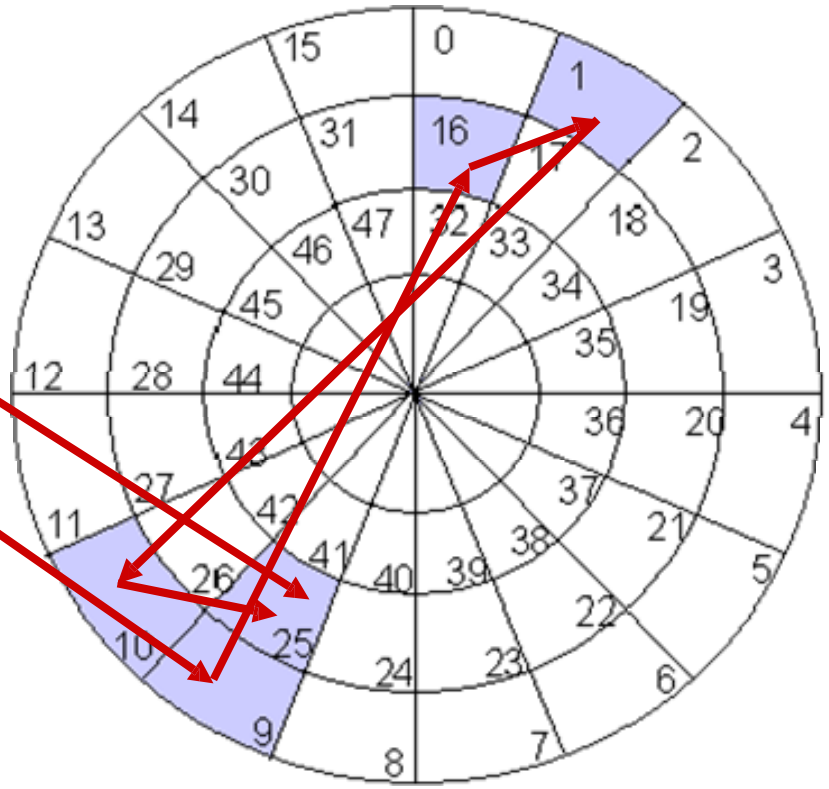
- Abhilfe:
  - Umkopieren (mit Geschwindigkeitsverlust), Garbage-Collection (Defragmentierung)

Verkettete Belegung

- Eine Datei ist als verkettete Liste von Blöcken realisiert.
- Ein Verzeichniseintrag enthält Zeiger auf den ersten und letzten Block einer Datei und deren Länge. Jeder Block enthält einen Zeiger auf den nächsten Block der Datei.

Verzeichnis

Datei	Start	Ende	Länge
Name1	9	25	5



Verkettete Belegung (2)

- Vorteile
  - Keine Fragmentierung, da Zeiger verwendet werden
  - Größe einer Datei muss im Voraus nicht bekannt sein

- Nachteile
  1. Nur effizient für sequenziellen Zugriff
  2. Zugriff auf Zeiger kostet zusätzliche Zeit
  3. Zeiger verbrauchen zusätzlichen Speicher
  4. Was machen, wenn ein Block kaputt ist?

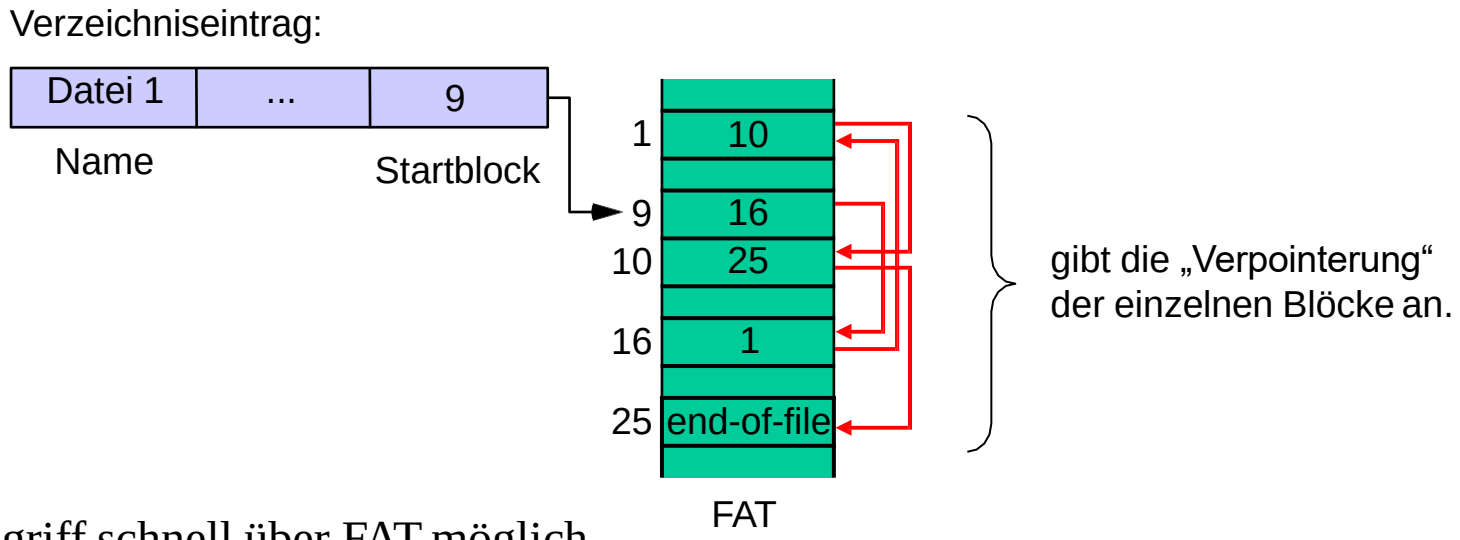
- Lösungen:

  Für 1., 2. und 3. ⇨ Lesen von Clustern (Cluster = Mehrere Blöcke)

  Für 4. ⇨ Verwendung von doppelt verketteten Listen

File Allocation Table FAT

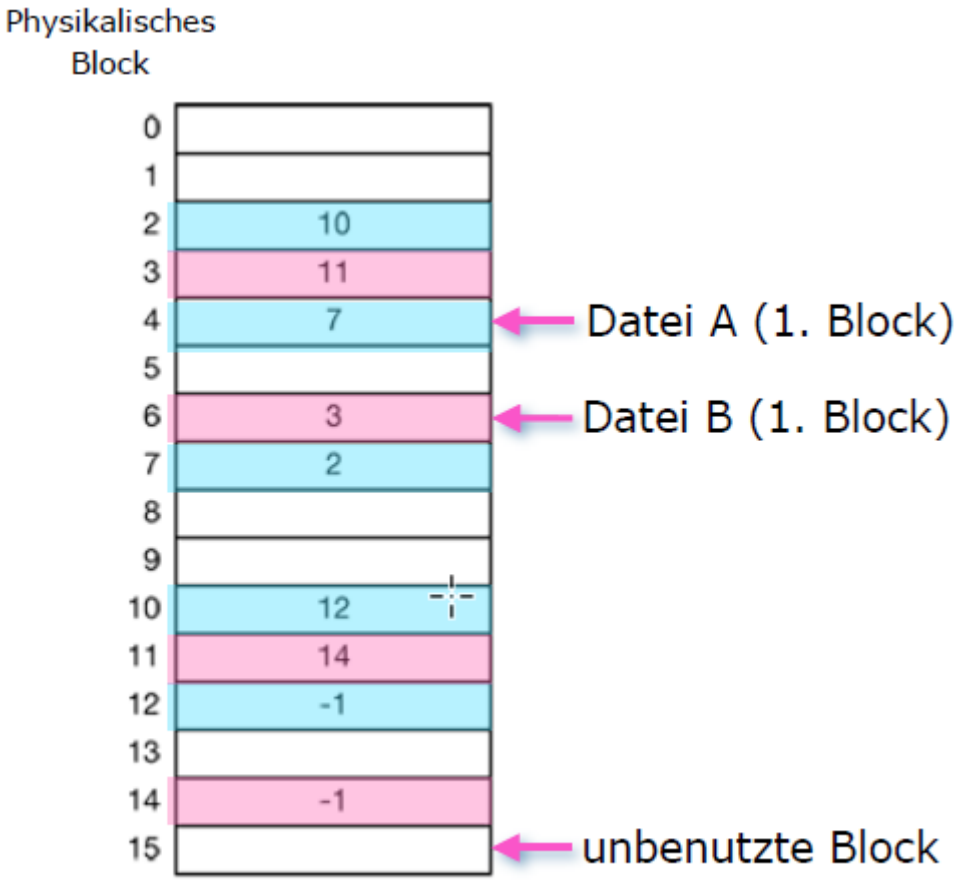
- Variante: **FAT** (File-Allocation-Table, Dateibelegungstabelle)
  - Am Anfang jeder Partition ist eine Tabelle mit je einem Feld für jeden Block.
  - Im Verzeichnis steht die Nummer *m* des ersten physikalischen Blocks der Datei.
  - An *m*-ter Stelle der Tabelle steht die Nummer des folgenden Blocks.
  - Verkettung bis zum letzten Block der Datei mit speziellem EOF-Wert in der Tabelle.
  - Freie Blöcke in der Tabelle sind mit einer 0 gekennzeichnet.



+ Direkter Zugriff schnell über FAT möglich
− Erhöhter Aufwand durch ständiges Springen der Schreib-/Leseköpfe zwischen FAT und Blöcken
− Verlust der Tabelle

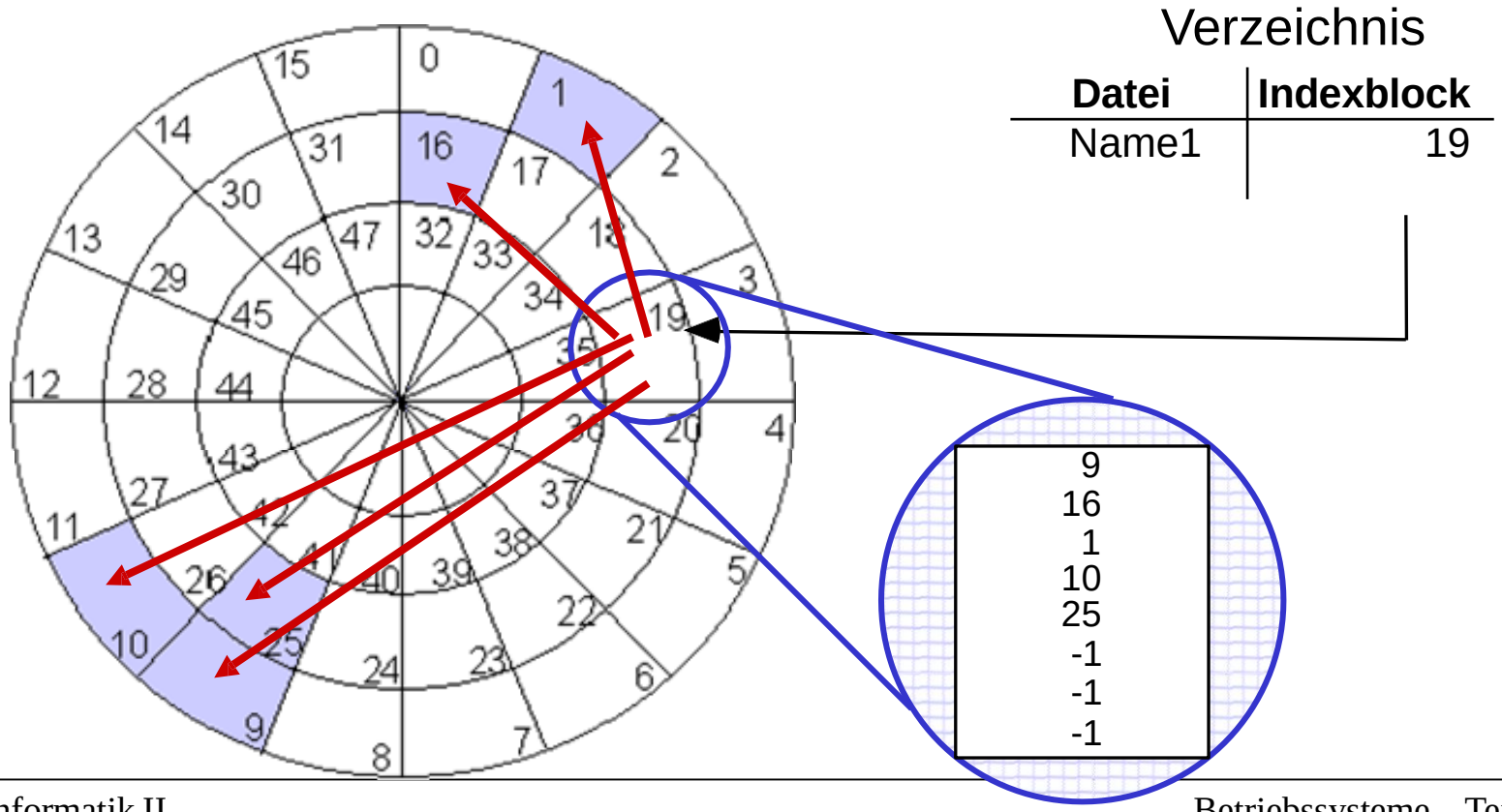
File Allocation Table FAT (2)

- Vorteile
  - Dateien können sehr leicht und effizient vergrößert werden
- Nachteile
  - Interne Fragmentierung
  - schlecht für random accesses
  - fehleranfällig



Indizierte Belegung

- Ein **Indexblock** enthält alle Zeiger einer Datei ➔ Zeiger = Blockadresse
- Der Verzeichniseintrag für eine Datei enthält nur noch die Adresse des Indexblocks; die Adresse des *i*-ten Datenblocks der Datei steht an der *i*-ten Stelle des Indexblocks.



Indizierte Belegung (2)

- Vorteil:
  - \+ direkter Zugriff ohne externe Fragmentierung
- Nachteil:
  - – Indexblock belegt Speicherplatz
- „Richtige“ Größe des Indexblocks?

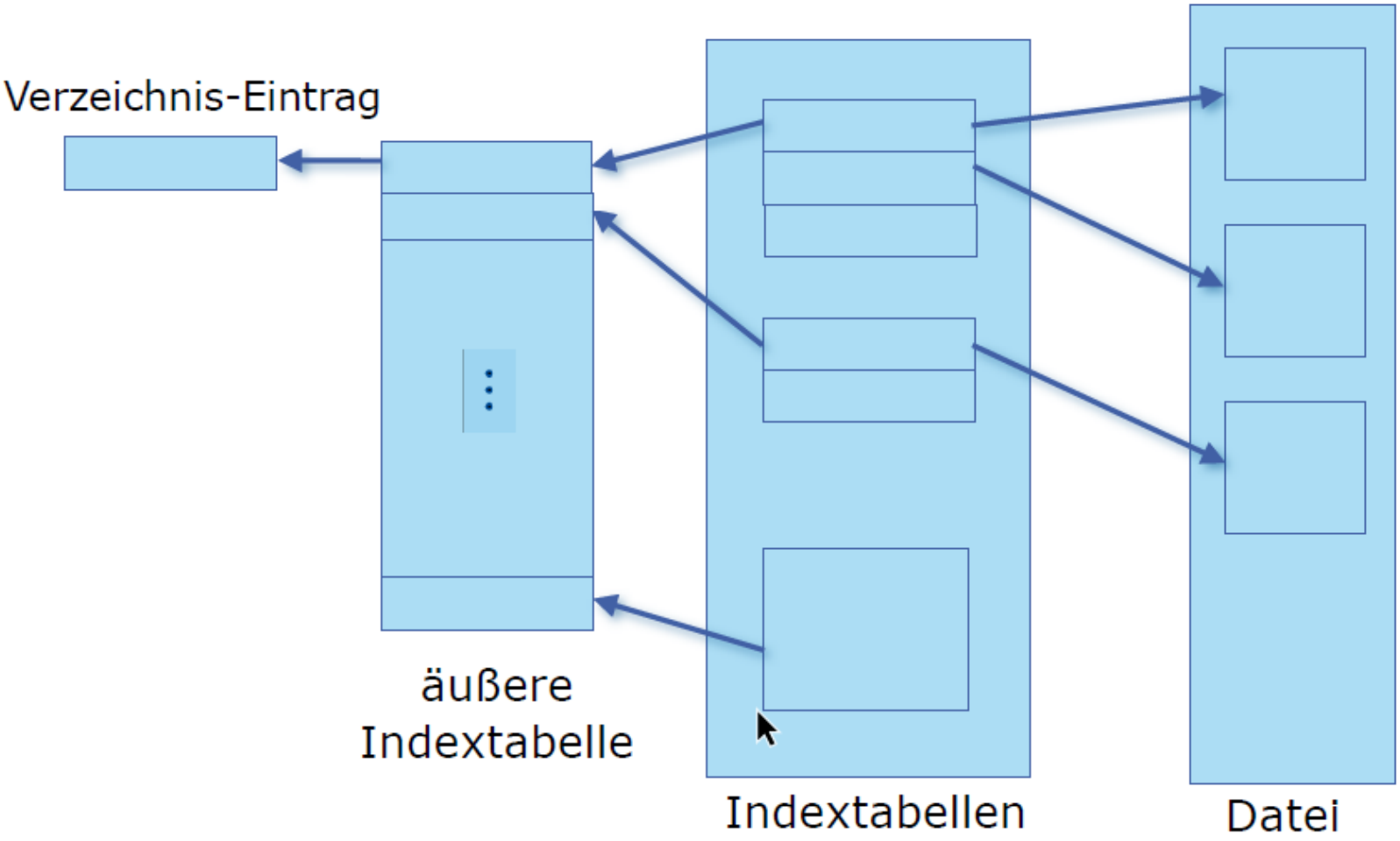
  Zu groß ⇒ Speicherplatz wird verschwendet

  Zu klein ⇒ Speicherplatz des Indexblocks reicht nicht für alle Zeiger
- Große Dateien
  - – Verkettung von Indexblöcken
  - – Hierarchie von Indexblöcken ➔ Indexblock mit Zeiger auf Indexblöcke

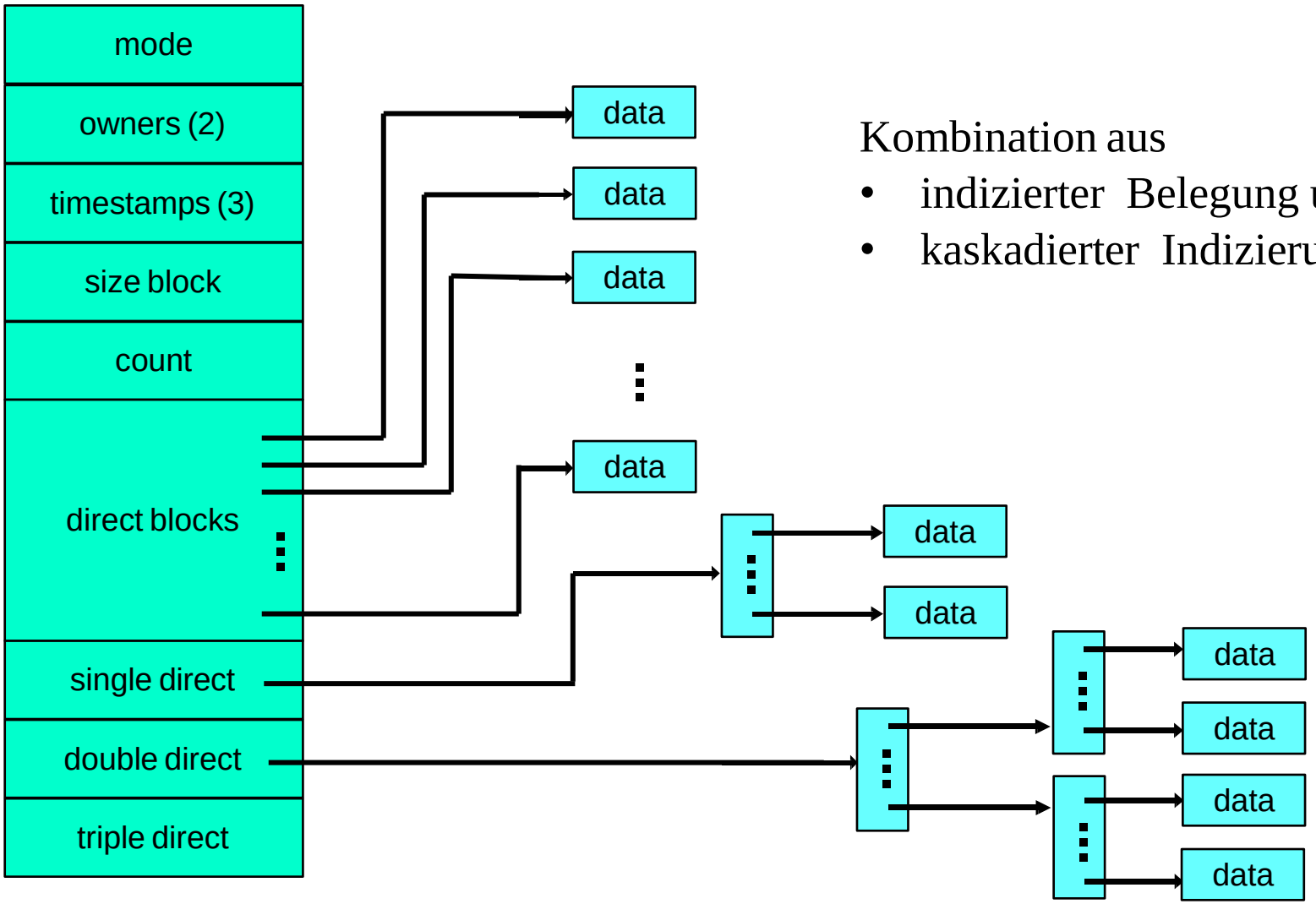
Verkettete Listen mit indizierter Tabelle

- Fragen
  - Wie groß soll ein Indexblock sein?
    - zu groß → Speicher-Overhead
    - zu klein → Probleme mit großen Dateien
- Lösungen
  - Linked scheme
    - Mehrere Indexblocks werden verkettet
  - Multilevel index
    - Zweistufige Indexblocktabelle wird verwendet
  - Combined scheme
    - Für kleine Dateien wird nur eine einfache Indextabelle verwendet. Wenn aber die Dateien groß werden, verwandelt sie sich in eine mehrstufige Indextabelle.

Multilevel index



Indizierte Belegung: UNIX inode



Kombination aus

- indizierter Belegung und
- kaskadierter Indizierung

UNIX inode (2)

- Vorteil:
  - \+ schneller Zugriff für kleine Dateien
  - \+ keine externe Fragmentierung
- Nachteil:
  - – Interne Fragmentierung
  - – Die maximale Dateigröße ist begrenzt
- ext2-Dateisystem-Grenzen im Linux

Blockgröße:	**1 kB**	**2 kB**	**4 kB**	**8 kB**
max. Dateigröße:	16 GB	256 GB	2048 GB	2048 GB
max. Dateisystemgröße:	2047 GB	8192 GB	16384 GB	32768 GB

Das Dateisystem begrenzt die Anzahl von Unterverzeichnissen in einem gegebenen Verzeichnis auf 32.768 Stücke.

Verwaltung des freien Speichers

- Die sogenannte **free-space-list** ist eine Tabelle, in der alle nicht belegten Speicherblöcke verzeichnet sind. Sie wird beim Erzeugen und Löschen einer Datei modifiziert.

- Mögliche Implementierungen:
  - Liste
  - Bit-Vektor ➔ ein Bit pro physikalischem Block
  - Verkettete Liste
  - Gruppierung freier Blöcke

    Im ersten freien Block sind *n* Adressen, davon *n-1* Zeiger auf freie Blöcke und ein Zeiger auf die nächste Blockadresse mit wiederum *n* Adressen.

    ➔ V o r t e i l :

    \+ Adressen vieler freier Blöcke können schnell gefunden werden.

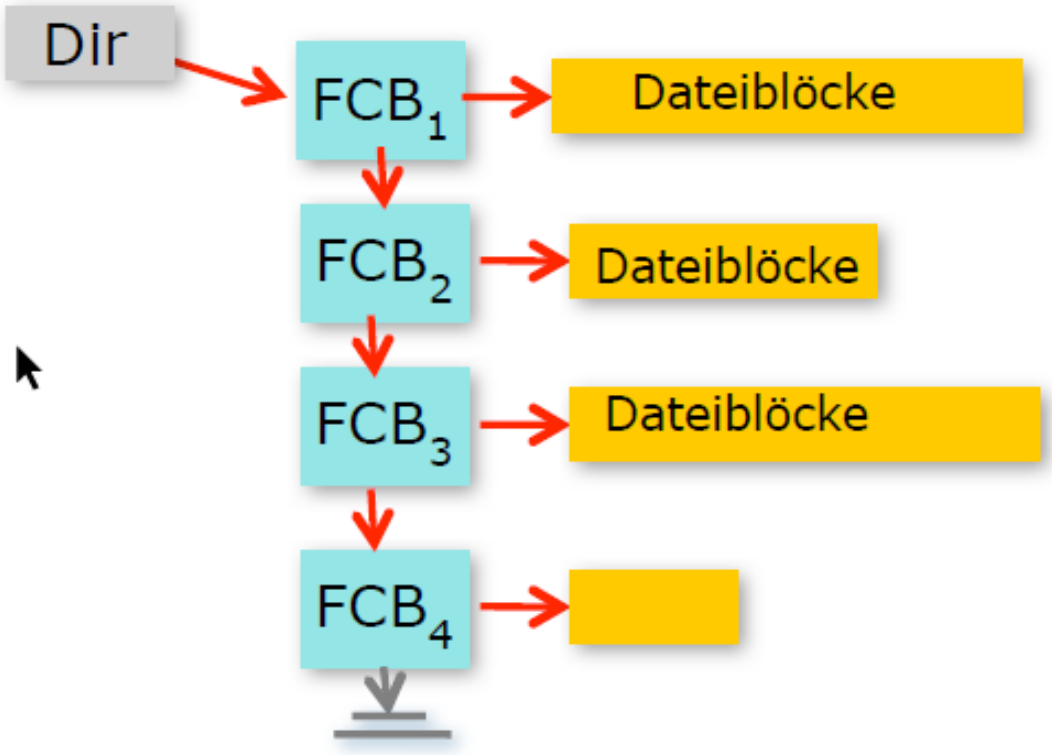
    \+ Bei vielen aufeinanderfolgenden freien Blöcken muss nur die Adresse des ersten sowie die Anzahl abgespeichert zu werden.

Implementierung von Verzeichnissen

- Lineare Liste
  - Neu erstellte Dateien hinten anhängen
  - Löschen einer Datei:
    - als „frei“ markieren oder
    - Eintrag in Liste freier Verzeichnisplätze oder
    - letzten Verzeichniseintrag an die freigewordene Stelle verschieben
  - \+ einfach
  - − sequenzielle Suche, zeitaufwendig ➔ Abhilfe: Sortieren oder **Cache**
- Hashtabelle
  - Viel effizienter als lineare Listen
  - Probleme:
    - als „frei“ markieren oder
    - Eintrag in Liste freier Verzeichnisplätze oder
- Eine Alternative sind *chained-overflow hash tables*

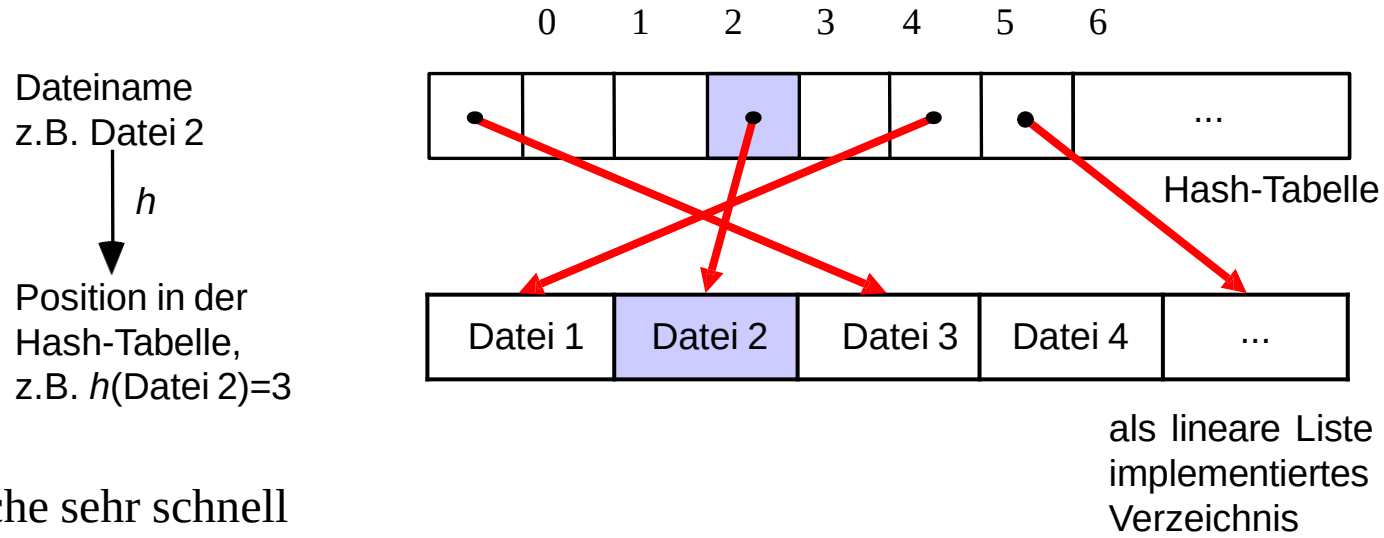
Implementierung von Verzeichnissen (2)

Lineare Liste



Hashtabelle

- Hashtabelle:
  - *h*: {Schlüssel}→{Speicherpositionen in Hash-Tabelle}
  - einfach zu berechnen
  - *h* soll verfügbaren Speicherraum gleichmäßig ausnutzen
  - *h* nicht notwendig injektiv ➔ Menge der Schlüssel kann auf wesentlich kleinere Hash-Tabellen reduziert werden.



+ Suche sehr schnell
+ Einfügen/Löschen einer Datei
− Vorkehrungen für Kollisionen notwendig ➔ Überlauftabelle
− Änderung der Hash-Tabellengröße erfordert neue Hash-Funktion

Hashtabelle (2)