4 Ersetzungsstrategie bei Solaris (6)

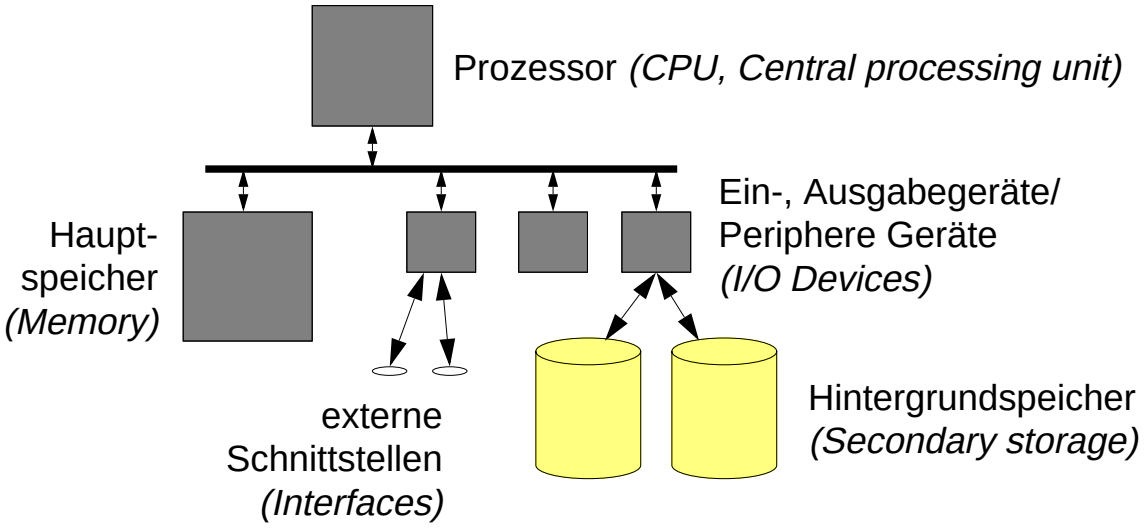
- Typische Werte
  - *minfree:* 1/64 des Hauptspeichers (Solaris 2.2), 25 Seiten (Solaris 2.4)
  - *desfree:* 1/32 des Hauptspeichers (Solaris 2.2), 50 Seiten (Solaris 2.4)
  - *lotsfree:* 1/16 des Hauptspeichers (Solaris 2.2), 128 Seiten (Solaris 2.4)
  - *deficit:* 0 bis *lotsfree*
  - *fastscan:* min(1/4 Hauptspeicher, 64 MByte) pro Sekunde (Solaris 2.4)
  - *slowscan:* 800 kBytes pro Sekunde (Solaris 2.4)
  - *handspread:* wie *fastscan* (Solaris 2.4)

E.9 Zusammenfassung

- Freispeicherverwaltung
  - Speicherrepräsentation, Zuteilungsverfahren
- Mehrprogrammbetrieb
  - Relokation, Ein- und Auslagerung
  - Segmentierung
  - Seitenadressierung, Seitenadressierung und Segmentierung, TLB
  - gemeinsamer Speicher
- Virtueller Speicher
  - Demand paging
  - Seitenersetzungsstrategien: FIFO, B_0, LRU, 2nd chance (Clock)
- Seitenflattern
  - Super-Zustände, Arbeitsmengenmodell

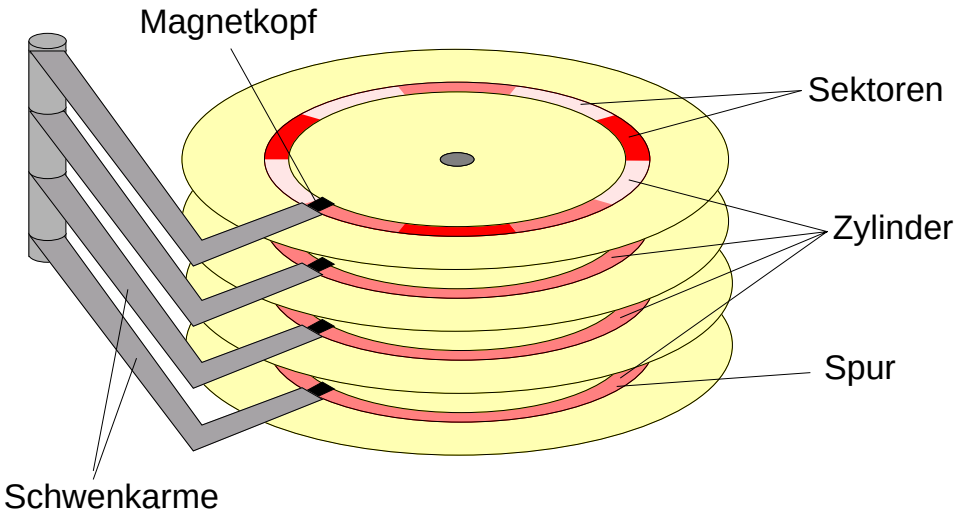
F Implementierung von Dateien

- Einordnung



F.1 Festplatten

- Häufigstes Medium zum Speichern von Dateien
  - ◆ Floppy Disks sind in der Handhabung ähnlich
- Aufbau einer Festplatte



F.1 Festplatten (2)

- Datenblätter zweier Beispielplatten

Plattentyp		Fujitsu M2344K	Fujitsu M2652
Kapazität		690 MB	2,0 GB
Zylinderzahl		624	1893
Spuren pro Zylinder		27	20
Sektoren pro Spur		1–128	
Positionierzeiten			
	Spur zu Spur	4 ms	2 ms
	mittlere	16 ms	11 ms
	maximale	33 ms	22 ms
Transferrate		2,458 MB/s	4,750 MB/s
Rotationsgeschw.		3.600 U/min	5.400 U/min
eine Plattenumdrehung		17 ms	11 ms

F.1 Festplatten (3)

- 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.)

F.2 Speicherung von Dateien

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

1 Kontinuierliche Speicherung

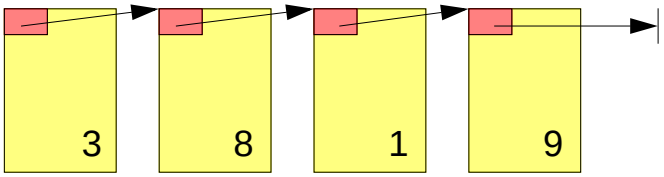
- Datei wird in Blöcken mit aufsteigenden Blocknummern gespeichert
  - Zugriff auf alle Blöcke mit minimaler Positionierzeit des Schwenkarms
  - Einsatz z.B. bei Systemen mit Echtzeitanforderungen
- Probleme
  - Finden des freien Platzes auf der Festplatte (Menge aufeinanderfolgender und freier Plattenblöcke)
  - Fragmentierungsproblem (Verschnitt: nicht nutzbare Plattenblöcke; siehe auch Speicherverwaltung)

1 Kontinuierliche Speicherung (2)

- Weiteres Problem
  - Größe bei neuen Dateien oft nicht im voraus bekannt
  - Erweitern ist problematisch
    - Umkopieren, falls kein freier angrenzender Block mehr verfügbar
- Variation
  - Unterteilen einer Datei in Folgen von Blocks *(Chunks, Extents)*
  - Blockfolgen werden kontinuierlich gespeichert
- Problem
  - Verschnitt innerhalb einer Folge

2 Verkettete Speicherung

■ Blöcke einer Datei sind verkettet



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

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

★ File kann wachsen und verlängert werden

2 Verkettete Speicherung (2)

▲ Probleme

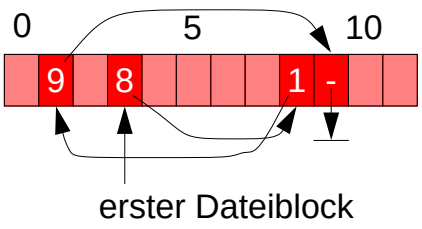
◆ Speicher für Verzeigerung geht von den Nutzdaten im Block ab (ungünstig im Zusammenhang mit Paging: Seite besteht immer aus Teilen von zwei Plattenblöcken)

◆ Fehleranfälligkeit: Datei ist nicht restaurierbar, falls einmal Verzeigerung fehlerhaft

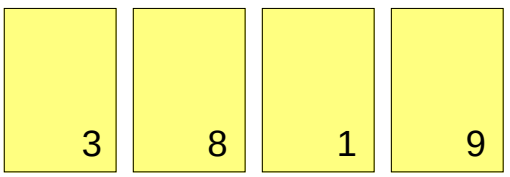
■ Verkettung wird in speziellen Plattenblocks gespeichert

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

FAT-Block



Blöcke der Datei: 3, 8, 1, 9



2 Verkette Speicherung (3)

★ Vorteile

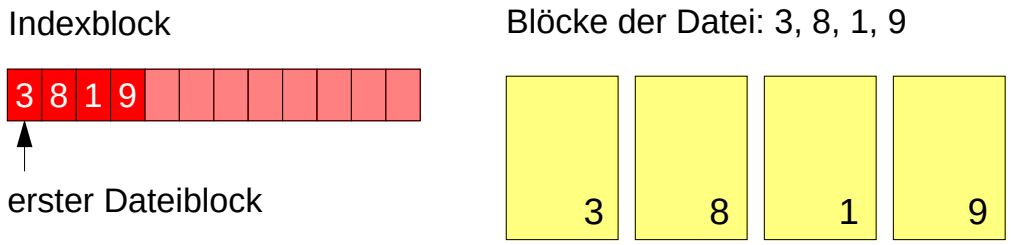
- ◆ kompletter Inhalt des Datenblocks ist nutzbar (günstig bei Paging)
- ◆ mehrfache Speicherung der FAT möglich: Einschränkung der Fehleranfälligkeit

▲ 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
- ◆ aufwendige Suche nach dem zugehörigen Datenblock bei bekannter Position in der Datei

3 Indiziertes Speichern

■ Spezieller Plattenblock enthält Blocknummern der Datenblocks einer Datei

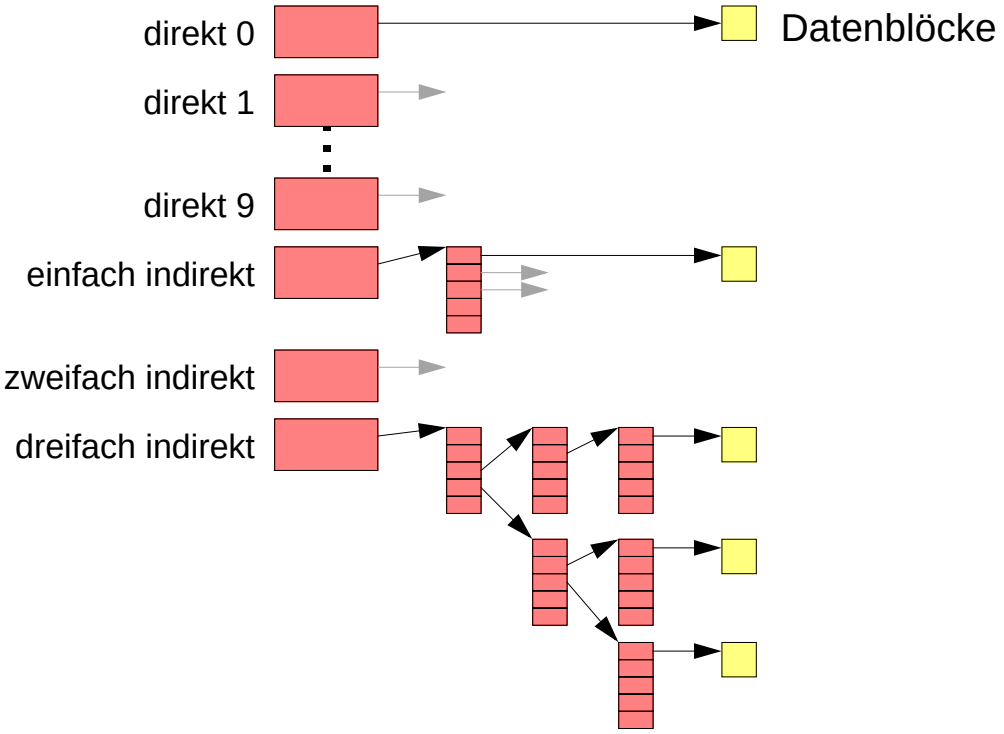


▲ Problem

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

3 Indiziertes Speichern (2)

- Beispiel UNIX Inode



3 Indiziertes Speichern (3)

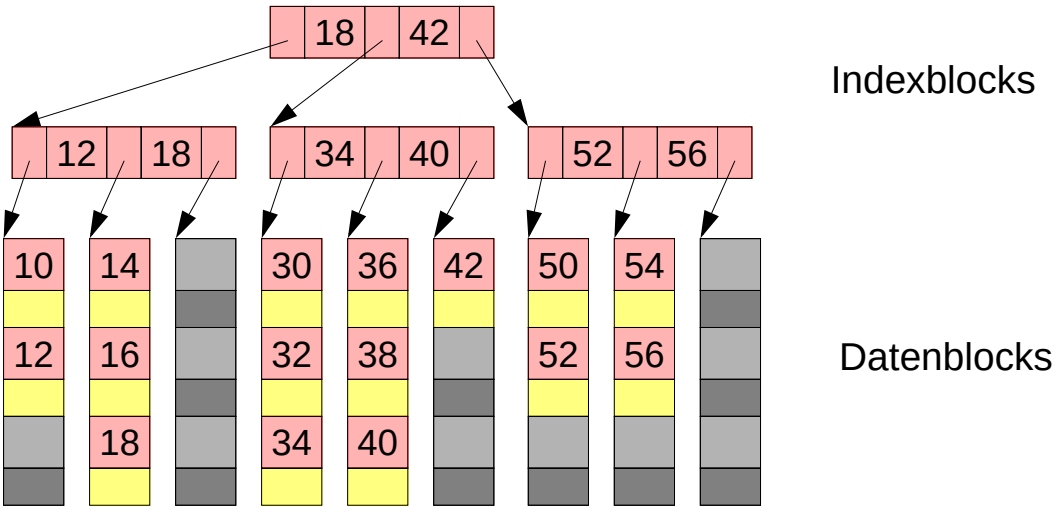
- ★ Einsatz von mehreren Stufen der Indizierung
  - ◆ Inode benötigt sowieso einen Block auf der Platte (Verschnitt unproblematisch bei kleinen Dateien)
  - ◆ durch mehrere Stufen der Indizierung auch große Dateien adressierbar
- ▲ Nachteil
  - ◆ mehrere Blöcke müssen geladen werden (nur bei langen Dateien)

4 Baumsequentielle Speicherung

- Satzorientierte Dateien
  - ◆ Schlüssel + Datensatz
  - ◆ effizientes Auffinden des Datensatz mit einem bekannten Schlüssel
  - ◆ Schlüsselmenge spärlich besetzt
  - ◆ häufiges Einfügungen 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

4 Baumsequentielle Speicherung (2)

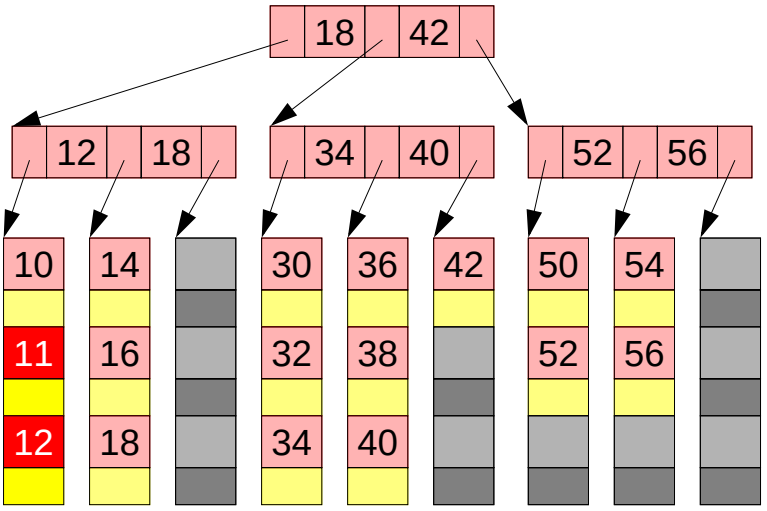
- Beispiel eines B*-Baums: Schlüssel sind Integer-Zahlen



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

4 Baumsequentielle Speicherung (3)

- Einfügen des Satzes mit Schlüssel „11“



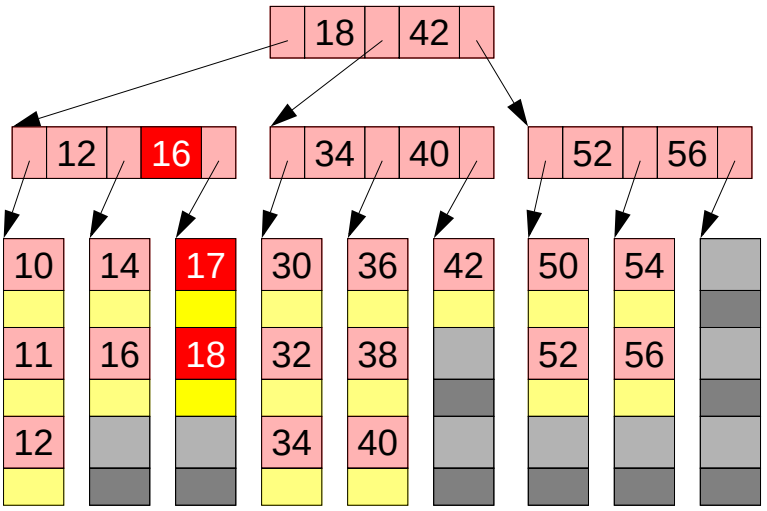
Indexblocks

Datenblocks

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

4 Baumsequentielle Speicherung (4)

- Einfügen des Satzes mit Schlüssel „17“



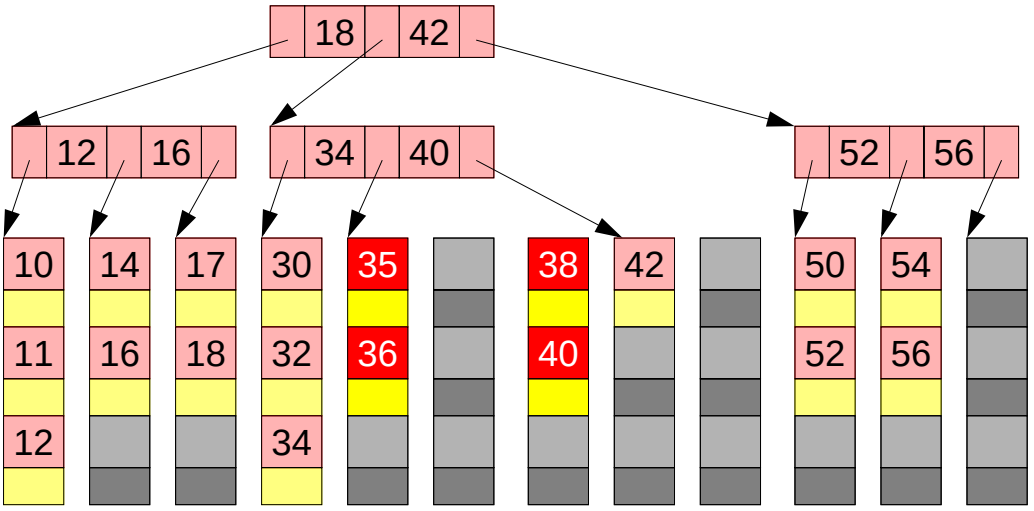
Indexblocks

Datenblocks

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

4 Baumsequentielle Speicherung (5)

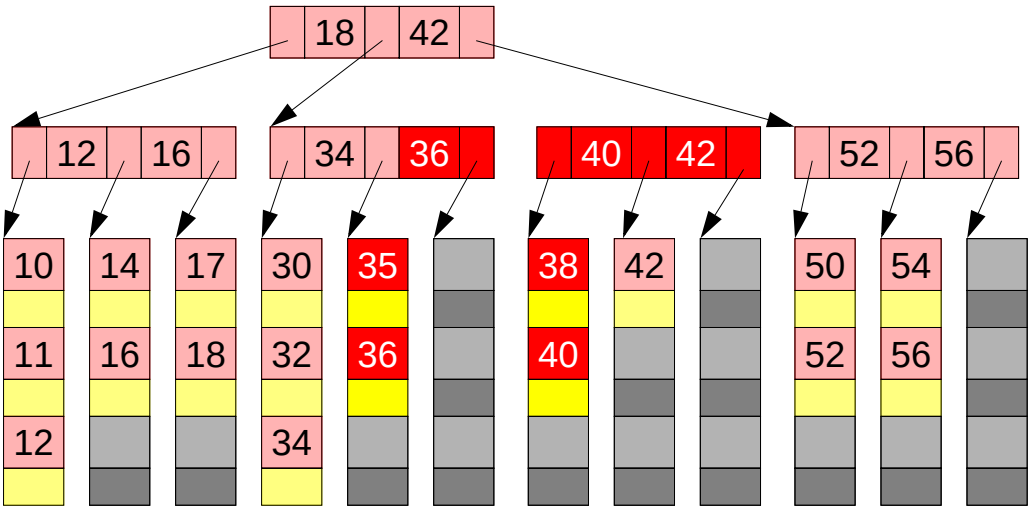
- Einfügen des Satzes mit Schlüssel „35“ (1. Schritt)



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

4 Baumsequentielle Speicherung (6)

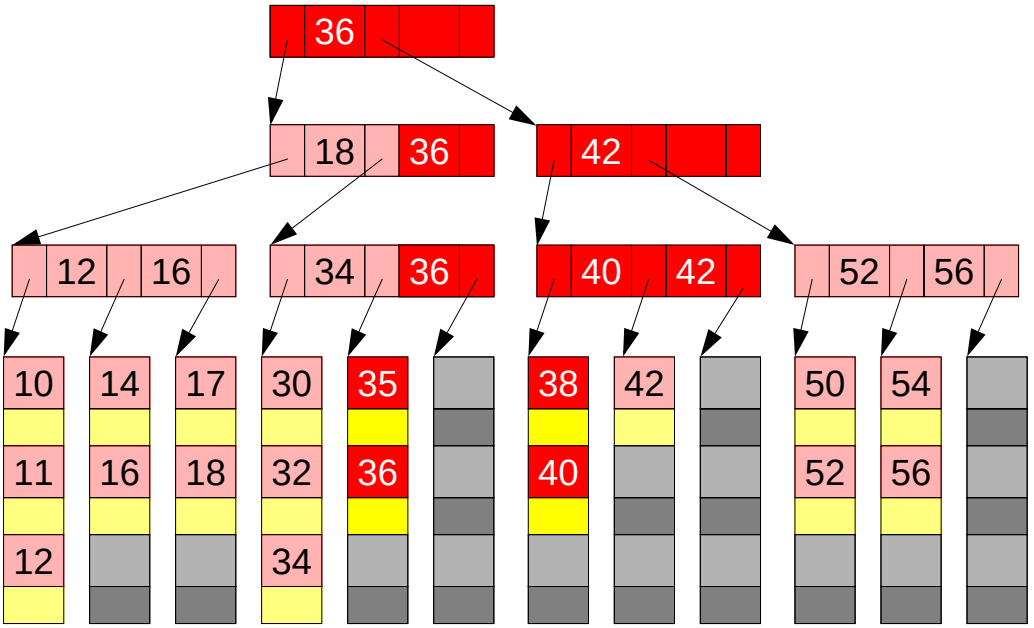
- Einfügen des Satzes mit Schlüssel „35“ (2. Schritt)



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

4 Baumsequentielle Speicherung (7)

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



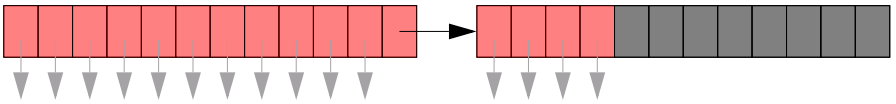
  - Spaltung des alten Wurzelknotens und Erzeugen eines neuer neuen Wurzel

4 Baumsequentielle Speicherung (8)

- ★ Effizientes Finden von Sätzen
  - Baum ist sehr niedrig im Vergleich zur Menge der Sätze
    - viele Schlüssel 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

F.3 Freispeicherverwaltung

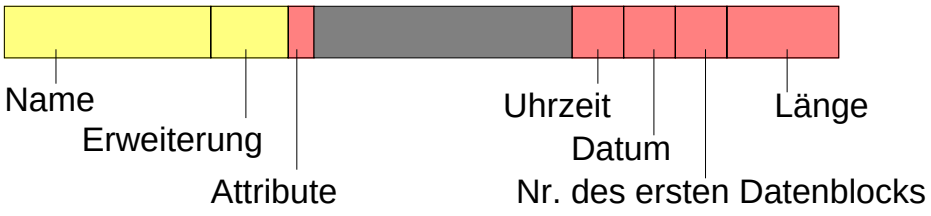
- prinzipiell ähnlich wie Verwaltung von freiem Hauptspeicher
  - Bitvektoren zeigen für jeden Block Belegung an
  - verkettete Listen repräsentieren freie Blöcke
    - Verkettung kann in den freien Blöcken vorgenommen werden
    - Optimierung: aufeinanderfolgende Blöcke werden nicht einzeln aufgenommen, 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



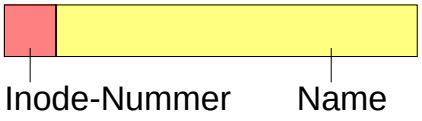
F.4 Implementierung von Katalogen

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*



1 Kataloge als Liste (2)

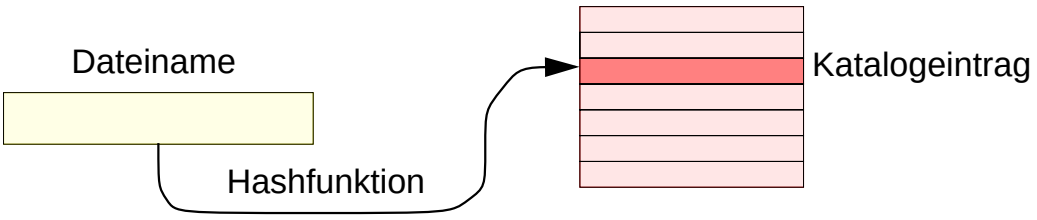
- Problem
  - Lineare Suche durch die Liste nach bestimmtem Eintrag
  - Sortierte Liste: binäre Suche, aber Sortieraufwand

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 = Länge der Katalogliste)

2 Einsatz von Hashfunktionen (2)

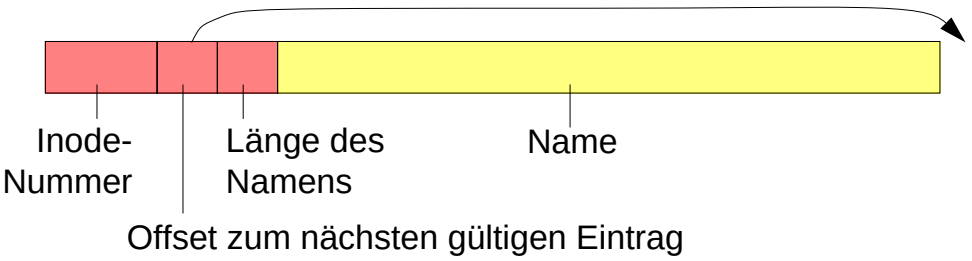
- Hashfunktion
  - Funktion bildet Dateinamen auf einen Index in die Katalogliste ab schnellerer Zugriff auf den Eintrag möglich (kein lineares Suchen)
  - (einfaches aber schlechtes) Beispiel: $(\sum \text{Zeichen}) \bmod N$



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

3 Variabel lange Listenelemente

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

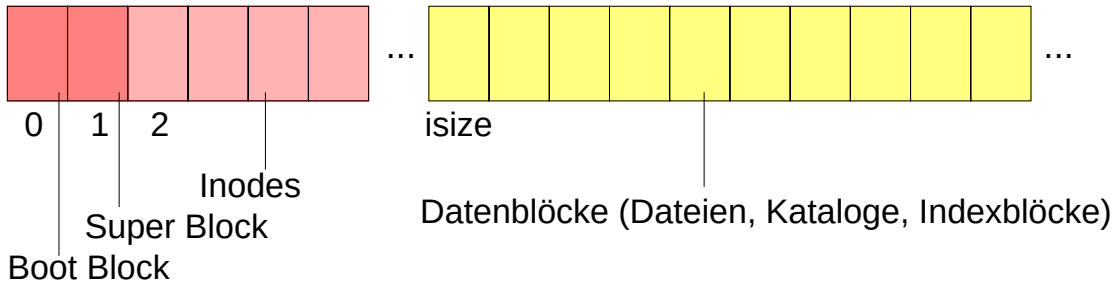


- Probleme
  - Verwaltung von freien Einträgen in der Liste
  - Speicherverschnitt (Kompaktifizieren, etc.)

F.5 Beispiel: UNIX File Systems

1 System V File System

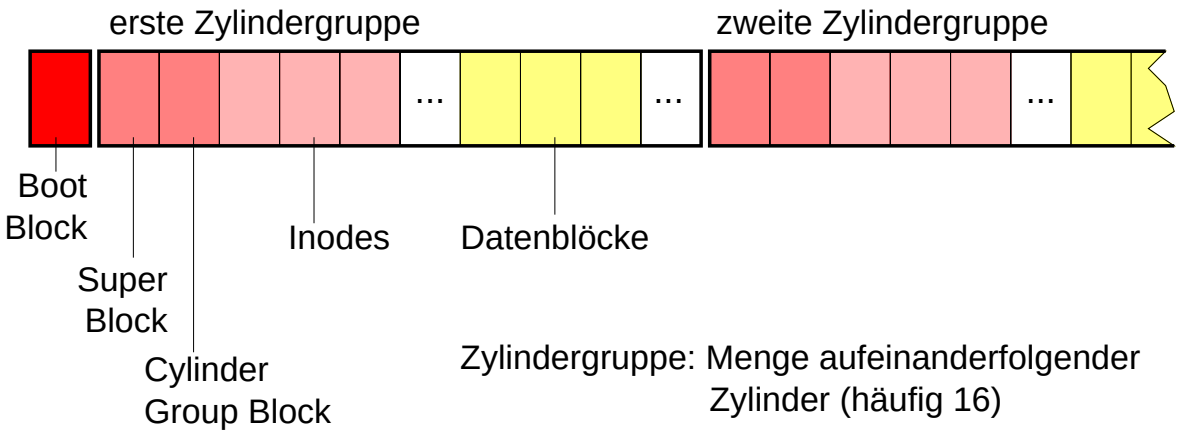
- Blockorganisation



  - 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
    - Anzahl und Liste freier Inodes
    - Attribute (z.B. *Modified flag*)

2 BSD 4.2 (Berkeley Fast File System)

- Blockorganisation



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

- ★ Vorteil: kürzere Positionierungszeiten

3 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)*

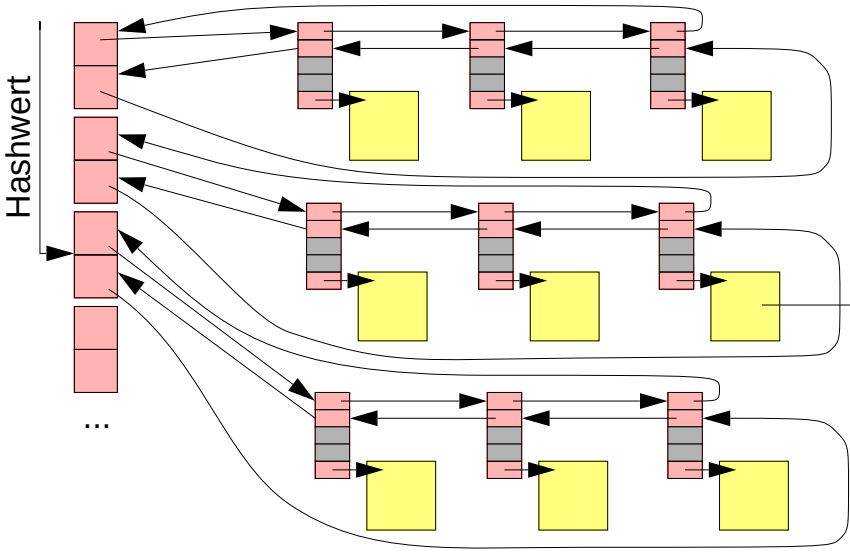
3 Block Buffer Cache (2)

- 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 Schreibaufruf 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

3 Block Buffer Cache (3)

- Aufbau des Block buffer cache

Pufferlisten (Queues)



Jede Liste ist vorwärts und rückwärts verkettet:

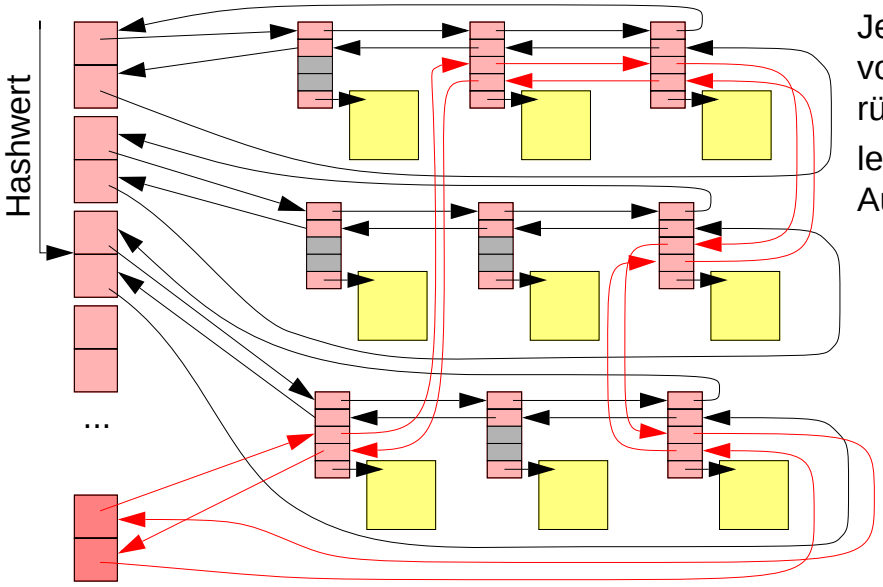
leichteres Ein- und Austragen

gepufferter Block

3 Block Buffer Cache (4)

- Aufbau des Block buffer cache

Pufferlisten (Queues)



Jede Liste ist vorwärts und rückwärts verkettet:

leichteres Ein- und Austragen

Freiliste

3 Block Buffer Cache (5)

- Block Buffer Cache teilweise obsolet durch moderne Pageing-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

4 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 );
```


4 Systemaufrufe (2)

- ◆ Kontrolloperation

```
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

F.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 erweiterbar
  - ◆ Rechte verknüpft mit NT Benutzern und Gruppen
    - *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*

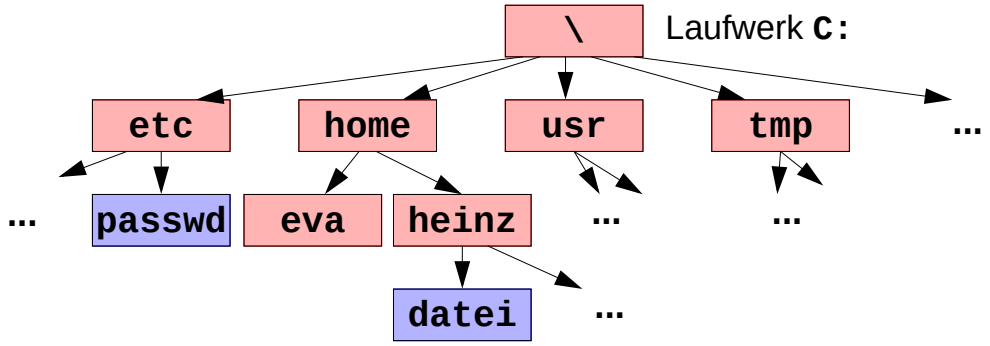
F.6 Beispiel: NTFS (2)

- ◆ 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

- 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 durch einen Laufwerksbuchstaben dargestellt z.B. **`C:`**

1 Pfadnamen

- Baumstruktur



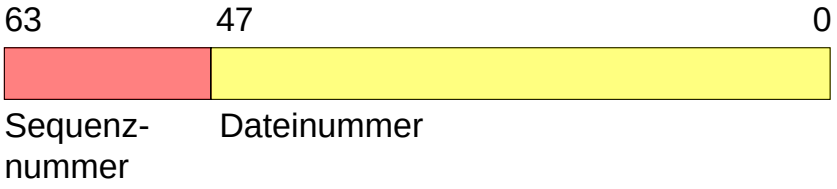
- Pfade
  - wie unter FAT File system
  - z.B. „**C:\home\heinz\datei**“, „**\tmp**“, „**C:..\heinz\datei**“

1 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

2 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 Datei
  - adressiert durch eine *File reference*



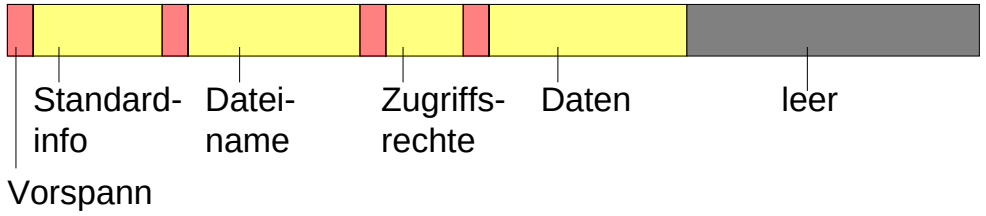
  - Dateinummer ist Index in eine globale Tabelle *(MFT: Master file table)*
  - Sequenznummer wird hochgezählt, für jede neue Datei mit gleicher Dateinummer

2 Dateiverwaltung (2)

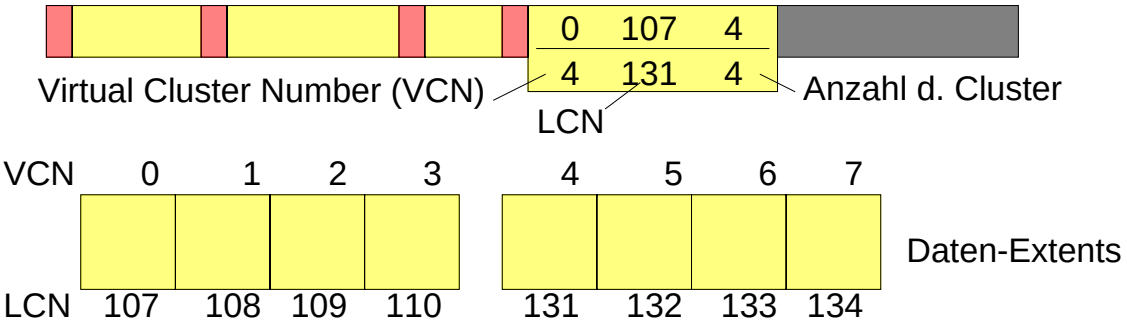
- Ströme
  - jede Datei kann mehrere Datenströme speichern
  - ein Datenstrom wird für die eigentlichen Daten verwendet
  - Dateiname, MS-DOS Dateiname, Zugriffsrechte, Attribute und Zeitstempel werden jeweils in eigenen Datenströmen gespeichert
    (leichte Erweiterbarkeit des Systems)
- Master file table
  - große Tabelle mit gleich langen Elementen (1KB, 2KB oder 4KB groß, je nach Clustergröße)
  - Index in die Tabelle ist Teil der *File reference*
  - entsprechender Eintrag für eine *File reference* enthält Informationen über bzw. die Ströme der Datei

3 Master File Table

- Eintrag für eine kurze Datei



- Eintrag für eine längere Datei



  - ◆ Extents werden außerhalb der MFT gespeichert

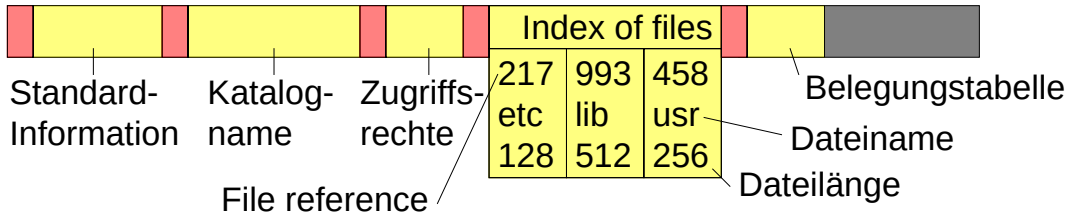
3 Master File Table (2)

- Mögliche Ströme *(Attributes)*
  - ◆ Standard Information (immer in der MFT)
    - enthält Länge, MS-DOS Attribute, Zeitstempel, Anzahl der Hard links
  - ◆ Dateiname (immer in der MFT)
    - kann mehrfach vorkommen (Hard links, MS-DOS Name)
  - ◆ Zugriffsrechte
    - *Security descriptor*
  - ◆ Daten
    - die eigentlichen Daten
  - ◆ Index
    - Index über einen Attributschlüssel (z.B. Dateinamen) implementiert Katalog
  - ◆ Indexbelegungstabelle

3 Master File Table (3)

- ◆ Attributliste (immer in der MFT)
  - wird benötigt, falls nicht alle Ströme in einen MFT Eintrag passen
  - referenzieren weitere MFT Einträge und deren Inhalt

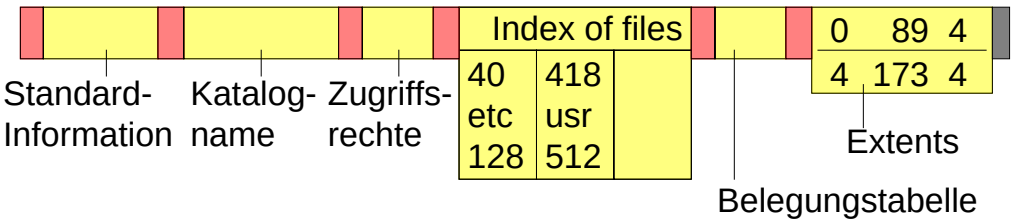
■ Eintrag für einen kurzen Katalog



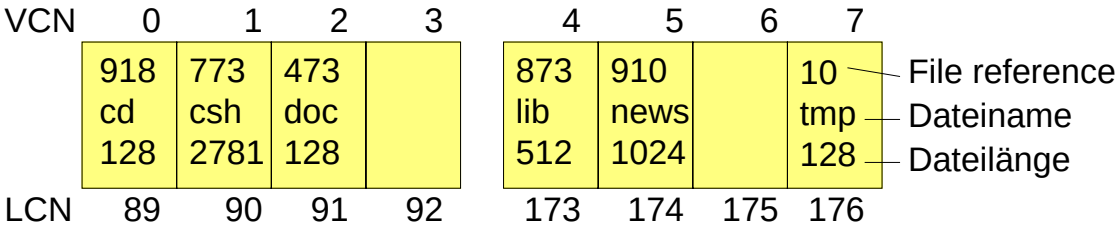
- ◆ Dateien des Katalogs werden mit File references benannt
- ◆ Name und Länge der im Katalog enthaltenen Dateien und Kataloge werden auch im Index gespeichert
  (doppelter Aufwand beim Update; schnellerer Zugriff beim Kataloglisten)

3 Master File Table (4)

■ Eintrag für einen längeren Katalog



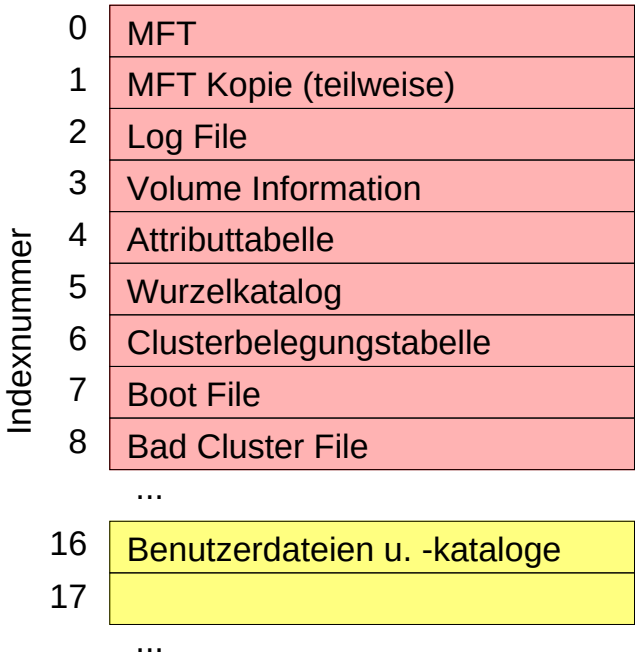
Daten-Extents



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

4 Metadaten

- Alle Metadaten werden in Dateien gehalten



Feste Dateien in der MFT

4 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
  - Attributtabelle: 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

5 Fehlererholung

- Mögliche Fehler
  - ◆ Stromausfall (dummer Benutzer schaltet Rechner aus)
  - ◆ Systemabsturz
- Auswirkungen auf das Dateisystem
  - ◆ inkonsistente Metadaten
    z.B. Katalogeintrag fehlt zur Datei oder umgekehrt; Block ist benutzt aber nicht als belegt markiert
    - Programme wie **chkdsk** oder **fsck** können inkonsistente Metadaten reparieren
    - Datenverluste möglich
  - ◆ unvollständige Daten in Dateien

5 Fehlererholung (2)

- ★ Log structured file system
  - ◆ Protokollieren aller Änderungen am Dateisystem in einer Protokolldatei *(Log file)*
  - ◆ Änderungen treten als Teil von Transaktionen auf
    - Erzeugen, löschen, erweitern, verkürzen von Dateien
    - Dateiattribute verändern
    - Datei umbenennen
  - ◆ Schreiben der Logdatei bevor die Änderungen auf Platte geschrieben werden (wurde etwas auf Platte geändert steht auch das Protokoll auf Platte)
  - ◆ Beim Bootvorgang wird überprüft, ob die protokollierten Änderungen vorhanden sind:
    - Transaktion kann evtl. wiederholt bzw. abgeschlossen werden *(Redo)*
    - angefangene aber nicht beendete Transaktionen werden rückgängig gemacht *(Undo)*

5 Fehlererholung (3)

- Beispiel: Löschen einer Datei
  - ◆ 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*

5 Fehlererholung (4)

- ◆ 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: rückwärts werden alle Operation rückgängig gemacht

- Checkpoints
  - ◆ Log file ist nicht unendlich groß
  - ◆ 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

5 Fehlererholung (5)

★ Ergebnis

- ◆ eine Transaktion ist entweder vollständig durchgeführt oder gar nicht
- ◆ Benutzer kann ebenfalls Transaktionen über mehrere Dateizugriffe definieren (?)
- ◆ keine inkonsistente 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

- ◆ etwas ineffizienter
- ◆ nur für Volumes >400 MB geeignet

F.7 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)

F.8 Fehlerhafte Plattenblöcke

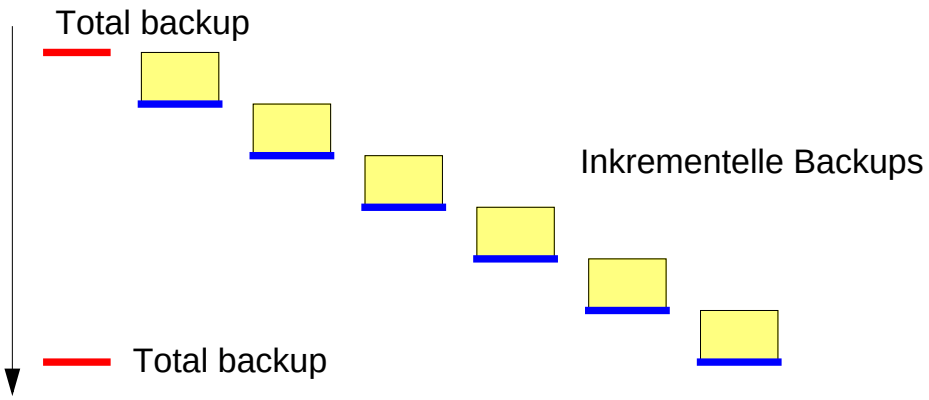
- Blöcke, die beim Lesen Fehlermeldungen erzeugen
  - z.B. Checksummenfehler
- Hardwarelösung
  - Platte und Plattencontroller bemerken selbst fehlerhafte Blöcke und maskieren diese aus
  - Zugriff auf den Block wird vom Controller automatisch auf einen „gesunden“ Block umgeleitet
- Softwarelösung
 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

F.9 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

1 Beispiele für Backup Scheduling

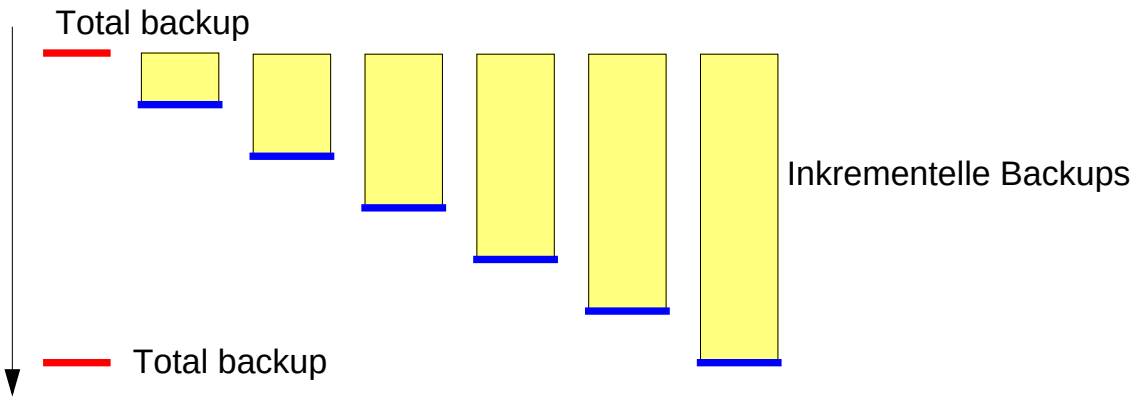
- Gestaffelte inkrementelle Backups



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

1 Beispiele für Backup Scheduling (2)

- Gestaffelte inkrementelle Backups zum letzten Total backup

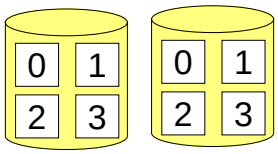


  - z.B. alle Woche ein Total backup und jeden Tag ein inkrementelles Backup zum letzten Total backup: maximal 2 Backups müssen eingespielt werden

- Hierarchie von Backupläufen
  - mehrstufige inkrementelle Backups zum Backup der nächst höheren Stufe
  - optimiert Archivmaterial und Restaurierungszeit

2 Einsatz mehrere redundanter Platten

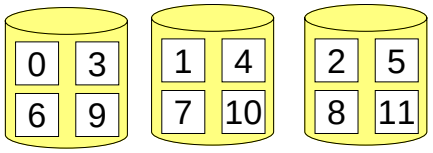
- Gespiegelte Platten (*Mirroring*; RAID 0)
  - Daten werden auf zwei Platten gleichzeitig gespeichert



  - 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

2 Einsatz mehrere redundanter Platten (2)

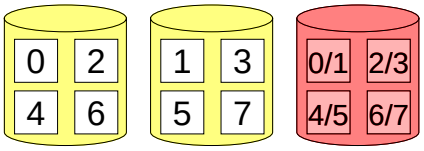
- Gestreifte Platten (*Striping*; RAID 1)
  - Daten werden über mehrere Platten gespeichert



  - 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)

2 Einsatz mehrere redundanter Platten (3)

- Paritätsplatte (RAID 4)
  - Daten werden über mehrere Platten gespeichert, eine Platte enthält Parität



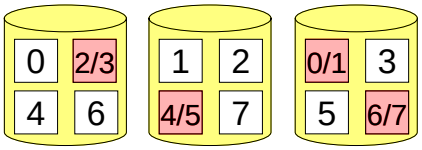
  - Paritätsblock enthält byteweise XOR-Verknüpfungen von den zugehörigen Blöcken aus den anderen Streifen
  - eine Platte kann ausfallen
  - schnelles Lesen
  - prinzipiell beliebige Plattenanzahl (ab drei)

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: $P_{neu} = P_{alt} \oplus B_{alt} \oplus B_{neu}$ (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])

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
- Problem
  - fehlerhafter Paritätsblock zerstört mehrere Blöcke, falls eine Platte ausfällt

G Ein- und Ausgabe

- Einordnung