M15$_{red}$ Dateisystem (Grundlagen)

Voraussetzungen

Ziele

Verständnis von logischer Sicht der Dateinutzung und physischer Realisierung

Inhalt

- Dateidefinition und –charakteristiken
- Arbeiten mit Dateien und Filedeskriptoren (UNIX)
- physischer Plattenzugriff
- Datenträgerverwaltung
- Dateiverwaltung
- FAT
- i-nodes
- Verzeichnisse
- Links

M15$_{red}$ Dateien und Speichermedien

- Dateien sind Datencontainer.
- Die Daten werden langlebig (persistent) gespeichert.
- Speichermedien sind magnetische (HD, FD), magneto-optische (MO) oder optische (DVD) Platten.
- Speichermedien sind schreib- und lesbar (HD, FD, CD-RW) oder nur lesbar (CD-ROM).
- Massenspeicher (HD, CD)
  - Kapazitäten im Bereich ~ 100 MB bis ~ GB
  - Zugriffszeiten 5 bis 15 ms (wegen Mechanik).
  - Sehr günstiges Preis-Leistungsverhältnis.

M15$_{red}$ Dateicharakteristiken

- Dateien sind meist klein (≈ KB).
- Dateien werden häufig gelesen, seltener geschrieben, noch seltener gelöscht.
- Sequentieller Zugriff ist dominant.
- Selten gleichzeitige Benutzung durch mehrere Programme.
- Aber: Nutzungscharakteristiken ändern sich (Multimedia):
  - sehr große Dateien,
  - hohe Übertragungsraten, Echtzeit (Audio, Video)

M15$_{red}$ Zahlenbeispiele

- Unkomprimierte Videoaufzeichnung

  1024 x 768, 3B/Pixel, 50 Bilder/s:

  Übertragungsrate = 112,5 MB/s
- Audiodatei, CD-Qualität

  10 MB/min

M15$_{red}$ Arbeiten mit Dateien

- Damit man mit einer Datei arbeiten kann, muss sie

  erzeugt **`creat`**

  geöffnet **`open`**

  und anschließend wieder

  geschlossen **`close`**

  werden.

- Das Erzeugen und Öffnen resultiert in einem Filedeskriptor **`fd`**, welcher im weiteren Verlauf benutzt wird, um die Datei anzusprechen.

```
fd = creat („dateiname", ooo)
fd = open („dateiname", flags, ooo)
ooo : Schutzbits
```


M15$_{red}$ Lesen und Schreiben von Dateien (1)

- Schreiben und Lesen

```
n = write (fd, buf, nbyte)
n = read (fd, buf, nbyte)
```

  Schreibt in das File **`fd`** bzw. liest aus dem File **`fd`**.

  Quelle / Ziel: Puffer **`buf`**.

  Anzahl Bytes: **`nbyte`**.

  Anzahl tatsächlich gelesener Bytes: **`n`**.

M15$_{red}$ Lesen und Schreiben von Dateien (2)

- Ein Lese- oder Schreibzugriff auf eine Datei erfolgt Byte-sequentiell, beginnend mit der Position, auf welche der Filepointer zeigt. Er kann mit **`lseek`** positioniert werden.
- **`lseek (fd, offset, whence)`**

  positioniert den zu **`fd`** gehörenden Pointer entsprechend **`offset`** und **`whence`**, wobei gilt

  `whence` = SEEK_SET: pointer = **`offset`** bytes

  `whence` = SEEK_CUR : pointer = pointer + **`offset`**

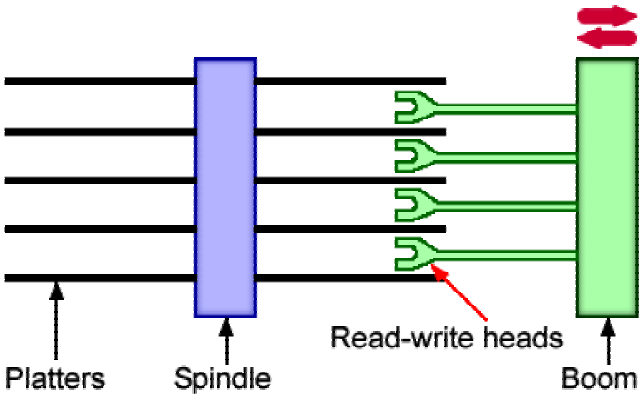
  `whence` = SEEK_END : pointer = file size + **`offset`**

M15$_{red}$ Informationen über Dateien

- Shellkommando **`ls -l`** gibt die Dateien des aktuellen Verzeichnisses zusammen mit ihren Attributen aus.
- Das Programm **`ls`** verwendet dazu die Systemaufrufe **`stat()`** oder **`fstat().`**
- **`stat`**/**`fstat`** greift auf interne Verwaltungsinformation (i-node) zu.

$M15_{red}$ Plattenzugriff

- Daten und Programme können im (virt.) Adressraum oder in Files (Dateien) gespeichert werden.
- Files werden i.d.R. auf einem nicht-flüchtigen Speicher (Floppy Disk FD, Hard Disk HD, Band) gehalten, heute meist HD. Abspeicherung in Blöcken der Größe B.
- Aufbau einer HD:



- Angaben zur Beurteilung von Massenspeichersystemen:
  - Zugriffszeit
  - Bandbreite (Datenrate)

$M15_{red}$ Datenträgerverwaltung (1)

- Problem: Verwaltung freier und belegter Blöcke
- Lösung a) Liste freier Blöcke (Free Disk Block List) belegt mehrere Blöcke

Beispiel

B = 1 KB ; Plattengröße = 1 GB (2^{20} Blöcke)

→ Disk-Blocknummer: 20 Bit (→ 32 Bit)

1 Block: 256 Block Nummern

nötig: 2^{12} Listenblöcke, 4 MB, 4 $^{o}/_{oo}$

- Lösung b) Bit Map

Pro Disk Block : 1 Bit

0 : frei

1 : besetzt

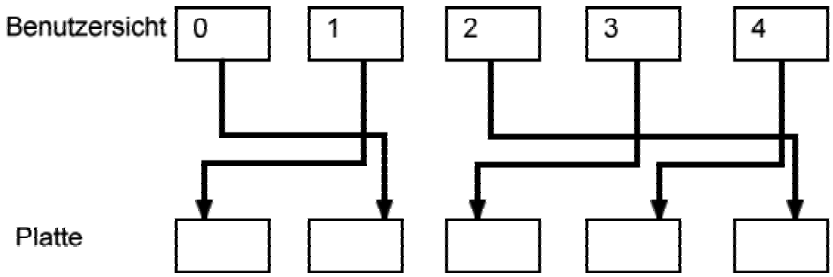
Beispiel

B = 1 KB, Plattengröße = 1 GB

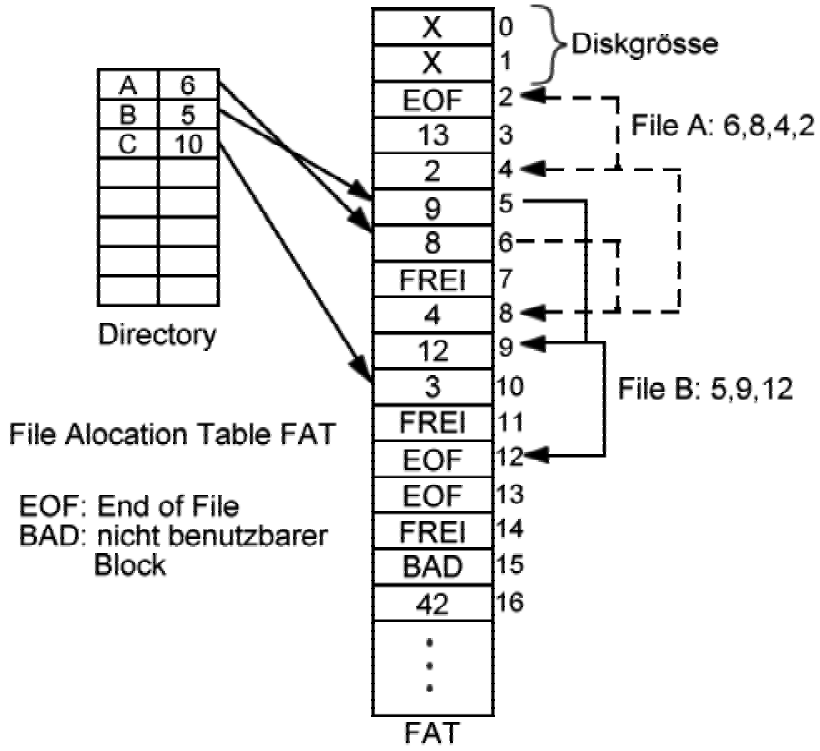
2^{20} Blöcke → 2^{20} Bit = 128 KB (0.1 $^{o}/_{oo}$)

- Bewertung
  - Bit Map braucht weniger Platz, aber benötigt u.U. mehrere Plattenzugriffe, bis freier Block gefunden.
  - Liste freier Blöcke: mehr Platzbedarf, schneller: 1 neuer Listenblock enthält 256 freie Blockadressen

- Abbildung der Benutzersicht auf die Blockanordnung auf Platte



- Problem: Zuordnung der File-Blöcke zu Diskblöcken
- Lösung 1: Verkettete Liste, Zeiger im Block (letztes und vorletztes Byte)
  - Blockgröße $\neq 2^n$ Bytes
  - Random-Suche erfordert einen Plattenzugriff pro Listenelement.
- Lösung 2: Verkettete Listen (MS-DOS):
  File Allocation Table (FAT)



- FAT-Umfang:

  a) FD 320 KB, Blockgröße = 1 KB, 12-bit-Blocknummern

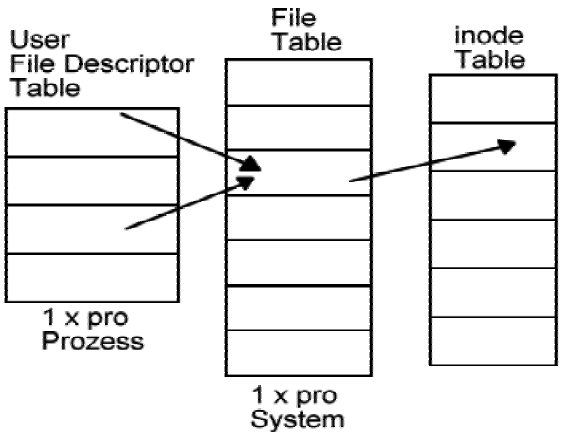
  → FAT-Umfang = 480 Bytes

  b) HD 600 MB, Blockgröße = 1 KB, 20-bit-Blocknummern

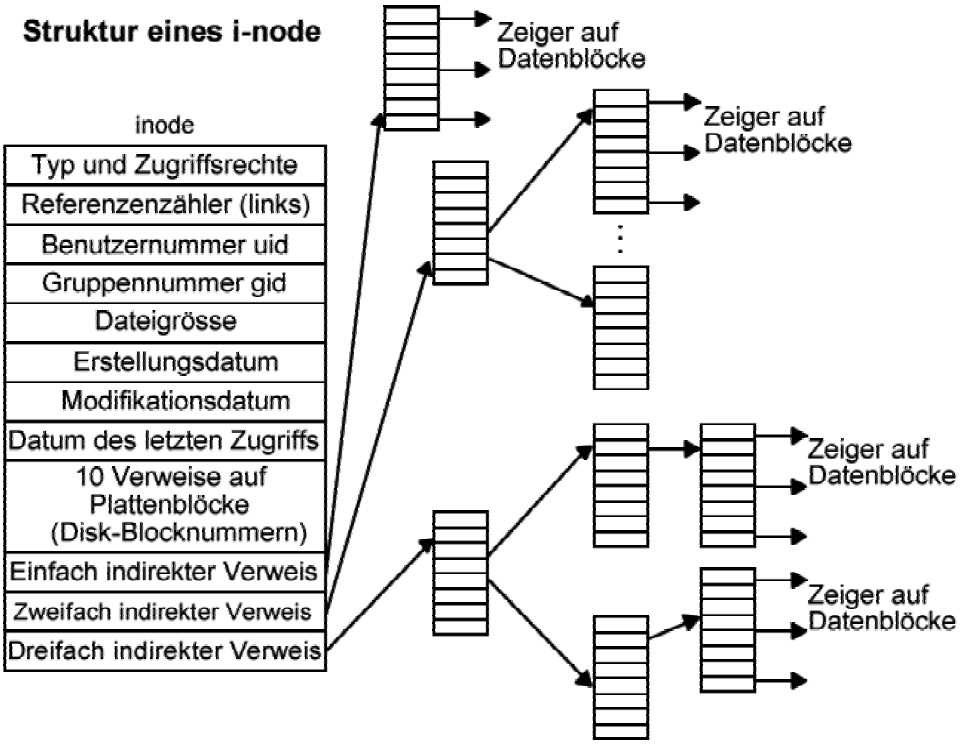
  → FAT-Umfang = 1.54 MB (zu groß für Hauptspeicher)

M15$_{red}$ UNIX-Lösung: inode

- Ein inode (index node) ist eine interne Repräsentation einer Datei.
  Er enthält Information über Besitzer, Zugriffsrechte und –zeiten sowie über den (psysikalischen) Ort der zugehörigen Datenblöcke auf der Platte.
- Ein Dateizugriff mittels Dateinamen *path* führt über Zwischenschritte zum inode der Datei. Außerdem werden Verweise in diversen Systemtabellen eingetragen.



M15$_{red}$ Inode - Struktur



M15$_{red}$ Datei-Größe (1)

- Für kleine Files (F ≤ 10*B) : Platz für 10 Disk-Blocknummern direkt im inode (mit B = Blockgröße).
- Wenn F > 10*B : „Single indirect“ zeigt auf Erweiterungsblock .0 (Größe B), welcher Blocknummern enthält.
- Beispiel

  B = 1 KB, Länge (Blocknummer) = 4 B

  Erweiterungsblock erschließt zusätzliche 256 KB (+ 10 aus inode).

M15$_{red}$ Datei-Größe (2)

- Wenn F > 266 KB : „Double indirect“ zeigt auf Erweiterungsblock .1 (Größe B), dessen Einträge zeigen auf n Erweiterungsblocke .2
- Beispiel

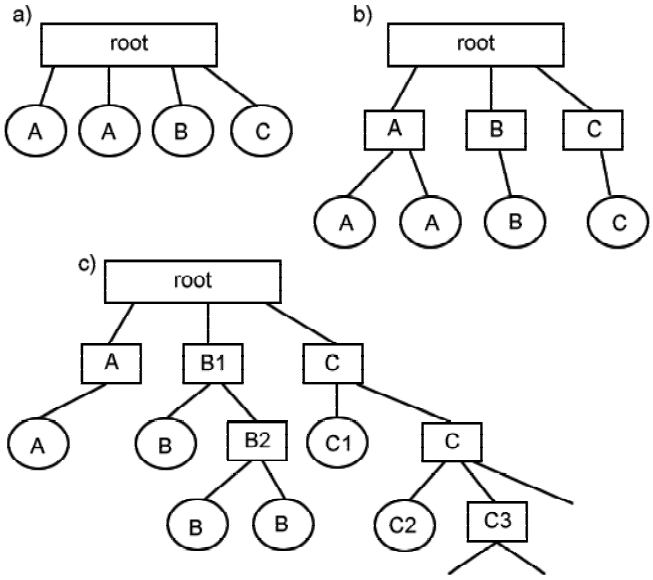
  B = 1 KB

  „Double indirect“ erschließt zusätzlich 256*256 KB

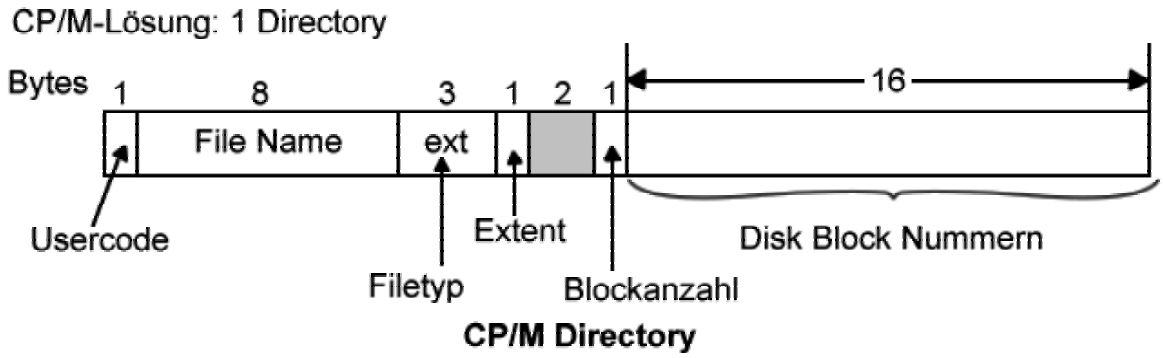
  Gesamt: $(266 + 256^2)*B$
- Wenn F > 65802 KB ➔“Triple indirect“

  F ≤ 16.8 GB

- Verzeichnisorganisation:
  - a) 1 Verzeichnis, linear für alle Benutzer (CP/M)
  - b) 1 Verzeichnis / Benutzer
  - c) beliebige benutzerspezifische Verzeichnisse

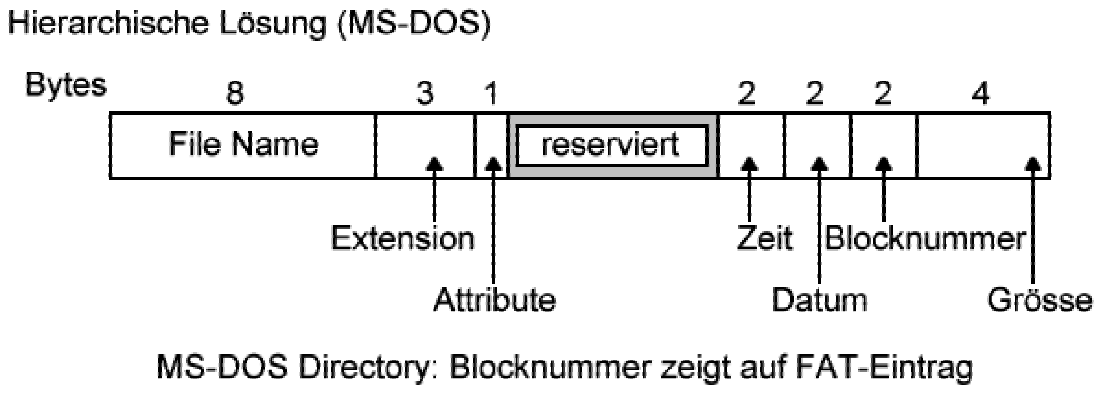


- Systemaufruf: `open (pathname, ...)`
  Aktion im BS: durchsuche alle Verzeichnisse nach `pathname`, liefere zugehörige Blocknummern (oder inode-Nummer).



Usercode: erlaubt beim Suchen Unterscheidung nach Benutzern

Extent: zeigt an, dass File > 16 Blöcke hat -> File besitzt zusätzlichen Eintrag im Directory



MS-DOS Directory: Blocknummer zeigt auf FAT-Eintrag

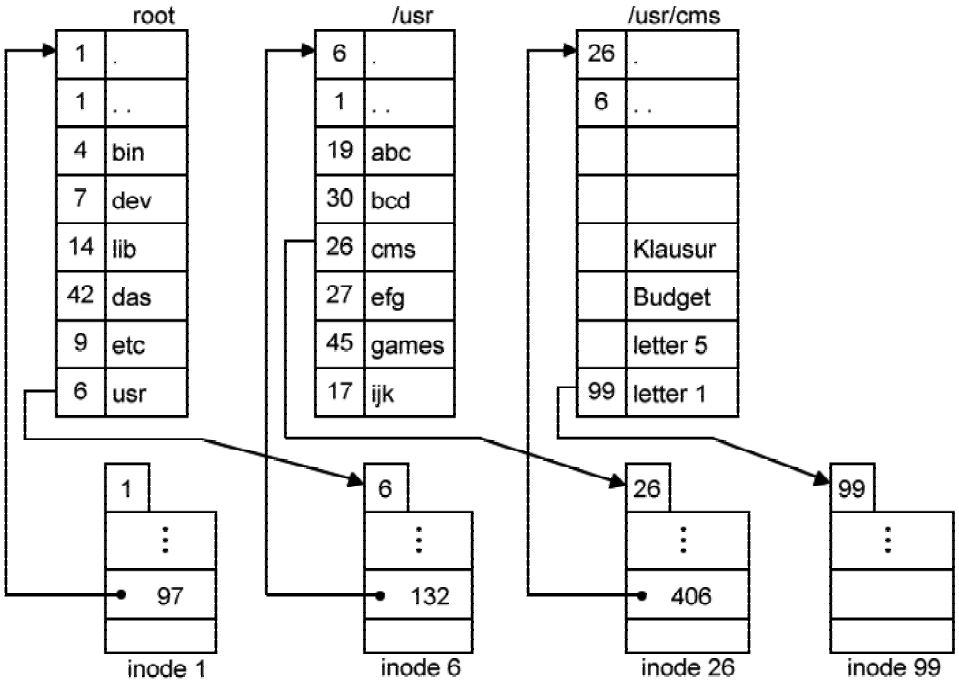
- Ein UNIX-Verzeichnis ist eine Zuordnung von inode-Nummern zu Datei-/ Verzeichnisnamen.

Verzeichnisnamen

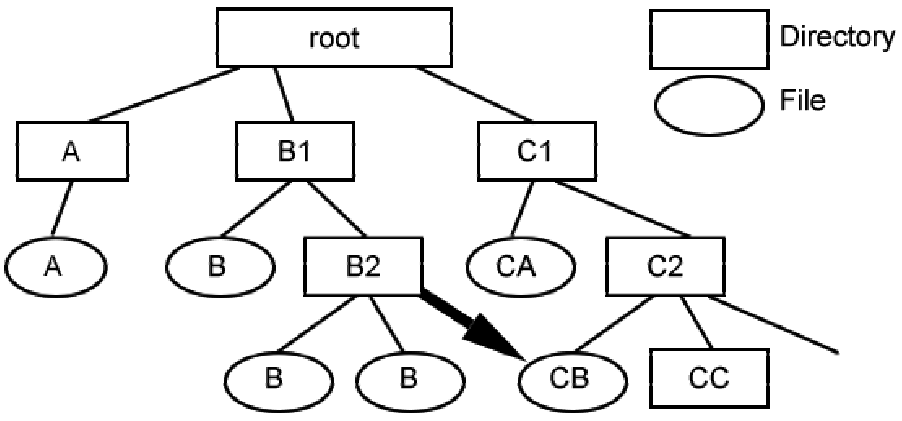
Inode-Nr.	Datei-/Verz.name
2 byte	14 byte

- Die Übersetzung eines Pfadnamens (Algorithmus **`namei`**) erfolgt schrittweise, beginnend mit dem Working Directory oder /. Nach der Suche wird ein File-Deskriptor zugeordnet.

- Beispiel: Zugriff auf File /usr/cms/letter1 mit der inode-Nr. 99



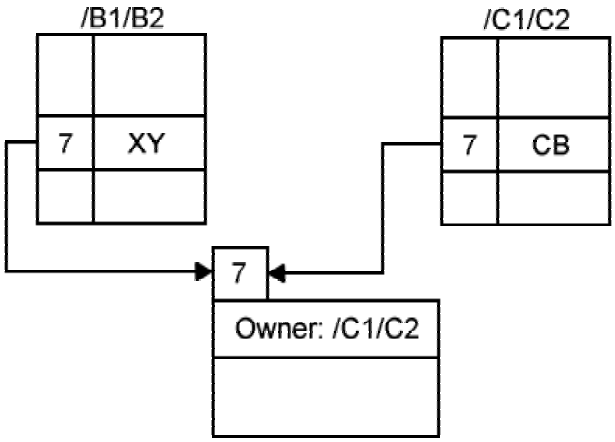
- Wunsch: Außer dem Besitzer eines Files sollen weitere Benutzer (über ihre eigenen Directories) darauf zugreifen können.
- Lösung 1: Link



File-Baum mit gesetztem Link (von B2 auf CB)

M15$_{red}$ Gemeinsame Files (2)

- Ein Link vom Directory B2 auf das File CB wird realisiert durch Eintrag der inode-Nummer von CB in B2. Der in B2 benutzte Name muss nicht gleich dem in C2 benutzten sein!



Link-Realisierung durch inode-Einträge

- Aus der Baumstruktur wird ein gerichteter azyklischer Graph.

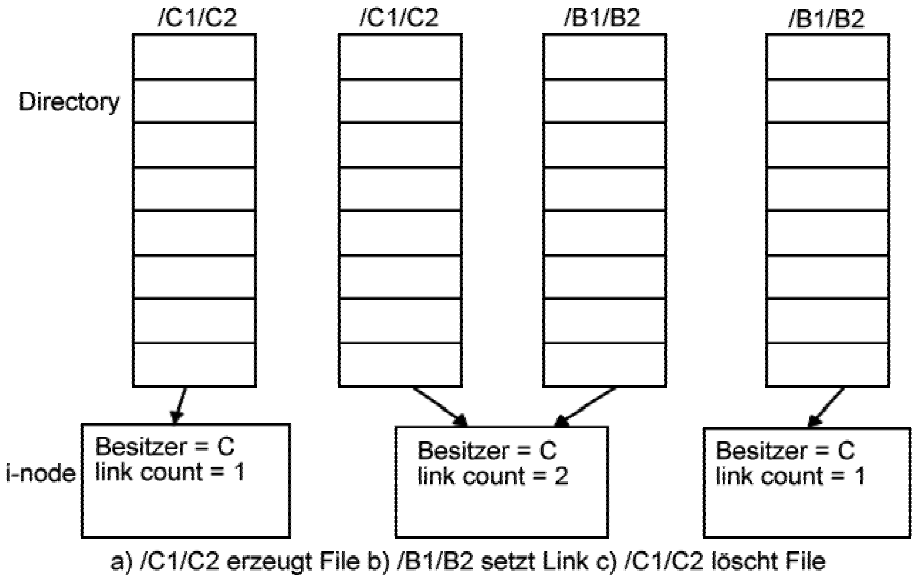
M15$_{red}$ Gemeinsame Files (3)

- Problem: /C1/C2 will File CB löschen:

  Directory-Eintrag wird aus /C1/C2 gelöscht, Link-Zähler (link count) im inode wird dekrementiert.

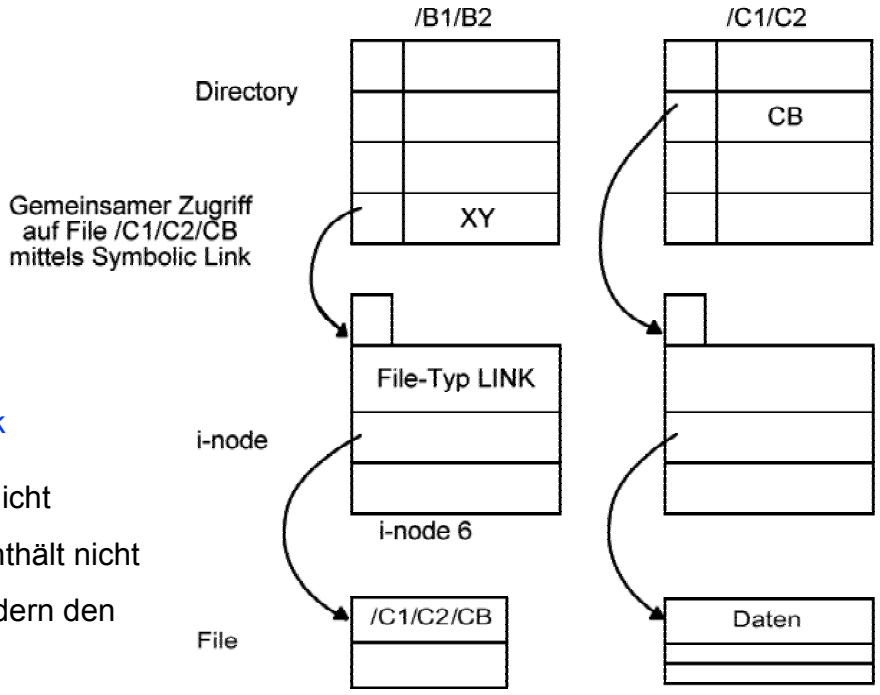
  Nur wenn link count = 0, werden inode und File gelöscht.

- Folge: B2 hat allein Zugang zu CB, der bei CB eingetragene Besitzer nicht. (Gebühren!)



a) /C1/C2 erzeugt File b) /B1/B2 setzt Link c) /C1/C2 löscht File

- Bereinigung würde Rückwärtszeiger oder Suche erfordern!



- Lösung 2: Symbolic Link

  Ein Directory, welches nicht Besitzer des Files ist, enthält nicht die inode-Nummer, sondern den Pfadnamen.

- Problem: Symbolic Link ist aufwendiger.
- Vorteil: Symbolic Link kann auch Maschinen- und Systemgrenzen überschreiten.