Betriebssysteme und Netzwerke Vorlesung 13

Artur Andrzejak

Dateien und Dateisysteme

Abstraktion der Hardware

- Erinnerung: Eine der zwei Funktionen eines BS ist die „Erweiterte Maschine“ - Abstraktion der Hardware
- Wir haben u.a. zwei wichtige Abstraktionen der HW kennengelernt – welche könnten das sein?

- Logischer / virtueller Adressraum abstrahiert den physischen Speicher
- Prozesse sind Abstraktionen von Prozessoren

Dateien und Dateisysteme

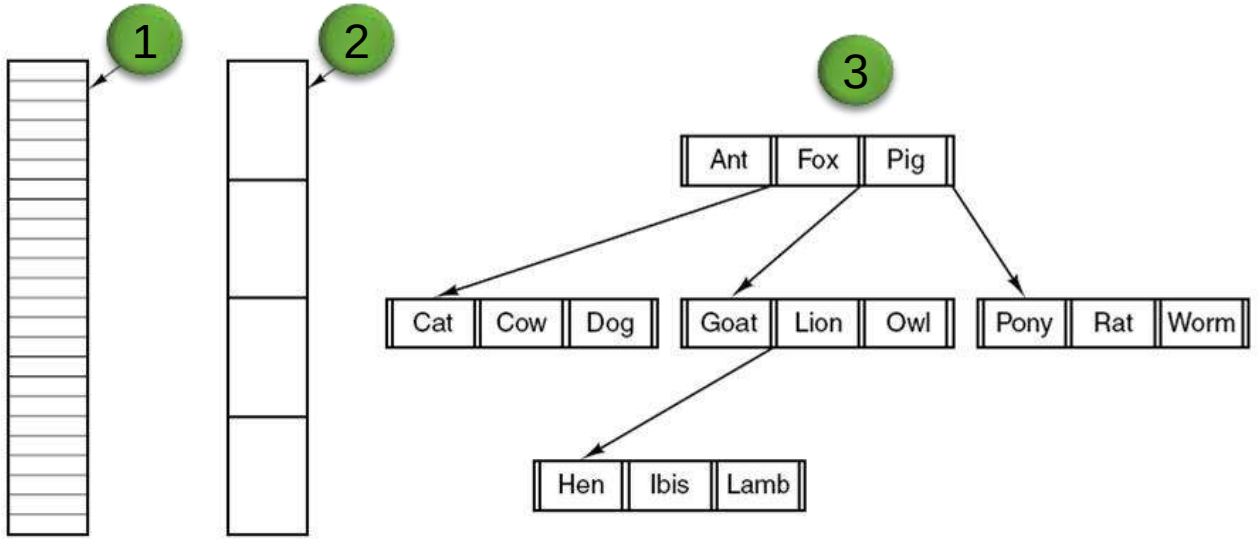
- **Dateien** (**Files**) sind die dritte (und letzte) wichtige Abstraktion
- Definition*:* Dateien sind abstrakte Informationseinheiten, die von Prozessen erzeugt und manipuliert werden
- Alternative Definition: Eine Datei ist eine persistente, strukturierte, benannte, und geschützte Informationseinheit
- Abstraktion der Hardware-Operationen wie ...
  - „Lesen von Block k“ und „Schreiben von Block k“
- **Dateisystem**: Teil des BS zuständig für die Dateien

Warum braucht man Dateien?

- Begrenzte Kapazität des Hauptspeichers
  - Nicht ausreichend z.B. bei Flugreservierungen, Datenbanksysteme, Videodatenverarbeitung
- Persistenz: Daten speichern nach Prozess-“Tod“
- Zugriff auf Informationen durch mehrere Prozesse
- => Drei wesentliche Anforderungen
  - Möglich, große Mengen von Informationen zu speichern
  - Daten sollen auch nach Beendigung des / der Prozesse erhalten bleiben
  - Mehrere Prozesse müssen gleichzeitig auf die Information zugreifen können

Strukturierung von Dateien

- 1. Dateien als Byte-Folge
  - Höchst flexibel, von Linux / UNIX und Windows benutzt
- 2. Sequenz von Datensätzen
  - Lesen und Schreiben von jeweils einem Datensatz
  - Geschichtlich: 80 Zeichen wegen Lochkarten
- 3. Datei ist ein Baum von Datensätzen, mit je einem **Schlüssel**
  - Basisoperation ist die Suche nach einem Datensatz zum Schlüssel



Typen von Dateien - Beispiel UNIX /1

- **Reguläre Dateien** (**regular files**)
  - Unstrukturierte Bytefolgen - Programme oder Daten
- **Verzeichnisse** (**directories**) oder **Ordner** (**folders**)
  - Virtuelle „Schublade“ mit einer Liste von Dateien/Ordnern
- **Benannte(s) Pipe**
  - Verbindung zwischen Prozessen
  - Schon bei IPC behandelt
- **Sockets** (**socket devices**)
  - Spezielle Dateien für Prozesskommunikation
  - Basisbaustein der Netzwerkkommunikation

Typen von Dateien - Beispiel UNIX /2

- **Virtuelle** Gerätedateien
  - Keine echten Geräte, sondern Routinen im BS-Kern
  - Z.B. **/dev/random** - produziert Zufallszahlen
- **Gerätedateien** (**device files**) - /dev/…
  - Interfaces zu den Geräten „verkleidet“ als Dateien
  - **Blockorientierte** Geräte (**block devices**)
    - Übertragen ganze Blöcke
    - Erlauben **wahlfreien Zugriff** (**random access**)
  - **Zeichenorientierte** Geräte (**character devices**)
    - Übertragen nur ein Zeichen auf einmal, oft ungepuffert
    - z.B. serielle / parallele Schnittstelle, UBS-Geräte

Dateioperationen – welche brauchen wir?

- **Create, Delete, Open, Close** - klar
- **Read** / **Write**
  - Lesen / Schreiben der Daten von der aktuellen / an die aktuelle Position; Puffer muss zur Verfügung stehen
- **Append**
  - Spezialfall von Write: Daten ans Ende der Datei anfügen
- **Seek**
  - Bei Dateien mit wahlfreien Zugriff (random access) positioniert einen „Zeiger“ an die gewünschte Stelle in der Datei
  - Anschließend wird read oder write ausgeführt
- **Get Attributes** / **Set Attributes**, **Rename** - klar

Beispielprogramm - eine Datei kopieren

```
/* Dateikopierprogramm. Fehlerbehandlung und -bericht sind minimal */
#include <sys/types.h>          /* Header-Dateien */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
int main(int, char*[]);         /* ANSI-Prototyp */

#define BUF_SIZE 4096           /* Buffer size of 4096 Byte */
#define OUTPUT_MODE 0700        /* Output permission bits: Link */

int main(int argc, char *argv[])
    int in_fd, out_fd, rd_count, wt_count;
    char buffer[BUF_SIZE];
    if (argc != 3) exit(1);                         /* Check the input “argc” ungleich 3 */

    /* Open input file and create output file */
    in_fd = open (argv[1], 0_ RDONLY);              /* Open source */
    if (in_fd < 0) exit(2);                 /* Finish in case of open fails*/
    out_fd = creat (argv[2], OUTPUT_MODE);  /* Create output file */
    if (out_fd < 0) exit(3);                /* Finish if reading fails */
```

Beispielprogramm - eine Datei kopieren

```
    /* Kopierschleife*/
    while (1) {
        /* Datenblock lesen */
        rd_count = read (in_fd, buffer, BUF_SIZE);
        if (rd_count <= 0) break;          /* Fehler oder Ende ? */
        wt_count = write (out_fd, buffer, rd_count);      /* Schreibe */
        if (wt_count <= 0) exit(4);          /* Fehler? */
    }
    /* Schließe Dateien*/
    close (in_fd);
    close (out_fd);
    if (rd_count == 0)
        exit(0);
    else
        exit(5);
}
```

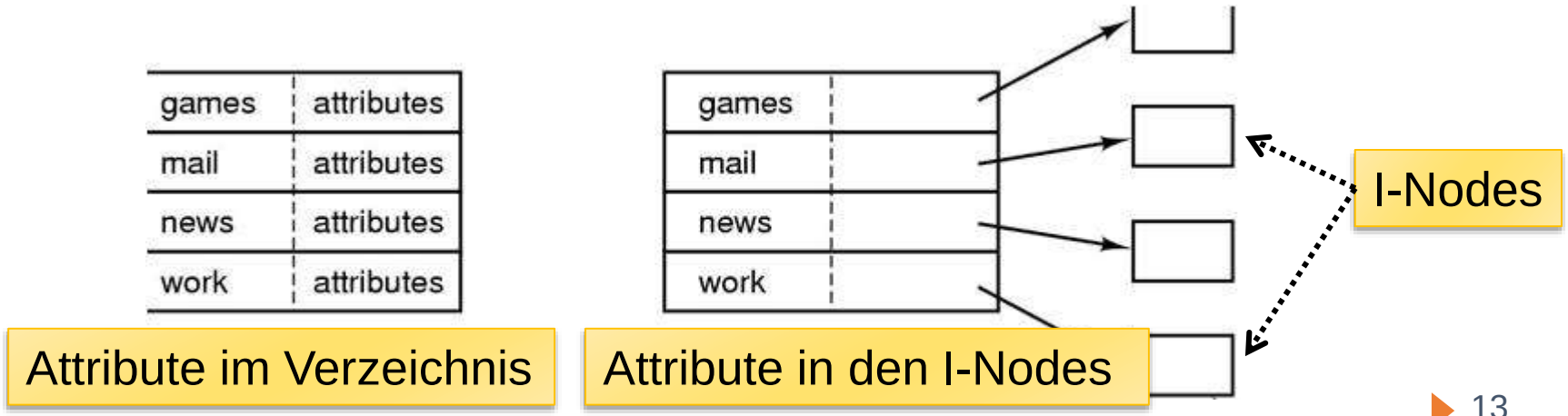
Dateiattribute - Metadaten einer Datei

- Zugriffsrechte — Wer kann wie auf die Datei zugreifen?
- Passwort — Passwort für den Zugriff auf die Datei
- Urheber — ID der Person, die die Datei erzeugt hat
- Eigentümer — Aktueller Eigentümer
- Read-only-Flag — 0: Lesen/Schreiben; 1: nur Lesen
- Hidden-Flag — 0: normal; 1: in Listen nicht sichtbar
- System-Flag — 0: normale Datei; 1: Systemdatei
- Archiv-Flag — 0: wurde gesichert; 1: wird noch gesichert
- Random-Access — 0: nur sequenzieller Zugriff; 1: wahlfreier Zugriff
- Erstellungszeit — Datum und Zeitpunkt der Dateierstellung
- Aktuelle Größe — Anzahl der Bytes in der Datei

- Zeitpunkt des letzten Zugriffs
- Zeitpunkt der letzten Änderung

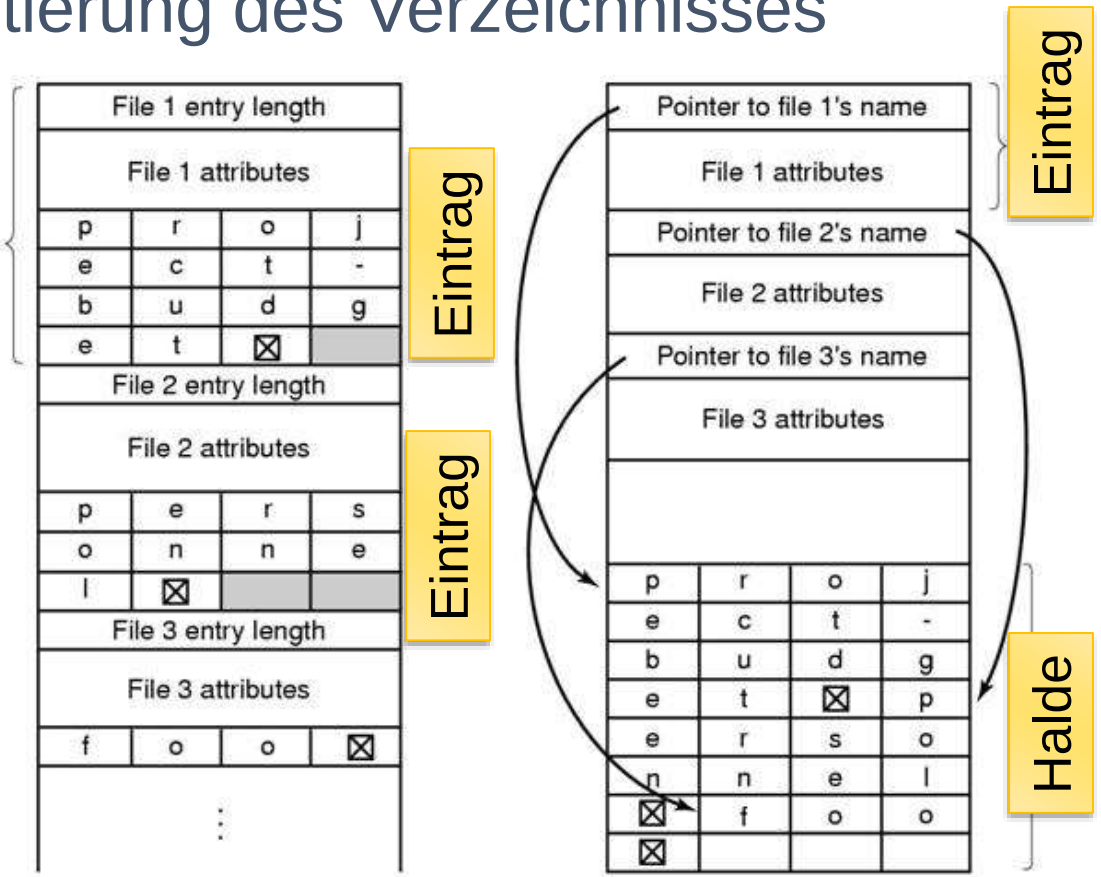
Verzeichnisse

- Listen von Dateinamen bzw. Verzeichnisnamen
  - Jeder Eintrag enthält neben Namen die logische Adresse des sog. **I-Nodes** (=Metadaten über die Speicherorte der Daten auf dem Datenträger)
- Hauptprobleme:
  - 1. Wie speichert man Dateinamen variabler Länge?
  - 2. Wie sucht man effizient den Eintrag zu einem Namen?



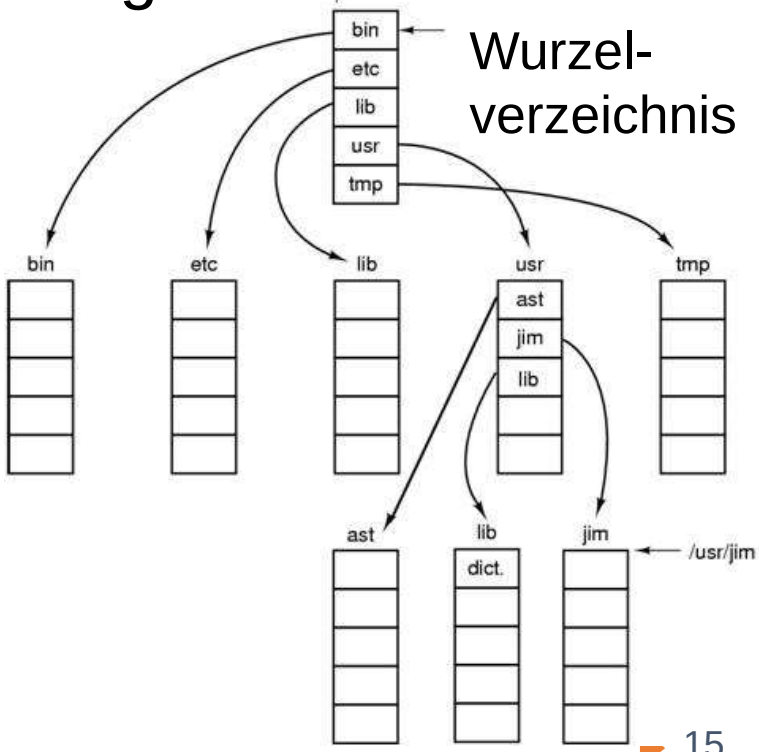
Verzeichnisse - Namen variabler Länge

- A. Jeder Eintrag hat eine variable Länge, erstes Feld speichert die Länge
  - Problem: Fragmentierung des Verzeichnisses
- B. Zeiger fester Länge, aber Namen werden separat in einer Halde (Heap) gespeichert, am Ende des Verzeichnisses
  - Halde muss auch verwaltet werden, ggf. verdichtet



Verzeichnisse und Pfadnamen

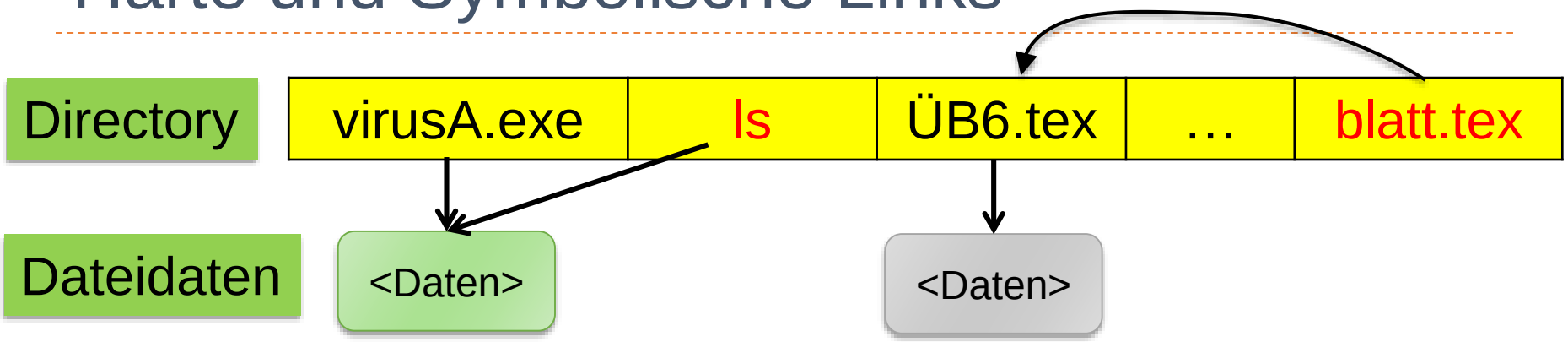
- Hierarchische Verzeichnisse erleichtern die Organisation
  - **Wurzelverzeichnis** (**root directory**) beinhaltet alle Dateien und Verzeichnisse
- **Pfadnamen** (**paths**) zur Benennung der Dateien
  - **Absoluter** Pfadname: gesamter Pfad von Wurzel zur Datei
  - **Relativer** Pfadname: Pfad vom aktuellen Verzeichnis zur Datei
- Jeder Eintrag kann eine reguläre Datei oder wieder ein Directory beschreiben => **hierarchisches Dateiensystem**



Links

- Ein **Link** ist ein Verweis auf eine Datei oder ein Verzeichnis im Dateisystem eines Computers
- Wie unterscheiden sich **Verknüpfungen** („Links“) von Windows und Links bei Unix / Linux?
- Win: Verknüpfungen (*.lnk)
  - Dateien, die nur den Pfadnamen der Zieldatei enthalten
  - Intrasparent: Ein Programm, das auf die Verknüpfung zugreift, behandelt diese anders als die Zieldatei
- UNIX / Linux: Links
  - Links sind transparent: greift eine Anwendung auf eine Verknüpfung zu, wird ihr vom BS stattdessen das Ziel der Verknüpfung geliefert
  - Neuere Windows haben sie auch (siehe mklink)

Harte und Symbolische Links



- Beide: ls und blatt.tex sind Links
- Von welchem Typ sind sie - hart oder symbolisch?
- Harter Link
  - Alle Verzeichniseinträge auf Daten sind identisch
  - Erst wenn der letzte V-E gelöscht wird, sind Daten weg
- Symbolischer Link
  - Ein Verweis auf ein Verzeichniseintrag; Interpretation durch BS nötig

Heutige Dateisysteme (DS)

- Die meisten BS unterstützten mehrere DS - Link
- Windows NT, XP, …
  - **FAT**, **FAT32**, **NTFS**, als auch CD-ROM, DVD, und Floppy-Disk-Dateisysteme
- UNIX
  - **UNIX file system** (**UFS**), basiert auf **Berkeley Fast File System** (**FFS**)
- Linux unterstützt über 40 verschiedene DS
  - Wichtigstes ist **extended file system** (**ext2**, **ext3**, **ext4**)
  - Interessant: **FUSE** - Dateisystem im Benutzerraum

Kurze Geschichte der Dateisysteme /1

Dateisystem	Urheber	Jahr	BS
DECtape	DEC	1964	PDP-6 Monitor
DOS (GEC)	GEC	1973	Core Operating System
CP/M file system	Gary Kildall	1974	CP/M
FAT12	Microsoft	1977	Microsoft Disk BASIC
DOS 3.x	Apple Computer	1978	Apple DOS
V7FS	Bell Labs	1979	Version 7 Unix
FFS	Kirk McKusick	1983	4.2BSD
FAT16	Microsoft	1987	MS-DOS 3.31
HPFS	IBM & Microsoft	1988	OS/2
ISO 9660:1988	Ecma International, Microsoft	1988	MS-DOS, Mac OS, AmigaOS
NTFS Version 1.0	Microsoft, T. Miller, G. Kimura	1993	Windows NT 3.1
ext2	Rémy Card	1993	Linux, Hurd

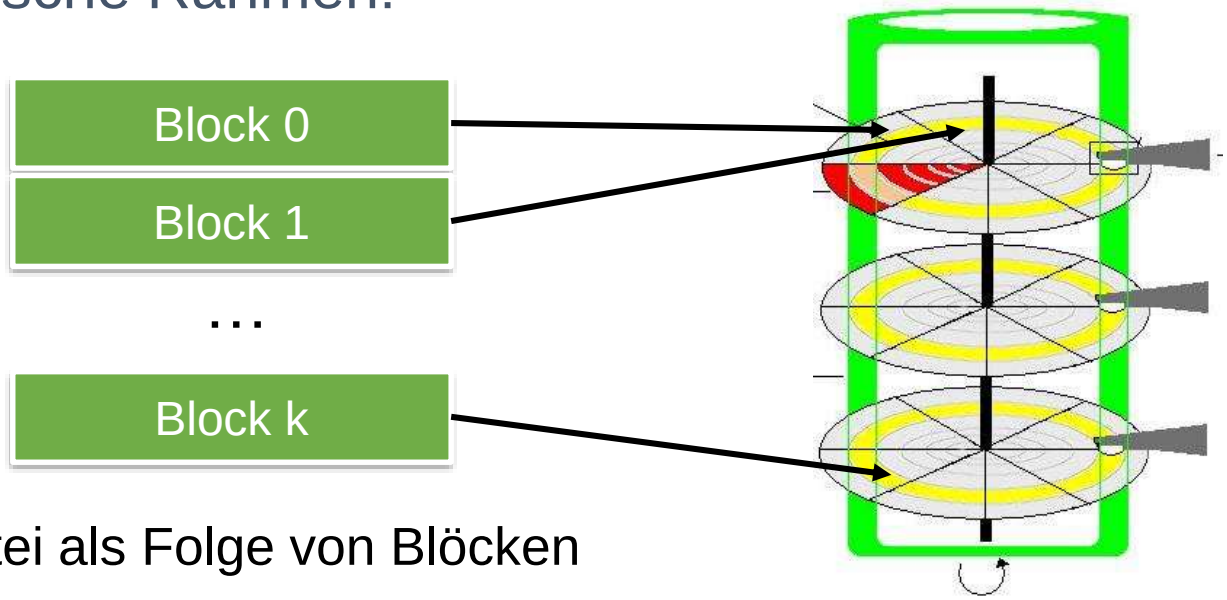
Kurze Geschichte der Dateisysteme /2

Dateisystem	Urheber	Jahr	BS
Joliet ("CDFS")	Microsoft	1995	viele
FAT32	Microsoft	1996	Windows 95b[3]
GFS	Sistina (Red Hat)	2000	Linux
NTFS Version 5.1	Microsoft	2001	Windows XP
ReiserFS	Namesys	2001	Linux
Lustre	Sun Microsystems/Cluster FS	2002	Linux
Google File System	Google	2003	Linux
Reiser4	Namesys	2004	Linux
Minix V3 FS	Andrew S. Tanenbaum	2005	MINIX 3
GFS2	Red Hat	2006	Linux
ext4	various	2006	Linux
NTFS Version 6.0	Microsoft	2006	Windows Vista

Speicherung von Dateien als Blocksequenzen

Speicherung von Dateien auf Festplatte

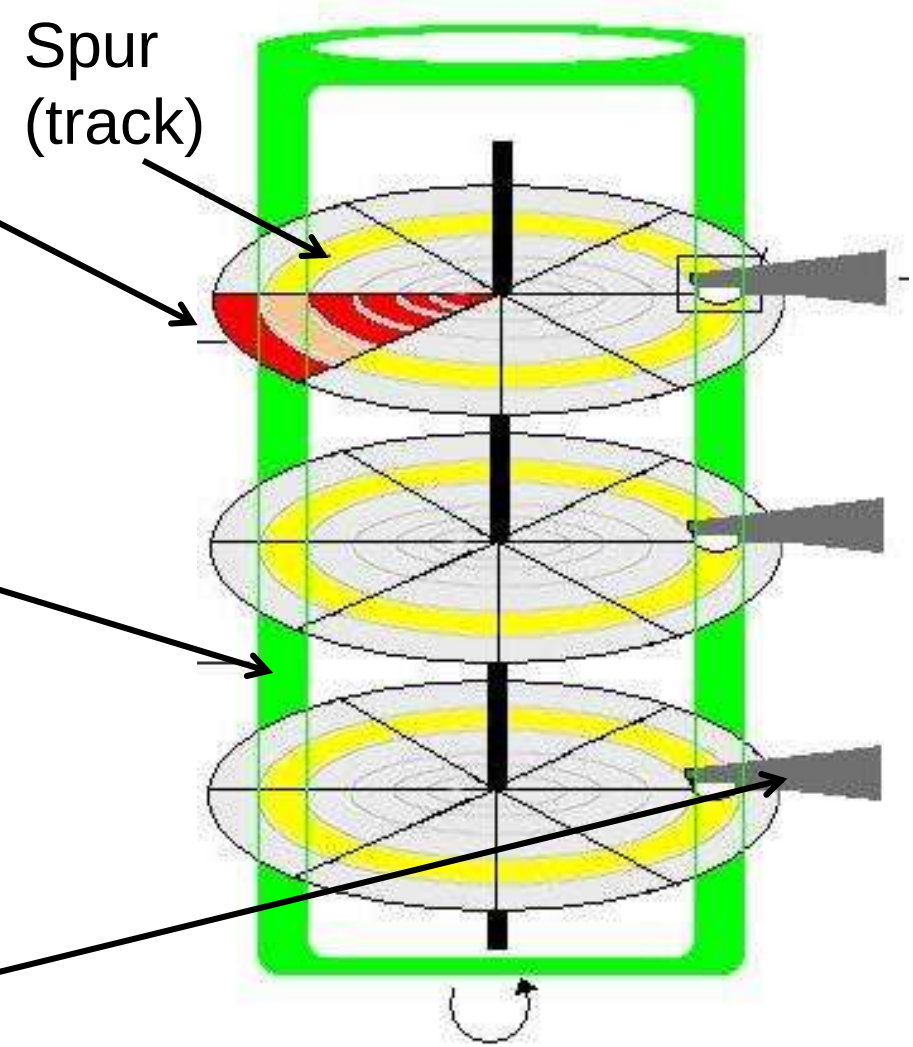
- Eine reguläre Datei wird auf einer Festplatte als eine Sequenz von (physischen) Blöcken gespeichert
  - Blöcke haben oft Größen 256 oder 512 Bytes
- Wie merken wir uns, in welchen FP-Blöcken eine Datei abgespeichert ist?
  - Ähnliches Problem wie Abbilden von logischen Seiten auf physische Rahmen!



Datei als Folge von Blöcken

Adressierung der Blöcke bei Rotierenden Festplatten

- **Sektor** = „Segment“ innerhalb einer Spur
- **Zylinder** = die „Ringnummer“, Abstand in Spuren von der Mitte
- **Head** = die Nummer der Oberfläche einer magnetischen Scheibe (Reihenfolge: oben, unten, oben, …)



Adressierung der Blöcke: CHS und LBL

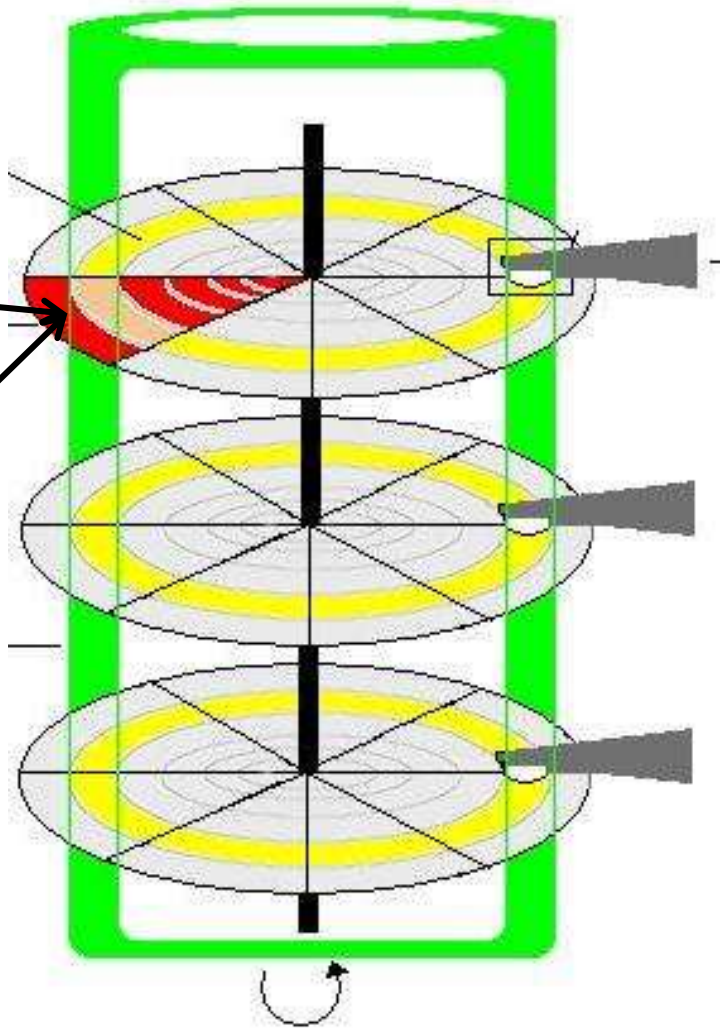
Cylinder Head Sector
(**CHS**)-Adressierung

Physische Adresse:
Zylinder 2, Kopf 0, Sektor 0

… entspricht einer …

Logischer Adresse
(dem Index): 7252

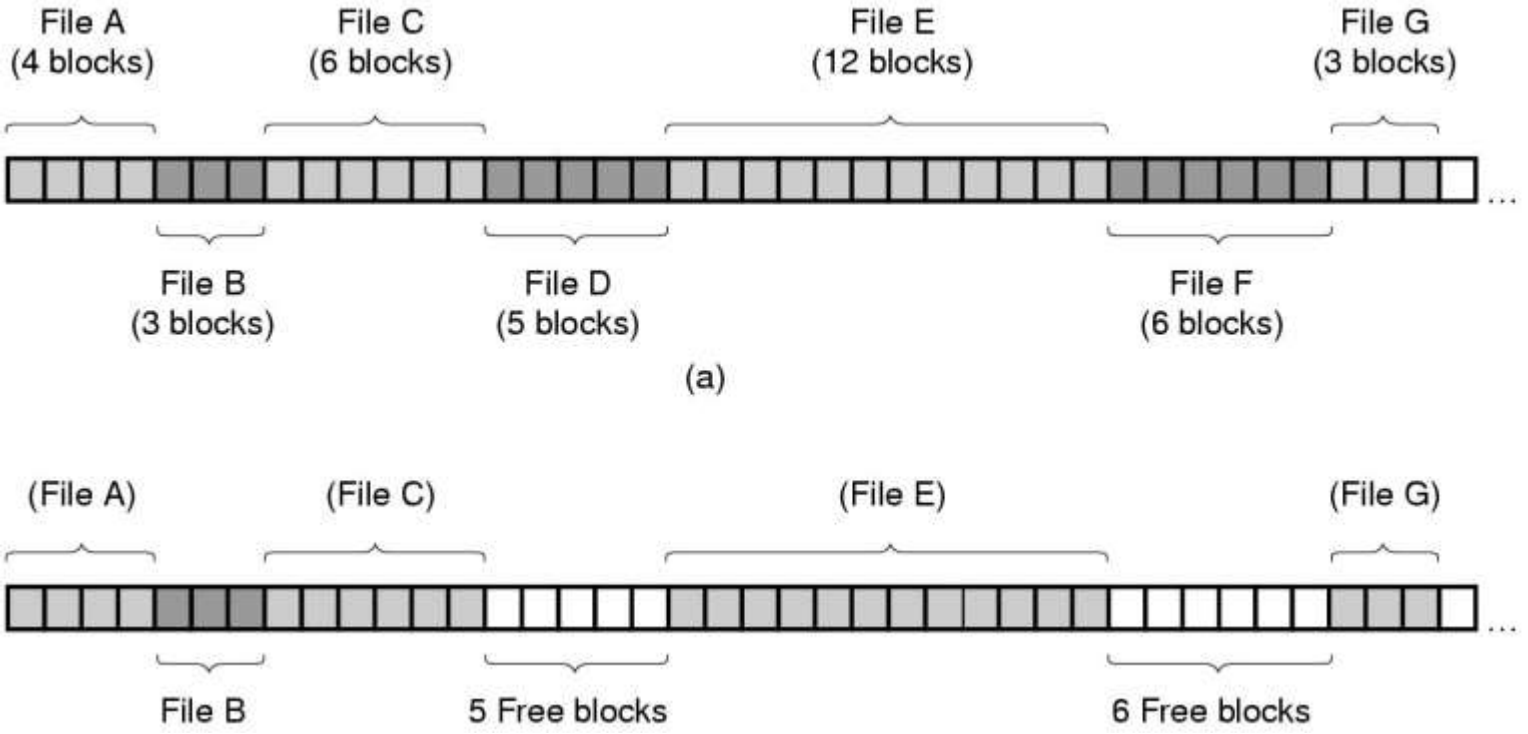
Logische Blockadressierung
(**logical block addressing**, **LBA**)



Speicherung von Dateien auf Festplatte /2

- Wie merken wir uns, in welchen FP-Blöcken eine Datei abgespeichert ist?
- Welche Eigenschaften soll eine Lösung haben?
- Schneller „random access“ zu den Inhalten der Datei (d.h. schnelle seek-Operation)
- Effizient: Wenig Overhead auf der Festplatte und in RAM
- Geringer oder keine Fragmentierung der Festplatte

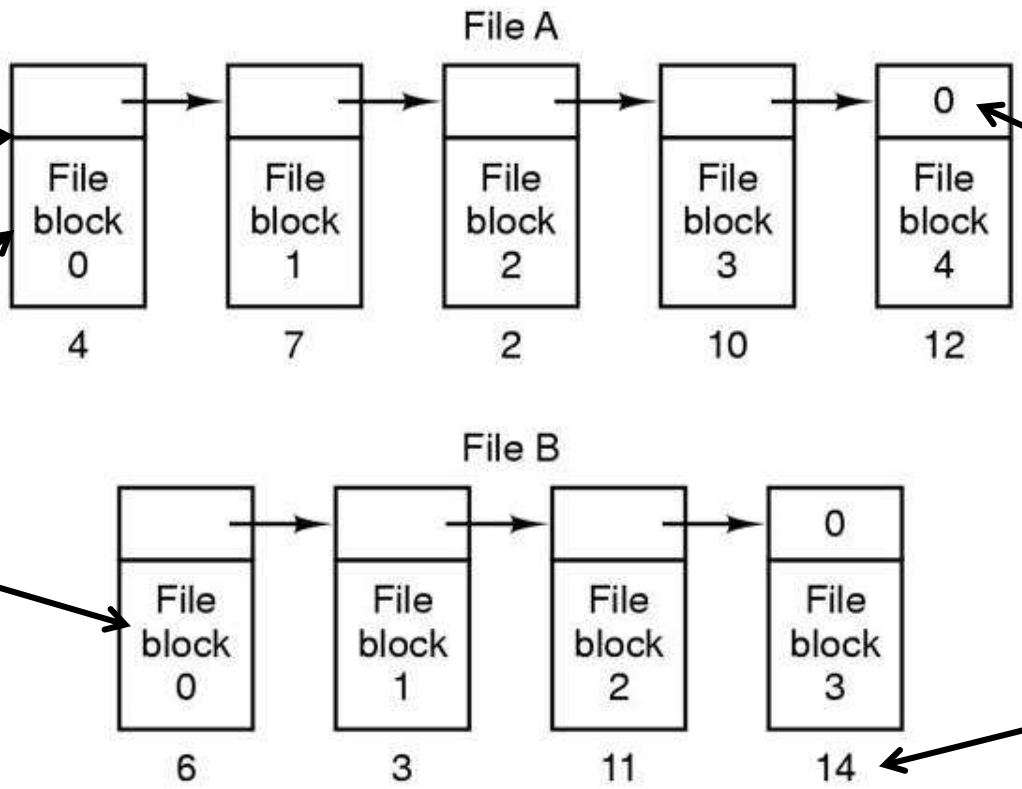
A. Zusammenhängende Belegung



- Vorteile: einfach zu implementieren; Leseleistung hervorragend, da man nur einmal Metadaten holen muss
- Nachteile: unvermeidbare Fragmentierung; Dateilänge muss bei Erstellung festgelegt werden

B. Belegung durch **Verkettete Listen**

Startblock, Adresse (4) bekannt aus dem Directory-eintrag



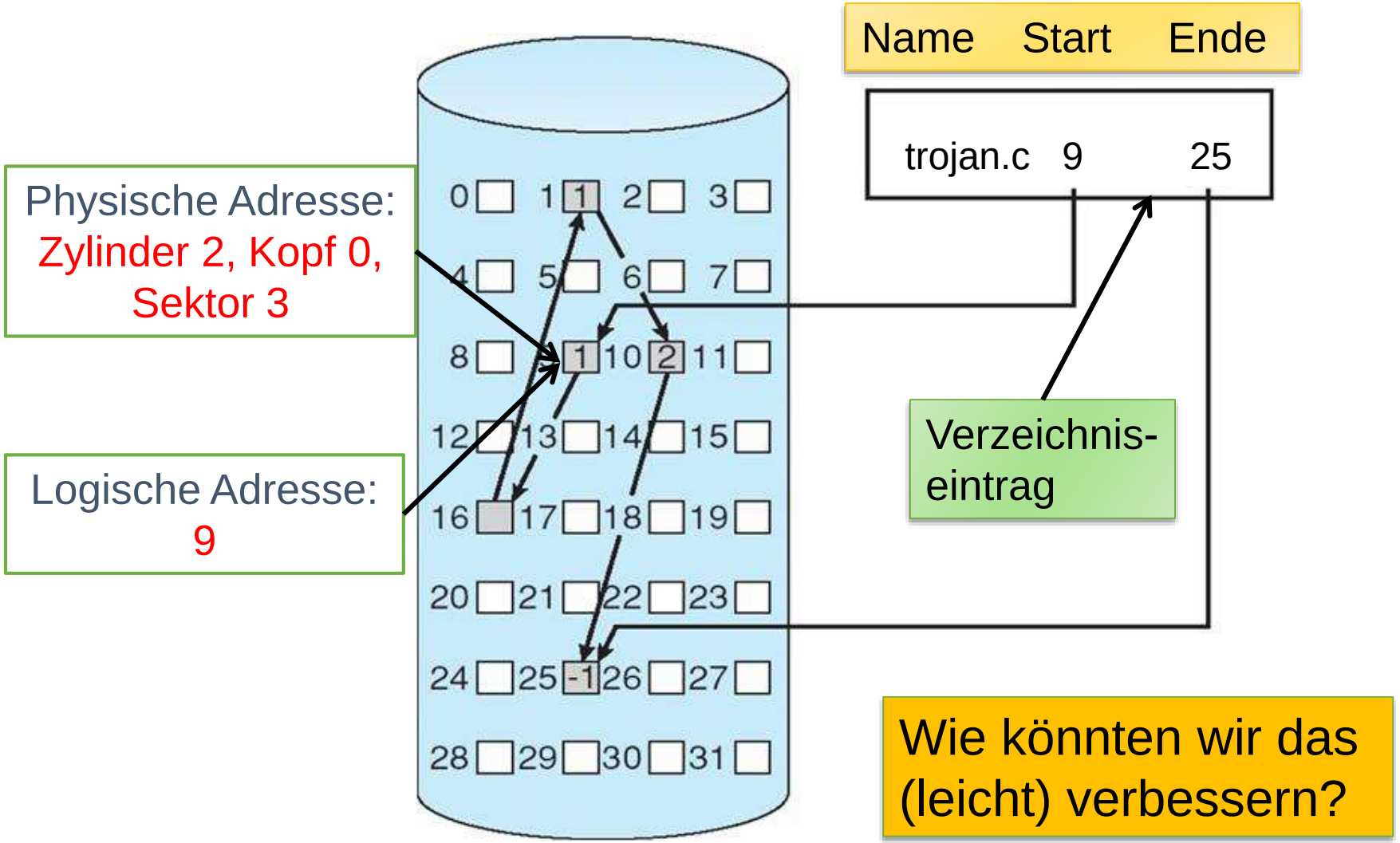
Block

Zeiger auf den nächsten Block oder 0 für Ende

Blockindex

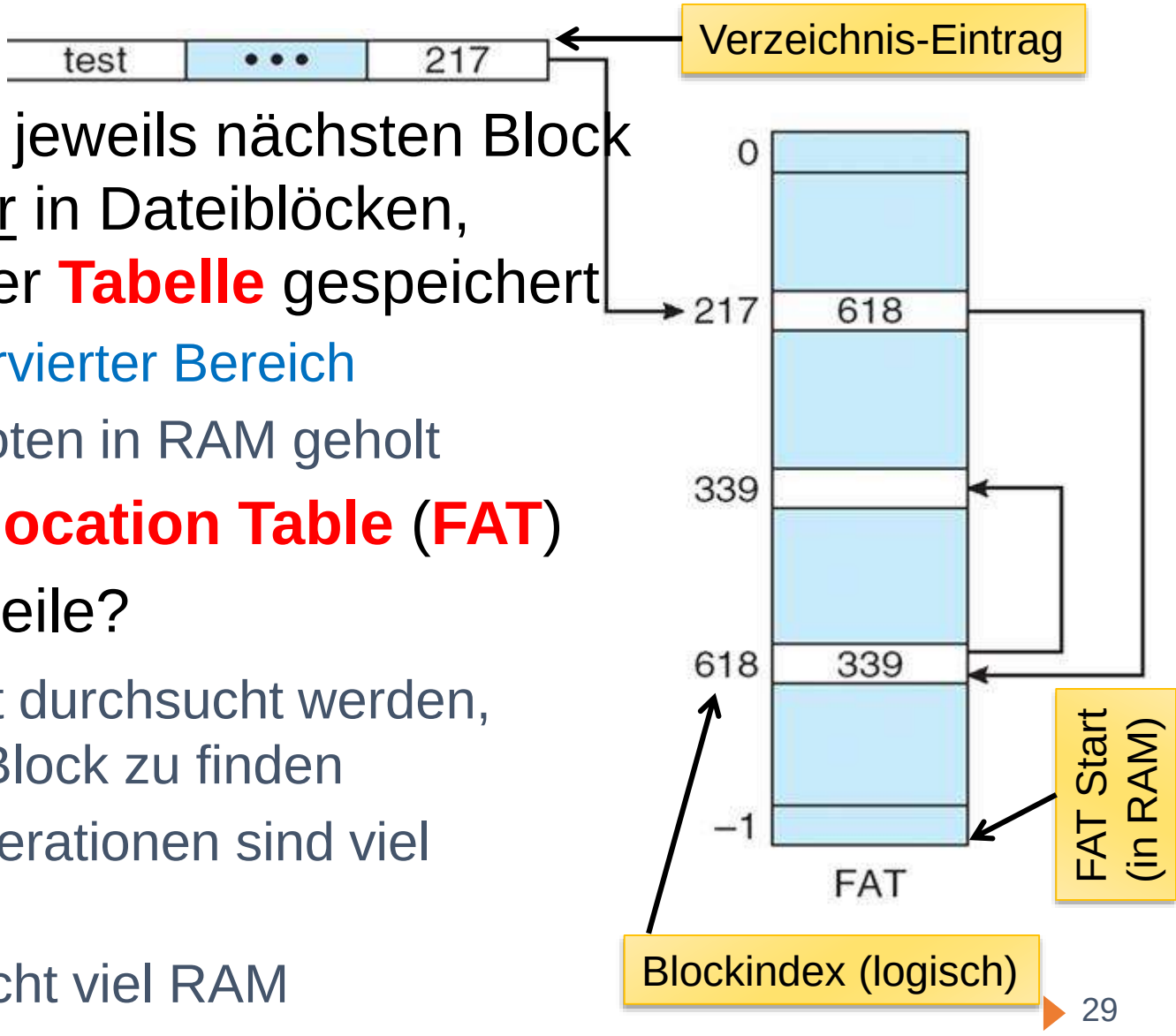
- Nachteile?
  - Wahlfreier Zugriff (random access) ist sehr langsam, da alle Blöcke der Datei (bis zur „seek“-Position) durchgegangen werden müssen
  - Kapazität eines Blocks ist keine Zweierpotenz mehr

B. Details - Verkettete Listen / Adressierung



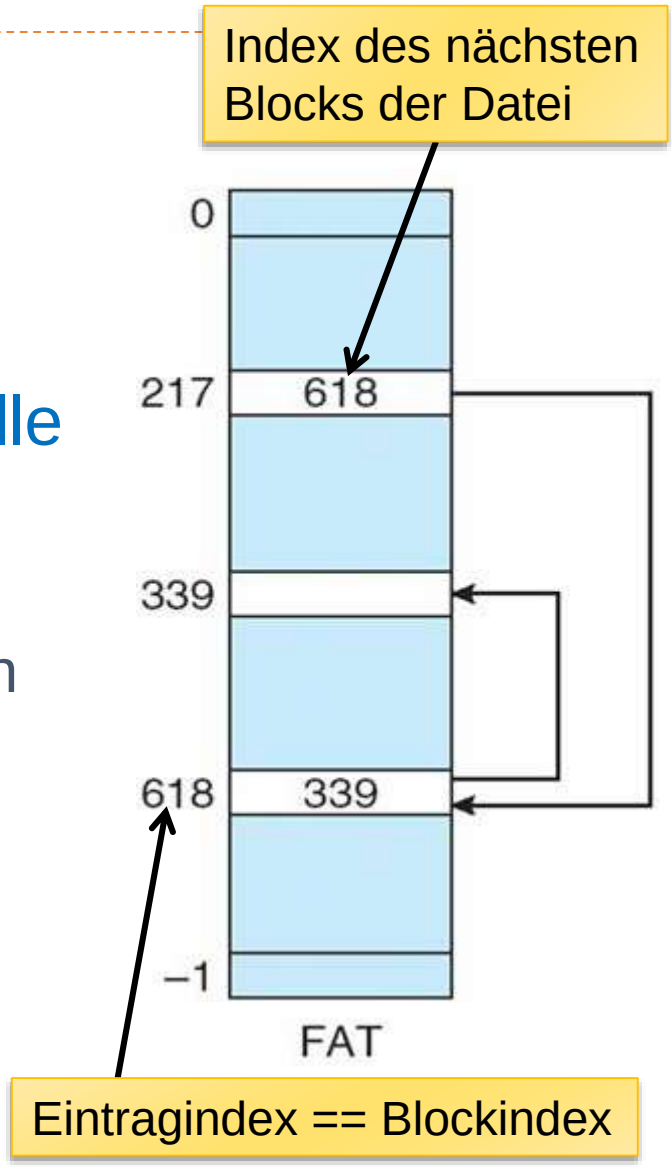
C. Eine Verbesserung - FAT

- Zeiger auf den jeweils nächsten Block wird nicht mehr in Dateiblöcken, sondern in einer **Tabelle** gespeichert
  - Auf Disk: reservierter Bereich
  - Nach dem Booten in RAM geholt
- Name: **File-Allocation Table** (**FAT**)
- Vorteile, Nachteile?
  - Sie kann leicht durchsucht werden, um ein freies Block zu finden
  - Auch seek-Operationen sind viel schneller
  - Nachteil: braucht viel RAM



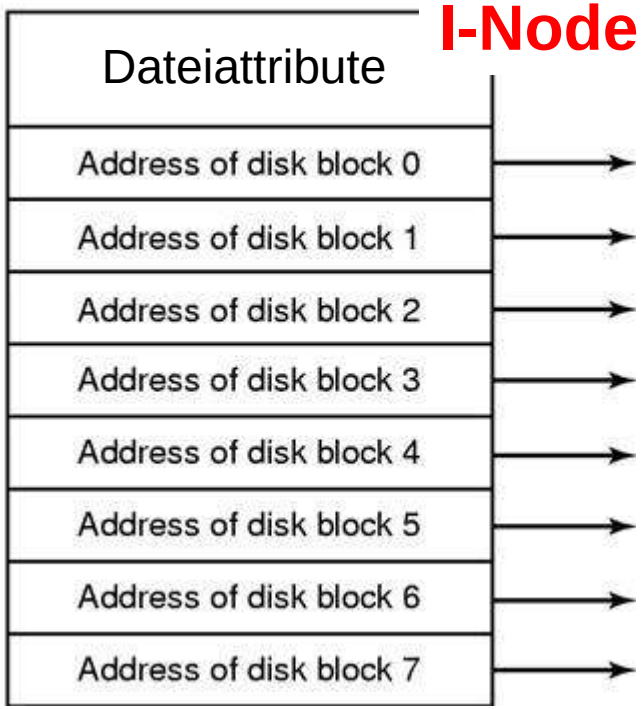
C. FAT – Ähnlichkeit?

- FAT ist entfernt ähnlich zu einem Konzept der Speicherverwaltung – zu welchem, und wie?
- Ähnlich zur invertierten Seitentabelle
- Ein FAT-Eintrag entspricht einem eindeutigen physischen Block
  - Bei invert. Seitentabelle entspricht ein Eintrag einem physischen Rahmen
- Unterschiede:
  - FAT-Eintrag-Wert ist Index eines FP-Blocks
  - Bei invertierter Seitentabelle war das die logische Seitennummer

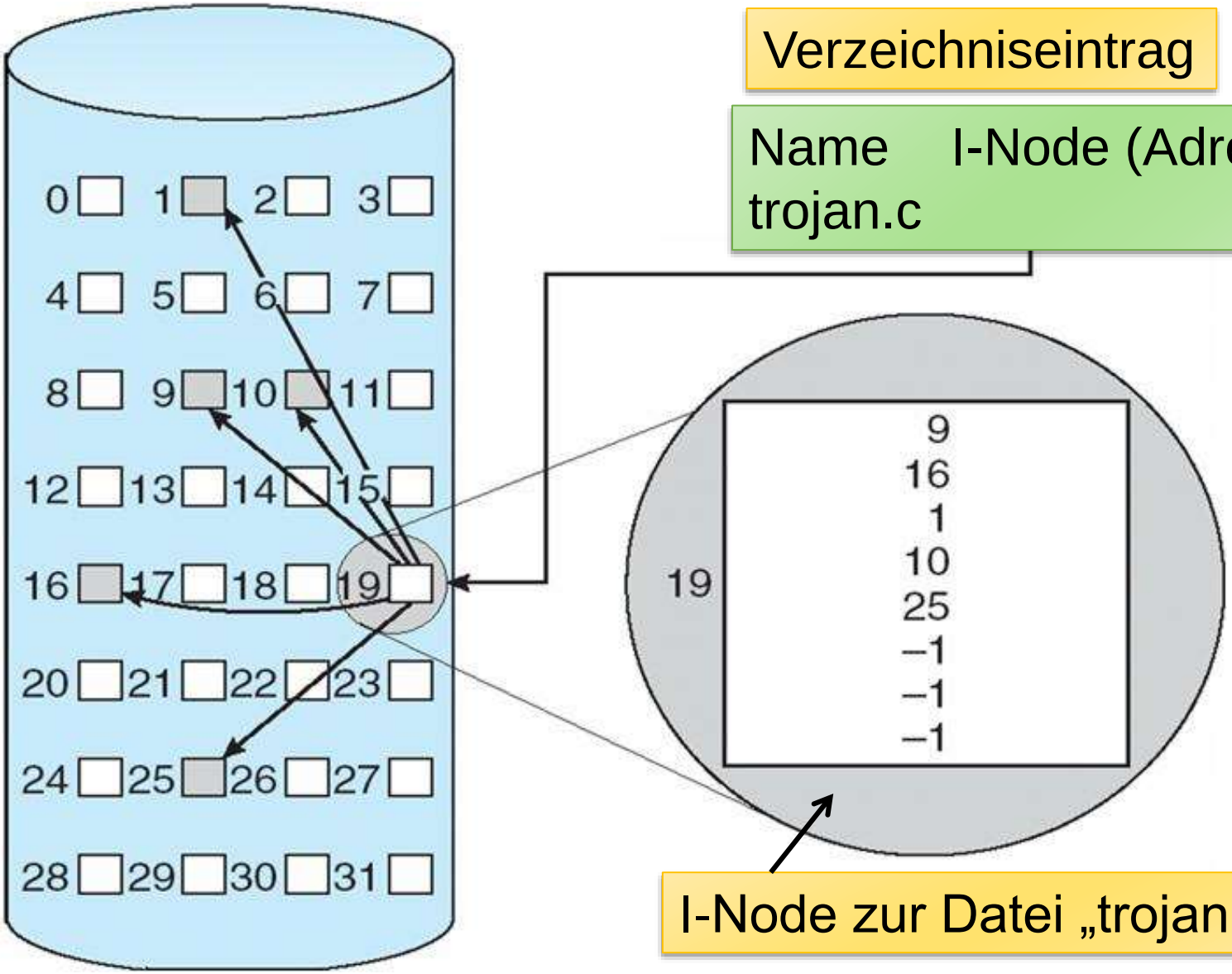


D. Weitere Verbesserung: I-Nodes

- Jede Datei erhält eine spezielle Tabelle mit der Liste der logischen Indizes ihrer Datenblöcke
- Name: **I-Node** (**inode**) bzw. der **Indexknoten**
- Gespeichert in einem Block der Festplatte
  - Kommt beim Öffnen in den RAM
- D.h. Datei hat nun 3 Teile:
- 1. Verzeichniseintrag (zeigt auf ein I-Node)
- 2. I-Node
- 3. Die eigentlichen Datenblöcke



D. I-Nodes - Beispiel



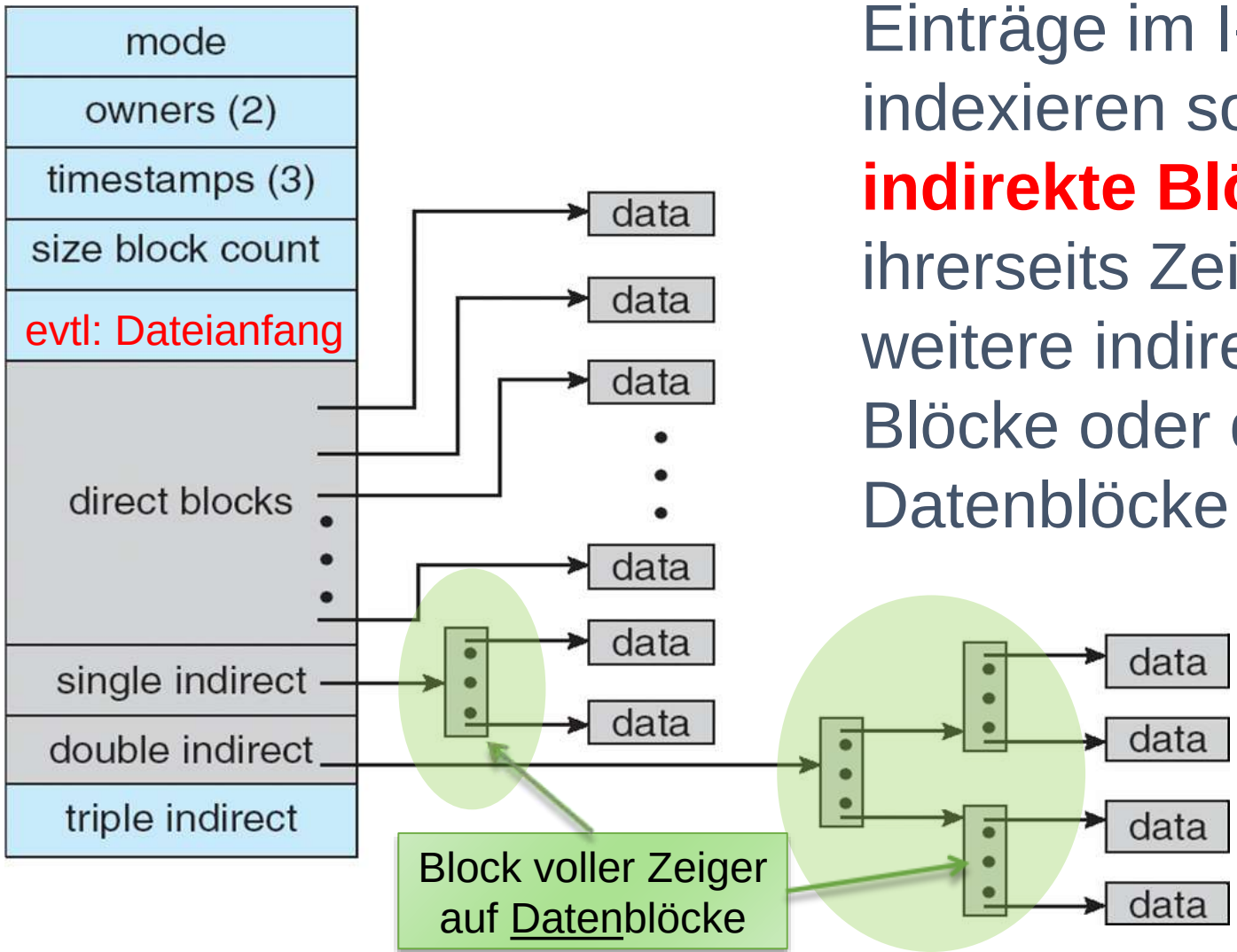
Vorteile und Nachteile

- Vorteile?
- Weniger RAM-Speicher nötig
- Platz auf Disk: proportional zur Gesamtlänge der Datei
- Platz im RAM: proportional zur Anzahl der geöffneten Dateien

- Nachteile?
- Was tun, wenn die Datei so lang ist, dass ein I-Node nicht ausreicht?
- Was könnte man machen?

Mehrstufige I-Nodes

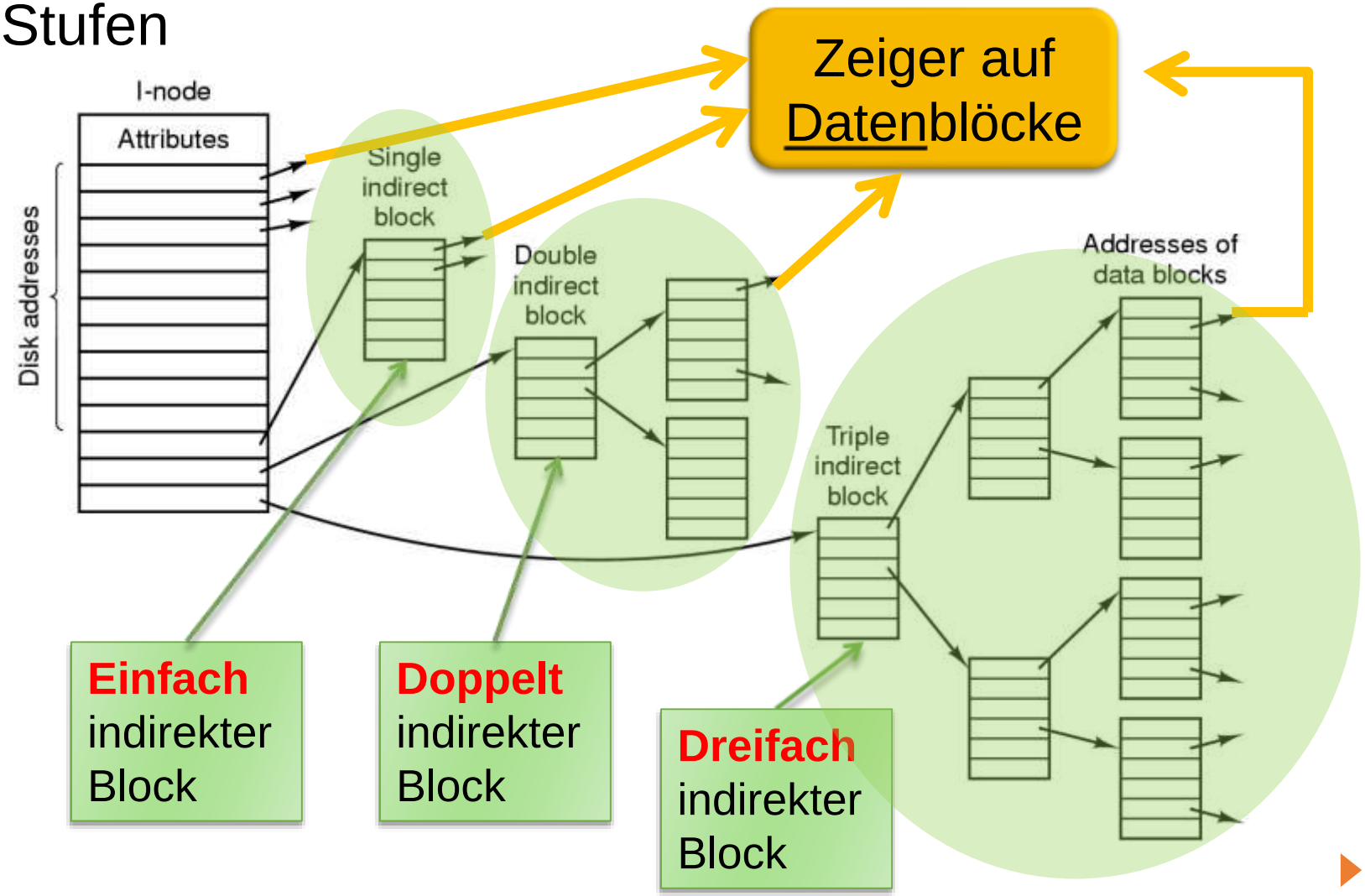
Beispiel: UNIX I-Nodes



- Lösung: Die letzten Einträge im I-Node indexieren sog. **indirekte Blöcke**, die ihrerseits Zeiger auf weitere indirekte Blöcke oder die Datenblöcke enthalten

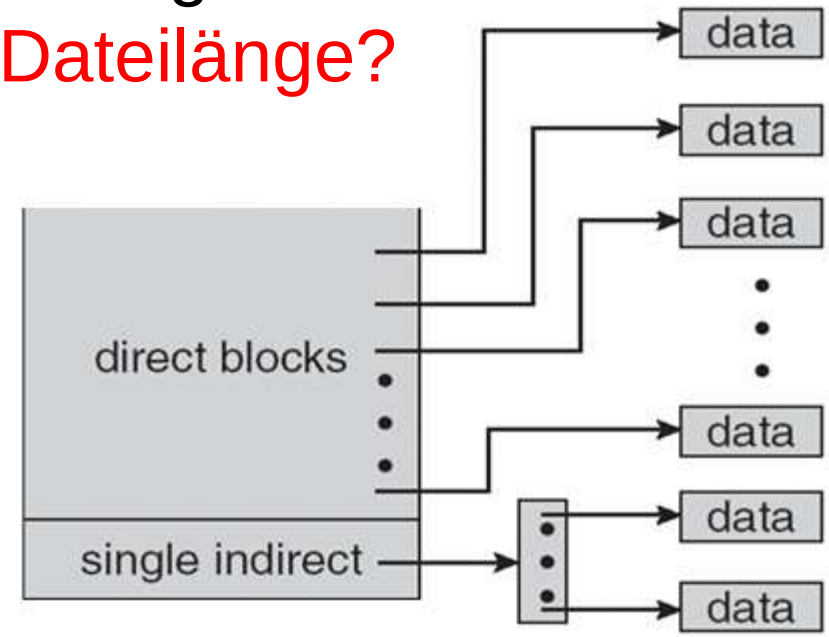
Beispiel: I-Nodes bei UNIX-V7 (Veraltet)

- Verwaltung der Dateien durch I-Nodes mit mehreren Stufen



Mehrstufige I-Nodes: Aufgabe

Ein I-Node enthält nur: n direkte Zeiger (je 4 Byte) und ein (1) „single indirect“ Zeiger (auf indirekten Block, nur mit Zeigern). Alle Blöcke: 1 kB groß. Was ist die max. Dateilänge?



- Wir haben (1024-4)/4 = 255 direkte Adressen, jede auf ein 1 kByte Block => 255 kByte
- Dazu 1 „single indirect“ Adresse auf ein Block (voll nur mit Adressen)
  - Also 1024/4 = 256 weitere Adressen => 256 kByte
- Insgesamt: 255+256 kByte = 511 kByte

Video - Inodes

- Shell-Befehle – was machen sie?
  - touch
  - ln
  - stat
  - df
- Video **LINUX Understanding inodes** von **theurbanpenguin [13a]**
  - https://www.youtube.com/watch?v=_6VJ8WfWI4k&list=PLqE63EN7m04eoD84hdrHK7rt9-zsZHe78&index=21
  - Ab ca. 3:35 bis ca. 7:20 (min:sec)

Speicherung von Dateien auf Festplatte

Zusammenfassung der Lösungen

1. **Zusammenhängende Belegung**
   - Einfach, aber gravierender Nachteil: Fragmentierung
2. **Belegung durch verkettete Listen** (Linked List Allocation), gespeichert in den Datenblöcken
3. **File Allocation Table** (**FAT**): Verkettete Listen, gespeichert in einer einzigen Tabelle (auf Disk/RAM)
   - Alle Listen für alle Dateien zusammen
4. **I-Nodes** (**Indexknoten**)
   - Datenblock/Blöcke mit der verketteten Listen, separat pro Datei

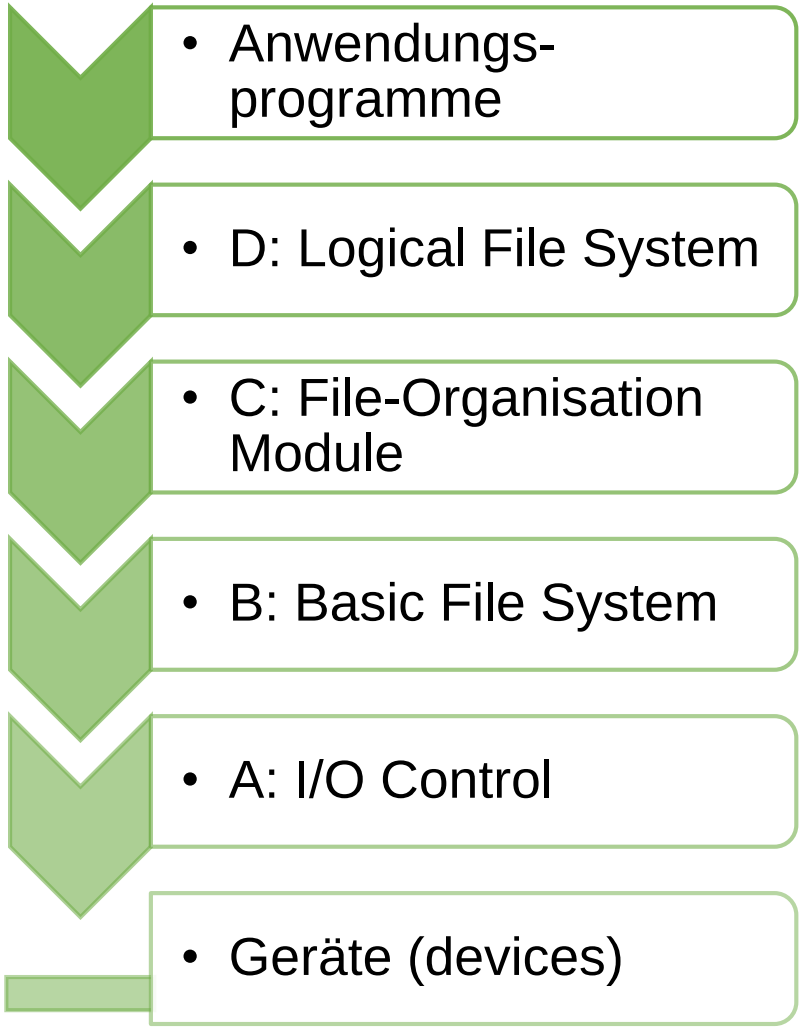
Zusammenfassung

- Dateien
  - Aufgaben, Attribute, Operationen, Dateien kopieren
  - Verzeichnisse, Links
- Implementierung von Dateisystemen
  - CHS und LBL – Adressierung; Schichtenaufbau
- Speicherung von Dateien als Blocksequenzen
  - Zusammenhängende Belegung, Verkettete Listen, FAT, Inodes
- Quellen: Silberschatz et al. Kap. 11+12; Tanenbaum Kap. 4, 11, 10; Wikipedia

Zusätzliche Folien: Ergänzungen Dateisysteme

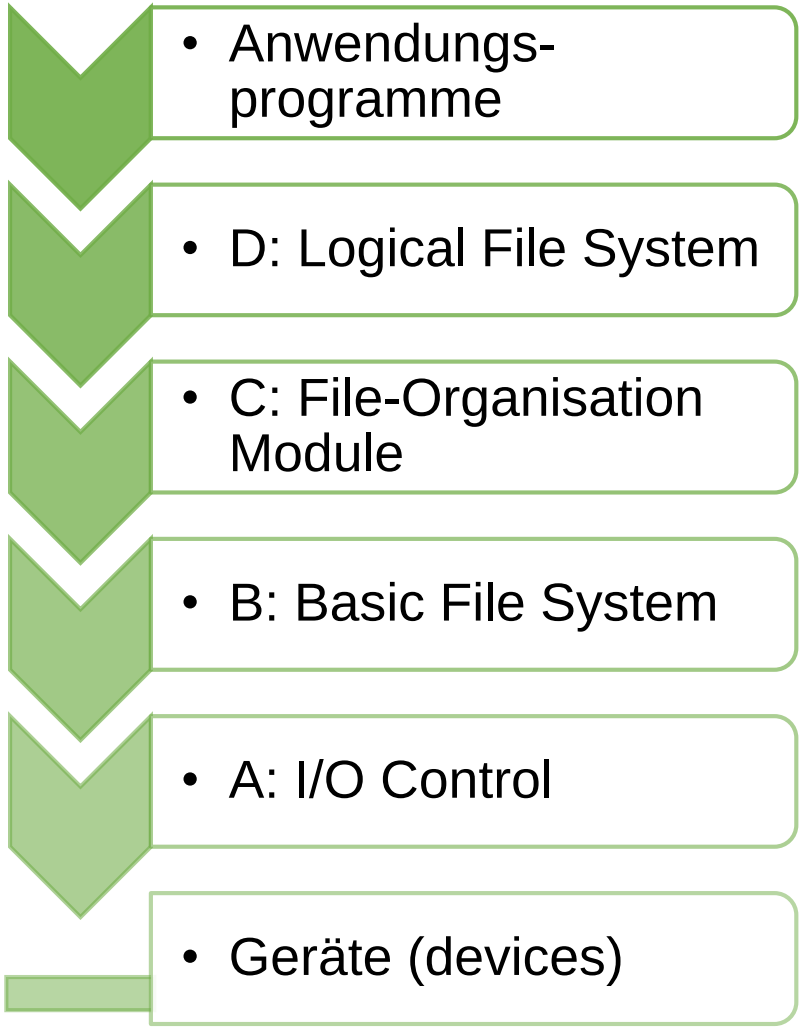
Schichtenaufbau eines Dateisystems

- **A: I/O-Control**
  - Gerätetreiber: Übersetzen Funktionsaufrufe in HW-Befehle des Controllers, z.B. „lese Block x/y/z“
- **B: Basic File System**
  - Verwaltet Puffer und Caches, aber „sieht“ nur physische Blöcke (z.B. „lese Zylinder 73, Spur 2, Sektor 5“)



Schichtenaufbau eines Dateisystems

- **C: File-Organisation Module** (Optional)
  - Kennt Dateien
  - Übersetzt Adressen von logischen Blöcken (0 bis N) in Adressen von physischen Blöcken
- **D: Logical File System**
  - Verwaltet Metadaten, inklusive Verzeichnisdaten
  - Manipuliert **File-Control Blocks** mit Informationen wie (logische) Datenadresse, Rechte, Eigentümer, …



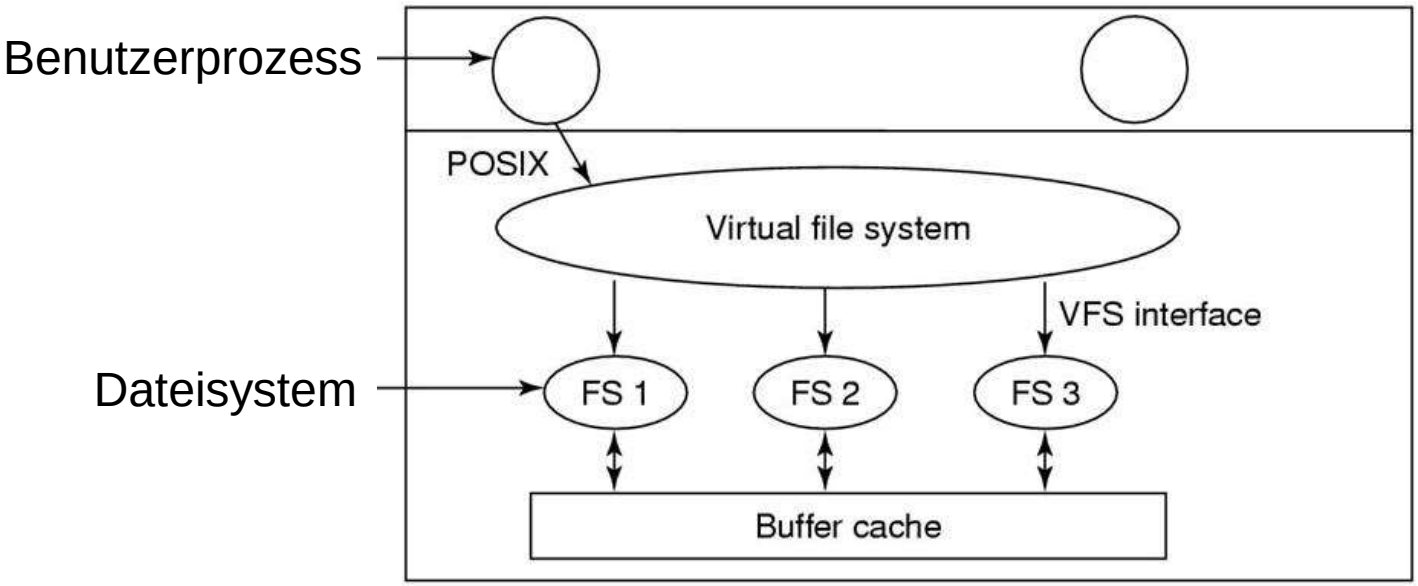
Zusätzliche Folien: Ergänzungen Dateisysteme

Verzeichnisse - Effizientes Suchen

- Problem „Gegeben ein Schlüssel (hier: Name), wie finde ich den Wert dazu (hier: Eintrag)?“
  - Fundamental in Informatik, tritt immer wieder auf
- Spezielle Datenstrukturen dafür vorhanden
  - **Dictionaries** bzw. **Hashtabellen**
- In Java: HashMap oder TreeMap
  - HashMap: Zugriff O(1), aber keine Ordnung
  - TreeMap: Zugriff O(log(# Einträge)), aber mit Ordnung
- Alternative bei Verzeichnissen
  - Lineare Liste, aber mit Caching der Suchergebnisse

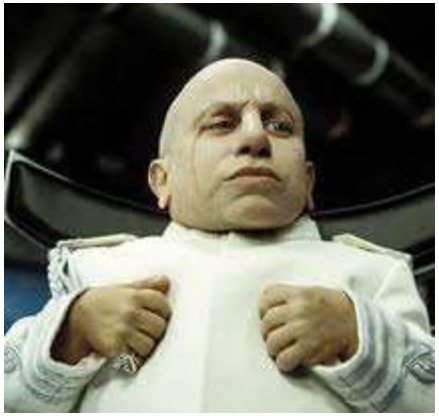
Virtuelle Dateisysteme

- **Virtuelle Dateisysteme** (**Virtual File Systems** (**VFS**)) ist eine Objektorientierte Umsetzung von Dateisystemen
  - Entwickelt von Sun Microsystems um 1986
- VFS bietet die gleiche API für verschiedene Arten von Dateisystemen
  - Der Benutzer kann auf verschiedene Dateisysteme (ext3, ReiserFS, …) uniform und transparent zugreifen



Virtuelles Dateisystem in Linux

- Linux VFS hat vier Haupttypen von Objekten
- D.h. Objekte in BS, die die API zur Datenmanipulation realisieren
  - **inode** object: repräsentiert eine individuelle Datei
  - **file** object: repräsentiert eine geöffnete Datei
  - **superblock** object: steht für das gesamte Dateisystem
  - **dentry** object: repräsentiert ein Verzeichniseintrag
- Jeder dieser Objekttypen gibt es mehrere Methoden
- Z.B. file object hat Methoden
  - int **open**(…) - Datei eröffnen
  - ssize_t **read**(…) / ssize_t **write**(…) - lesen und schreiben
  - int **mmap**(…) - Memory-mapping einer Datei (einblenden in den logischen Adressraum)



Zusätzliche Folien: Windows Dateisystem NTFS

NTFS - New Technology File System (Link)

- Entwickelt als Nachfolge von FAT32 Dateisystem, Alternative zu OS/2-Dateisystem HPFS
  - *„Während der größte Teil von NT auf dem Festland entwickelt wurde, ist NTFS unter den Komponenten des BS in der Hinsicht einmalig, dass vieles seines ursprünglichen Entwurfs auf einem Segelboot auf dem Puget Sound stattfand (streng nach dem Protokoll „Arbeit am Vormittag - Bier am Nachmittag“)“*

Tanenbaum, Seite 1041

- Hauptunterschiede zu FAT32
  - Max. Dateigröße von theoretisch (16 Exbibyte (EiB) - 1KB)
    - Implementation: 16 TB minus 64 KB
  - Zugriffsschutz auf Dateiebene durch **Access Control Lists**
  - Größere Datensicherheit durch **Journaling**
    - Erhält Konsistenz des Dateisystems bei Fehlern / Abstürzen

Weitere Eigenschaften von NTFS

- Namen und Verzeichnisse
  - Dateinamen bis zu 255 Zeichen
  - Pfadnamen bis 32767 Zeichen
  - Unicode-Unterstützung für Namen
- Unterstützung für Dateien mit geringer Datendichte
  - D.h. Abschnitte mit 0en werden automatisch komprimiert
- Transparente Kompression
- Verschlüsselung
- Fehlertoleranz
- Harte Links
- Symbolische Links ab Windows Vista (NT 6.0)

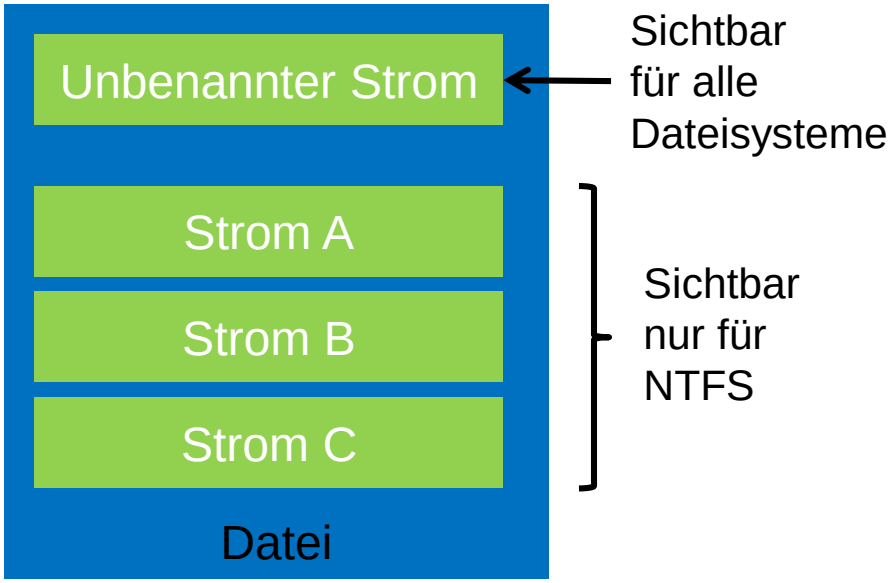
Links in NTFS

- Harte Links (Link)
  - Es sind bis zu 1023 pro Datei möglich
- Erzeugen unter Windows XP mit fsutil
  - fsutil hardlink create “new-link.txt“ “existiting-name.txt“
- Erzeugen ab Windows Vista mit mklink
  - mklink /H “new-link.txt“ “existiting-name.txt“
- Symbolische Links (Link) ab Windows NT 6.0
  - Admin-Recht Create Symbolic Link nötig
  - mklink “new-link.txt“ “existiting-name.txt“
  - mklink /d „new-dir-link" "..\up\existing-dir" (relativer Pfad)
  - mklink /d „new-dir-link" "\\other-host\abs-path-to-dir"

NTFS-Byteströme

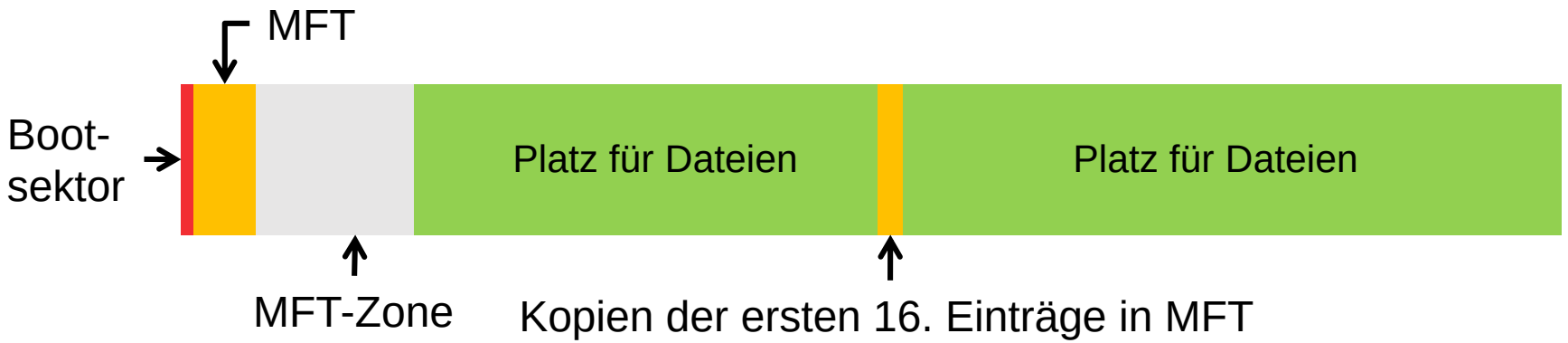
- Eine NTFS-Datei ist nicht nur eine Folge von Bytes
  - Sie ist wie ein Objekt mit mehreren **Attributen**
  - Jedes Attribut ist durch einen **Bytestrom** repräsentiert
  - Die nicht-Standard Ströme sind zerbrechlich, da viele Programme sie ignorieren (beim Kopieren / Übertragung)

- Typische Ströme
  - der **unbenannte Strom** - die eigentlichen Daten
  - Name
  - 64-Bit Objekt-ID
  - Metadaten, z.B. Bildvorschau von Bildern



Struktur einer NTFS-Partition (**Volume**)

- Jede NTFS-Partition (Teil der Festplatte) enthält eine lineare Folge von Blöcken (**Cluster** in MS-Sprache)
  - Blockgröße von 512 Byte bis 64 KB, meistens 4 KB
  - Blöcke werden durch einen 64-Bit-Offset zu Beginn der Partition referenziert
- Eine NTFS-Partition hat nach dem Bootsektor die **MFT-Zone** für die Datei **Master File Table** (**MFT**)
  - Meist 12.5% der Partitionsgröße, kann bis 50% erreichen



Master File Table, MFT - Struktur

- MFT ist eine Datei und kann auch woanders als in MFT-Zone stehen, falls Blöcke defekt sind
- Sie enthält **Einträge** (oder **Datensätze** - **records**)
  - Zwischen 512 Bytes und 4 KB groß, meist 1 KB groß
  - Jeder entspricht einer Datei oder einem Verzeichnis
- Jeder Datensatz besteht aus
  - Datensatz -**Header**
  - einer Folge von Attributen
- Jedes Attribut ist wiederum ein Paar
  - (**Attribut-Header**, **Attribut-Datenstrom**)

MFT – Datensatz - Header

- MFT-Datensatz-Header enthält u.a.
  - **magische Zahl** für die Gültigkeitsprüfung
  - eine **Sequenznummer**, die jedes Mal erneuert wird, wenn der Eintrag für eine neue Datei wiederverwendet wird
  - einen **Zähler der Verweise auf diese Datei**
  - die **aktuelle Länge** des Datensatzes
  - diverse andere Felder

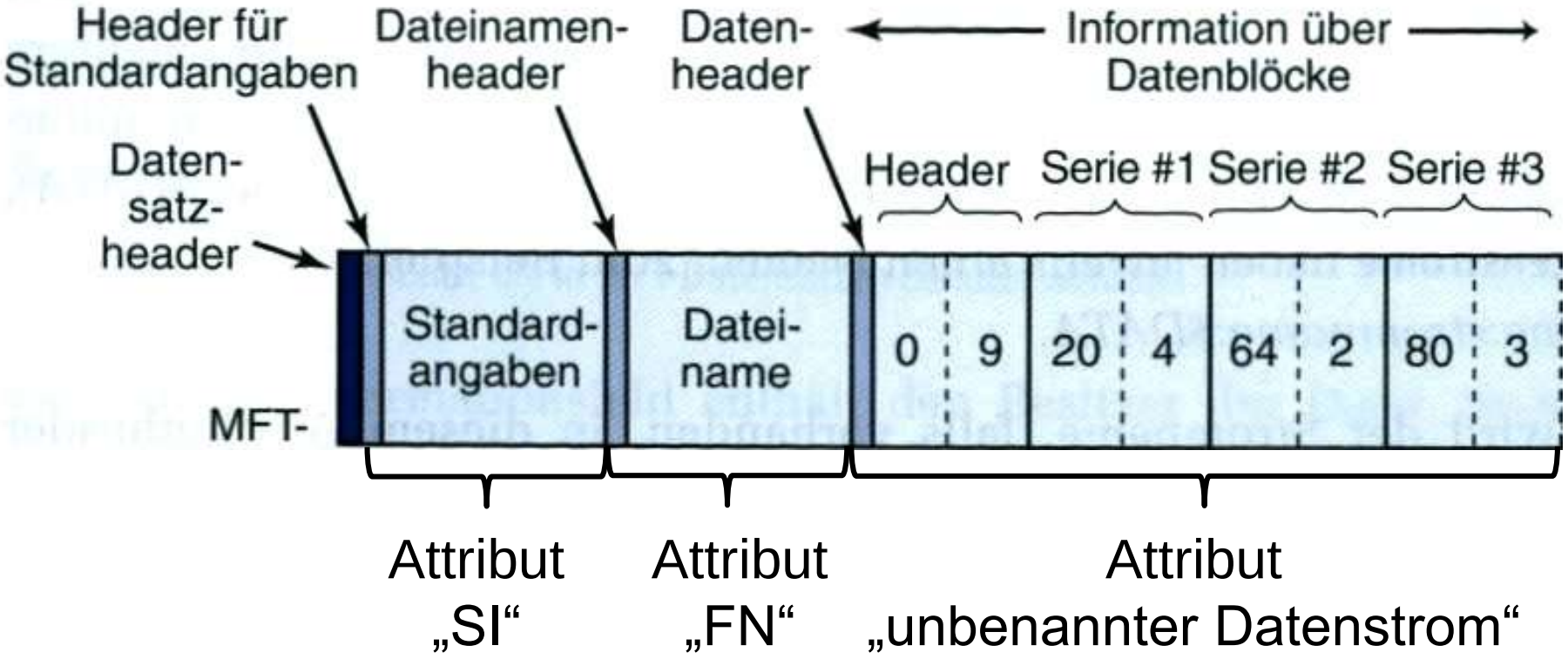
Wichtigste Attribute eines MFT-Eintrags

- Standardinformation - standard information (**SI**)
  - Besitzer der Datei, Sicherheitsinformationen, die von POSIX benötigten Zeitstempel, den Zähler der harten Links, das Read-only-Flag und das Archiv-Flag usw.
  - Hat eine feste Länge und ist immer vorhanden
- Dateiname(n) - file name (**FN**)
  - Unicode-Zeichenkette von variabler Länge, zusätzlich kann es auch noch einen 8+3-MS-DOS Namen geben
- Objekt-ID
  - Eindeutiger 64-Bit-Dateiidentifikator
- Daten - der Datenstrom; kann wiederholt werden
  - Der **Standarddatenstrom** ist unbenannt
  - **Alternative Datenströme** haben jeweils einen Namen

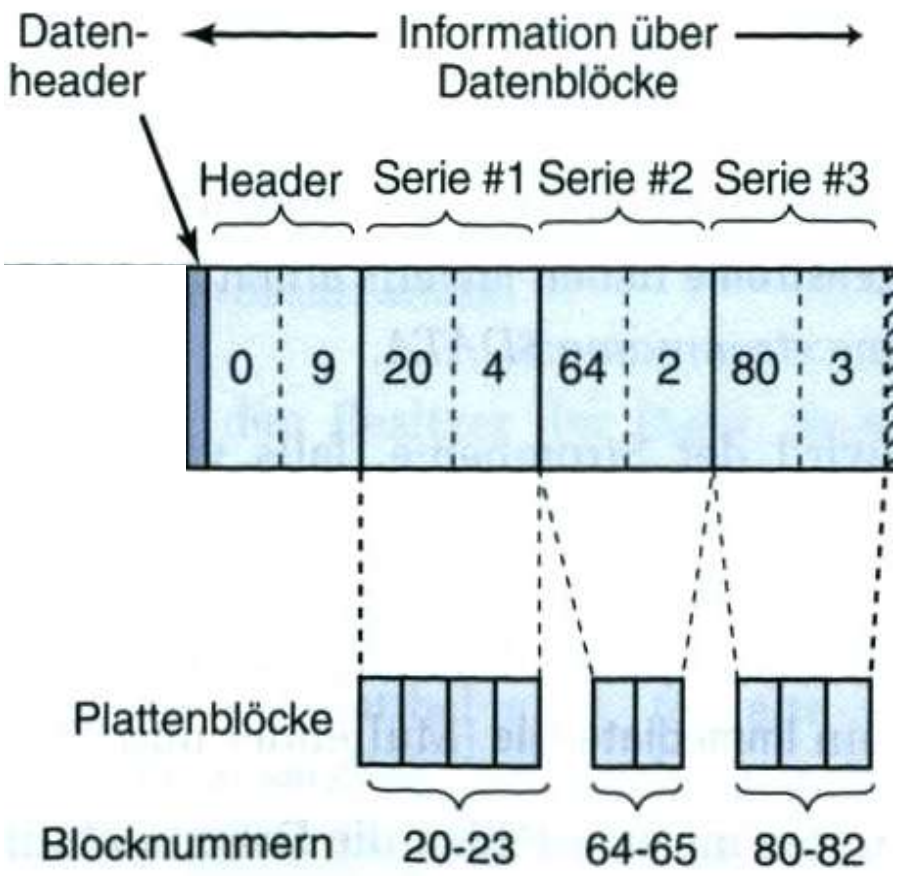
Attributstruktur

- Attributstruktur: (Attribut-Header, Attribut-Datenstrom)
  - Attribut-Header enthält ggf. einen Stromnamen
  - Attribut-Datenstrom hat zwei Varianten …
- Bei **resident attributes**
  - Stromdaten in dem MFT-Eintrag selbst gespeichert
- Bei **non-resident attributes**
  - Hier werden nach dem Attribut-Header nur die Adressen der Blöcke gespeichert, die den Strom enthalten
  - Ähnlich wie bei welchem Konzept?
- Was wird oft als ein non-resident attrib. gespeichert?
- Der unbenannte Datenstrom, d.h. eigentliche Daten

Beispiel MFT-Datensatz



Unbenannter Datenstrom - „Non-Resident“

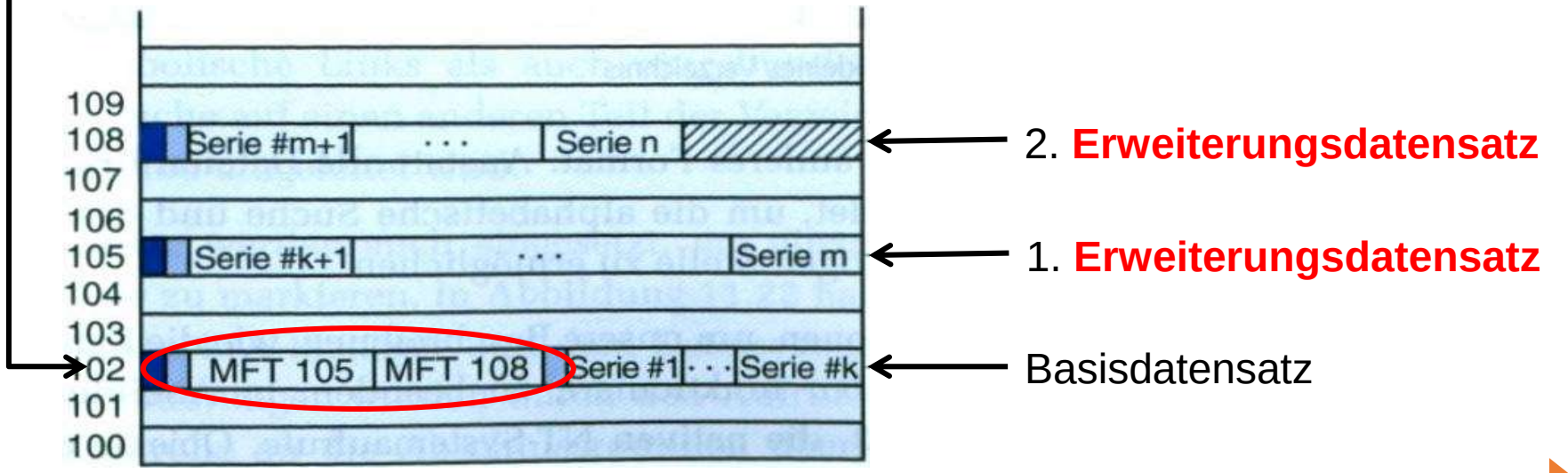


Wie viele Blöcke kann eine Serie angeben?

- Falls der unbenannte Datenstrom lang ist, wird er außerhalb der MFT gespeichert
- Zunächst ein „Header“ – (später)
- Dann eine Blockliste gespeichert als eine Folge von **Serien** (**runs**)
  - Zwei 8-Byte-Werte pro Serie: (Adresse des 1. Blocks, Länge)
  - Warum auf diese Weise?

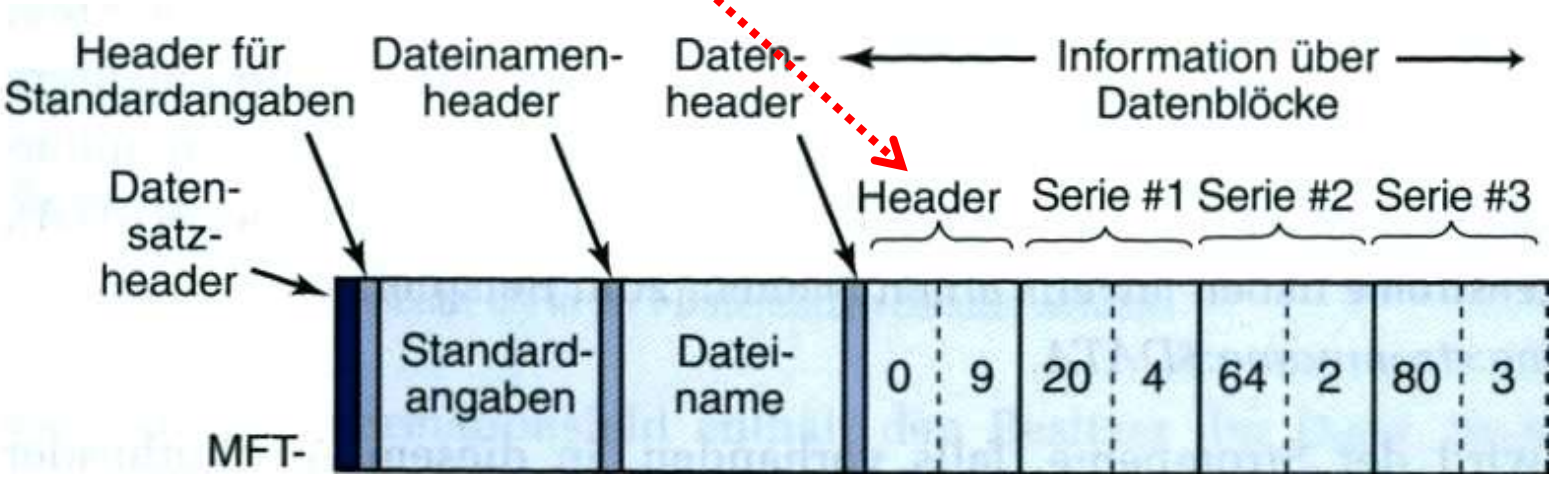
MFT-Eintrag Reicht Nicht Aus – Was Tun?

- Was tun, wenn ein Strom so viele Serien hat, dass ein MFT-Eintrag nicht ausreicht?
- Man nutzt gleiches Prinzip wie bei I-Nodes
  - Der **MFT-Basisdatensatz** enthält Adressen von weiteren MFT-Datensätzen, die die Serien eines Stroms angeben
  - Reicht auch das nicht aus, wird die Liste der Erweiterungsdatensätze nicht-resident gespeichert



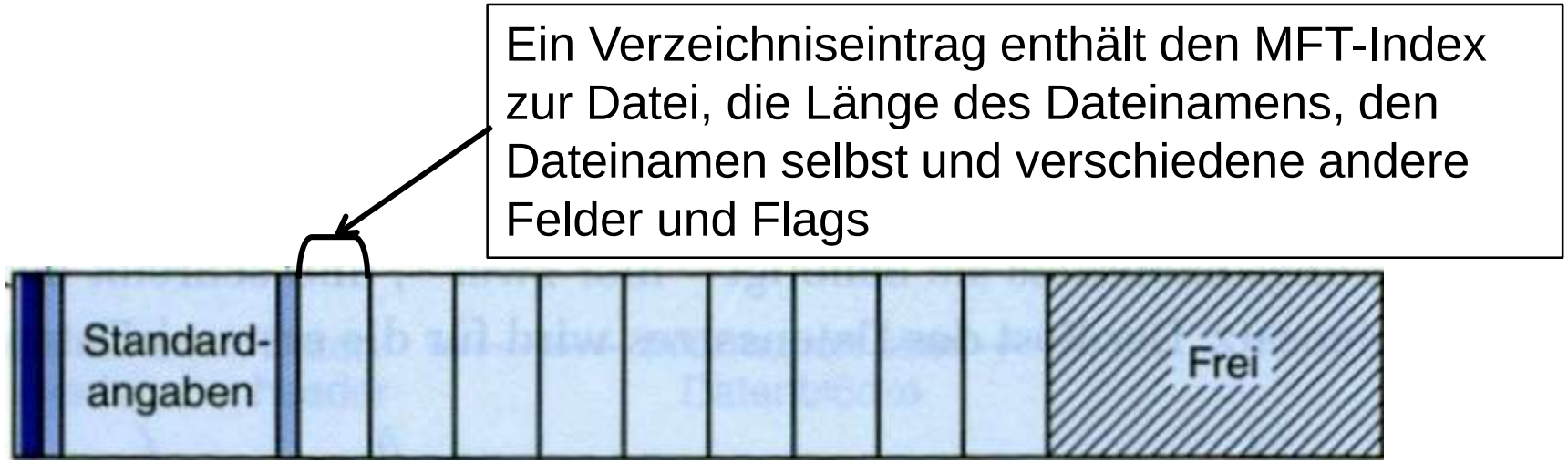
Dateien mit Geringer Dichte (sparse files)

- NTFS speichert effizient Datenströme mit „Lücken“
  - D.h. Regionen, die aus Nullen bestehen
- Hat ein Strom Lücken, wird nach jeder Lücke ein neuer Datensatz verwendet
  - Der „Header nach dem Attribut-Header“ gibt den Abschnitt des Stroms an, der im aktuellen Datensatz beschrieben wird



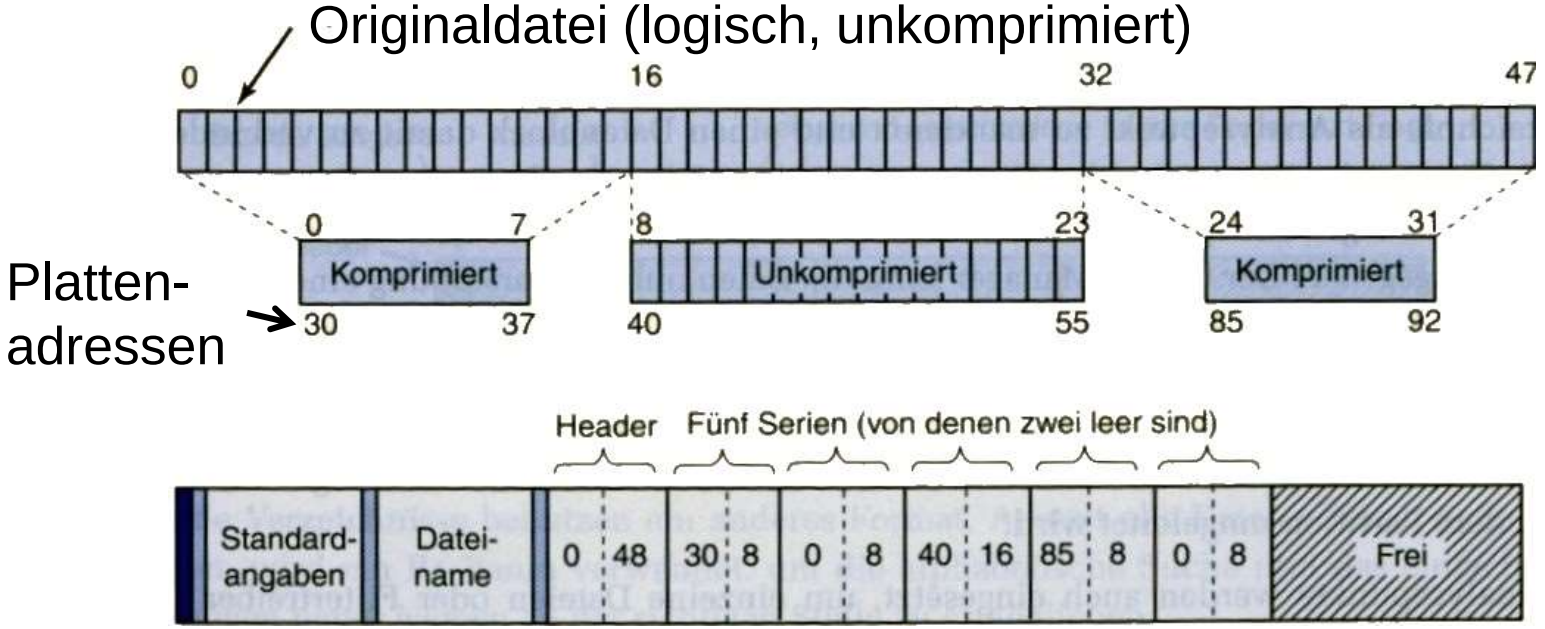
NTFS-Verzeichnisse

- Zwei Datenstrukturen für Verzeichnisse
- Lineare Liste für kleine Verzeichnisse
  - Suchen nach einem Eintrag in linearer Zeit
- Ein **B+-Baum** (Link) für große Verzeichnisse
  - Ein **Balancierter Baum** mit Modifikationen
- Beispiel: Verzeichnis als lineare Liste



Kompression in NTFS

- NTFS unterstützt **transparente Kompression**
- Bei einer zu komprimierenden Datei werden jeweils 16 aufeinanderfolgende logische Blöcke untersucht
  - Bei Ersparnis von mind. 1 Block werden komprimierte Daten auf die FP geschrieben, sonst die Originaldaten
  - Ein Ersparnis von k Blöcken wird als die Serie (0,k) markiert
- Bsp.: Blöcke 0-15 und 32-47 um jeweils 50% komprimiert



Analysepunkte (Reparse Points)

- NTFS bietet an, eine Datei oder ein Verzeichnis als **Analysepunkt** zu markieren und einen Datenblock damit zu verbinden
- Trifft man eine solche Datei oder das Verzeichnis während der Analyse eines Dateinamens, dann wird der Datenblock an den **NTFS-Objekt-Manager** zurückgegeben
  - Dieser kann die Daten als Darstellung eines alternativen Pfadnamens interpretieren und die Zeichenkette aktualisieren
  - Danach kann die Ein-/Ausgabeoperation wiederholt werden
- Nützlich, um symbolische Links als auch eingebundene Dateisysteme zu unterstützen
  - Indem die Suche auf ein anderes Verzeichnis oder eine andere Partition umgeleitet wird
- Tools (Server 2008 / Vista): Mountvol, Mklink, Fsutil

Die ersten 16 MFT-Einträge sind Metadaten

#	Name	Beschreibung
12-15		Für spätere Nutzung reserviert
11	$Extend	Erweiterungen: Kontingente etc.
10	$Upcase	Tabelle für Z-Konvertierung (klein →Unicode)
9	$Secure	Sicherheitsdeskriptoren für alle Dateien
8	$BadClus	Liste der fehlerhaften Blöcke
7	$Boot	Bootlader
6	$Bitmap	(Cluster) Bitmap der belegten Blöcke
5	$	Root Folder - Wurzelverzeichnis
4	$AttrDef	Definitionen der möglichen Attribute
3	$Volume	Partitions-Daten (Name, Version)
2	$LogFile	Logdatei für Wiederherstellung
1	$MftMirr	Spiegelkopie der MFT
0	$Mft	Masterdateitabelle selbst (MFT)

Mehr dazu hier (siehe „Metadata Files Stored in the MFT")

MFT Metadaten

- $Mft
  - Der Eintrag zu der MFT selbst; dadurch kann MFT verschoben werden
  - Nur der 1. Block von MFT muss beim Start bekannt sein, wird im Bootsektor der Partition spezifiziert
- $LogFile
  - Logdatei: Strukturelle Änderungen am Dateisystem (Hinzufügen / Löschen eines Verzeichnisses) werden zuerst hier festgehalten, ehe sie ausgeführt werden
- $AttrDef
  - Definiert 13 Attribute, die in MFT-Einträgen erlaubt sind
- $: Wurzelverzeichnis der Partition (root directory)
- $Extend
  - Informationen über Kontingente, Analysepunkte, usw.