Betriebssysteme I – Sommersemester 2009
Kapitel 6: Speicherverwaltung und Dateisysteme

Hans-Georg Eßer
Hochschule München

Teil 3: Zusammenhängende Speicherzuordnung

06/2009

Naiver Ansatz

Blöcke und Seiten

- Aufteilung der Festplatte in Blöcke gleicher Größe (z. B. 1 KByte)
- analog: Aufteilung des Hauptspeichers in Seiten gleicher Größe
- Verwalten einer Zuordnung
  - Datei → Blockliste
  - Prozess → Seitenliste
- 1. Problem: Zuordnungstabelle wird sehr groß
  - Datei der Größe 1 GByte besteht aus 1 M Blöcken
  - Prozess mit 1 GByte Speicherbedarf nutzt 1 M Seiten

Probleme im Dateisystem

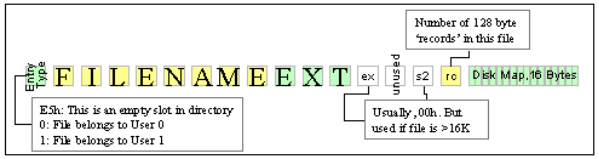
- (Größe der Zuordnungstabelle)
- sehr starke Fragmentierung (einzelne Blöcke ggf. wie zufällig über Platte verteilt)
- sequentieller Dateizugriff (von vorne nach hinten): stark verlangsamt
- wahlfreier/zufälliger Dateizugriff: nicht betroffen
- Verwaltung des freien Plattenplatzes: große Bitmap nötig

Probleme im Hauptspeicher

- (Größe der Zuordnungstabelle)
- sehr starke Fragmentierung (einzelne Seiten ggf. wie zufällig über RAM verteilt)
- Probleme der Code-Verschiebung und des Speicherschutzes (vgl. zus.-hgd. Speicherung) noch verstärkt
- Verwaltung des freien Plattenplatzes: große Bitmap nötig
- RAM oft zu klein für Prozess (Programm + Daten) oder Gesamtmenge der Prozesse

Beispiel CP/M-Dateisystem (1/3)

- typischer Datenträger: 180-KByte-Diskette
- FAT mit 32 Byte pro Datei



- Blockgröße 1 KByte
- Ein FAT-Eintrag enthält bis zu 16 Blockadressen $\Rightarrow$ 16 KByte Dateigröße
- größere Dateien: bis zu 15 zusätzliche FAT-Einträge („Extents“) $\Rightarrow$ 16 $\times$ 16 KByte = 256 KByte

Beispiel CP/M-Dateisystem (2/3)

CP/M-Dateisystem skaliert schlecht:

- Beispiel: Platte mit 20 GByte, Wunschdateigröße 1 GByte
- Größe eines FAT-Eintrags wächst, weil Anzahl der Blockadressen steigt
- Blockgröße 1 KB: Maximalgröße unverändert
- Mögliche Änderungen:
  - Mehr Extents: 1 GByte / 256 KByte = 65536 FAT-Einträge für eine Datei!
  - Größere Blockgröße: Bei Blockgröße 4 MByte geht es – zu groß, enorme interne Fragmentierung
  - Größere FAT-Einträge (mehr Blockadressen)

Beispiel CP/M-Dateisystem (3/3)

Problem mit sehr großen FAT-Einträgen:

- Alle FAT-Einträge gleich groß
  $\Rightarrow$ auch großer Eintrag für kleine Datei
- $\Rightarrow$ im Mittel Hälfte der FAT ungenutzt

Beispiel Speicherverwaltung (1/3)

- Zum leichteren Rechnen: RAM-Größe 100.000, Seitengröße 1.000 (also 1000 Seiten)
  Seite 0: 0–999, Seite 999: 999.000–999.999
- Prozess 1 benötigt 3.500 Byte und erhält die Speicherseiten 9, 12, 104 und 401,
- kann also die physikalischen Adressen
  - 009.000–009.999,
  - 012.000–012.999,
  - 104.000–104.999 und
  - 401.000–401.999

  nutzen

Beispiel Speicherverwaltung (2/3)

- Code-Verschiebung und Speicherschutz müssen Aufteilung des Speichers berücksichtigen:
  - Ist Programmcode z. B. 1500 Byte lang, muss Computer das Programm nach Ausführung des Befehls an logischer Position 999 (Ende 1. Seite) an einer anderen Stelle fortsetzen (Anfang 2. Seite)
  - So arbeiten CPUs nicht $\Rightarrow$ Programm muss schon beim Kompilieren alle 1000 Byte einen künstlichen Sprung zum nächsten Blockanfang erhalten
- Zur Vermeidung größere Seiten wählen
  $\Rightarrow$ mehr interne Fragmentierung

Beispiel Speicherverwaltung (3/3)

Speicherschutz wird praktisch unmöglich:

- Bei zusammenhängendem Speicher: Basisregister + Längenregister, CPU kann Limits prüfen
- Bei (beliebig vielen) separaten Bereichen: CPU müsste bei jedem Speicherzugriff ganze Speichertabelle durchgehen (aufwendige Schleife – nichts, was die CPU nebenbei macht)

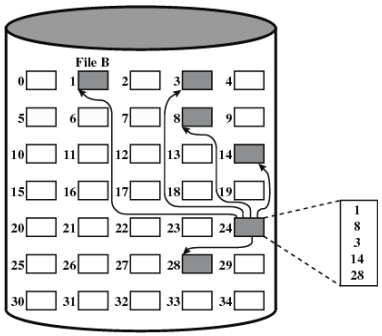
Leicht verbesserte Ansätze

Dateisystem: Blockliste

Ziel: FAT-Eintrag klein halten, trotzdem große Zahl Blöcke erlauben

Index-Allokation

- FAT enthält keine Liste der Blöcke, sondern Zeiger auf einen speziellen Index-Block
- Index-Block enthält Blockadressen
- Wenn Index-Block voll ist: Verweis auf nächsten Index-Block

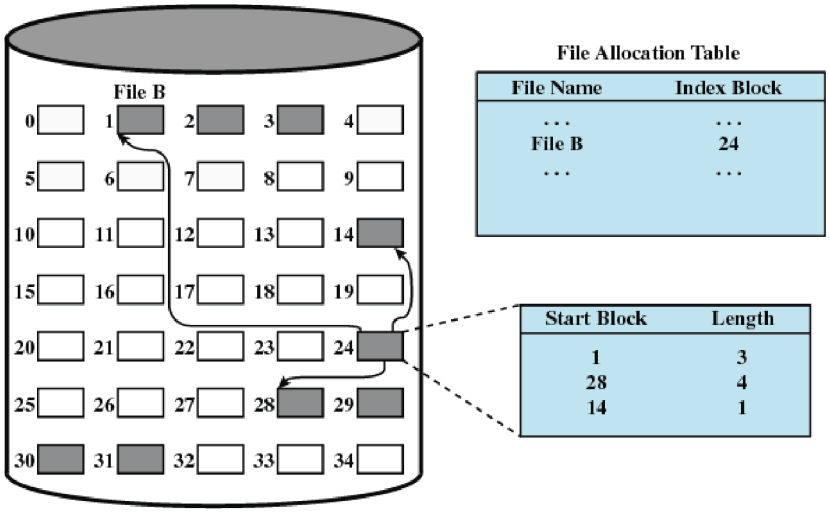


Dateisystem: teilweise zusammenhängend

Mischung Index-Allokation / zusammenhängende Zuordnung:

- Versuche, jeweils mehrere zusammenhängende Blöcke zu belegen
- FAT: Mehrere Einträge der Form (Anfang, Länge)
- Mit der Zeit: auch hohe Fragmentierung (artet in einfaches Modell/Index-Allokation aus)

Dateisystem: teilweise zusammenhängend



Erfundenes Dateisystem: Index-CP/M

- FAT-Eintrag wie bei CP/M, aber nicht 16 Blockadressen, sondern 16 Adressen von Index-Blöcken
- Jeder Index-Block (1 KByte groß) nimmt 128 Blockadressen auf
- $\Rightarrow 16 \times 128 = 2048$ Blockadressen
- $\Rightarrow 2048 \times 1$ KByte $= 2$ MByte max. Dateigröße (und das ohne Extents)
- mit Extents sogar 32 MByte max. Dateigröße
- Achtung: Größere Datenträger $\Rightarrow$ größere FAT-Einträge wegen „längerer“ Adressen

Speicher: teilweise zusammenhängend

Segmentierung: Code + Daten nicht zwingend in einem zusammenhängenden Bereich

- Code-Segment: zusammenhängend, Daten-Segement: zusammenhängend (an anderer Stelle)
- je nach Programm evtl. weitere unabhängige Daten-Segmente
- Verwaltungs-Overhead relativ gering ...
- ... Gewinn an Flexibilität aber auch

Speicher: Was tun, wenn RAM voll? (1/2)

Früher Ansatz: Overlay-Programmierung

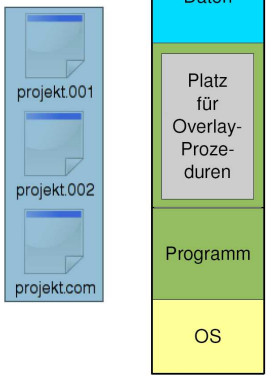
- Speicher zu knapp für große Programme $\Rightarrow$ Overlay-Programmierung
- Programmteile dynamisch nachladen, wenn sie benötigt werden
- Programmierer muss sich um Aufteilung in Overlays kümmern

Speicher: Was tun, wenn RAM voll? (2/2)

Früher Ansatz: Overlay-Programmierung

Turbo Pascal, um 1985-90:

```
program grossesprojekt;
overlay procedure kundendaten;
...
overlay procedure lagerbestand;
...
{ Hauptprogramm }
begin
  while input <> "exit" do begin
    case input of
      "kunden": kundendaten;
      "lager": lagerbestand;
    end;
  end;
end.
```



Lösung aller Speicherprobleme

Alle Probleme im Hauptspeicher:

- Komplexität nicht-zusammenhängender Speicherzuteilung
- Speicherschutz
- Code-Verschiebung
- Grenzen des RAM

löst man mit **virtuellem Speicher** (keine „direkte“ Adressierung des physikalischen Speichers durch Prozess). Mehr dazu später

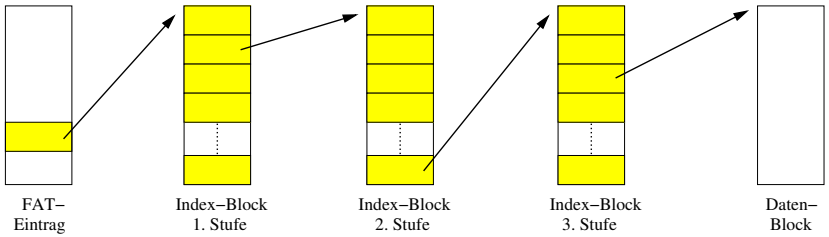
Methoden aktueller Betriebssysteme

Dateisystem: Mehrstufige Index-Blöcke

Ansatz mit Index-Blöcken jetzt mehrstufig

- FAT-Eintrag enthält Adressen von Index-Blöcken (1. Stufe)
- Index-Blöcke enthalten Adressen von weiteren Index-Blöcken (2. Stufe)
- . . . usw.
- in der letzten Stufe enthalten die Index-Blöcke Adressen von Datenblöcken

Beispiel: Drei-Stufen-Index

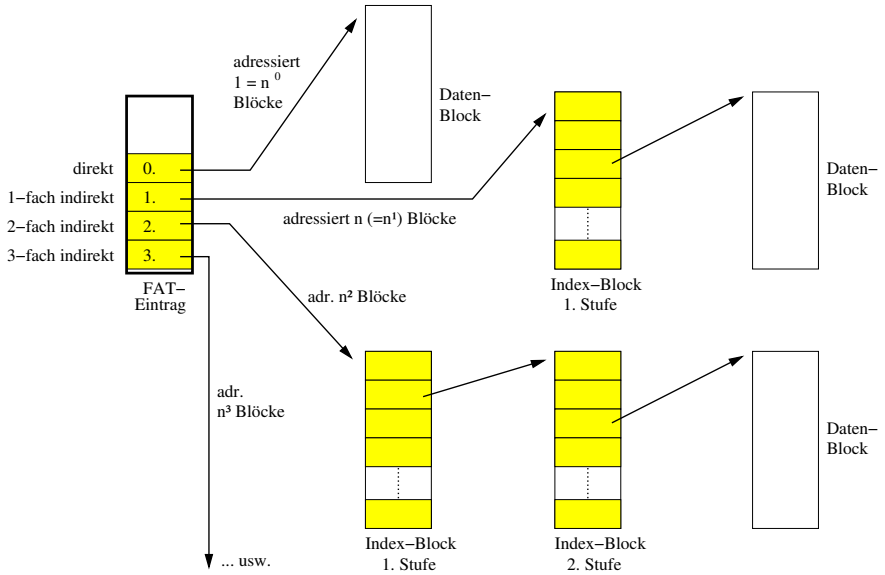


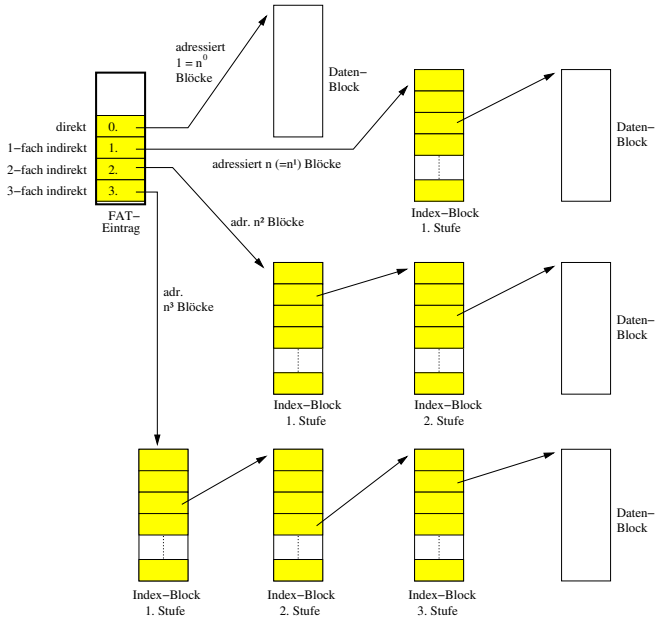
- Passen n Blockadressen in einen Index-Block, lassen sich so n^3 Datenblöcke adressieren – die maximale Dateigröße ist dann $n^3 \times B$ (B = Blockgröße)
- Problem: Für jedes Lesen eines Datenblocks sind fünf Plattenzugriffe (FAT, 1.–3. Index + Datenblock) nötig

Mehrere Stufen kombinieren (1/2)

Ziel: Bei kleinen und mittleren Dateien die Zahl der Blockzugriffe reduzieren

- Einige Blöcke direkt adressieren (Datenblocknummern in FAT eintragen)
- Einige Blöcke nur einfach indirekt adressieren
- Weitere Blöcke zweifach indirekt adressieren
- Weitere Blöcke dreifach indirekt adressieren





Indirektion und Dateigrößen

Beim Erstellen des Dateisystems unterschiedliche Blockgrößen möglich (Beispiel mit 3-facher Indirektion):

- 1024-Byte-Blöcke $\Rightarrow$ max. Größe 16 GByte
- 2048-Byte-Blöcke $\Rightarrow$ max. Größe 256 GByte
- 4096-Byte-Blöcke $\Rightarrow$ max. Größe 4096 GByte (4 TB)

Bei großen Dateien: Speichern der Blocknummern in Indirektionsblöcken (bis zu 3 Stufen): Verdoppeln der Blockgröße = Ver-16-fachen (2^4) der Block- nummern: Faktor 2^3 aus Indirektion, Faktor 2 aus Blockgröße

Berechnung der max. Dateigröße (1/4)

Gegeben sind:

- B: Blockgröße (z. B. 32 KByte)
- D: Größe des Dateisystems (16 GByte)
- b_0: Anzahl direkter Verweise (z. B. 9)
- b_1: Anzahl 1-fach indirekter Verweise (4)
- b_2: Anzahl 2-fach indirekter Verweise (2)
- b_3: Anzahl 3-fach indirekter Verweise (1)

Berechnung der max. Dateigröße (2/4)

1. Wie viele Blöcke passen in das Dateisystem?
   D/B 16 GByte / 32 KByte = 0,5 M = 512 K = 2^{19}
2. Wie groß ist eine Blockadresse? $L = \log_2(D/B)$
   $\log_2(2^{19}) = 19$: 19 Bit, aufrunden auf 2er-Potenz: 4 Byte
3. Adressen pro Block: $N = B/L$
   32 KByte / 4 Byte = 8 K = 8192
4. max. Dateigröße = (Anzahl adressierbare Blocks) × (Blockgröße) = $(\sum_i b_i N^i) \times B$
   $(9 \times 1 + 4 \times 8192 + 2 \times 8192^2 + 1 \times 8192^3) \times 32$ KByte = 549.890.064.393 × 32 KByte ≈ 16388 TByte (Terabyte)

Berechnung der max. Dateigröße (3/4)

Noch mal in der Übersicht:

Dateisystem mit folgenden Parametern:
- Dateisystemgröße: **16 GB**
- Blockgröße: **32 KB**
- I-Node enthält:
  - **9 direkte** Verweise
  - **4 1-fach indirekte** Verweise
  - **2 2-fach indirekte** Verweise
  - **1 3-fach indirekter** Verweis

Zu berechnen:
a) Größe der Blockadresse
b) # Adressen pro Block
c) Maximale Dateigröße, die ein Inode zulässt

a) Dateisystem: 16 GB, Block: 32 KB
-> Es gibt 16 G / 32 K
= 0,5 M = 512 K = 2^{19} Blöcke

Blockadressen sind also mind. 19 Bit lang,
in der Praxis: 4 Byte
(kleinste 2er Potenz, in die 19 Bit passen)

b) Blockgröße / Adressgröße = 32 KB / 4 Byte
= 8 K = 8192 = 2^{13}

c) # (adressierb. Blöcke) x Blockgröße =
(9 +
4 x 8192 +
2 x 8192^2 +
1 x 8192^3) x 32 KB
= 17.596.482.060.576 KB ≈ 16.781.313 GB
≈16.388 Terabyte (sehr groß)

Berechnung der max. Dateigröße (4/4)

Rechenaufgabe

Ein Dateisystem habe folgende Eigenschaften:

- Blockgröße: 16 KByte
- Dateisystemgröße: 1 GByte
- 2-fache Indirektion;
  - 10 direkte Verweise,
  - 3 einfach indirekte Verweise
  - 1 zweifach indirekter Verweis

Berechnen Sie die maximale Dateigröße.

Unix-Dateisystem

- Dateisysteme unter Unix / Linux sind Beispiele für diese Art der Mehrfachindirektion.
- Nächstes Mal: mehr zu Unix-/Linux-Dateisystemen