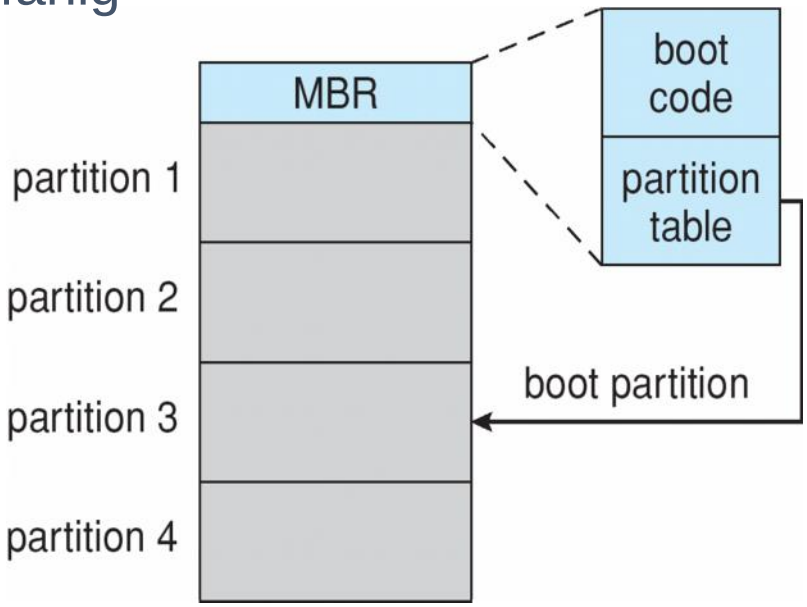
Betriebssysteme und Netzwerke Vorlesung 14

Artur Andrzejak

Implementierung von Dateisystemen - Historische Beispiele

Struktur einer Festplatte bei BIOS /1

- Bei BIOS-basierten Computern der x86-Architektur besteht eine Festplatte aus dem **Master Boot Record** (**MBR**) (Link) und 1 bis 4 **Partitionen** (**volumes**), d.h. Unterteilungen der FP
- Jede Partition kann ein anderes Dateisystem enthalten
  - Aber nur eine ist aktiv, d.h. bootfähig
- Die aktive Partition enthält als 1. Block den **boot sector**
- MBR und boot sector enthalten (u.a.) ausführbaren Code



Struktur einer Festplatte bei BIOS /2

- Das MBR ist 512 Bytes groß und enthält:
  - Programmcode des Boot-Loaders (440 Bytes)
  - Die **Partitionstabelle** mit bis 4 Einträgen von je 16 Bytes
- Struktur eines Eintrags

Start	Größe (B)	Inhalt
0	1	80_h=bootfähig (**active**), 00_h=nicht bootfähig
1	3	CHS-Eintrag: Cylinder - Head - Sector des 1. Blocks
4	1	Typ der Partition (Partitionstyp)
5	3	CHS-Eintrag des letzten (physischen) Blocks
8	4	Startblock als logische Adresse, relativ zum FP-Anfang
12	4	Anzahl der physischen Blöcke in der Partition

- Die 24-Bit CHS Felder sind bei ca. 8 GB erschöpft
  - Deshalb wurden bei größeren FP nur die Einträge an den Stellen 8 und 12 benutzt (d.h. logische Angaben)

FATx – Dateisystem von MS-DOS (Link)

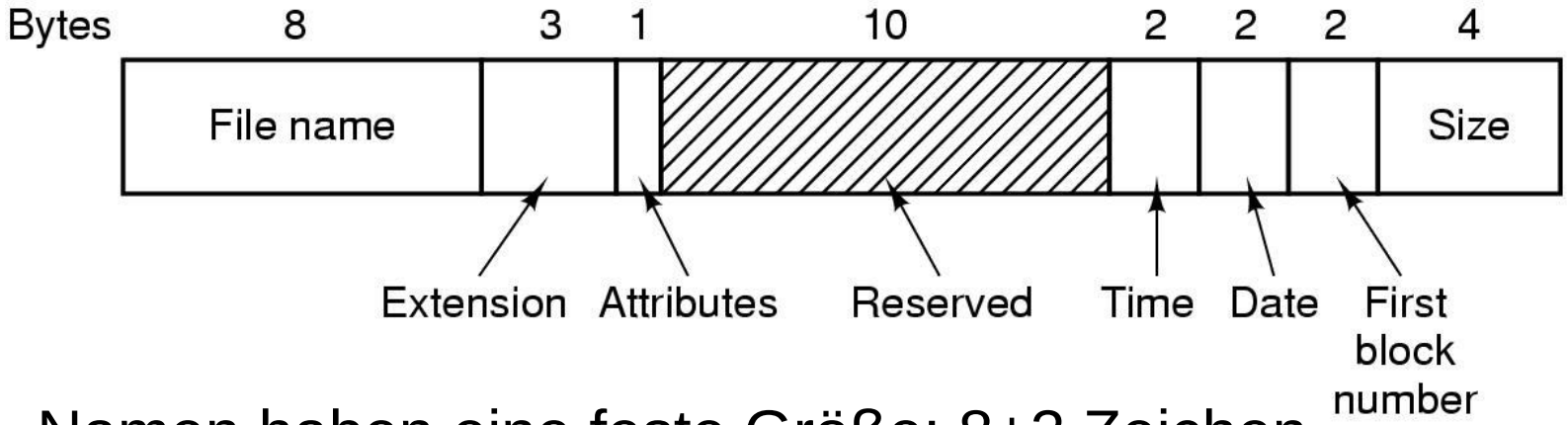
- Benutzt eine FAT Struktur, d.h. Liste von Zeigern, die in den Speicher geladen werden
- Zeigergröße (in Bits) abhängig von der Version
  - FAT12: 12 Bits, FAT16: 16 Bits, FAT32: **28** Bits
- **Logische Blockgröße** (bei Microsoft: **Clustergröße**) ist k*512 Bytes

Blockgröße	FAT-12	FAT-16	FAT-32
0.5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

Max. Kapazitäten einer Festplatte

FATx - Verzeichnisse

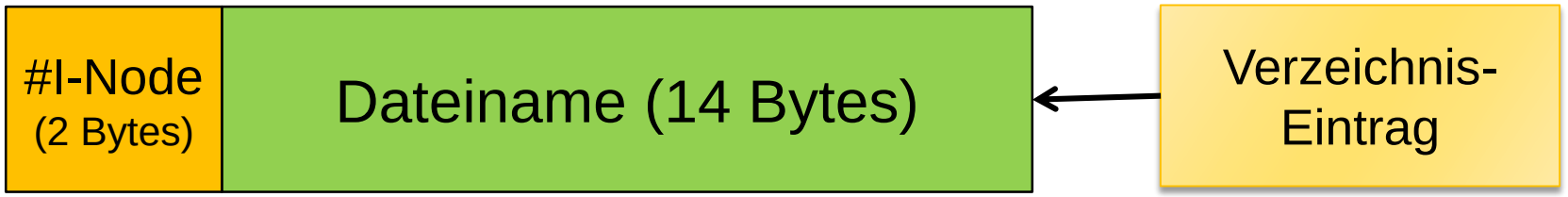
- Struktur eines Verzeichniseintrags



  - Namen haben eine feste Größe: 8+3 Zeichen
  - Ab Windows 95 mit **VFAT** (**Virtual File Allocation Table**) auch Namen variabler Länge (bis 255 Zeichen)
  - 32 Bits für Dateilänge, also max. 4 GB (-1) groß
  - Dateiattribute: Bit 0: Schreibgeschützt; Bit 1: Versteckt; Bit 2: Systemdatei; Bit 3: Volume-Label; Bit 4: Unterverzeichnis; Bit 5: Archiv; Bit 6–7: ungenutzt

UNIX-V7 Dateisystem

- Verzeichniseintrag einer Datei
  - Name (14 Bytes), log. Nummer des Blocks mit I-Node
  - Welche Beschränkungen erzeugte das?



- Attribute sind in den I-Nodes
  - Dateigröße, Zeiten (creation, last access, last modification), Besitzer, Gruppe, Zugriffsrechte, …
  - Und „Anzahl der Verzeichniseinträge, die auf diesen I-Node zeigen“ – wozu?

Dateisysteme
ext2, ext3, ext4
zfs

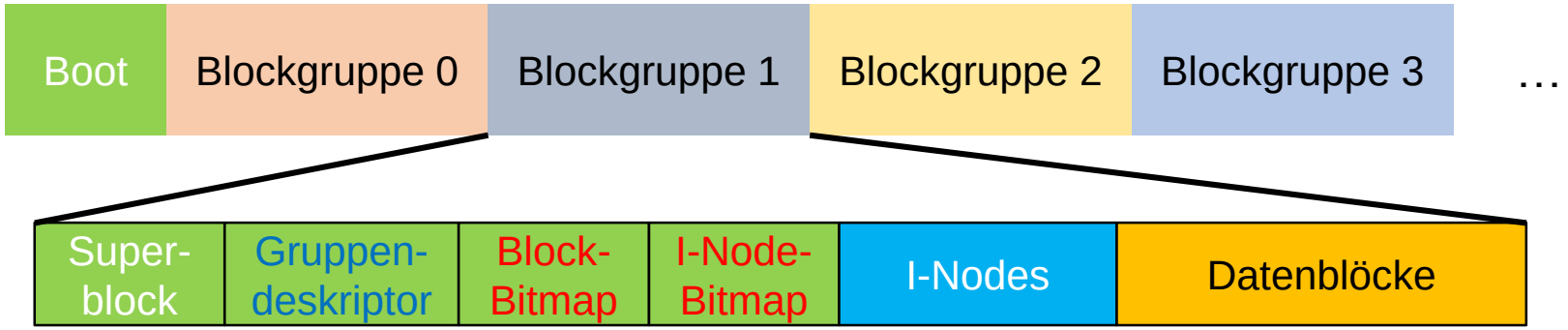
ext2 - Second Extended Filesystem

- Eingeführt in 1993, jahrelang das Standarddateisystem von Linux
- Fakten
  - Max. Größe einer Datei: 2 TiB, $2*(2^{40}\sim10^{12}$ Bytes, d.h. 1 TB)
  - Max. Anzahl der Dateien: 10^{18}
  - Max. Plattengröße: 16 TiB
  - Max. Länge eines Dateinamens: 255 Byte
- Viele Eigenschaften traditioneller UNIX-Dateisysteme
  - I-Nodes, Verzeichnisse, Zugriffskontrolllisten, Kompression,..
- Dateirechteverwaltung wie in POSIX
  - D.h. Besitzer-Gruppe-Welt-Modell

Struktur des Dateisystems

- Eine Partition enthält logische Blöcke (1, 2, 4 KB)
- Um die Fragmentierung zu vermeiden, werden Blöcke zu **Blockgruppen** (**BG**) zusammengefasst
  - Eine BG entspricht etwa einem Zylinder (Spurengruppe)
- Der **Superblock** enthält wichtige Informationen und wird am Anfang einer BG gespeichert (repliziert)
  - Anzahl der Blöcke und I-Nodes im Dateisystem
  - …wie viele davon frei sind
  - … wie viele I-Nodes und Blöcke in jeder Blockgruppe vorhanden sind
  - … wann das Dateisystem eingebunden wurde, ob es beim letzten Mal korrekt ausgehängt wurde, …

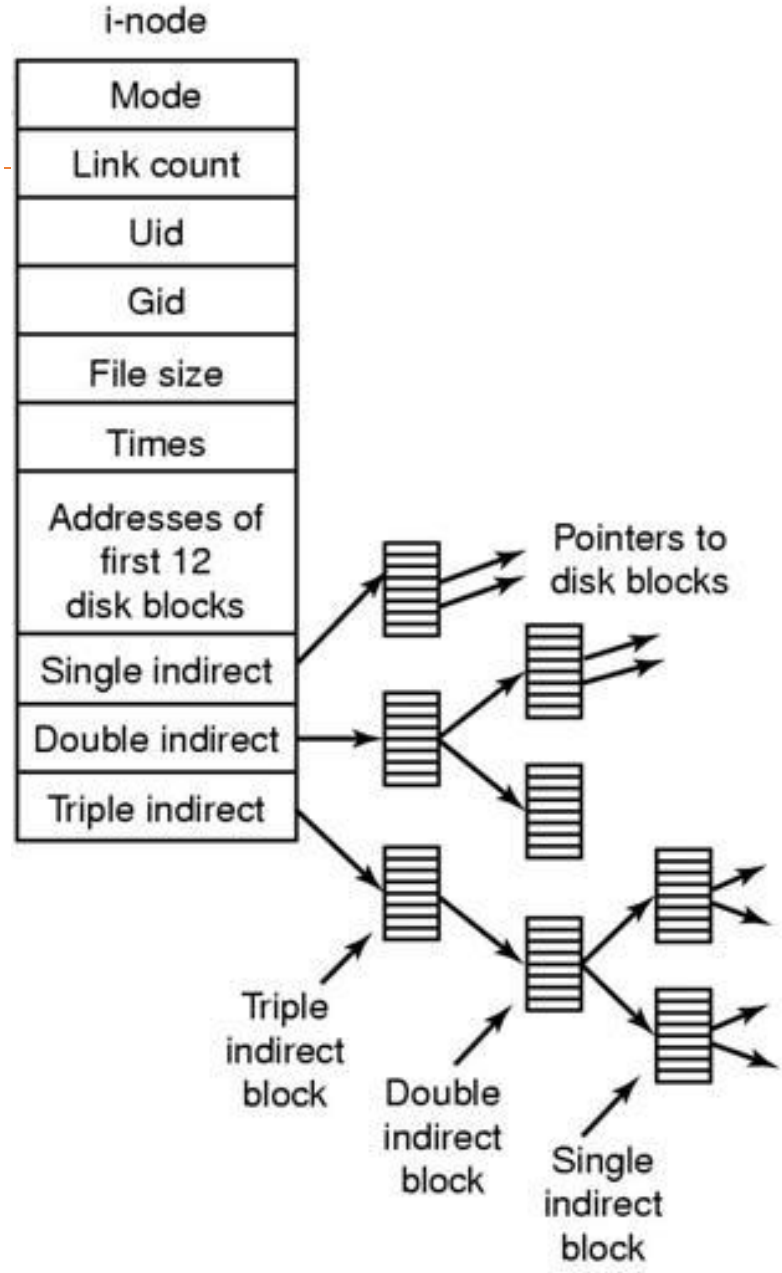
Blockgruppen



- G-Deskriptor: Anzahl freier Blöcke, freier I-Nodes und Verzeichnisse in dieser BG
- Bitmaps: Da sie je 1 KB groß sind, ist die max. Anzahl der Blöcke / I-Nodes jeweils 8192 pro BG
- I-Nodes: jeweils 128 Bytes lang

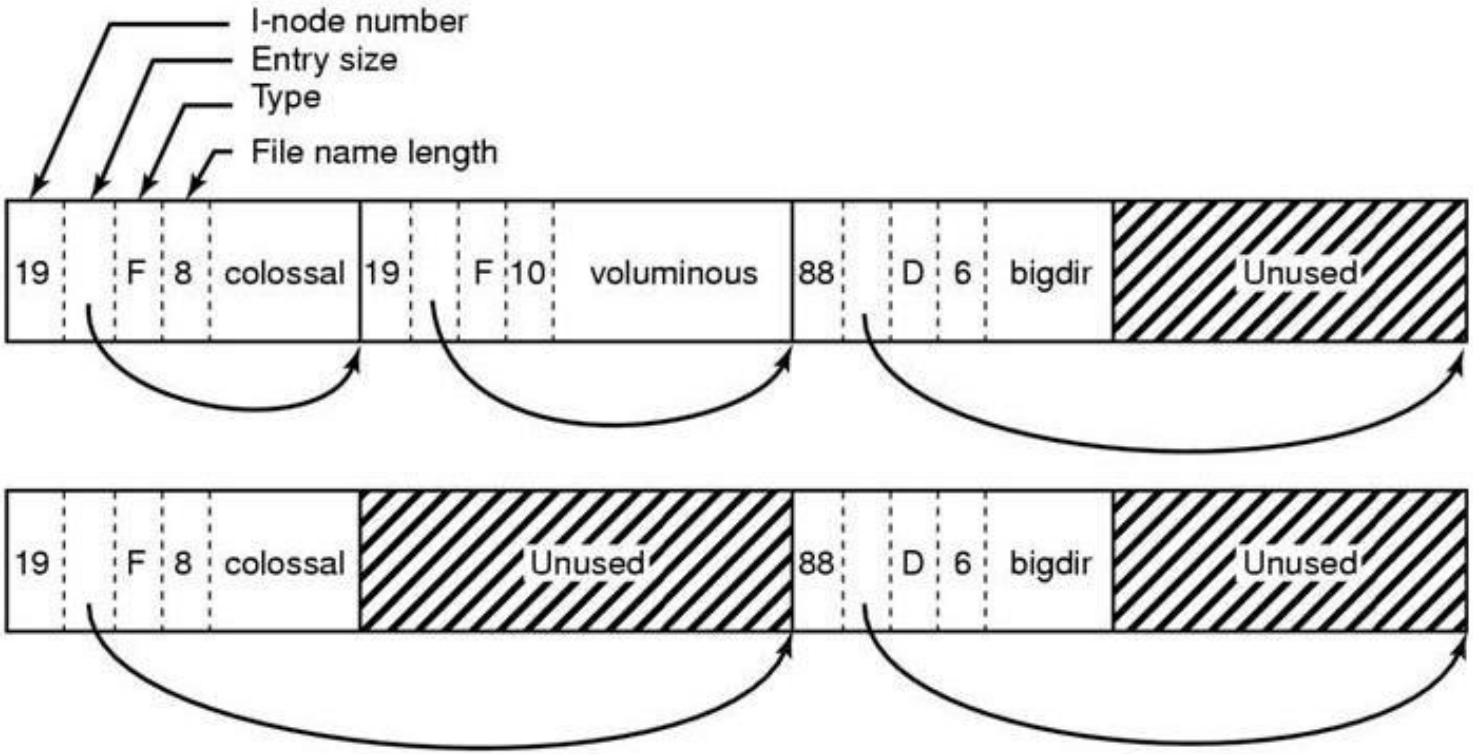
I-Nodes von ext2

- Die Attribute sind am Anfang
- Danach Zeiger auf Blöcke
  - Die ersten 12 Zeiger sind direkt
  - Dann gibt es je einen Zeiger auf
    - 1-fach indirekte
    - 2-fach indirekte, und
    - 3-fach indirekte Blöcke



Verzeichnisse in ext2

- Werden linear durchsucht, was lange dauern dann
  - Es wird deshalb ein Cache mit den zuletzt gesuchten Verzeichnissen aufrechterhalten



ext3 - Third Extended Filesystem (Link)

- Erstveröffentlichung: Nov. 2001 (Linux 2.4.15)
- Kombination von ext2 mit **Journal-Erweiterung**
  - Daten können mit einem ext2-Treiber gelesen werden
- Das **Journal** ist eine reguläre Datei, in die Metadaten (optional die Nutzdaten) geschrieben werden, bevor sie auf das tatsächliche Dateisystem geschrieben werden
  - Metadaten werden nicht mehr beschädigt
- Weitere Erweiterungen gegenüber ext2
  - H-Baum-Verzeichnisindizes (eine Version vom B-Bäumen)
  - Online-Änderung der Dateisystemgröße

Journaling-Stufen von ext3

- **Ordered** (Option *data=ordered*) - Standardeinstellung
  - Nur Metadaten werden ins Journal geschrieben
  - Die Dateiinhalte werden direkt ins Dateisystem geschrieben, erst danach werden die Metadaten im Journal aktualisiert
- **Writeback** (Option *data=writeback*)
  - Nur Metadaten werden ins Journal geschrieben
  - Das Aktualisieren der Dateiinhalte durch *sync*-Prozess
  - Schnell, jedoch die Gefahr von Datenverlust durch abgebrochene Out-of-Order-Schreibvorgänge
- **Full** (Option *data=journal*)
  - Sowohl Metadaten als auch Dateiinhalte werden erst ins Journal geschrieben
  - Erhöht die Zuverlässigkeit, ist jedoch langsam beim Schreiben, da alle Daten zweimal geschrieben werden

ext4 - Fourth Extended Filesystem

- Seit Oktober 2008, Linux 2.6.28
- Verbesserungen gegenüber ext3
  - Partitionen bis zu 1 EiB (Exbibyte d.h. $2^{60} \sim 10^{18}$ Byte)
  - Zeitstempel auf Nanosekunden-Basis
  - Online-Defragmentierung (Defragmentierung, während die Partition eingehängt ist)
  - Verwendung von Prüfsummen im Journal
- ext3-Partitionen können ohne Neuformatierung in ext4-Partitionen konvertiert werden
  - ext4 ist nur teilweise rückwärtskompatibel

Leistungsvergleich aus pro-linux.de

- http://www.pro-linux.de/artikel/2/224/3,das-dateisystem-ext4.html

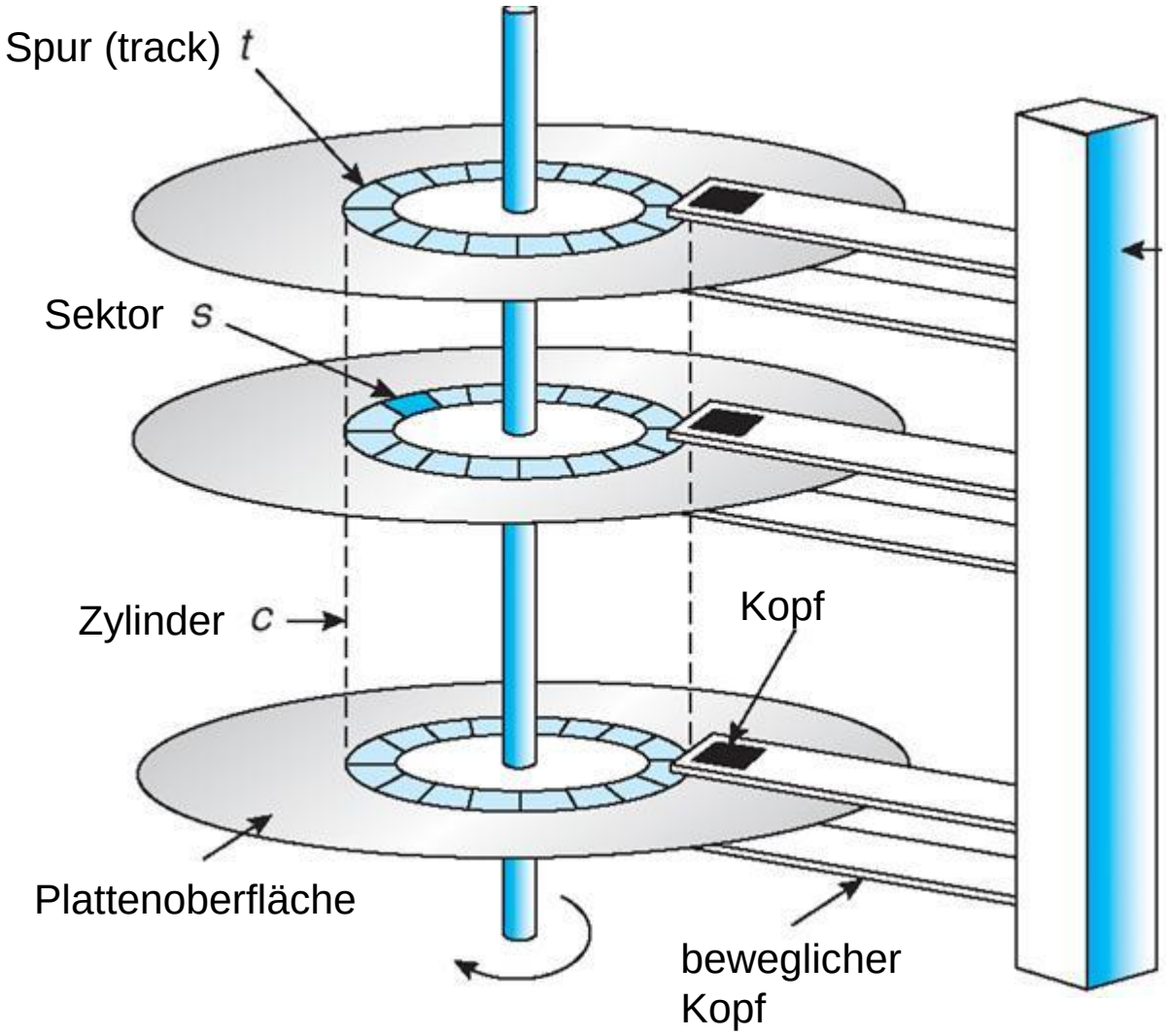
	ext2	ext3	ext4	xfs	jfs	reiser3
Dateisystem anlegen [Sek]	6,58	9,03	8,36	0,28	0,49	0,81
Dateisystem mounten [sek]	0,56	0,89	0,58	0,17	0,19	1,47
Datei 8 GB lesen [MB/s]	61,6	62,3	66,4	64,9	62,5	59,3
Datei 8 GB schreiben [MB/s]	63,1	57,1	60,4	61,1	56,3	54,5
Seq. lesen [MB/s]	60,2	59,3	64,8	62,1	62,7	61,5
Seq. schreiben [MB/s]	60,0	55,6	58,3	55,0	54,6	52,4
Seeks [1/s]	123	110	129	116	116	133
Datei erzeugen [1/Sek]	844	69760	59872	3031	12213	18995
Datei löschen [1/Sek]	1964	19777	19574	527	405	5328
Datei 8 GB löschen [Sek]	0,235	0,534	0,457	0,284	0,0	1,267
Datei 8 GB löschen [min Ges Sek]	0,498	0,612	0,500	0,308	0,007	1,296
Datei 8 GB löschen [max Ges Sek]	9,588	7,596	7,029	6,382	0,059	7,914

zfs

- Seit 2005 von Sun, jetzt Oracle entwickelt
- Von Problemen vorheriger Dateisysteme gelernt
  - 128 bit Zeiger, Partitionen bis zu 2^{128} bytes
  - Copy-On-Write für Konsistenz, erlaubt snapshots
  - Prüfsummen, Kompression und Verschlüsselung
  - Storage pool zur Verwaltung von devices
  - Mehrere RAID Level
  - Deduplikation
  - Backup streams
- Solaris, BSD derivate, Linux

Struktur und Scheduling von Festplatten

Struktur einer Festplatte (FP)



- Die Zeit, einen **Lese-/Schreib-kopf** über den richtigen Sektor zu positionieren ist der Flaschenhals bei kleinen Datenmengen

Besteht aus:

- **Zugriffszeit (seek time)**
  - Zeit der Kopfbewegung
- **Drehlatenz (rotational latency)**
  - Zeit der Drehung, bis der Kopf über dem richtigen Sektor ist

Festplatte - Spezifikationen (Beispiele)

- WD 1 TB Desktop Hard Drive WD10EARS (2009)
  - 7200/60 = 120 Umdrehungen pro Sek. (RPM)
  - 1.953.525.168 Sektoren, 2 Scheiben
  - **Zugriffszeit**: 8.9 ms; **Seq. R/W:** ~100 MB/s
  - SATA 2: 3 Gbit/s, effektiv 2.4 Gbit/s (300 MB/s)

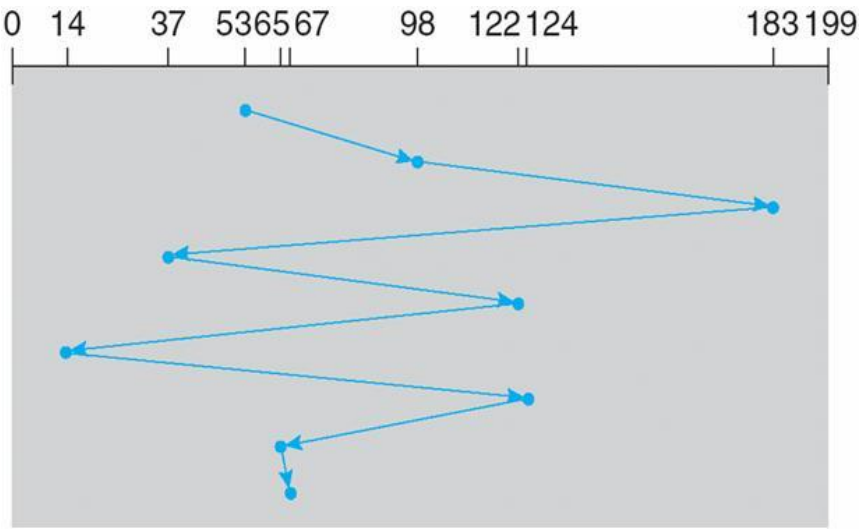
- Toshiba 900GB AL14SXB (2019)
  - 15000/60 = 250 Umdrehungen pro Sek.
  - **Zugriffszeit**: 5.3R/1.2W ms; **Seq. R/W:** ~220 MB/s
  - SAS 3: 12 Gbit/s, effektiv 9.6 Gbit/s (1200 MB/s)

Ab welcher Datenmenge dominiert Transferzeit?

- Zugriffszeit (seek time): 8.9 ms
- Rotational latency (Drehlatenz)
  - 7200/60 = 120 Umdrehungen pro Sekunde
  - => worst-case: **8.3 Millisekunden**
  - Aber Latenz ist 4.17 ms – Köpfe auf beiden Seiten der Achse?
- Transferzeit: 300 MB/sec
- Ab welcher Datenmenge ist Transferzeit == totale Latenz?
  - Annahme: Totale Zeit bis zum Transferstart 8.9+4.17 ~ 13 ms
- => (300 MB)* 13/1000 = 3.9 MB (Megabyte)
- Fazit: Totale Latenz ist entscheidend beim Lesen / schreiben von Datenmengen bis einige MBs
- Wie kann man den Betrieb beschleunigen?

Scheduling von Diskzugriffen

- Idee: Mehrere wartende Anfragen werden so umgeordnet, dass bei der Ausführung die Kopfbewegungen reduziert sind
  - Anfragen warten z.B. auf das Ende aktueller Operation
  - Dieses „Umordnen“ nennt man Scheduling (Ablaufsteuerung, Zuteilung)
- Kein Umordnen führt zu **First-Come First-Serve** (**FCFS**)
  - Ausführung in der Reihenfolge der Anfragen

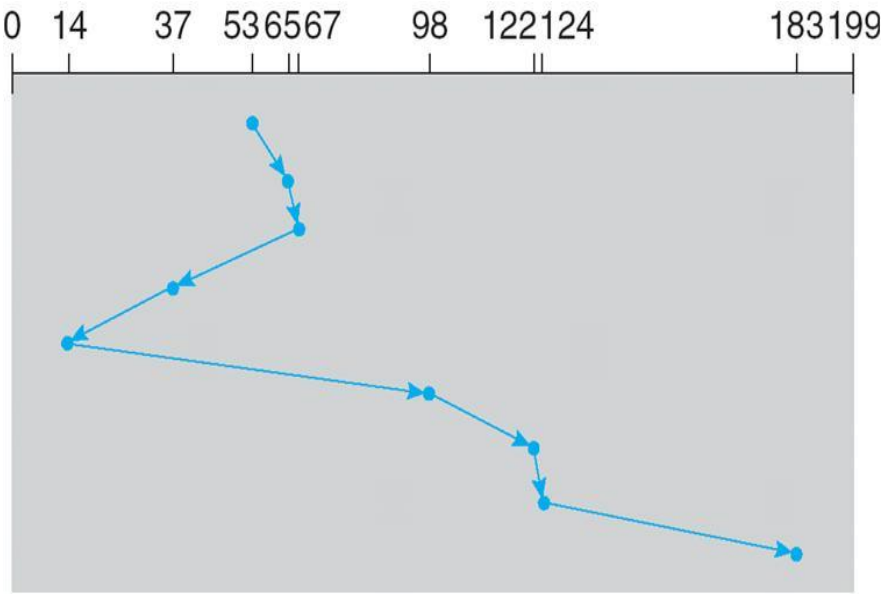


Start beim Zylinder 53, dann Zylinder 98, 183, 37, 122, 14, 124, 65, 67

Shortest Seek Time First (SSTF)

- Als nächste Anfrage wird diejenige ausgewählt, die die kürzeste Zugriffszeit (bezüglich der aktuellen Position) hat
- Was könnte hier problematisch werden?

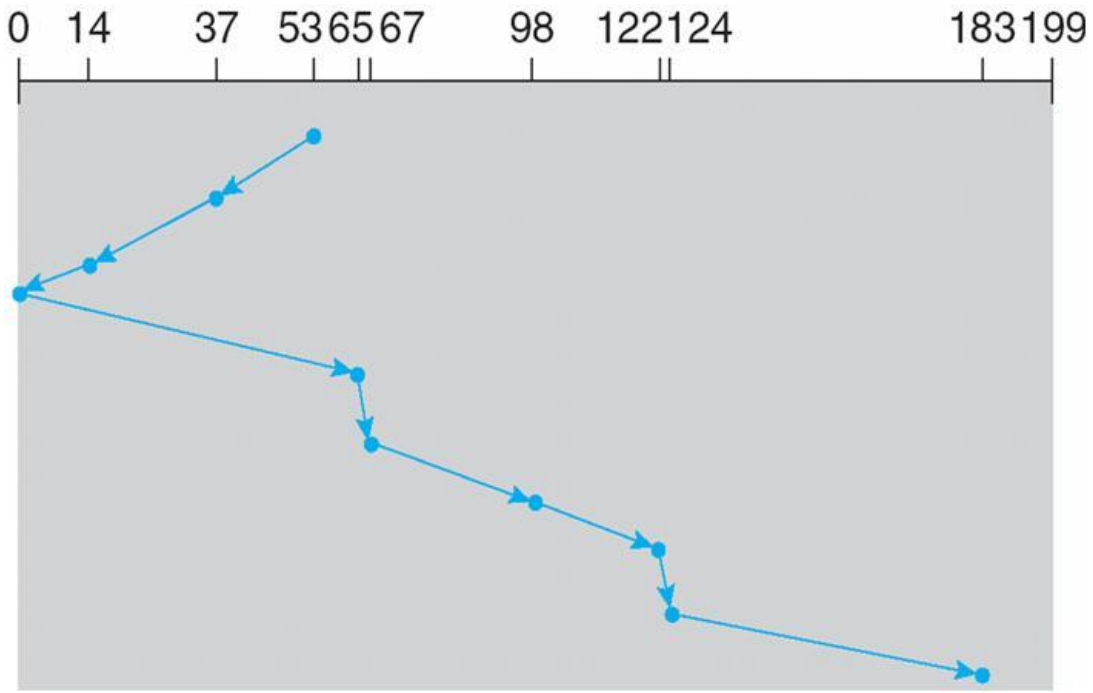
- Anfragen werden ggf. nie ausgeführt – **starvation ("Verhungern")**
  - Z.B. wenn viele neue Anfragen nahe der aktuellen Position kommen
  - … eine ferne Position wird nie angesteuert



SSFT

SCAN - Algorithmus

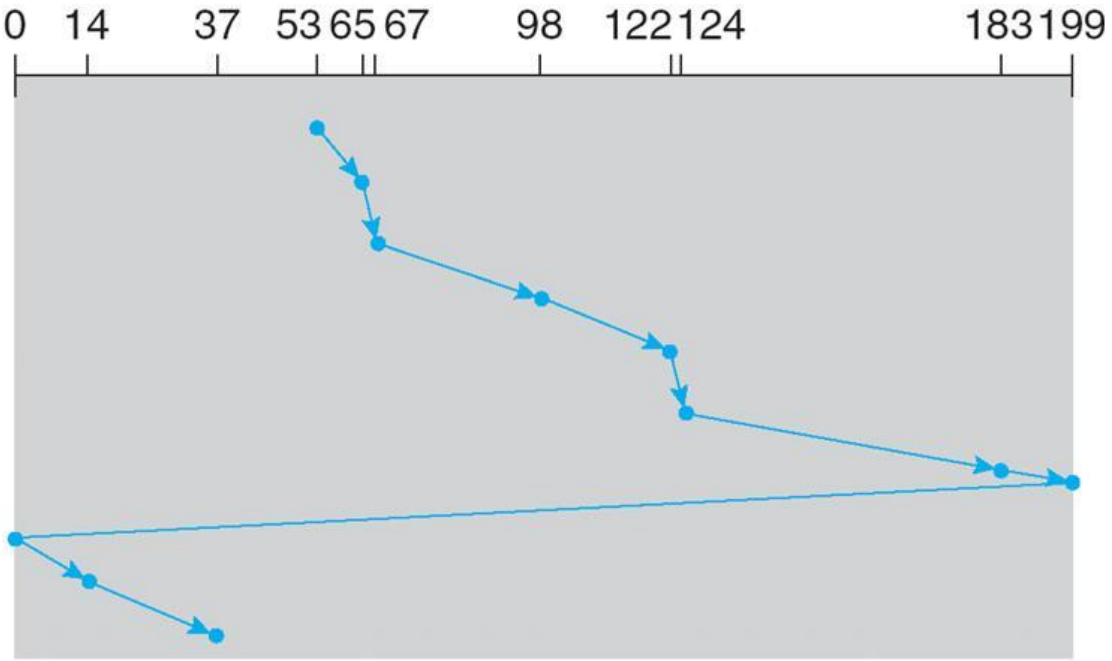
- Der Arm geht von einem Ende der Scheibe (z.B. außen) und geht bis zum anderen Ende (innen), und dann zurück
  - Dabei werden alle Anfragen „auf dem Weg“ bearbeitet
- Genannt auch der „Fahrstuhl-Alg.“ (**elevator algorithm**)
- Problem?
- Nicht-uniforme Abarbeitung: neue Anfragen am aktuellen Ende kommen schnell dran, Anfragen am anderen Ende warten



SCAN

C-SCAN - Algorithmus

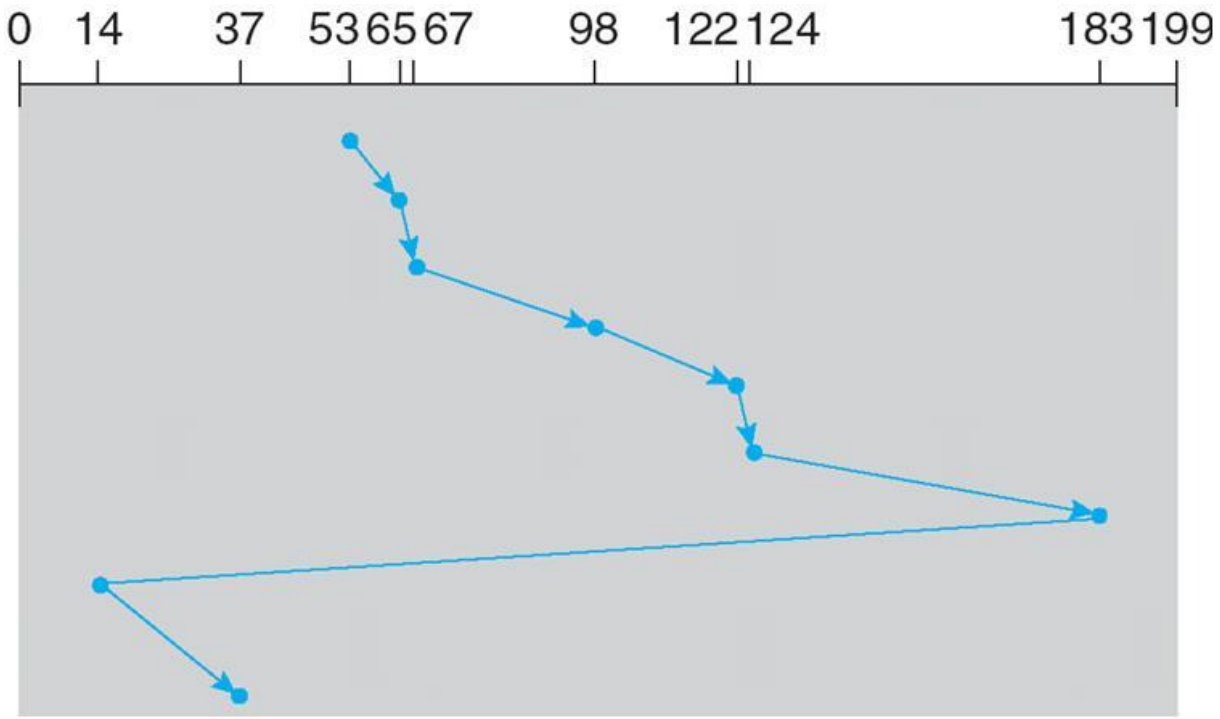
- Der Arm geht von einem Ende der Scheibe (z.B. außen) und geht bis zum anderen Ende (hier: innen)
  - Unterwegs werden alle Anfragen „auf dem Weg“ bearbeitet
- Dann aber fährt der Arm an den „Start“ zurück und beginnt von vorne



C-SCAN

C-LOOK - Algorithmus

- Eine Verbesserung von C-SCAN
- Hier fährt der Arm nicht ans Ende der Disk, sondern nur bis zur Spur mit der weitesten Anfrage



C-LOOK

SSD und NVM

- Solid state drive
  - Flash: persistente Speicherung ohne Energieverbrauch
  - **Zugriffszeit**: 0.1 – 0.01 ms; **Seq. R/W:** ~600 MB/s
  - SATA 3: 6 Gbit/s, effektiv 4.8 Gbit/s (600 MB/s)
    - Oder mehr über PCI express, bis zu ~16GB/s bei PCIe 3.0 x16
- Non volatile memory storage
  - **Zugriffszeit**: <0.01 ms; **Seq. R/W:** ~600 MB/s
  - SATA 3: 6 Gbit/s, effektiv 4.8 Gbit/s (600 MB/s)
    - Oder mehr über PCI express, bis zu ~16GB/s bei PCIe 3.0 x16
  - Ersetzt Hauptspeicher?
    - Auswirkungen?

RAID-Systeme

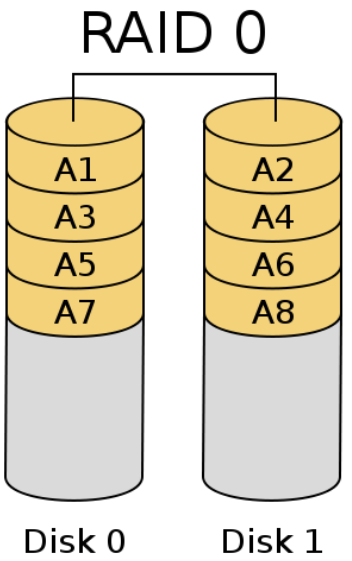
RAID

- Idee: Organisiere mehrere physische Disks (FP) zu einer logischen Disk, für höhere …
  - Verlässlichkeit (**reliability**)
  - Leistungsfähigkeit (**performance**)
- RAID: **Redundand Array of ~~Inexpensive~~ *Independent* Disks**
  - Vorgeschlagen 1987 durch D. A. Patterson, G. Gibson und R. H. Katz
  - Ursprünglich 5 Level, später Level 0 und 6 hinzugefügt
- Bei RAID-Systemen werden u.a. redundante Daten erzeugt, damit beim Ausfall einzelner Komponenten das RAID („logische Disk“) als Ganzes seine Integrität und Funktionalität behält

RAID 0 - Striping ohne Redundanz

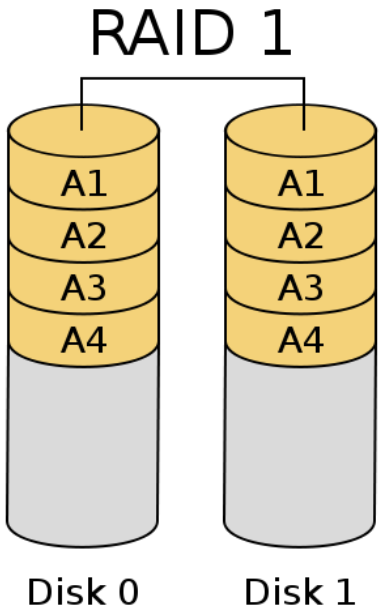
- Die FP werden in zusammenhängende Blöcke gleicher Größe aufgeteilt, und die Dateien dadurch (implizit) in „Streifen“ zerlegt (**striping**)
  - Die Daten können so parallel gelesen / geschrieben werden, wie in einem „Reißverschlussverfahren“
  - Oft wird als die Blockgröße (**chunk size**) 64 kB gewählt

- Es gibt aber keine Redundanz!
  - Es ist eigentlich nur „AID“ bzw. **JBOD** (**Just a Bunch Of Disks**)
  - Beim Defekt einer FP sind ggf. alle Daten weg!



RAID 1: Mirroring - Spiegelung

- Ein Verbund von mind. 2 FP für höhere Verlässlichkeit
- Ein RAID 1 speichert auf allen Festplatten die gleichen Daten (**Spiegelung**)
  - Die Kapazität des Arrays ist hierbei höchstens so groß wie die der kleinsten beteiligten FP
  - **Mirroring**: alle FP-“Scheiben“ am gleichen Controller; **Duplexing**: separate, selbständige FP
- Unverzichtbar für sicherheitskritische Echtzeitanwendungen (z.B. Kernkraftwerk, Computerspiele, …)
  - Beim Fehler einer FP läuft alles weiter
  - Bei „paranoiden“ Systemen werden mehrere Leseströme verglichen
    - Bei Unterschieden gib es Fehlermeldung



Verlässlichkeit von RAID 0 und RAID 1

- Annahme: Eine FP fällt in 3 Jahren mit Wahrscheinlichkeit (W-keit) von 0.05 aus
- Was ist W-keit, dass ein RAID 1 System (Spiegelung) in drei Jahren ausfällt (System mit zwei FP)?
- Was ist die analoge W-keit für ein RAID 0 System (JBOD)?
- RAID 1:
  - P(beide FP fallen aus) = 0.05^2 = 0.0025
- RAID 0:
  - P(mind. 1 FP fällt aus) = 1 – P(keine FP fällt aus) = 1 – (1-0.05)^2 = 1-(1-2*0.05+0.0025)=0.1 – 0.0025 = 0.0975
- => RAID 0 „halbiert“ die Verlässlichkeit, RAID 1 macht sie 20-fach höher (tatsächlich noch besser – warum?)

Ethik-Algebra /1

- Gegeben: **n** Festplatten (FP), jede von diesen hat die W-keit **p**, dass sie in einem Zeitraum T ausfällt
- Was ist W-keit des RAID-Ausfalls im Zeitraum T (d.h. mindestens eine FP fällt aus)?

$1-(1-p)^n$

- Generalisierung: gegeben seien **n** "Objekte" (z.B. FP oder Versuche), und zu jedem gibt es:
  - Ein böses bzw. schlechtes (bad) Ereignis **BE**

  - Ein gutes Ereignis (good) **GE**

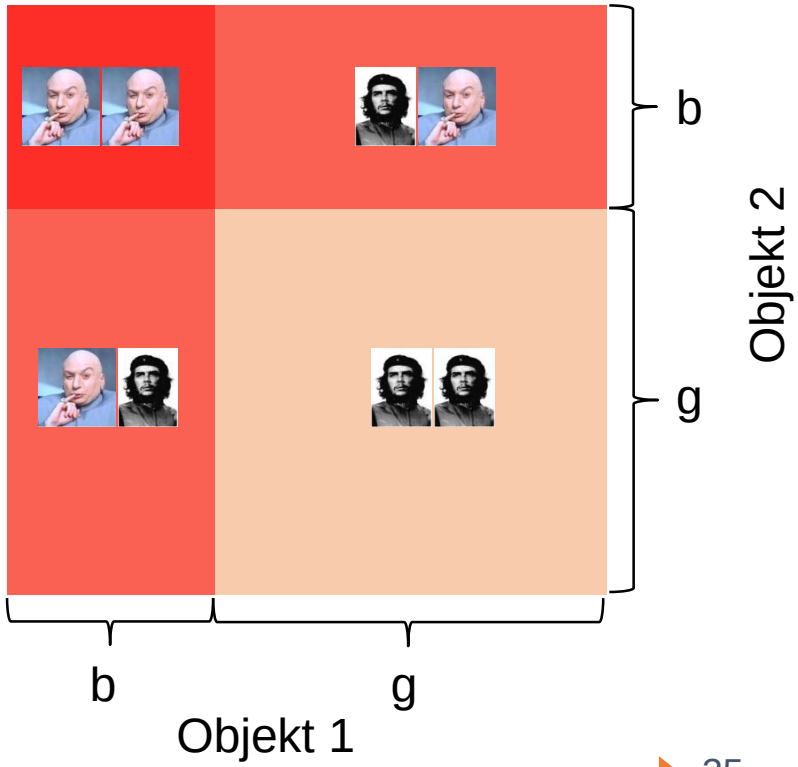
  - Für jedes Objekt sei …
    - **b**(ad) = W-keit, dass das böse Ereignis auftritt
    - **g**(ood) = W-keit, dass das gute Ereignis auftritt
  - Entweder BE oder GE tritt auf, also b+g = 1

Ethik-Algebra /2

- Was ist die W-keit, dass es mindestens 1 BE gibt?
- 1- (**W-keit von genau n GE**) = 1 – $\mathbf{g^n}$ (mit g := 1-p)
- Man kann das für n = 2 visualisieren - wie?

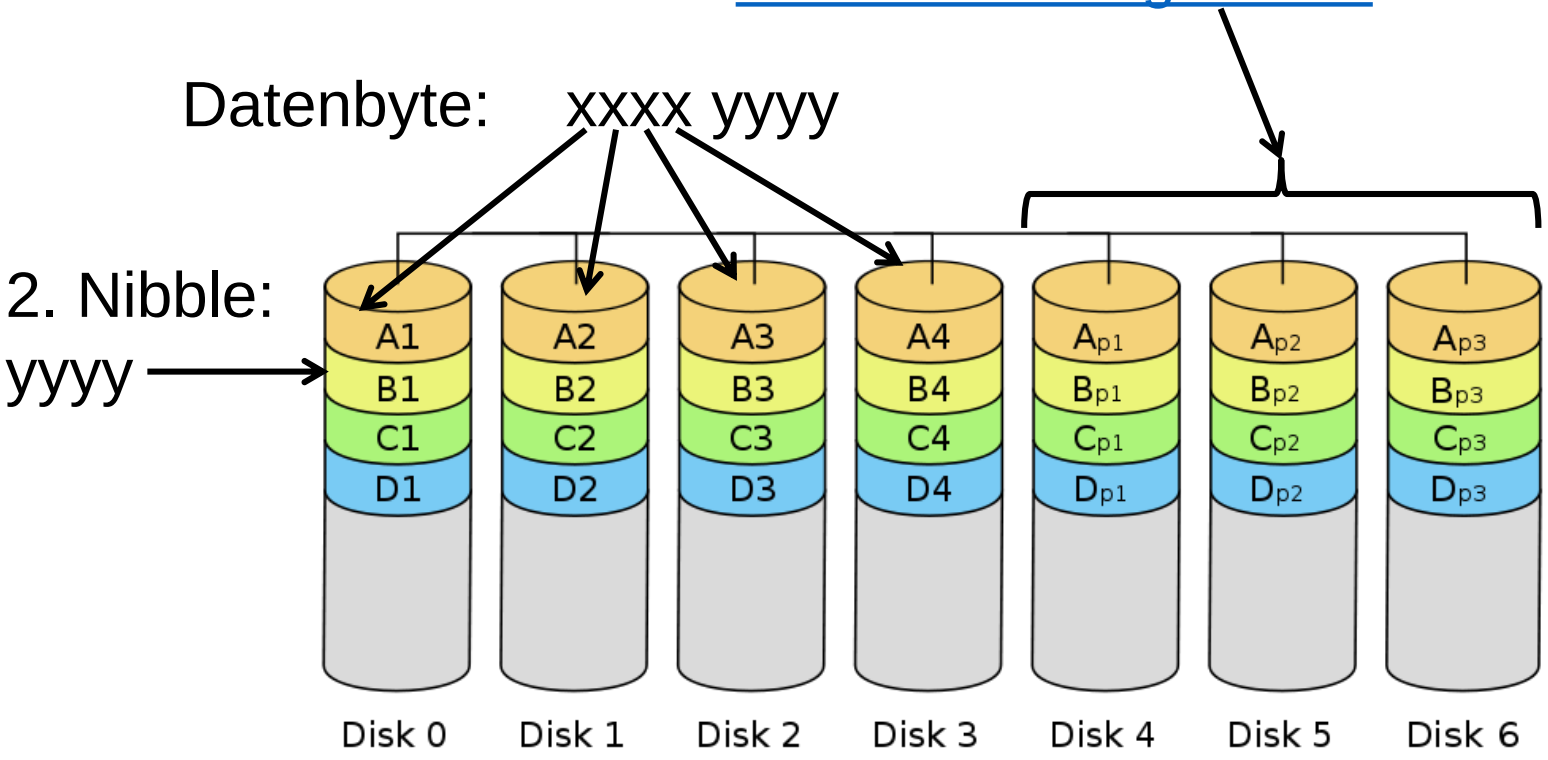
- Wir summieren auf:
  - W-keit, dass es genau 1 BE gibt
  - W-keit, dass es genau 2 BE gibt
  - …
  - W-keit, dass es genau n BE gibt

- Wie berechnet man, dass es genau k BE gibt, 0 < k < n?
- Was passiert für n > 2?



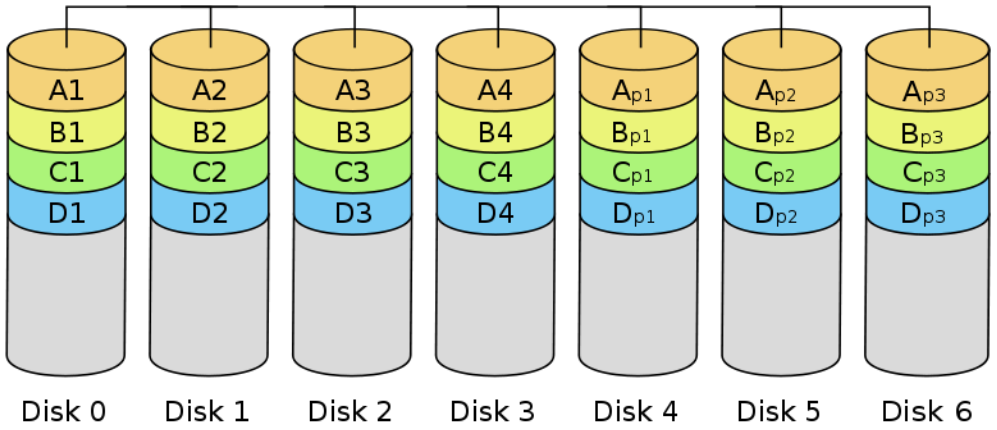
RAID 2 – Bit-Level-Striping

- Einzelne Bytes werden in Bitfolgen fester Größe zerlegt und mittels eines Hamming-Codes auf größere Bitfolgen abgebildet
  - **Hamming(7,4)**: 4 Bit für Daten und noch 3 Bits für den zusätzlichen Teil des Error-Correcting-Codes



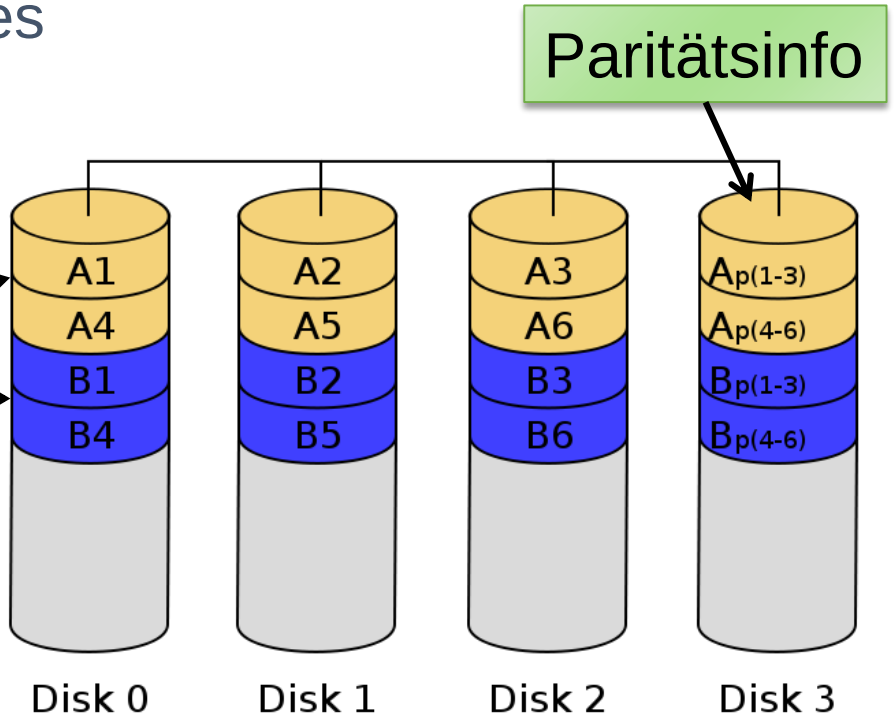
RAID 2 - Details

- RAID 2 ermöglicht sehr hohe Datentransferraten
- Aber die FP-Scheiben müssen sich synchron drehen
  - D.h. gleiche Sektoren zur gleichen Zeit
- Ermöglicht automatische Wiederherstellung von 1-Bit-Fehlern und Erkennung von 2-Bit-Fehlern
- Heute nicht mehr in der Praxis verwendet
  - Zu komplex
  - Heutige FP haben Error-Correcting-Codes (ECC) innerhalb eines Sektors



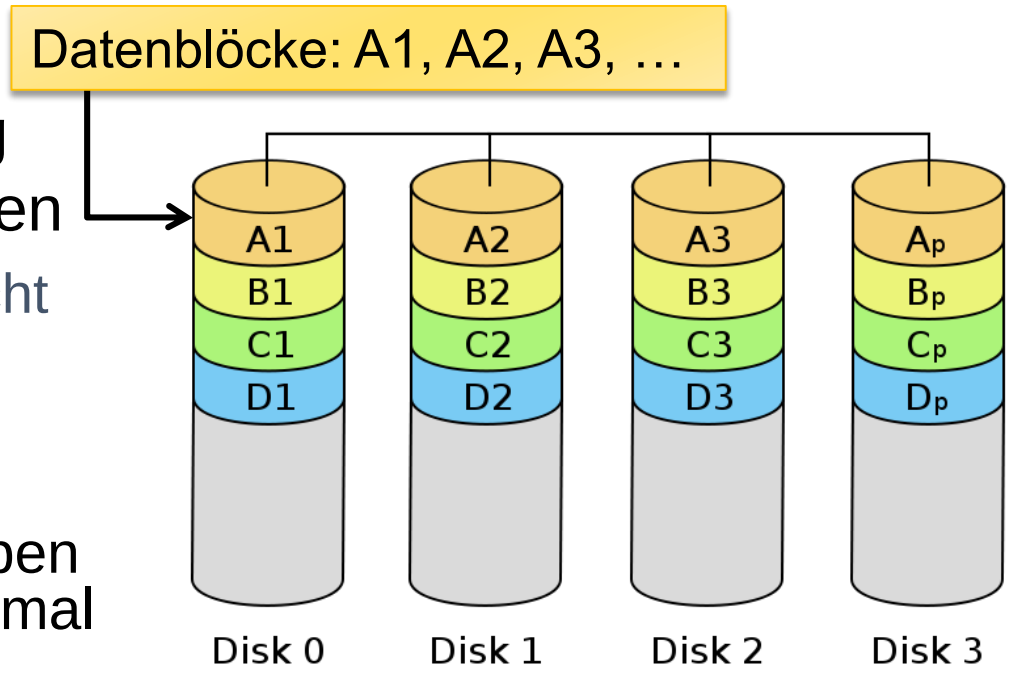
RAID 3 - Byte-Level-Striping

- Hier werden aufeinanderfolge Bytes auf separate FP geschrieben und die **Paritätsinformation** auf eine weitere FP
  - Diese **Paritätsinformation** ist Ergebnis der XOR-Verknüpfung einzelner Bytes
- Datenstrom (in ganzen Bytes):
  - A1, A2, A3, A4, A5, A6,
  - B1, B2, B3, B4, B5, B6,
  - …
- Umsetzung sehr komplex => Wird nicht mehr genutzt (Link)



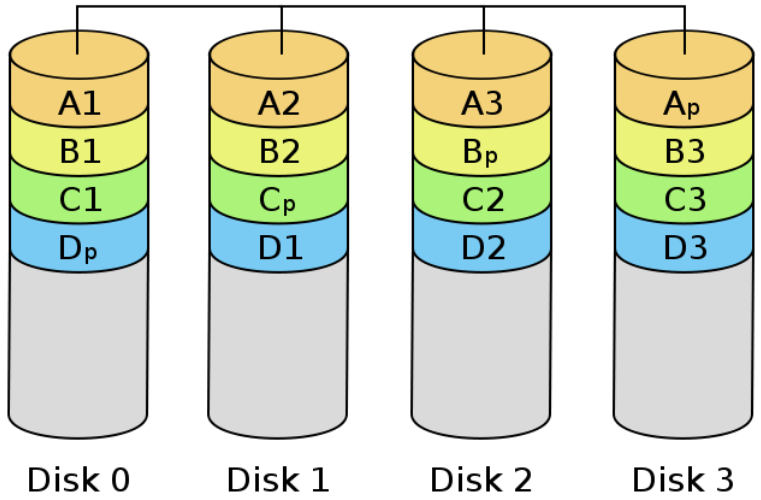
RAID 4 – Block-Level-Striping

- Aufeinanderfolgende Blöcke von Daten (z.B. 512-Bytes-Blöcke) werden auf verschiedene FP geschrieben
  - Eine zusätzliche FP speichert die Paritätsinformation
- Wenn eine FP ausfällt, können die n-1 Daten-FP + die Parität-FP zur Rekonstruktion benutzt werden
- Vorteile, Nachteile?
- (+) Parallele Abarbeitung mancher 1-Block-Anfragen
  - A1 || B2 geht, A1 & B1 nicht
- (-) Schreibvorgänge sind langsamer – warum?
- Wenn A1 und B2 geschrieben werden, muss Disk 3 zwei mal schreiben



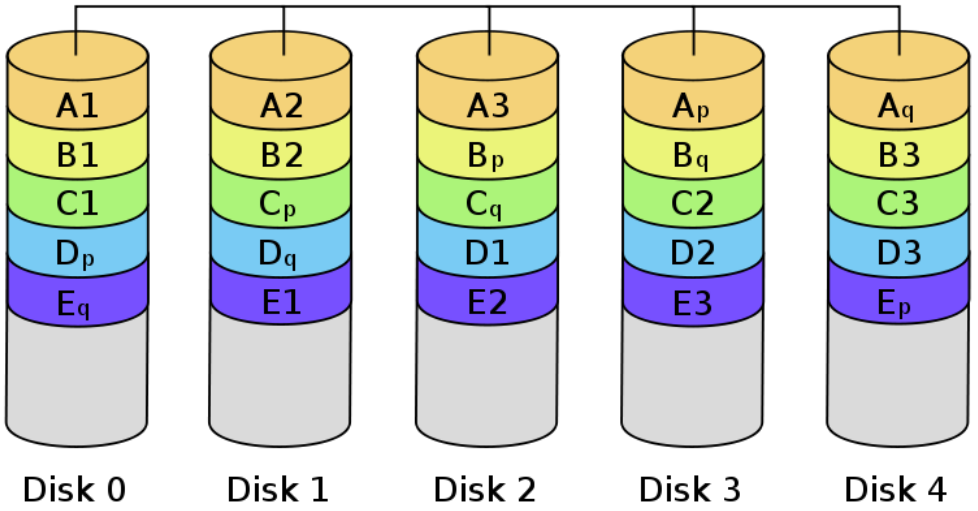
RAID 5: **Block-interleaved distributed parity**

- Wie RAID 4 werden aufeinanderfolgende Blöcke auf verschiedene FP geschrieben
- ABER: die Paritätsinformationen werden „gleichmäßig“ auf allen FP verteilt
  - Z.B. bei k FP wird die Paritätsinformation für Block n auf der FP mit Index (n mod k) gespeichert
- Warum dieser Unterschied zu RAID 4?
- So werden alle FP gleichmäßig benutzt; das Schreiben ist schneller
  - Bei RAID 4 wird die P.-FP übermäßig benutzt und fällt schneller aus
- Zurzeit häufig verwendet



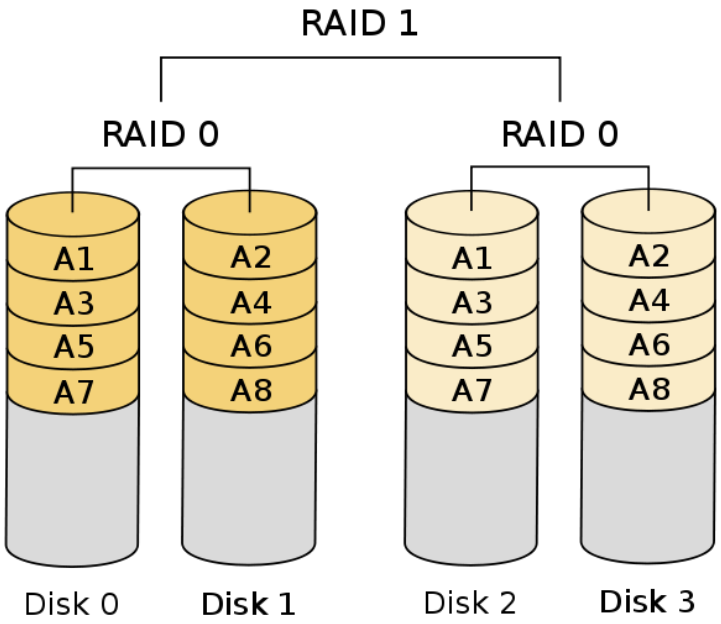
RAID 6 - **P + Q redundancy scheme**

- Wie RAID 5, aber mit mehr Redundanz
- Dadurch kann man auch 2-Bit-Fehler korrigieren
- Oft Verwendung von Reed-Solomon-Codes
  - Z.B. für je 4 Bits von Daten werden 2 Bits redundanter Daten gespeichert => Wiederherstellung bei 2-Bit-Fehlern möglich
- Ein RAID-6-Verbund benötigt mindestens vier Festplatten
- Mehr Informationen: Link

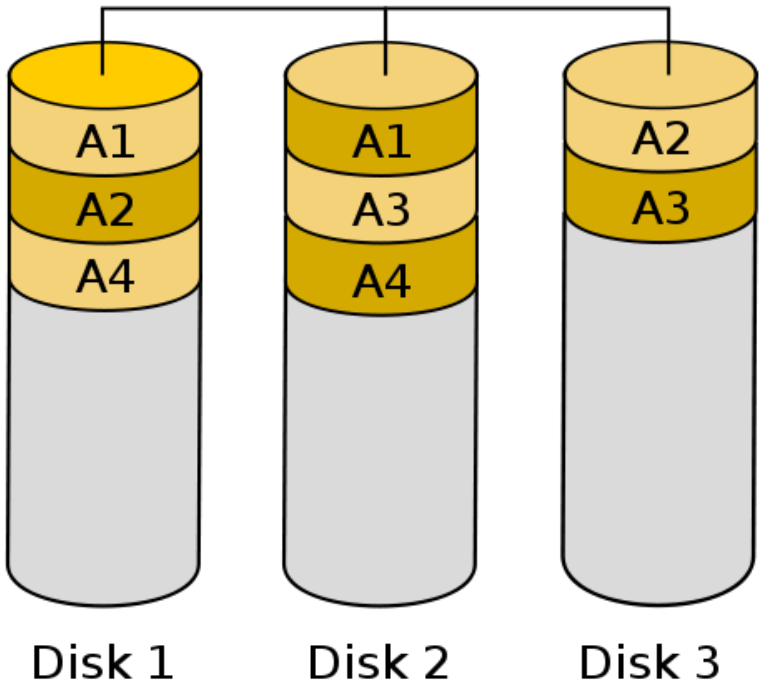


RAID 01 (bzw. 0+1)

- Es ist ein RAID 1 über mehrere RAID 0‘s
  - „Unten“ ist RAID 0: Striping auf Blockebene
  - Jedes solche FP-Paar (mit RAID 0) wird durch ein weiteres Paar repliziert

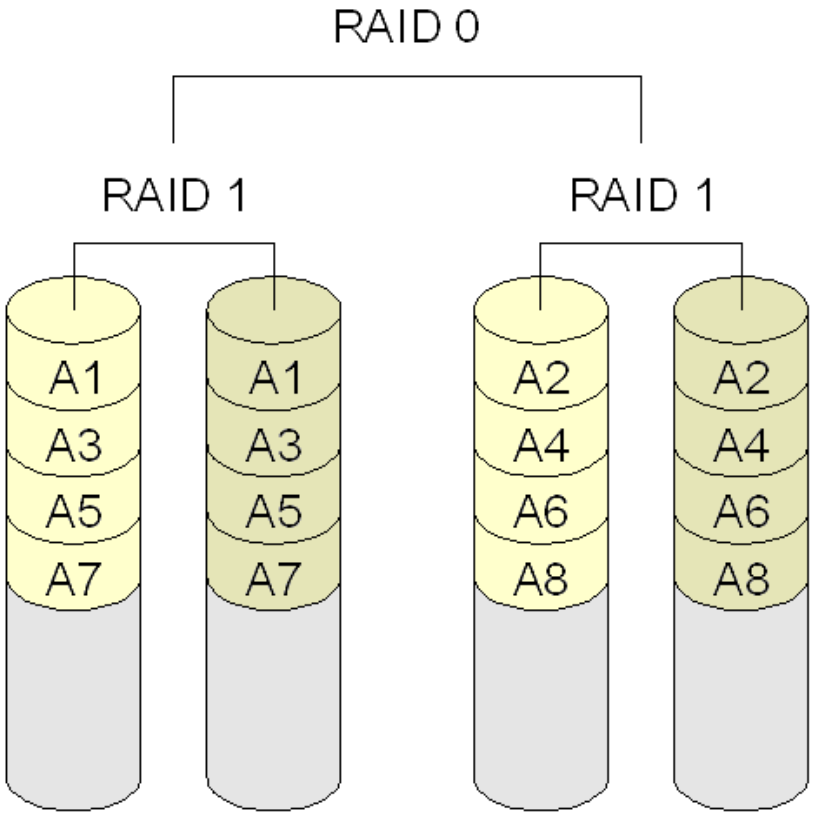


- Geht es auch mit 3 FP?
- Ja, siehe http://de.wikipedia.org/wiki/RAID



RAID 10 (bzw. 1+0)

- RAID 0 über mehrere RAID 1-Systeme
- Vorteil gegenüber 0+1: bessere Ausfallsicherheit und schnelle Rekonstruktion nach einem FP-Ausfall



Zusammenfassung

- Implementierung von Dateisystemen - historische Beispiele
- Dateisysteme ext2, ext3, ext4, zfs
- Struktur und Scheduling von Festplatten
  - Minimierung der Seek-Zeit einer Festplatte
  - Algorithmen: SSTF, SCAN, C-SCAN, C-LOOK
  - SSD und NVM verdrängen magnetische FP
- RAID: Höhere Leistung und Verlässlichkeit durch Verbund mehreren Festplatten
- Quellen (Dateien): Silberschatz et al. Kap. 11+12; Tanenbaum Kap. 4, 11, 10; Wikipedia

Zusätzliche Folien: Scheduling von Festplatten

Auswahl des Algorithmus

- SSTF ist weit verbreitet und „natürlich“
- SCAN und C-SCAN sind geeignet für Systeme, die eine FP intensiv nutzen
- Ein solcher Algorithmus sollte als ein separates Modul (d.h. Plug-in) implementiert werden, damit das BS bei Bedarf den Algorithmus ersetzen kann
  - Was wären mögliche API-Aufrufe?
- Moderne Disk-Controller führen diese Algorithmen selbst aus
  - **Vorsicht**: Manchmal muss die Reihenfolge der Anfragen als FCFS erhalten bleiben!
    - Z.B. Eine Datei erzeugen => I-Node schreiben; an die Datei anhängen