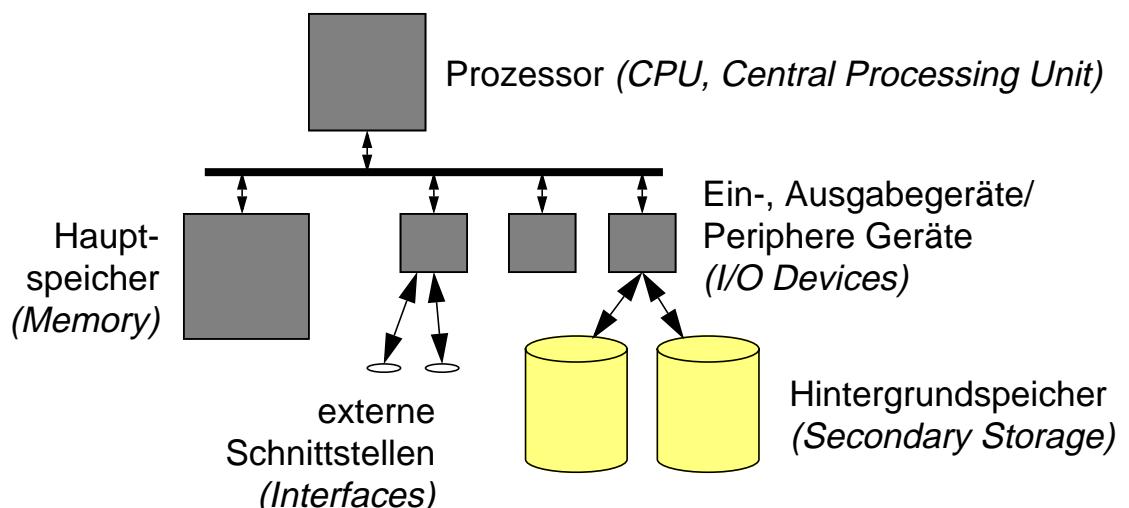


F Implementierung von Dateien

F Implementierung von Dateien

■ Einordnung

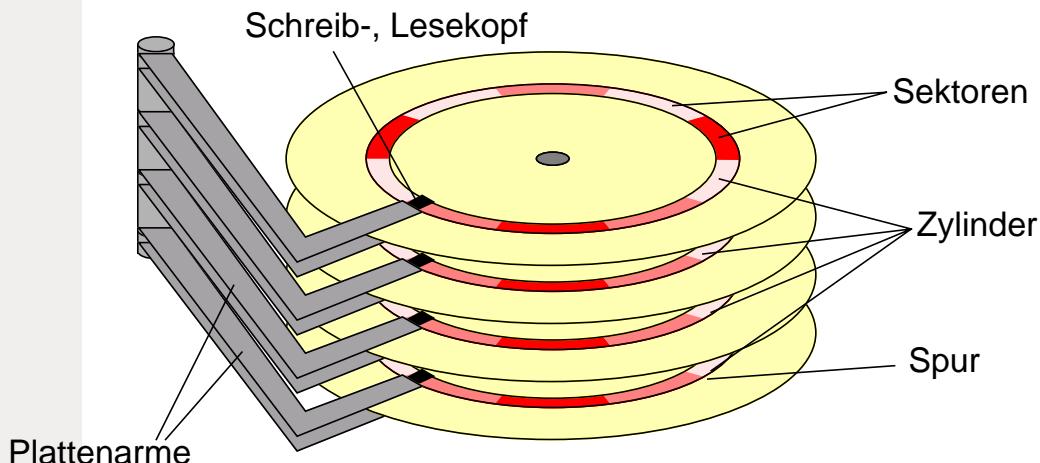


1 Medien

1.1 Festplatten

- Häufigstes Medium zum Speichern von Dateien

- ◆ Aufbau einer Festplatte



- ◆ Kopf schwiebt auf Luftpolster

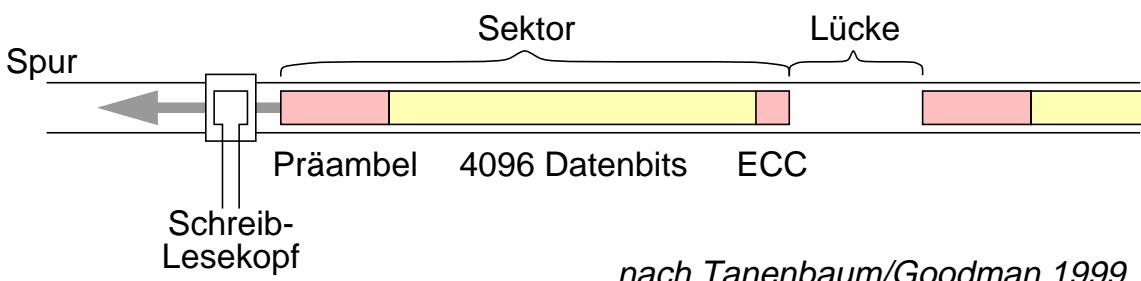
Systemprogrammierung I

© 1997-2001, Franz J. Hauck, 2002 F. Hofmann, Inf 4, Univ. Erlangen-Nürnberg[F-File.fm, 2002-12-11 11.52]
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

F – 3

1.1 Festplatten (2)

- Sektoraufbau



nach Tanenbaum/Goodman 1999

- ◆ Breite der Spur: 5–10 µm
 - ◆ Spuren pro Zentimeter: 800–2000
 - ◆ Breite einzelner Bits: 0,1–0,2 µm

- Zonen

- ◆ Mehrere Zylinder (10–30) bilden eine Zone mit gleicher Sektorenanzahl (bessere Plattenausnutzung)

Systemprogrammierung I

© 1997-2001, Franz J. Hauck, 2002 F. Hofmann, Inf 4, Univ. Erlangen-Nürnberg[F-File.fm, 2002-12-11 11.52]
Reproduktion jeder Art oder Verwendung dieser Unterlage, außer zu Lehrzwecken an der Universität Erlangen-Nürnberg, bedarf der Zustimmung des Autors.

F – 4

1.1 Festplatten (3)

■ Datenblätter zweier Beispielplatten

Plattentyp	Seagate Medalist	Seagate Cheetah
Kapazität	10,2 GB	36,4 GB
Platten/Köpfe	3/6	12/24
Zylinderzahl	CHS 16383/16/83	9772
Cache	512 kB	4 MB
Positionierzeiten	Spur zu Spur	0,6/0,9 ms
	mittlere	9,5 ms
	maximale	12/13 ms
Transferrate	8,5 MB/s	18,3–28 MB/s
Rotationsgeschw.	5.400 U/min	10.000 U/min
eine Plattenumdrehung	11 ms	6 ms
Stromaufnahme	4,5 W	14 W

1.1 Festplatten (4)

■ Zugriffsmerkmale

- ◆ blockorientierter und wahlfreier Zugriff
- ◆ Blockgröße zwischen 32 und 4096 Bytes (typisch 512 Bytes)
- ◆ Zugriff erfordert Positionierung des Schwenkarms auf den richtigen Zylinder und Warten auf den entsprechenden Sektor

■ Blöcke sind üblicherweise numeriert

- ◆ getrennte Numerierung: Zylindernummer, Sektornummer
- ◆ kombinierte Numerierung: durchgehende Nummern über alle Sektoren (Reihenfolge: aufsteigend innerhalb eines Zylinders, dann folgender Zylinder, etc.)

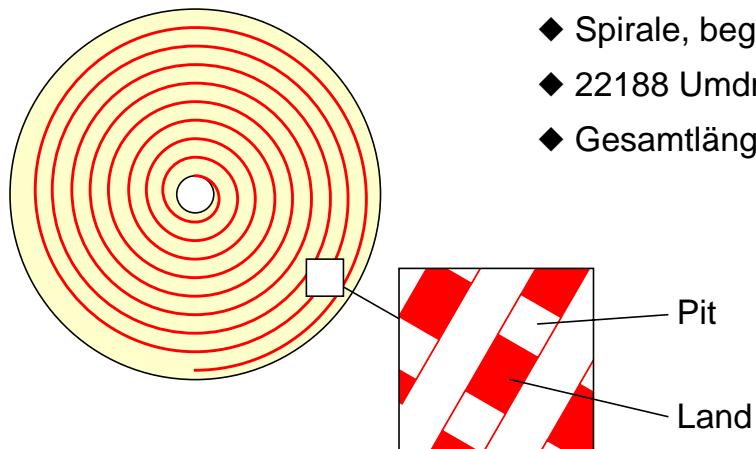
1.2 Disketten

- Ähnlicher Aufbau wie Festplatten
 - ◆ maximal zwei Schreib-, Leseköpfe (oben, unten)
 - ◆ Kopf berührt Diskettenoberfläche
- Typische Daten

Diskettentyp	3,5" HD
Kapazität	1,44 MB
Köpfe	2
Spuren	80
Sektoren pro Spur	18
Transferrate	62,5 kB/s
Rotationsgeschw.	300 U/min
eine Umdrehung	200 ms

1.3 CD-ROM

- Aufbau einer CD



- ◆ Spirale, beginnend im Inneren
- ◆ 22188 Umdrehungen (600 pro mm)
- ◆ Gesamtlänge 5,6 km

- ◆ **Pit:** Vertiefung, die von einem Laser abgetastet werden kann

1.3 CD-ROM (2)

- Kodierung
 - ◆ **Symbol**: ein Byte wird mit 14 Bits kodiert
(kann bereits bis zu zwei Bitfehler korrigieren)
 - ◆ **Frame**: 42 Symbole werden zusammengefasst
(192 Datenbits, 396 Fehlerkorrekturbits)
 - ◆ **Sektor**: 98 Frames werden zusammengefasst
(16 Bytes Präambel, 2048 Datenbytes, 288 Bytes Fehlerkorrektur)
 - ◆ *Effizienz*: 7203 Bytes transportieren 2048 Nutzbytes
- Transferrate
 - ◆ Single-Speed-Laufwerk:
75 Sektoren pro Sekunde (153.600 Bytes pro Sekunde)
 - ◆ 40-fach-Laufwerk:
3000 Sektoren pro Sekunde (6.144.000 Bytes pro Sekunde)

1.3 CD-ROM (3)

- Kapazität
 - ◆ ca. 650 MB
- Varianten
 - ◆ **CD-R** (Recordable): einmal beschreibbar
 - ◆ **CD-RW** (Rewritable): mehrfach beschreibbar
- DVD (Digital Versatile Disk)
 - ◆ kleinere Pits, engere Spirale, andere Laserlichfarbe
 - ◆ einseitig oder zweiseitig beschrieben
 - ◆ ein- oder zweischichtig beschrieben
 - ◆ Kapazität: 4,7 bis 17 GB

2 Speicherung von Dateien

- Dateien benötigen oft mehr als einen Block auf der Festplatte
 - ◆ Welche Blöcke werden für die Speicherung einer Datei verwendet?

2.1 Kontinuierliche Speicherung

- Datei wird in Blöcken mit aufsteigenden Blocknummern gespeichert
 - ◆ Nummer des ersten Blocks und Anzahl der Folgeblöcke muss gespeichert werden
- ★ Vorteile
 - ◆ Zugriff auf alle Blöcke mit minimaler Positionierzeit des Schwenkarms
 - ◆ Schneller direkter Zugriff auf bestimmter Dateiposition
 - ◆ Einsatz z.B. bei Systemen mit Echtzeitanforderungen

2.1 Kontinuierliche Speicherung (2)

- ▲ Probleme
 - ◆ Finden des freien Platzes auf der Festplatte (Menge aufeinanderfolgender und freier Plattenblöcke)
 - ◆ Fragmentierungsproblem (Verschnitt: nicht nutzbare Plattenblöcke; siehe auch Speicherverwaltung)
 - ◆ Größe bei neuen Dateien oft nicht im Voraus bekannt
 - ◆ Erweitern ist problematisch
 - Umkopieren, falls kein freier angrenzender Block mehr verfügbar

2.1 Kontinuierliche Speicherung (3)

■ Variation

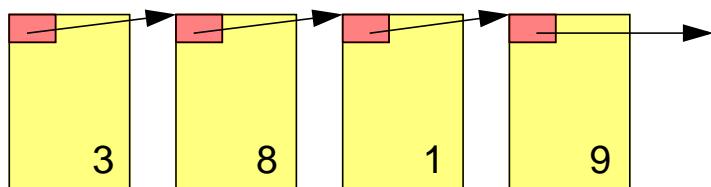
- ◆ Unterteilen einer Datei in Folgen von Blocks (*Chunks, Extents*)
- ◆ Blockfolgen werden kontinuierlich gespeichert
- ◆ Pro Datei muss erster Block und Länge jedes einzelnen Chunks gespeichert werden

▲ Problem

- ◆ Verschnitt innerhalb einer Folge (siehe auch Speicherverwaltung: interner Verschnitt bei Seitenadressierung)

2.2 Verkettete Speicherung

■ Blöcke einer Datei sind verkettet



- ◆ z.B. Commodore Systeme (CBM 64 etc.)

- Blockgröße 256 Bytes
- die ersten zwei Bytes bezeichnen Spur- und Sektornummer des nächsten Blocks
- wenn Spurnummer gleich Null: letzter Block
- 254 Bytes Nutzdaten

★ File kann wachsen und verlängert werden

2.2 Verkettete Speicherung (2)

▲ Probleme

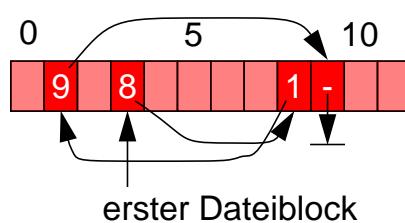
- ◆ Speicher für Verzeigerung geht von den Nutzdaten im Block ab (ungünstig im Zusammenhang mit Paging: Seite würde immer aus Teilen von zwei Plattenblöcken bestehen)
- ◆ Fehleranfälligkeit: Datei ist nicht restaurierbar, falls einmal Verzeigerung fehlerhaft
- ◆ schlechter direkter Zugriff auf bestimmte Dateiposition
- ◆ häufiges Positionieren des Schreib-, Lesekopfs bei verstreuten Datenblöcken

2.2 Verkettete Speicherung (3)

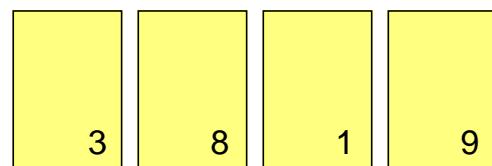
■ Verkettung wird in speziellen Plattenblocks gespeichert

- ◆ FAT-Ansatz (*FAT: File Allocation Table*), z.B. MS-DOS, Windows 95

FAT-Block



Blöcke der Datei: 3, 8, 1, 9



★ Vorteile

- ◆ kompletter Inhalt des Datenblocks ist nutzbar (günstig bei Paging)
- ◆ mehrfache Speicherung der FAT möglich: Einschränkung der Fehleranfälligkeit

2.2 Verkettete Speicherung (4)

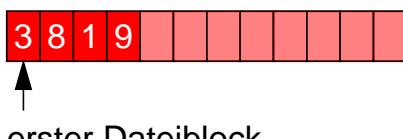
▲ Probleme

- ◆ mindestens ein zusätzlicher Block muss geladen werden
(Caching der FAT zur Effizienzsteigerung nötig)
- ◆ FAT enthält Verkettungen für alle Dateien: das Laden der FAT-Blöcke lädt auch nicht benötigte Informationen
- ◆ aufwändige Suche nach dem zugehörigen Datenblock bei bekannter Position in der Datei
- ◆ häufiges Positionieren des Schreib-, Lesekopfs bei verstreuten Datenblöcken

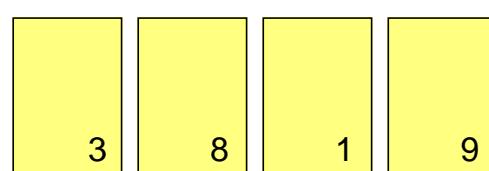
2.3 Indiziertes Speichern

■ Spezieller Plattenblock enthält Blocknummern der Datenblocks einer Datei

Indexblock



Blöcke der Datei: 3, 8, 1, 9

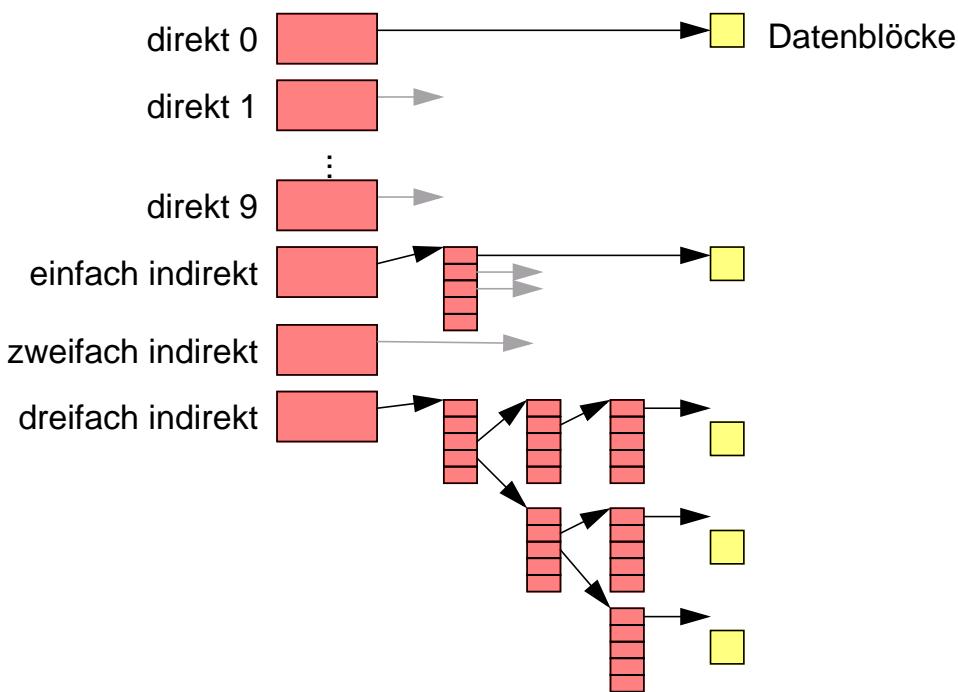


▲ Problem

- ◆ feste Anzahl von Blöcken im Indexblock
 - Verschnitt bei kleinen Dateien
 - Erweiterung nötig für große Dateien

2.3 Indiziertes Speichern (2)

Beispiel UNIX Inode



2.3 Indiziertes Speichern (3)

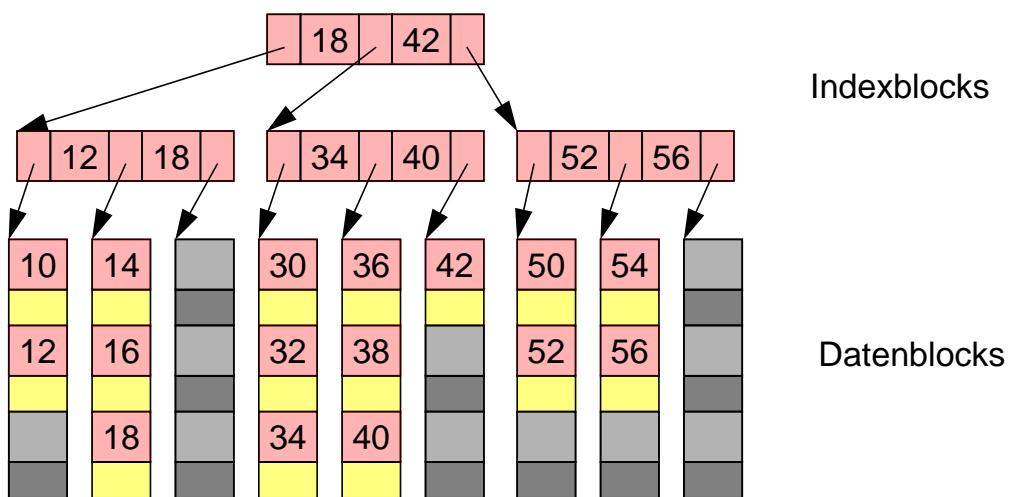
- ★ Einsatz von mehreren Stufen der Indizierung
 - ◆ Inode benötigt sowieso einen Block auf der Platte (Verschnitt unproblematisch bei kleinen Dateien)
 - ◆ durch mehrere Stufen der Indizierung auch große Dateien adressierbar
- ▲ Nachteil
 - ◆ mehrere Blöcke müssen geladen werden (nur bei langen Dateien)

2.4 Baumsequentielle Speicherung

- Satzorientierte Dateien
 - ◆ Schlüssel + Datensatz
 - ◆ effizientes Auffinden des Datensatz mit einem bekannten Schlüssel
 - ◆ Schlüsselmenge spärlich besetzt
 - ◆ häufiges Einfügen und Löschen von Datensätzen
- Einsatz von B-Bäumen zur Satzspeicherung
 - ◆ innerhalb von Datenbanksystemen
 - ◆ als Implementierung spezieller Dateitypen kommerzieller Betriebssysteme
 - z.B. VSAM-Dateien in MVS (*Virtual Storage Access Method*)
 - z.B. NTFS Katalogimplementierung

2.4 Baumsequentielle Speicherung (2)

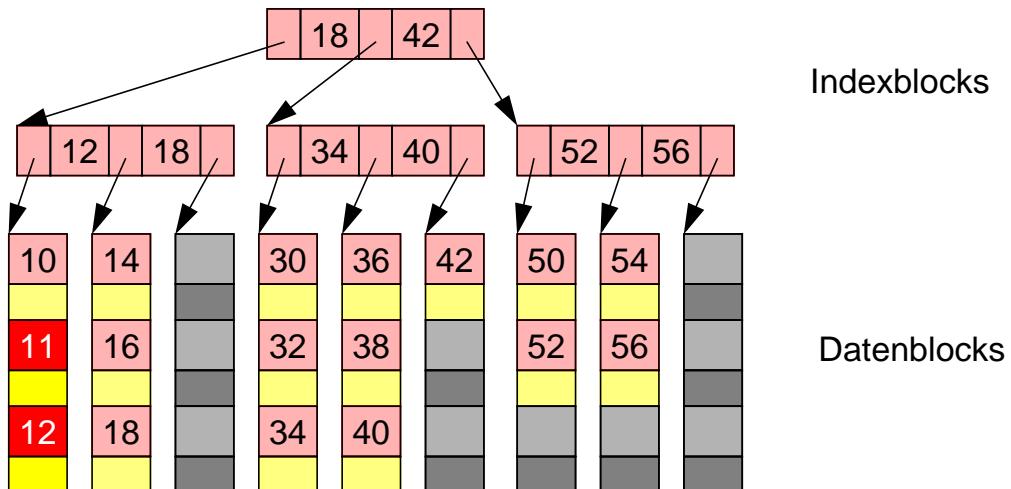
- Beispiel eines B*-Baums: Schlüssel sind Integer-Zahlen



- ◆ Blöcke enthalten Verweis auf nächste Ebene und den höchsten Schlüssel der nächsten Ebene
- ◆ Blocks der untersten Ebene enthalten Schlüssel und Sätze

2.4 Baumsequentielle Speicherung (3)

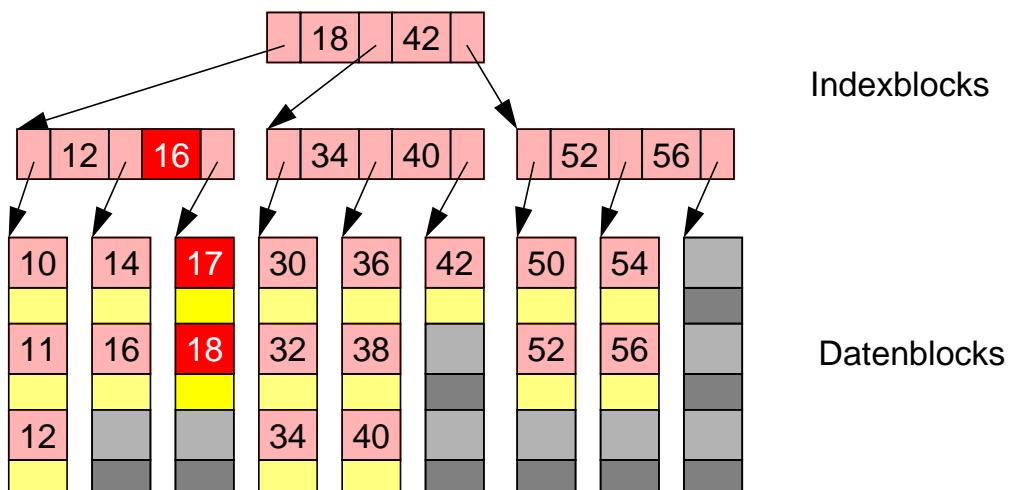
- Einfügen des Satzes mit Schlüssel „11“



- ◆ Satz mit Schlüssel „12“ wird verschoben
- ◆ Satz mit Schlüssel „11“ in freien Platz eingefügt

2.4 Baumsequentielle Speicherung (4)

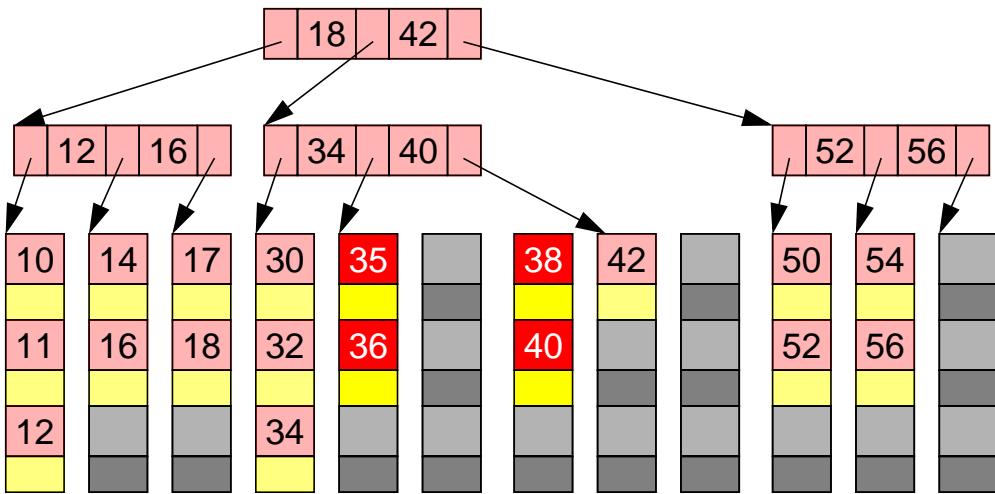
- Einfügen des Satzes mit Schlüssel „17“



- ◆ Satz mit Schlüssel „18“ wird verschoben (Indexblock wird angepasst)
- ◆ Satz mit Schlüssel „17“ in freien Platz eingefügt

2.4 Baumsequentielle Speicherung (5)

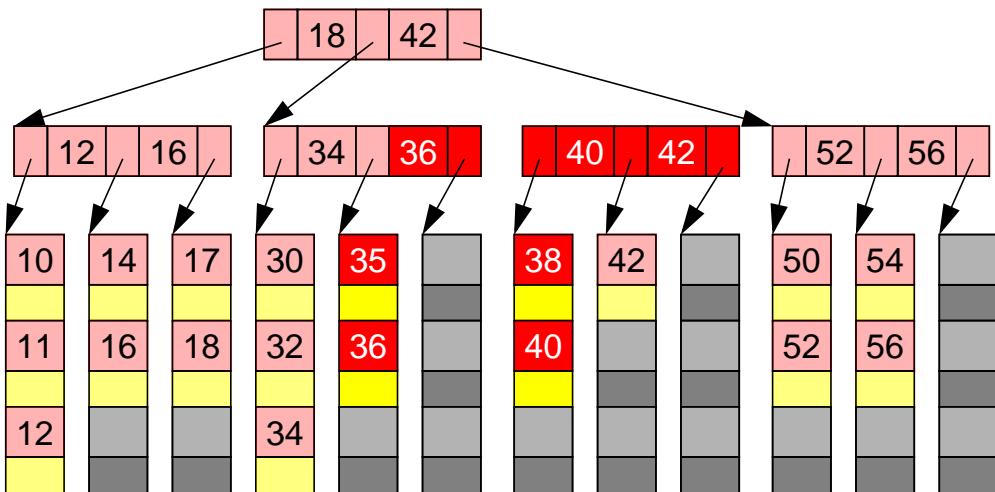
- Einfügen des Satzes mit Schlüssel „35“ (1. Schritt)



- ◆ Teilung des Blocks mit Satz „36“ und Einfügen des Satzes „35“
- ◆ Anfordern zweier weiterer, leerer Datenblöcke

2.4 Baumsequentielle Speicherung (6)

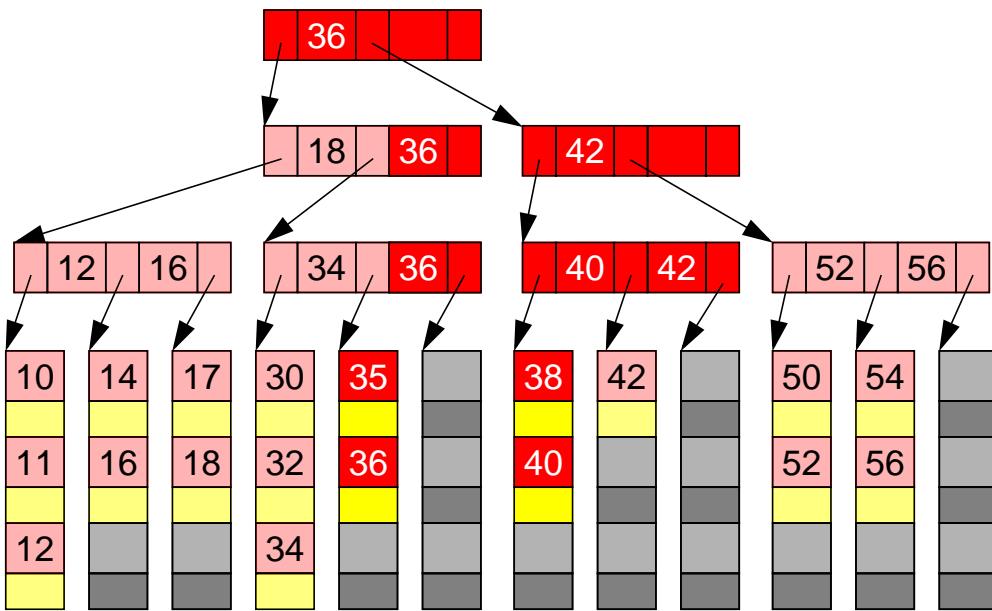
- Einfügen des Satzes mit Schlüssel „35“ (2. Schritt)



- ◆ Teilung bzw. Erzeugung eines neuen Indexblocks und dessen Verzeigerung

2.4 Baumsequentielle Speicherung (7)

- Einfügen des Satzes mit Schlüssel „35“ (3. Schritt)



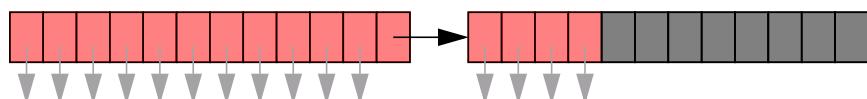
◆ Spaltung des alten Wurzelknotens, Erzeugen eines neuen Wurzel

2.4 Baumsequentielle Speicherung (8)

- ★ Effizientes Finden von Sätzen
 - ◆ Baum ist sehr niedrig im Vergleich zur Menge der Sätze
 - viele Schlüsse pro Indexblock vorhanden (je nach Schlüssellänge)
- ★ Gutes Verhalten im Zusammenhang mit Paging
 - ◆ jeder Block entspricht einer Seite
 - ◆ Demand paging sorgt für das automatische Anhäufen der oberen Indexblocks im Hauptspeicher
 - schneller Zugriff auf die Indexstrukturen
- ★ Erlaubt nebenläufige Operationen durch geeignetes Sperren von Indexblöcken
- Löschen erfolgt ähnlich wie Einfügen
 - ◆ Verschmelzen von schlecht belegten Datenblöcken nötig

3 Freispeicherverwaltung

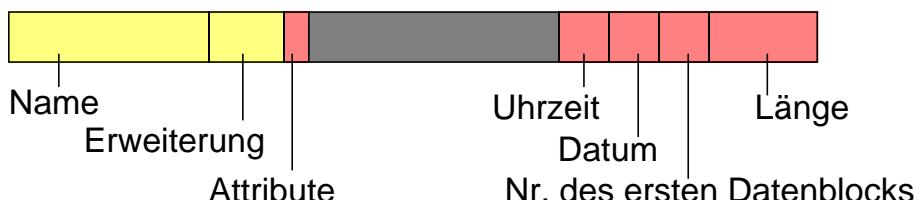
- Prinzipiell ähnlich wie Verwaltung von freiem Hauptspeicher
 - ◆ Bitvektoren zeigen für jeden Block Belegung an
 - ◆ verkettete Listen repräsentieren freie Blöcke
 - Verkettung kann in den freien Blöcken vorgenommen werden
 - Optimierung: aufeinanderfolgende Blöcke werden nicht einzeln aufgenommen, sondern als Stück verwaltet
 - Optimierung: ein freier Block enthält viele Blocknummern weiterer freier Blöcke und evtl. die Blocknummer eines weiteren Blocks mit den Nummern freier Blöcke



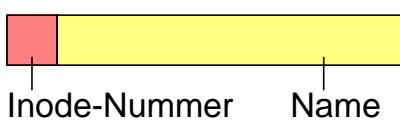
4 Implementierung von Katalogen

4.1 Kataloge als Liste

- Einträge gleicher Länge werden hintereinander in eine Liste gespeichert
 - ◆ z.B. *FAT File systems*



- ◆ für *VFAT* werden mehrere Einträge zusammen verwendet, um den langen Namen aufzunehmen
 - ◆ z.B. *UNIX System V.3*



4.1 Kataloge als Liste (2)

▲ Problem

- ◆ Lineare Suche durch die Liste nach bestimmtem Eintrag
- ◆ Sortierte Liste: binäre Suche, aber Sortieraufwand

4.2 Einsatz von Hashfunktionen

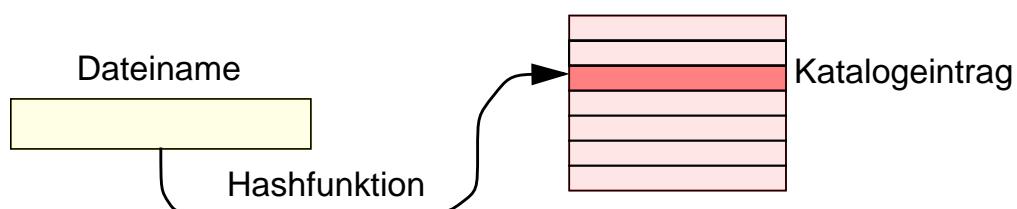
■ Hashing

- ◆ Spärlich besetzter Schlüsselraum wird auf einen anderen, meist dichter besetzten Schlüsselraum abgebildet
- ◆ Beispiel: Menge der möglichen Dateinamen wird nach [0 – N-1] abgebildet ($N = \text{Länge der Katalogliste}$)

4.2 Einsatz von Hashfunktionen (2)

■ Hashfunktion

- ◆ Funktion bildet Dateinamen auf einen Index in die Katalogliste ab schnellerer Zugriff auf den Eintrag möglich (kein lineares Suchen)
- ◆ (einfaches aber schlechtes) Beispiel: $(\sum \text{Zeichen}) \bmod N$

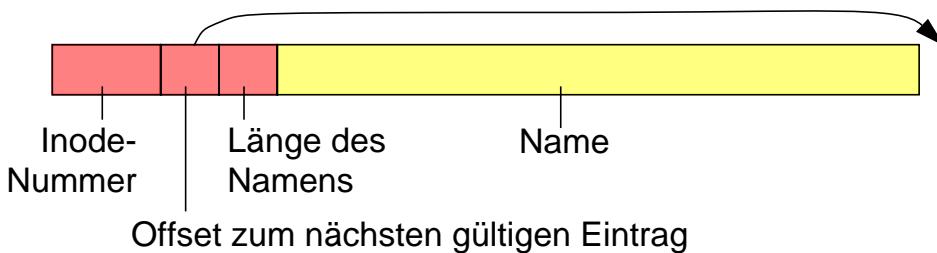


▲ Probleme

- ◆ Kollisionen (mehrere Dateinamen werden auf gleichen Eintrag abgebildet)
- ◆ Anpassung der Listengröße, wenn Liste voll

4.3 Variabel lange Listenelemente

- Beispiel *BSD 4.2, System V.4, u.a.*



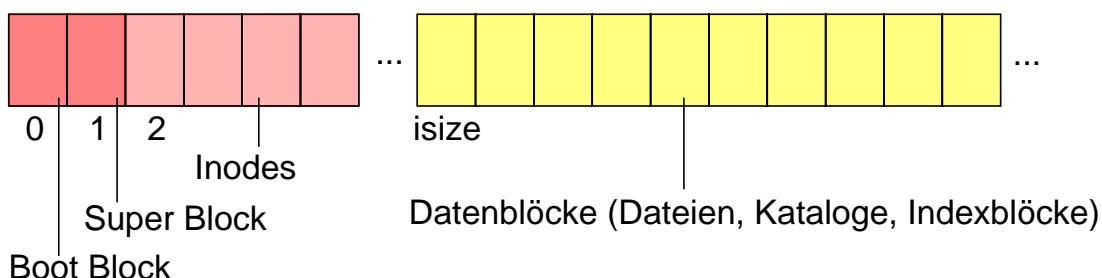
Probleme

- Verwaltung von freien Einträgen in der Liste
- Speicherverschnitt (Kompaktifizieren, etc.)

5 Beispiel: UNIX File Systems

5.1 System V File System

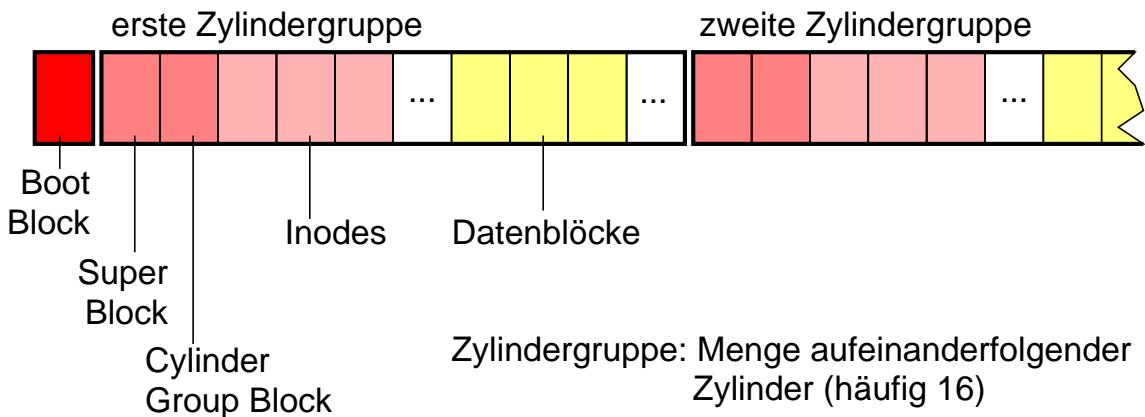
- Blockorganisation



- Boot Block enthält Informationen zum Laden eines initialen Programms
- Super Block enthält Verwaltungsinformation für ein Dateisystem
 - Anzahl der Blöcke, Anzahl der Inodes
 - Anzahl und Liste freier Blöcke und freier Inodes
 - Attribute (z.B. *Modified flag*)

5.2 BSD 4.2 (Berkeley Fast File System)

■ Blockorganisation

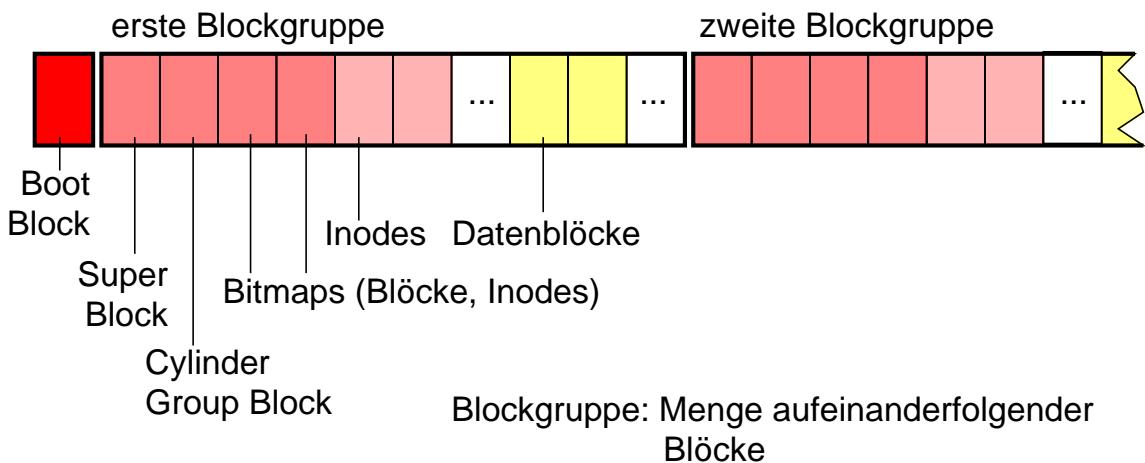


- ◆ Kopie des Super Blocks in jeder Zylindergruppe
 - ◆ freie Inodes u. freie Datenblöcke werden im Cylinder group block gehalten
 - ◆ eine Datei wird möglichst innerhalb einer Zylindergruppe gespeichert

★ Vorteil: kürzere Positionierungszeiten

5.3 Linux EXT2 File System

■ Blockorganisation



- ◆ Ähnliches Layout wie BSD FFS
 - ◆ Blockgruppen unabhängig von Zylindern

5.4 Block Buffer Cache

- Pufferspeicher für alle benötigten Plattenblocks
 - ◆ Verwaltung mit Algorithmen ähnlich wie bei Paging
 - ◆ *Read ahead*: beim sequentiellen Lesen wird auch der Transfer des Folgeblocks angestoßen
 - ◆ *Lazy write*: Block wird nicht sofort auf Platte geschrieben (erlaubt Optimierung der Schreibzugriffe und blockiert den Schreiber nicht)
 - ◆ Verwaltung freier Blöcke in einer Freiliste
 - Kandidaten für Freiliste werden nach LRU Verfahren bestimmt
 - bereits freie aber noch nicht anderweitig benutzte Blöcke können reaktiviert werden (*Reclaim*)

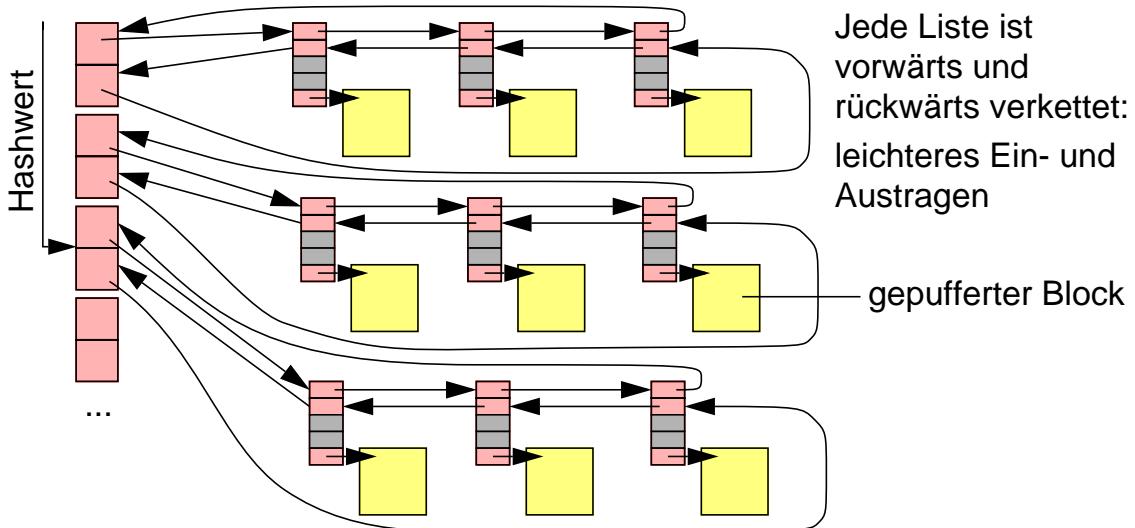
5.4 Block Buffer Cache (2)

- Schreiben erfolgt, wenn
 - ◆ Datei geschlossen wird,
 - ◆ keine freien Puffer mehr vorhanden sind,
 - ◆ regelmäßig vom System (*fsflush* Prozess, *update* Prozess),
 - ◆ beim Systemaufruf *sync()*,
 - ◆ und nach jedem Schreibauftrag im Modus *O_SYNC*.
- Adressierung
 - ◆ Adressierung eines Blocks erfolgt über ein Tupel:
(Gerätenummer, Blocknummer)
 - ◆ Über die Adresse wird ein Hashwert gebildet, der eine der möglichen Pufferliste auswählt

5.4 Block Buffer Cache (3)

Aufbau des Block buffer cache

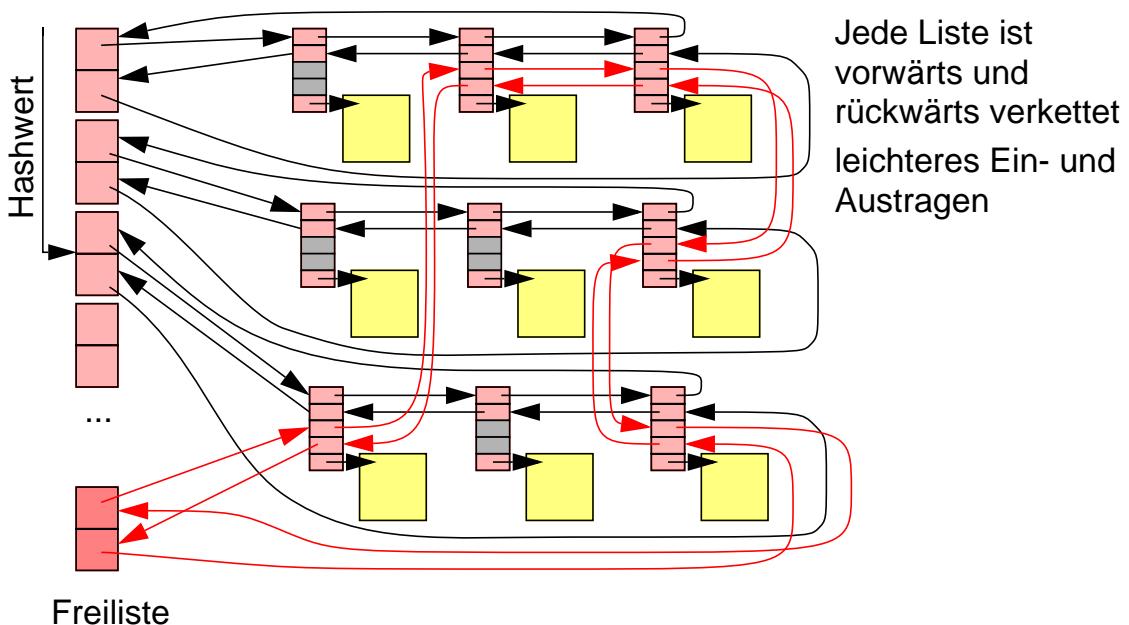
Pufferlisten (Queues)



5.4 Block Buffer Cache (4)

Aufbau des Block buffer cache

Pufferlisten (Queues)



5.4 Block Buffer Cache (5)

- Block Buffer Cache teilweise obsolet durch moderne Paging-Systeme
 - ◆ Kacheln des Hauptspeichers ersetzen den Block Buffer Cache
 - ◆ Kacheln können Seiten aus einem Adressraum
und/oder Seiten aus einer Datei beherbergen
- ▲ Problem
 - ◆ Kopieren großer Dateien führt zum Auslagern noch benötigter Adressraumseiten

5.5 Systemaufrufe

- Bestimmen der Kachelgröße

```
int getpagesize( void );
```
- Abbildung von Dateien in den virtuellen Adressraum
 - ◆ Einblenden einer Datei

```
caddr_t mmap( caddr_t addr, size_t len, int prot, int flags,
              int fd, off_t off );
```

 - Einblenden an bestimmte oder beliebige Adresse
 - lesbar, schreibbar, ausführbar
 - ◆ Ausblenden einer Datei

```
int munmap( caddr_t addr, size_t len );
```

5.5 Systemaufrufe (2)

◆ Kontolloperation

```
int mctl( caddr_t addr, size_t len, int func, void *arg );
```

- zum Ausnehmen von Seiten aus dem Paging (Fixieren im Hauptspeicher)
- zum Synchronisieren mit der Datei

6 Beispiel: Windows NT (NTFS)

■ File System für Windows NT

■ Datei

- ◆ einfache, unstrukturierte Folge von Bytes
- ◆ beliebiger Inhalt; für das Betriebssystem ist der Inhalt transparent
- ◆ dynamisch erweiterbare Dateien
- ◆ Rechte verknüpft mit NT Benutzern und Gruppen
- ◆ Datei kann automatisch komprimiert abgespeichert werden
- ◆ große Dateien bis zu 8.589.934.592 Gigabytes lang
- ◆ Hard links: mehrere Einträge derselben Datei in verschiedenen Katalogen möglich

6 Beispiel: NTFS (2)

■ Katalog

- ◆ baumförmig strukturiert
 - Knoten des Baums sind Kataloge
 - Blätter des Baums sind Dateien
- ◆ Rechte wie bei Dateien
- ◆ alle Dateien des Katalogs automatisch komprimierbar

■ Partitionen heißen Volumes

- ◆ Volume wird (in der Regel) durch einen Laufwerksbuchstaben dargestellt z.B. c:

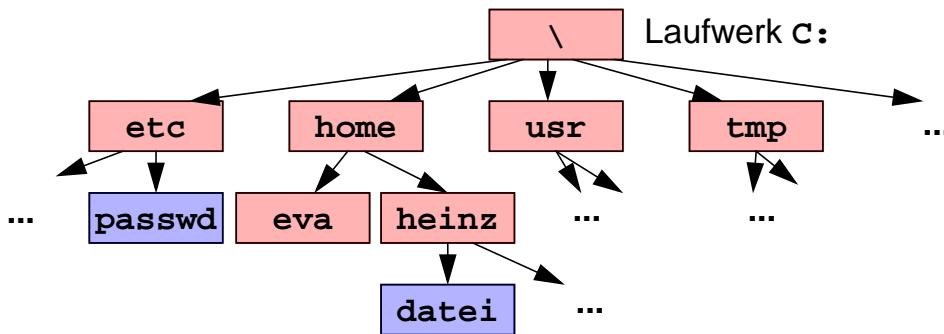
6.1 Rechte

■ Eines der folgenden Rechte pro Benutzer oder Benutzergruppe

- ◆ *no access*: kein Zugriff
- ◆ *list*: Anzeige von Dateien in Katalogen
- ◆ *read*: Inhalt von Dateien lesen und *list*
- ◆ *add*: Hinzufügen von Dateien zu einem Katalog und *list*
- ◆ *read&add*: wie *read* und *add*
- ◆ *change*: Ändern von Dateiinhalten, Löschen von Dateien und *read&add*
- ◆ *full*: Ändern von Eigentümer und Zugriffsrechten und *change*

6.2 Pfadnamen

■ Baumstruktur



■ Pfade

- ◆ wie unter FAT-Filesystem
- ◆ z.B. „C:\home\heinz\datei“, „\tmp“, „C:..\heinz\datei“

6.2 Pfadnamen (2)

■ Namenskonvention

- ◆ 255 Zeichen inklusive Sonderzeichen
(z.B. „**Eigene Programme**“)
- ◆ automatischer Kompatibilitätsmodus: 8 Zeichen Name, 3 Zeichen Erweiterung, falls „langer Name“ unter MS-DOS ungültig
(z.B. **AUTOEXEC.BAT**)

■ Kataloge

- ◆ Jeder Katalog enthält einen Verweis auf sich selbst („..“) und einen Verweis auf den darüberliegenden Katalog im Baum („...“)
- ◆ Hard links aber keine symbolischen Namen direkt im NTFS

6.3 Dateiverwaltung

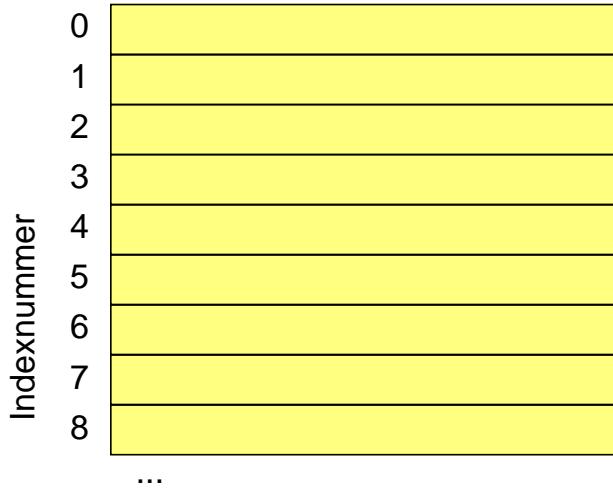
- Basiseinheit „Cluster“
 - ◆ 512 Bytes bis 4 Kilobytes (beim Formatieren festgelegt)
 - ◆ wird auf eine Menge von hintereinanderfolgenden Blöcken abgebildet
 - ◆ logische Cluster-Nummer als Adresse (LCN)
- Basiseinheit „Strom“
 - ◆ jede Datei kann mehrere (Daten-)Ströme speichern
 - ◆ einer der Ströme wird für die eigentlichen Daten verwendet
 - ◆ Dateiname, MS-DOS Dateiname, Zugriffsrechte, Attribute und Zeitstempel werden jeweils in eigenen Datenströmen gespeichert
(leichte Erweiterbarkeit des Systems)

6.3 Dateiverwaltung (2)

- *File-Reference*
 - ◆ Bezeichnet eindeutig eine Datei oder einen Katalog
- 63 47 0
-
- Sequenz- Dateinummer
nummer
- Dateinummer ist Index in eine globale Tabelle (*MFT: Master File Table*)
 - Sequenznummer wird hochgezählt, für jede neue Datei mit gleicher Dateinummer

6.4 Master-File-Table

- Rückgrat des gesamten Systems
 - ◆ große Tabelle mit gleich langen Elementen
(1KB, 2KB oder 4KB groß, je nach Clustergröße)

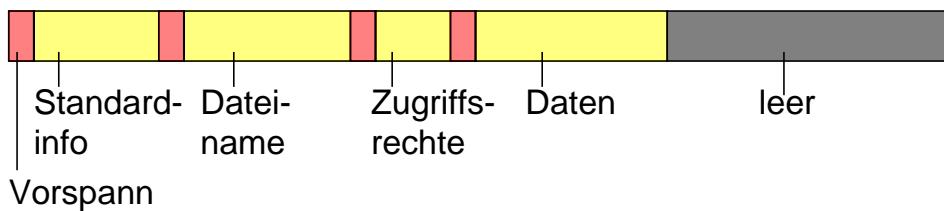


entsprechender Eintrag für eine *File-Reference* enthält Informationen über bzw. die Ströme der Datei

- ◆ Index in die Tabelle ist Teil der *File-Reference*

6.4 Master-File-Table (2)

- Eintrag für eine kurze Datei

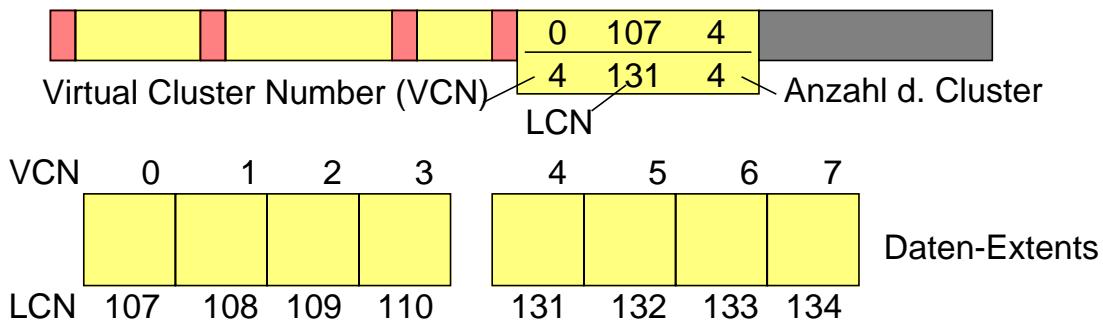


- Ströme

- ◆ Standard Information (immer in der MFT)
 - enthält Länge, MS-DOS Attribute, Zeitstempel, Anzahl der Hard links, Sequenznummer der gültigen File-Reference
- ◆ Dateiname (immer in der MFT)
 - kann mehrfach vorkommen (Hard links, MS-DOS Name)
- ◆ Zugriffsrechte (*Security Descriptor*)
- ◆ Eigentliche Daten

6.4 Master-File-Table (3)

- Eintrag für eine längere Datei



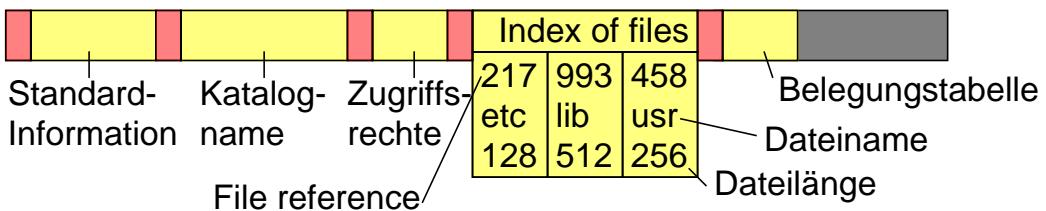
- ◆ Extents werden außerhalb der MFT in aufeinanderfolgenden Clustern gespeichert
- ◆ Lokalisierungsinformationen werden in einem eigenen Strom gespeichert

6.4 Master-File-Table (4)

- Mögliche weitere Ströme (*Attributes*)
 - ◆ Index
 - Index über einen Attributschlüssel (z.B. Dateinamen)
implementiert Katalog
 - ◆ Indexbelegungstabelle
 - Belegung der Struktur eines Index
 - ◆ Attributliste (immer in der MFT)
 - wird benötigt, falls nicht alle Ströme in einen MFT Eintrag passen
 - referenzieren weitere MFT Einträge und deren Inhalt

6.4 Master File Table (3)

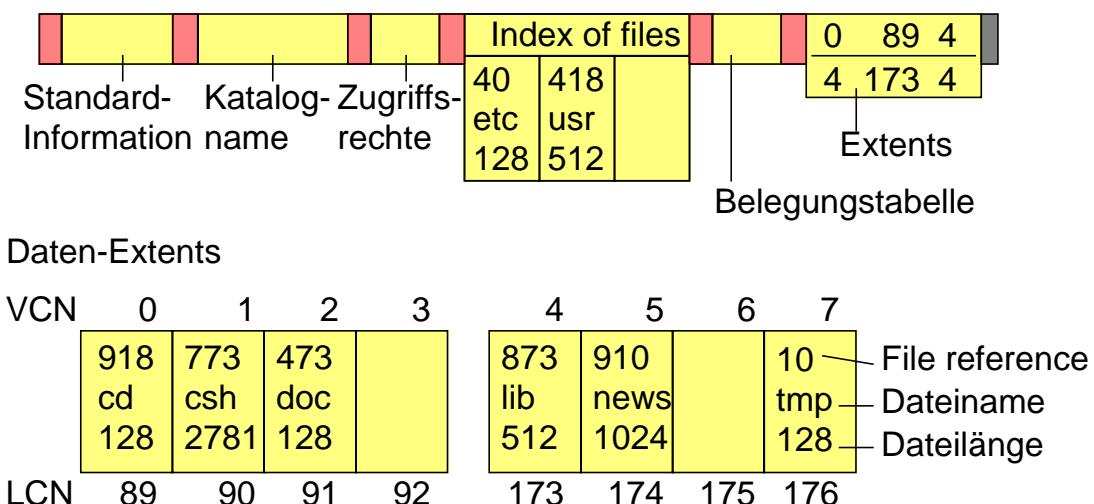
- Eintrag für einen kurzen Katalog



- ◆ Dateien des Katalogs werden mit File-References benannt
- ◆ Name und Länge der im Katalog enthaltenen Dateien und Kataloge werden auch im Index gespeichert
(doppelter Aufwand beim Update; schnellerer Zugriff beim Kataloglisten)

6.4 Master File Table (4)

- Eintrag für einen längeren Katalog



- ◆ Speicherung als B⁺-Baum (sortiert, schneller Zugriff)
- ◆ in einen Cluster passen zwischen 3 und 15 Dateien (im Bild nur eine)

6.5 Metadaten

- Alle Metadaten werden in Dateien gehalten

0	MFT	Feste Dateien in der MFT
1	MFT Kopie (teilweise)	
2	Log File	
3	Volume Information	
4	Attributabelle	
5	Wurzelkatalog	
6	Clusterbelegungstabelle	
7	Boot File	
8	Bad Cluster File	
...		
16	Benutzerdateien u. -kataloge	
17		
...		

6.5 Metadaten (2)

- Bedeutung der Metadateien

- ◆ MFT und MFT Kopie: MFT wird selbst als Datei gehalten
(d.h. Cluster der MFT stehen im Eintrag 0)
MFT Kopie enthält die ersten 16 Einträge der MFT (Fehlertoleranz)
- ◆ Log File: enthält protokollierte Änderungen am Dateisystem
- ◆ Volume Information: Name, Größe und ähnliche Attribute des Volumes
- ◆ Attributabelle: definiert mögliche Ströme in den Einträgen
- ◆ Wurzelkatalog
- ◆ Clusterbelegungstabelle: Bitmap für jeden Cluster des Volumes
- ◆ Boot File: enthält initiales Programm zum Laden, sowie ersten Cluster der MFT
- ◆ Bad Cluster File: enthält alle nicht lesbaren Cluster der Platte
NTFS markiert automatisch alle schlechten Cluster und versucht die Daten in einen anderen Cluster zu retten

6.6 Fehlererholung

- NTFS ist ein Journaled-File-System
 - ◆ Änderungen an der MFT und an Dateien werden protokolliert.
 - ◆ Konsistenz der Daten und Metadaten kann nach einem Systemausfall durch Abgleich des Protokolls mit den Daten wieder hergestellt werden.
- ▲ Nachteile
 - ◆ etwas ineffizienter
 - ◆ nur für Volumes >400 MB geeignet

7 Dateisysteme mit Fehlererholung

- Mögliche Fehler
 - ◆ Stromausfall (dummer Benutzer schaltet einfach Rechner aus)
 - ◆ Systemabsturz
- Auswirkungen auf das Dateisystem
 - ◆ inkonsistente Metadaten
 - z.B. Katalogeintrag fehlt zur Datei oder umgekehrt
 - z.B. Block ist benutzt aber nicht als belegt markiert
- ★ Reparaturprogramme
 - ◆ Programme wie **chkdsk**, **scandisk** oder **fsck** können inkonsistente Metadaten reparieren
- ▲ Datenverluste bei Reparatur möglich
- ▲ Große Platten induzieren lange Laufzeiten der Reparaturprogramme

7.1 Journalled-File-Systems

- Zusätzlich zum Schreiben der Daten und Meta-Daten (z.B. Inodes) wird ein Protokoll der Änderungen geführt
 - ◆ Alle Änderungen treten als Teil von Transaktionen auf.
 - ◆ Beispiele für Transaktionen:
 - Erzeugen, löschen, erweitern, verkürzen von Dateien
 - Dateiattribute verändern
 - Datei umbenennen
 - ◆ Protokollieren aller Änderungen am Dateisystem zusätzlich in einer Protokolldatei (*Log File*)
 - ◆ Beim Bootvorgang wird Protokolldatei mit den aktuellen Änderungen abgeglichen und damit werden Inkonsistenzen vermieden.

7.1 Journalled-File-Systems (2)

- Protokollierung
 - ◆ Für jeden Einzelvorgang einer Transaktion wird zunächst ein Logeintrag erzeugt und
 - ◆ danach die Änderung am Dateisystem vorgenommen.
 - ◆ Dabei gilt:
 - Der Logeintrag wird immer **vor** der eigentlichen Änderung auf Platte geschrieben.
 - Wurde etwas auf Platte geändert, steht auch der Protokolleintrag dazu auf der Platte.

7.1 Journalled-File-Systems (3)

■ Fehlererholung

- ◆ Beim Bootvorgang wird überprüft, ob die protokollierten Änderungen vorhanden sind:
 - Transaktion kann wiederholt bzw. abgeschlossen werden (*Redo*) falls alle Logeinträge vorhanden.
 - Angefangene aber nicht beendete Transaktionen werden rückgängig gemacht (*Undo*).

7.1 Journalled-File-Systems (4)

■ Beispiel: Löschen einer Datei im NTFS

- ◆ Vorgänge der Transaktion
 - Beginn der Transaktion
 - Freigeben der Extents durch Löschen der entsprechenden Bits in der Belegungstabelle (gesetzte Bits kennzeichnen belegten Cluster)
 - Freigeben des MFT Eintrags der Datei
 - Löschen des Katalogeintrags der Datei
(evtl. Freigeben eines Extents aus dem Index)
 - Ende der Transaktion
- ◆ Alle Vorgänge werden unter der File-Reference im Log-File protokolliert, danach jeweils durchgeführt.
 - Protokolleinträge enthalten Informationen zum *Redo* und zum *Undo*

7.1 Journalled-File-Systems (5)

- ◆ Log vollständig (Ende der Transaktion wurde protokolliert und steht auf Platte):
 - *Redo* der Transaktion:
alle Operationen werden wiederholt, falls nötig
- ◆ Log unvollständig (Ende der Transaktion steht nicht auf Platte):
 - *Undo* der Transaktion:
in umgekehrter Reihenfolge werden alle Operation rückgängig gemacht

■ Checkpoints

- ◆ Log-File kann nicht beliebig groß werden
- ◆ gelegentlich wird für einen konsistenten Zustand auf Platte gesorgt (*Checkpoint*) und dieser Zustand protokolliert (alle Protokolleinträge von vorher können gelöscht werden)
- ◆ Ähnlich verfährt NTFS, wenn Ende des Log-Files erreicht wird.

7.1 Journalled-File-Systems (6)

★ Ergebnis

- ◆ eine Transaktion ist entweder vollständig durchgeführt oder gar nicht
- ◆ Benutzer kann ebenfalls Transaktionen über mehrere Dateizugriffe definieren, wenn diese ebenfalls im Log erfasst werden.
- ◆ keine inkonsistenten Metadaten möglich
- ◆ Hochfahren eines abgestürzten Systems benötigt nur den relativ kurzen Durchgang durch das Log-File.
 - Alternative `chkdsk` benötigt viel Zeit bei großen Platten

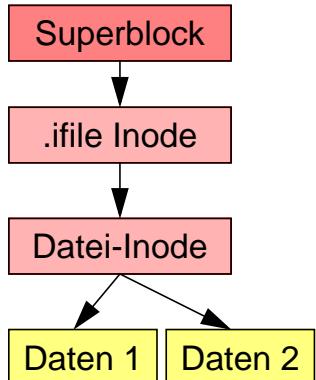
▲ Nachteile

- ◆ ineffizienter, da zusätzliches Log-File geschrieben wird

■ Beispiele: NTFS, EXT3, ReiserFS

7.2 Log-Structured-File-Systems

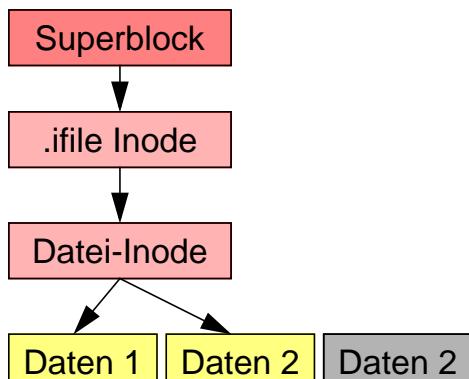
- Alle Änderungen im Dateisystem erfolgen auf Kopien
 - ◆ Der Inhalt veränderter Blöcke wird in einen neuen Block geschrieben



- ◆ Beispiel LinLogFS: Superblock einziger nicht ersetzer Block

7.2 Log-Structured-File-Systems

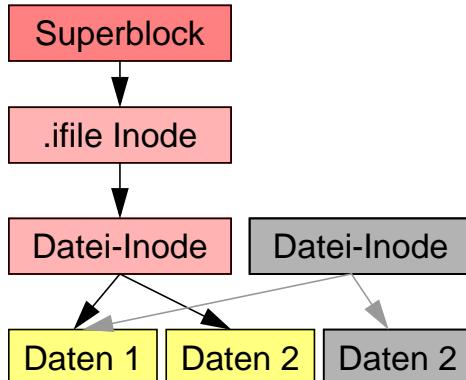
- Alle Änderungen im Dateisystem erfolgen auf Kopien
 - ◆ Der Inhalt veränderter Blöcke wird in einen neuen Block geschrieben



- ◆ Beispiel LinLogFS: Superblock einziger nicht ersetzer Block

7.2 Log-Structured-File-Systems

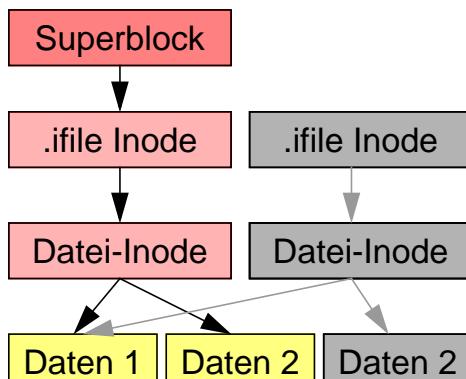
- Alle Änderungen im Dateisystem erfolgen auf Kopien
 - ◆ Der Inhalt veränderter Blöcke wird in einen neuen Block geschrieben



- ◆ Beispiel LinLogFS: Superblock einziger nicht erster Block

7.2 Log-Structured-File-Systems

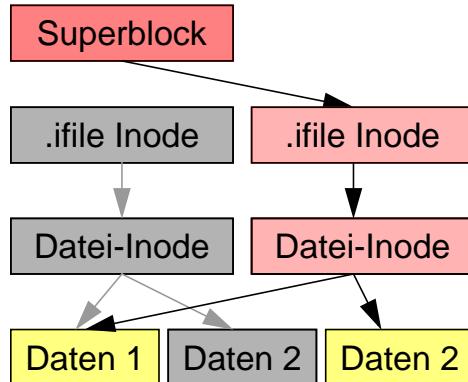
- Alle Änderungen im Dateisystem erfolgen auf Kopien
 - ◆ Der Inhalt veränderter Blöcke wird in einen neuen Block geschrieben



- ◆ Beispiel LinLogFS: Superblock einziger nicht erster Block

7.2 Log-Structured-File-Systems

- Alle Änderungen im Dateisystem erfolgen auf Kopien
 - ◆ Der Inhalt veränderter Blöcke wird in einen neuen Block geschrieben



- ◆ Beispiel LinLogFS: Superblock einziger nicht erster Block

7.2 Log-Structured-File-Systems (2)

- ★ Vorteile
 - ◆ Datenkonsistenz bei Systemausfällen
 - ein atomare Änderung macht alle zusammengehörigen Änderungen sichtbar
 - ◆ Schnappschüsse / Checkpoints einfach realisierbar
 - ◆ Gute Schreibeffizienz
 - Alle zu schreibenden Blöcke werden kontinuierlich geschrieben
- ▲ Nachteile
 - ◆ Gesamtperformanz geringer
- Beispiele: LinLogFS, BSD LFS, AIX XFS

8 Limitierung der Plattennutzung

- Mehrbenutzersysteme
 - ◆ einzelnen Benutzern sollen verschieden große Kontingente zur Verfügung stehen
 - ◆ gegenseitige Beeinflussung soll vermieden werden (*Disk-full* Fehlermeldung)
- Quota-Systeme (Quantensysteme)
 - ◆ Tabelle enthält maximale und augenblickliche Anzahl von Blöcken für die Dateien und Kataloge eines Benutzers
 - ◆ Tabelle steht auf Platte und wird vom File-System fortgeschrieben
 - ◆ Benutzer erhält Disk-full Meldung, wenn sein Quota verbraucht ist
 - ◆ üblicherweise gibt es eine weiche und eine harte Grenze (weiche Grenze kann für eine bestimmte Zeit überschritten werden)

9 Fehlerhafte Plattenblöcke

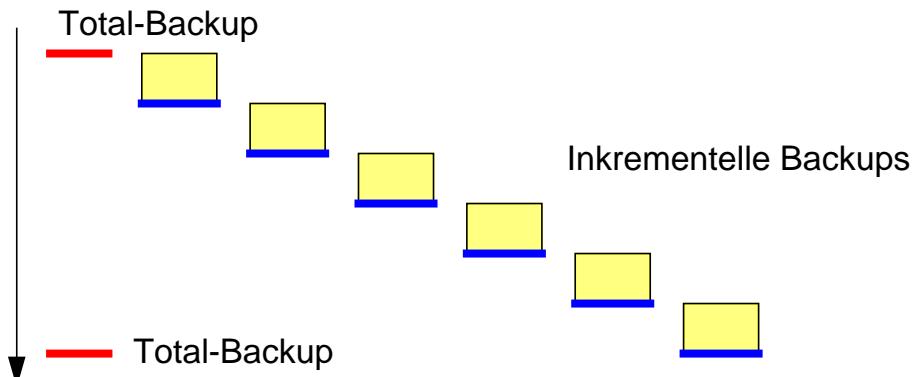
- Blöcke, die beim Lesen Fehlermeldungen erzeugen
 - ◆ z.B. Prüfsummenfehler
- Hardwarelösung
 - ◆ Platte und Plattencontroller bemerken selbst fehlerhafte Blöcke und maskieren diese aus
 - ◆ Zugriff auf den Block wird vom Controller automatisch auf einen „gesunden“ Block umgeleitet
- Softwarelösung
 - ◆ File-System bemerkt fehlerhafte Blöcke und markiert diese auch als belegt

10 Datensicherung

- Schutz vor dem Totalausfall von Platten
 - ◆ z.B. durch Head-Crash oder andere Fehler
- Sichern der Daten auf Tertiärspeicher
 - ◆ Bänder
 - ◆ WORM Speicherplatten (*Write Once Read Many*)
- Sichern großer Datenbestände
 - ◆ Total-Backups benötigen lange Zeit
 - ◆ Inkrementelle Backups sichern nur Änderungen ab einem bestimmten Zeitpunkt
 - ◆ Mischen von Total-Backups mit inkrementellen Backups

10.1 Beispiele für Backup Scheduling

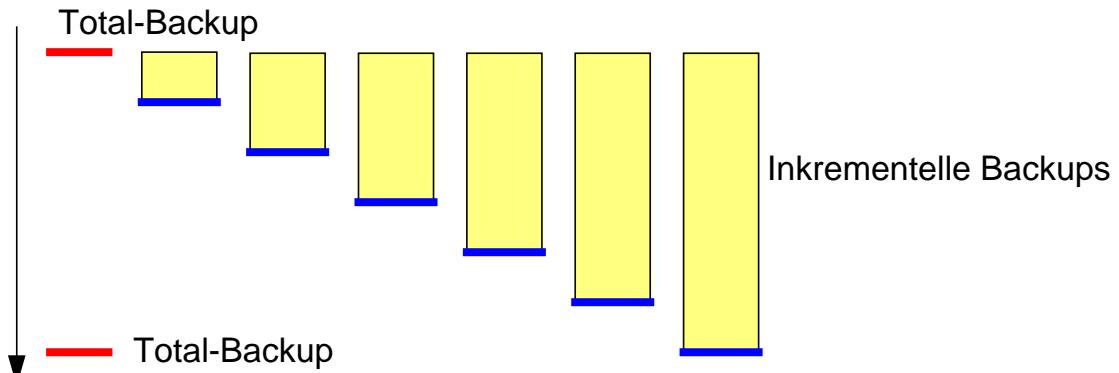
- Gestaffelte inkrementelle Backups



- ◆ z.B. alle Woche ein Total-Backup und jeden Tag ein inkrementelles Backup zum Vortag: maximal 7 Backups müssen eingespielt werden

10.1 Beispiele für Backup Scheduling (2)

- Gestaffelte inkrementelle Backups zum letzten Total-Backup



- ◆ z.B. alle Woche ein Total-Backup und jeden Tag ein inkrementelles Backup zum letzten Total-Backup: maximal 2 Backups müssen eingespielt werden

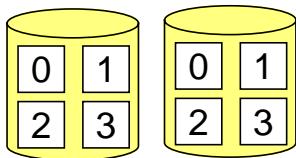
- Hierarchie von Backup-Läufen

- ◆ mehrstufige inkrementelle Backups zum Backup der nächst höheren Stufe
- ◆ optimiert Archivmaterial und Restaurierungszeit

10.2 Einsatz mehrere redundanter Platten

- Gespiegelte Platten (*Mirroring; RAID 0*)

- ◆ Daten werden auf zwei Platten gleichzeitig gespeichert



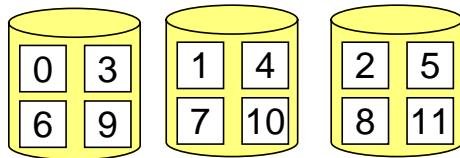
- ◆ Implementierung durch Software (File-System, Plattentreiber) oder Hardware (spez. Controller)
- ◆ eine Platte kann ausfallen
- ◆ schnelleres Lesen (da zwei Platten unabhängig voneinander beauftragt werden können)

▲ Nachteil

- ◆ doppelter Speicherbedarf
- ◆ wenig langsameres Schreiben durch Warten auf zwei Plattentransfers

10.2 Einsatz mehrere redundanter Platten (2)

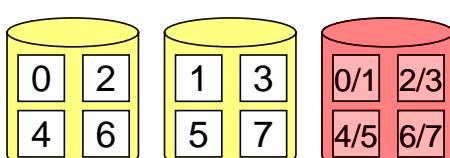
- Gestreifte Platten (*Striping*; RAID 1)
 - ◆ Daten werden über mehrere Platten gespeichert



- ◆ Datentransfers sind nun schneller, da mehrere Platten gleichzeitig angesprochen werden können
- ▲ Nachteil
 - ◆ keinerlei Datensicherung: Ausfall einer Platte lässt Gesamtsystem ausfallen
- Verknüpfung von RAID 0 und 1 möglich (RAID 0+1)

10.2 Einsatz mehrere redundanter Platten (3)

- Paritätsplatte (RAID 4)
 - ◆ Daten werden über mehrere Platten gespeichert, eine Platte enthält Parität
- ◆ Paritätsblock enthält byteweise XOR-Verknüpfungen von den zugehörigen Blöcken aus den anderen Streifen
- ◆ eine Platte kann ausfallen
- ◆ schnelles Lesen
- ◆ prinzipiell beliebige Plattenanzahl (ab drei)



10.2 Einsatz mehrerer redundanter Platten (4)

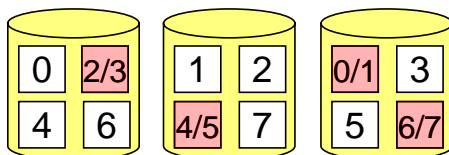
▲ Nachteil von RAID 4

- ◆ jeder Schreibvorgang erfordert auch das Schreiben des Paritätsblocks
- ◆ Erzeugung des Paritätsblocks durch Speichern des vorherigen Blockinhalts möglich: $P_{\text{neu}} = P_{\text{alt}} \oplus B_{\text{alt}} \oplus B_{\text{neu}}$ (P =Parity, B =Block)
- ◆ Schreiben eines kompletten Streifens benötigt nur einmaliges Schreiben des Paritätsblocks
- ◆ Paritätsplatte ist hoch belastet
(meist nur sinnvoll mit SSD [Solid state disk])

10.2 Einsatz mehrere redundanter Platten (5)

■ Verstreuter Paritätsblock (RAID 5)

- ◆ Paritätsblock wird über alle Platten verstreut



- ◆ zusätzliche Belastung durch Schreiben des Paritätsblocks wird auf alle Platten verteilt
- ◆ heute gängigstes Verfahren redundanter Platten
- ◆ Vor- und Nachteile wie RAID 4