5. Dateisystem

hier: **lokale** Dateisystem

Das Dateisystem ist eine logische Abstraktion des genutzten physikalischen Speichers.
Betriebssystemsicht auf eine Datei: Folge von Bytes

```
> ll folien.tex
-rw-r--r-- 1 schnor users 10098 Dec 12 09:12 folien.tex
```

Historie: Frühe Betriebssysteme gestatteten nur sequentiellen Zugriff auf Dateien.

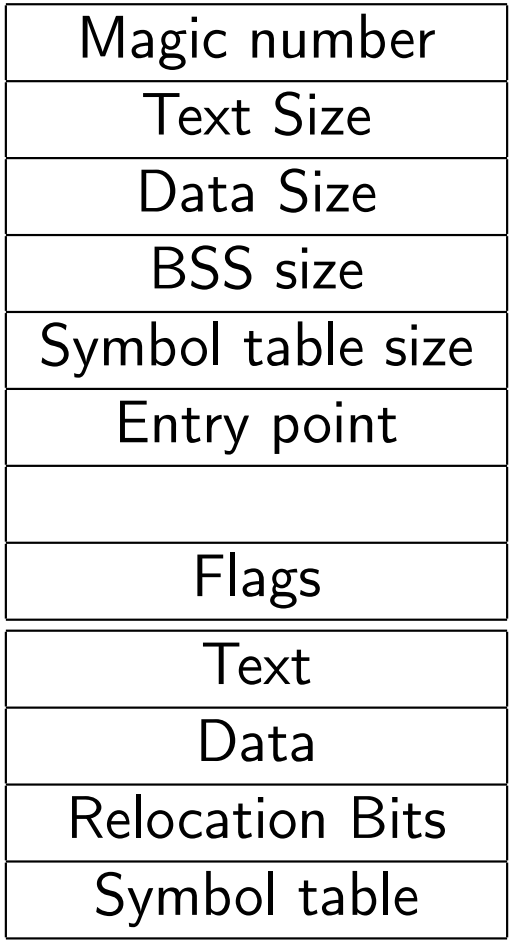
Dateitypen:

- Benutzerdateien: ASCII-Dateien, Binaries

  Beispiel: UNIX führt ein Programm nur aus, wenn es einem bestimmten Format genügt.

- Systemdateien: Directories, Dateien zur Modellierung von I/O-Geräten wie z.B. Terminals und Platten

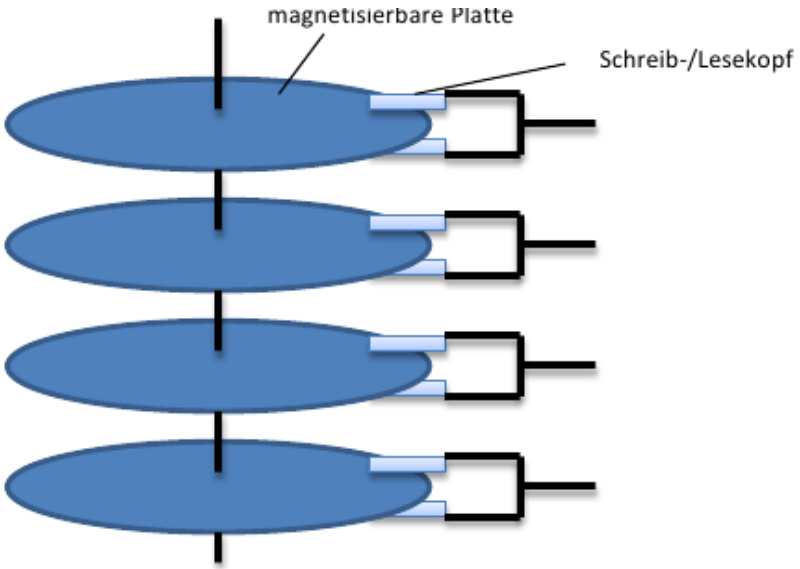
UNIX Executable Format `a.out` (aktuell: ELF-Format)

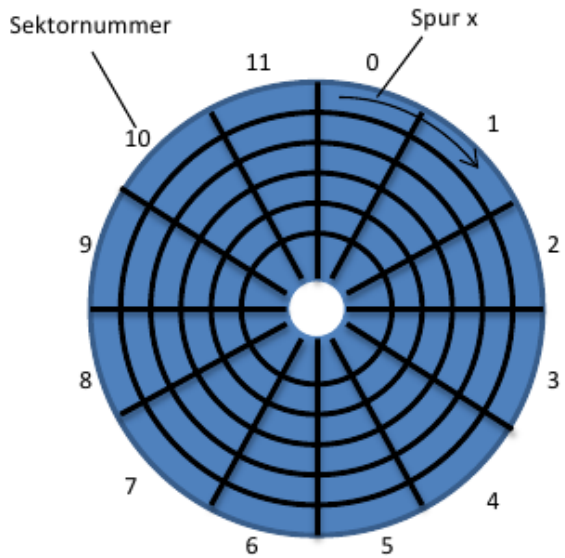


```
> ll folien.tex
-rw-r--r-- 1 schnor users 10098 Dec 12 09:12 folien.tex
```

Ansatz: Analog zum Hauptspeicher und dem virtuellen Adreßraum eines Prozesses werden Dateien und Festplatte in Blöcke gleicher Größe eingeteilt.

Physikalischer Aufbau Festplatte





Quelle: Mario Schölzel

Definition 30: Ein Platten- bzw. Diskettenlaufwerk (engl. drive) besteht aus einer oder mehreren Magnetplatten mit jeweils unter- oder oberseitigen Lese/Schreibköpfen.
Die Daten werden auf Kreislinien auf der Oberfläche der Magnetplatten gespeichert, den sogenannten Spuren (engl. tracks).
Die Spuren werden durchnummeriert. Die Spuren n aller Oberflächen heißen Zylinder n. Spuren werden in Abschnitte fester Datenlänge eingeteilt, den sogenannten Sektoren.

Häufig spricht man anstatt von Sektoren auch von *Blöcken*.

Beispiel:
ATA device, Model Number: SAMSUNG HD161HJ
Configuration:

Logical	max
cylinders	16383
heads	16
sectors/track	63
Logical/Physical Sector size:	**512 bytes**

seit ca. 2010 auch die Verwendung vom: Advanced Format mit 4096 Bytes großen Sektoren

Das Betriebssystem merkt sich zu jeder Datei Attribute wie z.B.

- Zugriffsrechte
- Eigentümer
- Erzeugungsdatum
- Zeitpunkt der letzten Änderung
- Dateigröße
- ...

Dateioperationen: `open()`, `read()`, `write()`, `close()`, `rm()`, Umbenennen, Attribute lesen und setzen.

```
int open(const char *path, int oflag)
```

- The first argument points to a pathname naming the file.
- Open Flags für read-only, write-only, read-write, append, create, ...
- The open() function **establishes the connection between a file and a file descriptor.** The file descriptor is used by other I/O functions to refer to that file.
- The open() function **returns a file descriptor** for the named file that is the lowest file descriptor not currently open for that process.
- The file offset used to mark the current position within the file is set to the beginning of the file.

Memory Mapped Files

Datei wird in den virtuellen Adreßraum eines Prozesses eingeblendet.
$\Longrightarrow$ Shared Memory zwischen Prozessen

Beispiel: Dynamischer Linker

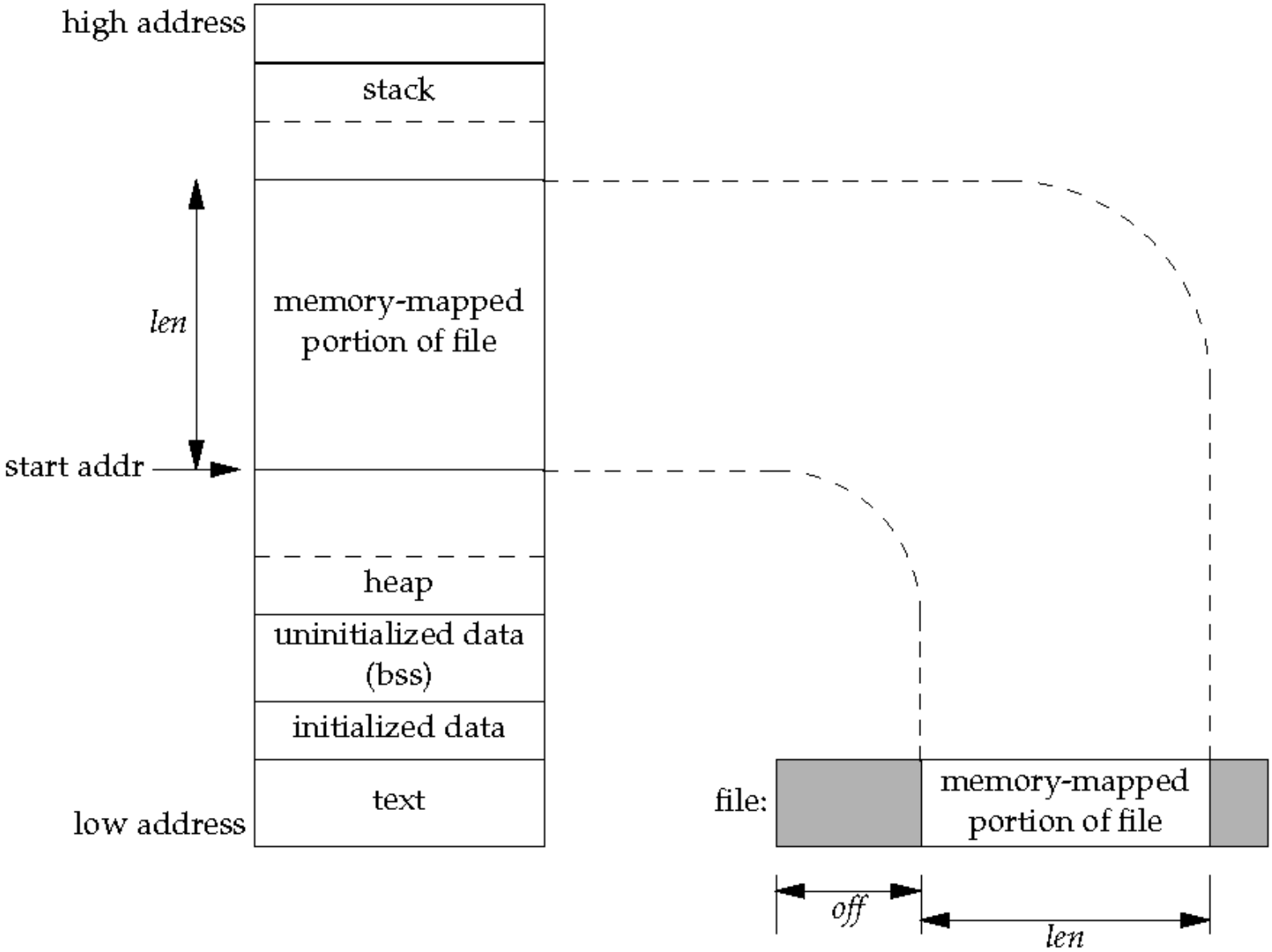


Figure 14.26 Example of a memory-mapped file

Quelle: Stevens, Rago: *Advanced Programming in the UNIX Environment*

Beispiel: Im 4.4 BSD enthalten[22]: `mmap()`

```
NAME
mmap - map files or devices into memory
mmap(caddr_t addr, size_t len, int prot, int flags,
int fd, off_t offset)


DESCRIPTION
The mmap() function causes the pages starting at addr and
continuing for at most len bytes to be mapped from the
object described by fd, starting at byte offset offset.
```

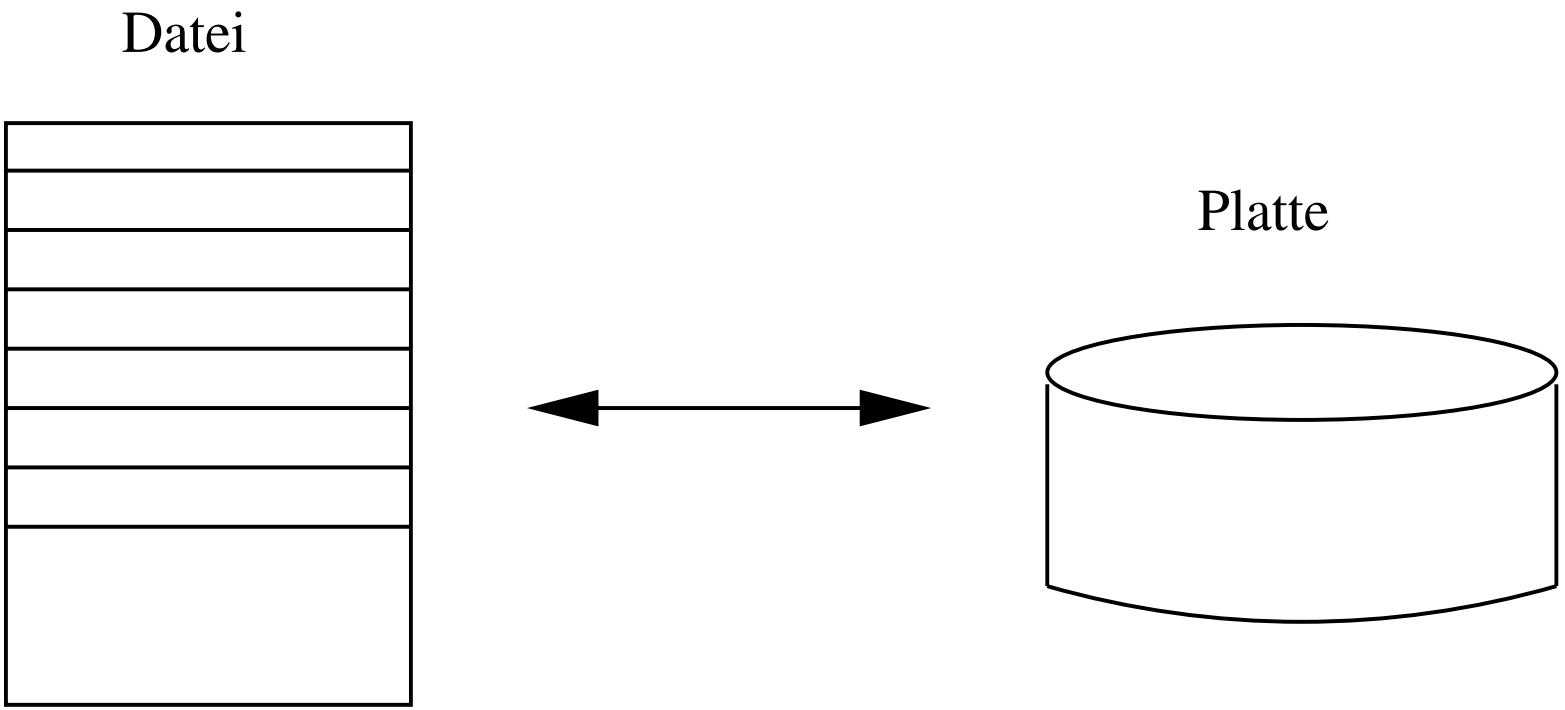
[22] Die virtuelle Speicherverwaltung vom 4.4 BSD basiert auf dem Mach 2.0 mit Updates von Mach 2.5 und Mach 3.0.

```
NAME
  vdso - overview of the virtual ELF dynamic shared object

SYNOPSIS
  #include <sys/auxv.h>

  void *vdso = (uintptr_t) getauxval(AT_SYSINFO_EHDR);
```

The "vDSO"(virtual dynamic shared object) is a small shared library that the kernel automatically maps into the address space of all user- space applications. Applications usually do not need to concern themselves with these details as the vDSO is most commonly called by the C library. This way you can code in the normal way using standard functions and the C library will take care of using any functionality that is available via the vDSO.

Why does the vDSO exist at all? There are some system calls the kernel provides that user-space code ends up using frequently, to the point that such calls can dominate overall performance. This is due both to the frequency of the call as well as the context-switch overhead that results from exiting user space and entering the kernel.

siehe `>man vdso`



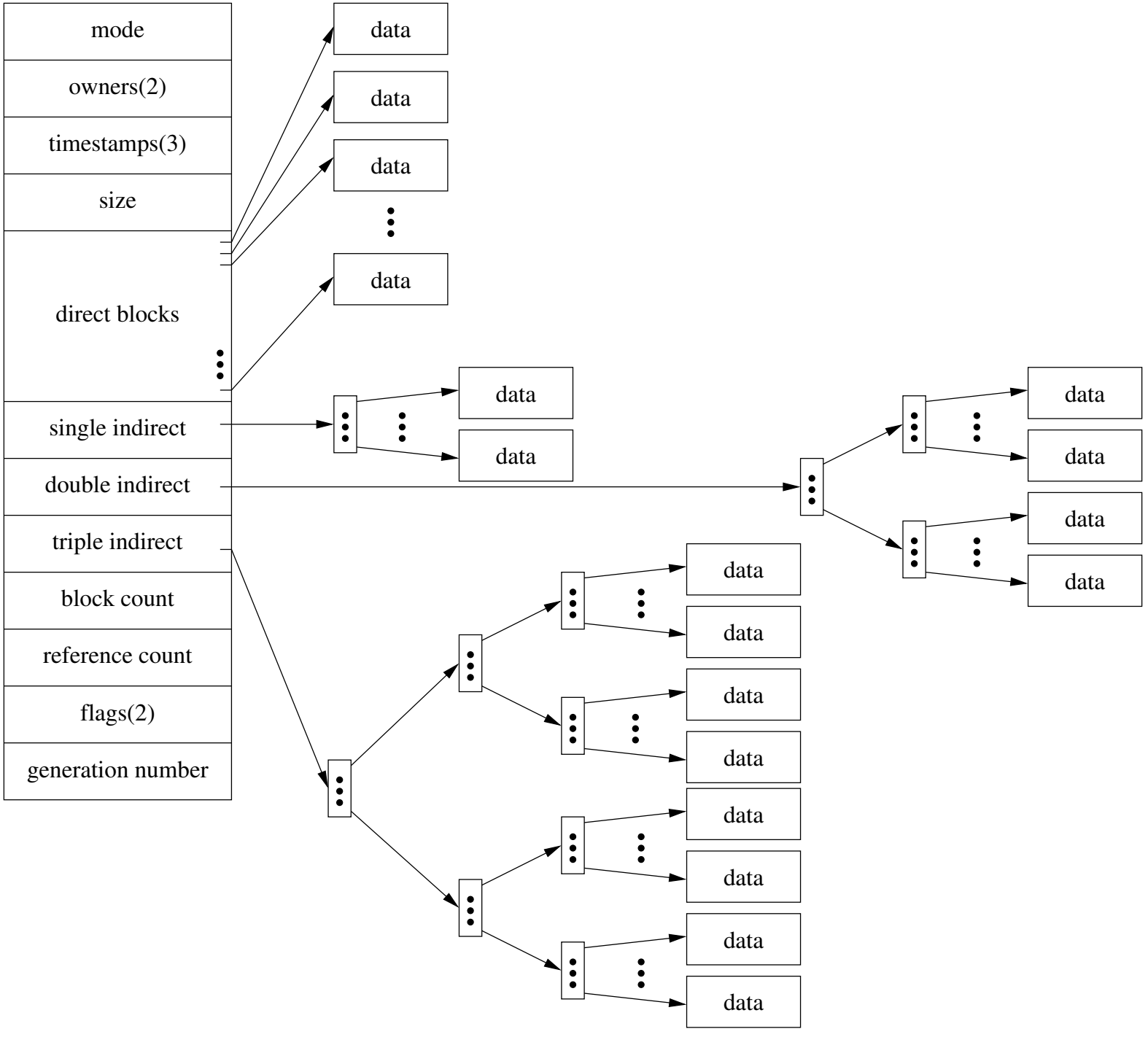
Beispiel: Eine 40 KB große Datei benötigt 10 Blöcke à 4KB
$\Longrightarrow$ Wie soll die Datei auf dem Hintergrundspeicher abgespeichert werden?

1. Möglichkeit: Die Datei wird auf einem zusammenhängenden Plattenbereich abgespeichert.

2. Möglichkeit: Die Datei wird auf n Blöcke, die auf der Platte verteilt sein können, abgespeichert.

Beispiel: UNIX i–node (index-node)

mode
owners (2)
timestamps (3)
size
12 direct blocks
single indirect
double indirect
triple indirect
block count
reference count
flags
generation number


mode
owners(2)
timestamps(3)
size
direct blocks
single indirect
double indirect
triple indirect
block count
reference count
flags(2)
generation number
data

Üblicherweise wird das Dateisystem mittels Dateiverzeichnissen (Directories) hierarchisch strukturiert.

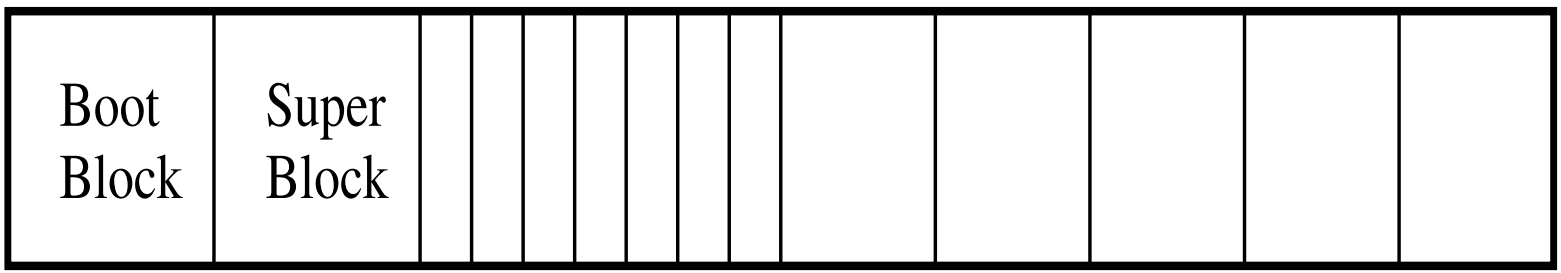
UNIX-Directories sind Dateien mit Einträgen von der Form:

inode number	file name

Auslesen der Inode-Nummer für die Datei `foo` aus dem aktuellen Dateiverzeichnis:

```
>ls -i foo
```

Traditionelle Platteneinteilung im UNIX:

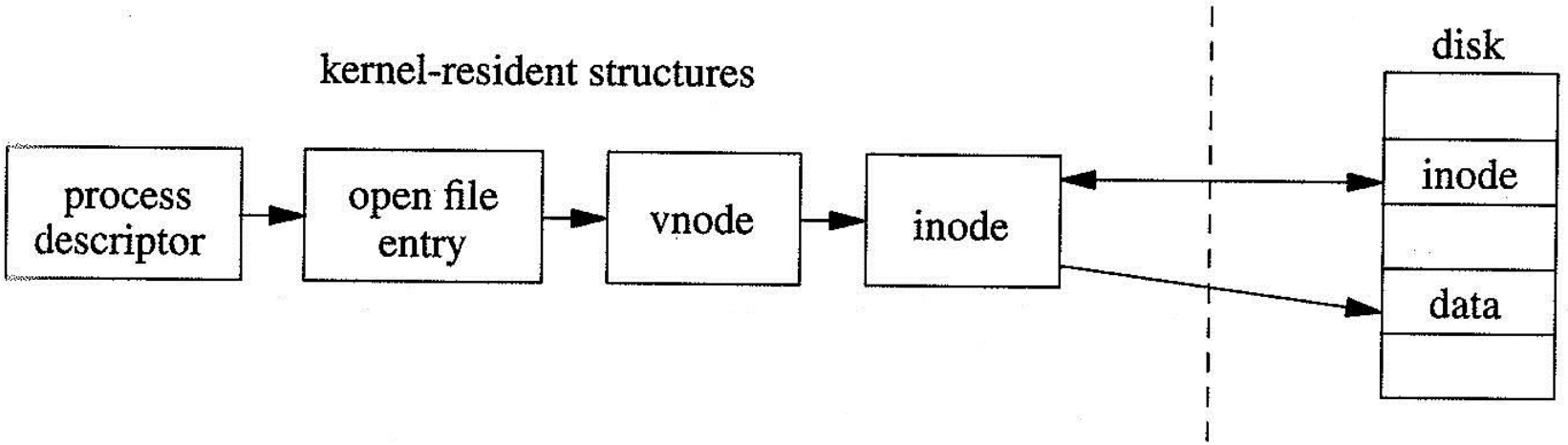


Superblock enthält Dateisystem-Informationen wie z.B.

- Blockgröße
- Anzahl und Lage der inodes
- ...

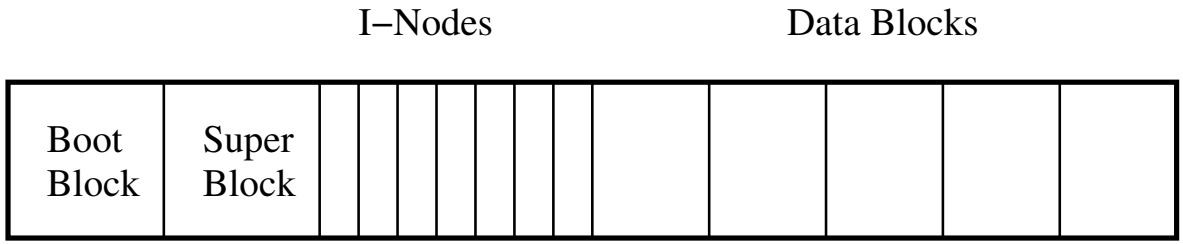
Datenstrukturen für Öffnen und Lesen einer Datei

Figure 7.2 Layout of kernel tables.



Quelle: M. K. McKusick, K. Bostic, M. J. Karels und J. S. Quaterman: *The Design and Implementation of the 4.4 BSD Operating System*

Beispiel: > `less` /home/schnor/mbox



Beispiel: `> less /home/schnor/mbox`

1. Anker: Root-Directory wird durch den inode 2 beschrieben
   $\Longrightarrow$ inode 2 lesen
2. Blockadresse für '/' aus dem inode 2 auslesen: BA1
   $\Longrightarrow$ Block BA1 einlesen
3. inode für /home bestimmen: IN1
   $\Longrightarrow$ ggf. inode IN1 einlesen
4. Blockadresse für /home aus dem inode IN1 auslesen: BA2
   $\Longrightarrow$ Block BA2 einlesen
5. inode für /home/schnor bestimmen: IN2
   $\Longrightarrow$ ggf. inode IN2 einlesen

6. Blockadresse für /home/schnor aus dem inode IN2 auslesen: BA3

   $\Longrightarrow$ Block BA3 einlesen

7. inode für /home/schnor/mbox bestimmen: IN3

   $\Longrightarrow$ ggf. inode IN3 einlesen

8. Blockadresse für /home/schnor/mbox aus dem inode IN3 auslesen: BA4

   $\Longrightarrow$ Block BA4 einlesen 

Operationen auf Dateiverzeichnissen:

- create
- delete
- opendir
- closedir
- readdir
- rename
- link
- unlink

bisher: Datei lesen

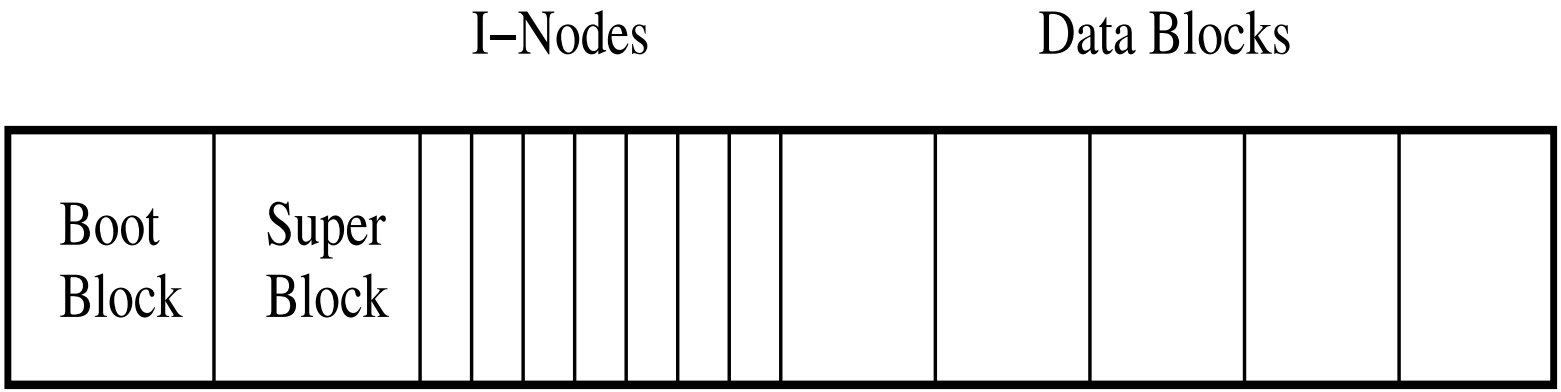
Wie finde ich eine Datei bzw. die zugehörigen Blöcke schnell auf der Platte?

fehlt noch: Datei schreiben

Wie finde ich freien Platz, d.h. freie Blöcke auf der Platte?

Marshall Kirk McKusick, 4.2 BSD, 1984
Ziel: Performance

Traditionelle Platteneinteilung im UNIX



Superblock enthält Dateisystem-Informationen wie z.B.

- Blockgröße
- Anzahl inodes
- Beginn der Free-Block-Liste

Verwaltung der freien Blöcke auf der Platte

1. Möglichkeit: Einfach verkettete Liste von Platten-Blöcken.
Blöcke enthalten als Daten die Blocknummern von freien Blöcken.

2. Möglichkeit: Bitmap

Platte enthält n Blöcke $\Longrightarrow$ n Bit

$$k\text{-te Bit} = \begin{cases} 0 & \text{Block } k \text{ ist belegt} \\ 1 & \text{Block } k \text{ ist frei} \end{cases}$$

Beispiel: 4.2 BSD: Liste, ab 4.4 BSD Bitmap

Optimierungen im Berkeley Fast File System

Hauptproblem: Lange Zugriffszeiten (engl. Seek Time) auf dem Speichermedium wie z.B. der Festplatte, bis der Lese-/Schreibkopf positioniert ist.

1. Änderung: Seek Time verkürzen, indem inode und zugehörige Datei näher auf der Platte zusammengelegt werden $\Longrightarrow$ Platte wird in Cylinder Groups eingeteilt. Jede Cylinder Group hat eigenen Superblock, inodes, Bit Map für freie Blöcke in der Cylinder Group und Datenblöcke. Sofern möglich, werden Datenblöcke in demselben Zylinder abgespeichert wie der zugehörige inode.

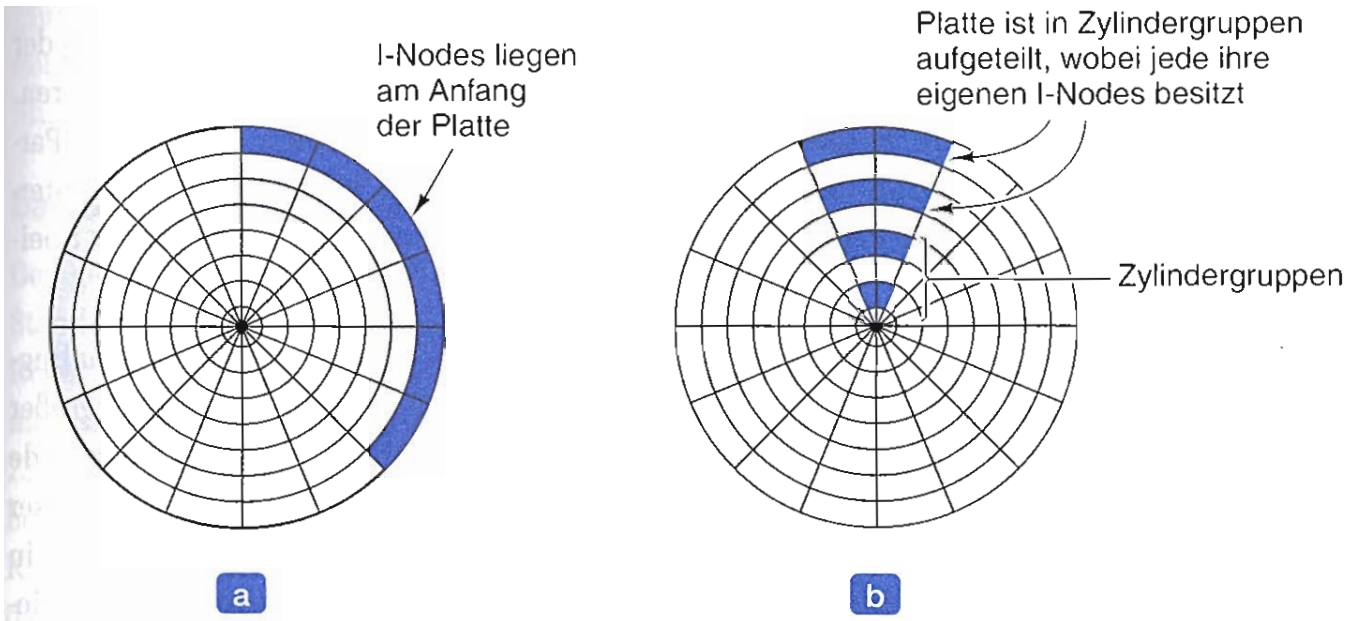


Abbildung 4.29: (a) Die Platzierung der I-Nodes am Anfang der Platte (b) Die Platte aufgeteilt in Zylindergruppen mit jeweils eigenen Blöcken und I-Nodes

Quelle: Andrew S. Tanenbaum: *Moderne Betriebssysteme*, 3. Auflage, 2009.

2. Änderung: Erhöhen der Blockgröße

Forderung: Dateien $> 2^{32}$ Bytes sollen mit Double Indirect Blocks auskommen $\Longrightarrow$ Blockgröße ≥ 4 KB

Monitoring-Studien:

1984 Mullender und Tanenbaum: Median der Dateigröße $\approx$ 1KB

1993 Irlam: Median der Dateigröße $<$ 2KB
mittlere Dateigröße $\approx$ 22 KB

Plattenverschnitt:

Platz für inodes + Fragmentierung auf dem letzten Block
+ Platz für Bitmap

Plattenverschnitt bei mittlerer Dateigröße 22 KB:

Blockgröße	Verschnitt
1 k	8,8 %
2 k	11,7 %
4 k	15,4 %
8 k	29,4 %

Studie auf Linux-Rechnern am IFI (2006)

Klemens Kittan, Heinrich Lenz

Dateien	0-1 KB		1-2 KB		2-3 KB		3-4 KB		< 4KB		> 4 KB	
gesamt	Anzahl	%	Anzahl	%	Anzahl	%	Anzahl	%	Anzahl	%	Anzahl	%
142 371	46 713	33	22 774	16	12 532	9	7 541	5	94 344	66	48 027	34

$\Longrightarrow$ 66 % der Dateien sind **kleiner als 4 KB**!

Studie IFI Fileserver (2016)

Fileserver hält 6 177 792 Dateien.
Median der Dateigröße: 2467 Bytes

Größe	Anzahl	Prozent
0 Bytes	259995	4,3 %
- 512 Bytes	1193957	20 %
- 1024 Bytes	685428	11 %
- 2048 Bytes	648567	11 %
- 3072 Bytes	463197	8 %
- 4096 Bytes	328636	5,5 %

$\Longrightarrow$ Knapp 60 % aller Dateien sind zwischen 0-4 KB groß bzw. nur 40 % größer als 4 KB.

Studie IFI Fileserver (2022)

Fileserver hält 11 070 093 Dateien.
Median der Dateigröße: knapp unter 3 KBytes

	2016		**2022**	
Größe	Anzahl	Prozent	Anzahl	Prozent
0 Bytes	259995	4,3 %	–	
- 512 Bytes	1193957	20 %	2204938	20 %
- 1024 Bytes	685428	11 %	1183160	11 %
- 2048 Bytes	648567	11 %	1420672	13 %
- 3072 Bytes	463197	8 %	797661	7 %
- 4096 Bytes	328636	5,5 %	604620	5 %

$\Longrightarrow$ 56 % aller Dateien sind zwischen 0-4 KB groß bzw. 44 % sind größer als 4 KB.

FFS unterteilt Blöcke in Fragmente. Die Fragmentgröße wird beim Erzeugen des Dateisystems festgelegt (im Superblock).

Belegungsregel: Bis auf den letzten belegt eine Datei nur vollständige Blöcke. Für den Rest der Datei sind freie Fragmente „in der Nähe“ zu suchen.

Beispiel: 4096/1024 Filesystem $\Longrightarrow$ Für jeden Block 4 Einträge in der Bit Map.

Bit Map	- - - -	- - 1 1	1 1 - -	1 1 1 1
Fragment-Nr.	0 - 3	4 - 7	8 - 11	12 - 15
Block-Nr.	0	1	2	3

Ext2 $\approx$ Berkeleys Fast FS:

- Platte wird in „Block Groups" aufgeteilt. Jede Blockgruppe hat eigene Verwaltungsdaten.
- Bitmap für freie Blöcke
- Bitmap für freie inodes
- 12 direkte, 3 indirekte Blockadressen im inode

Unterschiede:

Blockgröße zwischen 1–4 KByte konfigurierbar, üblich sind 4 KByte
keine Block/Fragment-Aufteilung

ext3 ist ein Journaling-Filesystem:

Idee: Das Dateisystem führt Logbuch „Journal".

Schreiben eines Blockes:

1. Kopie des Blockes wird im Journal gespeichert
2. Block wird im Dateisystem geschrieben
3. Kopie im Journal löschen

Beispiel: Löschen einer Datei

1. Löschen der Datei aus ihrem Verzeichnis
2. Inode aktualisieren $\Longrightarrow$ Reference-Count $--$
3. Falls Reference-Count$==0$ $\Longrightarrow$ Freigeben des Inodes in den Pool der freien Inodes (zugehörige Bitmap aktualisieren)
4. Freigeben der zugehörigen Plattenblöcke (zugehörige Bitmap aktualisieren)

Im Journal kann notiert werden, bei welchem Arbeitsschritt das Dateisystem aktuell ist.

$\Longrightarrow$ Schnellere Konsistenzchecks nach Systemabsturz.

Beispiel ext3 hat 3 Modi:

1. Journal: Alle Änderungen an Metadaten des Dateisystems und Dateien werden im Journal protokolliert. $\Longrightarrow$ sicherste, aber langsamste Modus
2. Ordered: Default-Einstellung, bei der nur die Änderungen an den Metadaten protokolliert werden. Zudem werden Datenblöcke vor den zugehörigen Metadatenänderungen auf Platte geschrieben.
3. Writeback: Nur die Änderungen an den Metadaten werden im Journal gemerkt. $\Longrightarrow$ schnellste Modus

Bem.: Journaling-Dateisysteme sind ReiserFS (erstes Journaling-Filesystems für Linux, aber nicht kompatibel zu ext2), ext3, xfs von SGI, JFS von IBM.
Die übrigen Journaling-Dateisysteme arbeiten im Writeback-Modus.

Literatur: Daniel P. Bovet, Marco Cesati: *Understanding the Linux Kernel*, O'Reilly, 3. Auflage, 2005.

Beispiel: ext4 (Default in Linux)

- Journaling-Dateisystem
- Unterstützung großer Dateisysteme und Dateien: Volumes bis 1 Exbibyte (EiB) und Dateien bis 16 Tebibytes (TiB) $\Longrightarrow$ von 32-Bit auf 48-Bit-Blockadressen umgestellt
- schneller Zugriff auf große Dateien: ext4 unterstützt sogenannte Extents (z.B. in JFS und XFS schon länger vorhanden), um Adressen von zusammenhängenden physikalischen Blocks effizienter abzuspeichern.
- Änderungen erfordern das Anpassen der inode-Struktur

Quelle: Avantika Mathur et al.: *The new ext4 filesystem: current status and future plans*, Linux Conference, 2007.

IEC 60027-2

International Electrotechnical Commission (IEC) spezifiziert in IEC 60027-2 2005, third Edition, Einheiten zur Basis 2: kibibytes, mebibytes, gibibytes, exbibytes

Faktor	Name	Symbol
2^{10}	kibi	Ki
2^{20}	mebi	Mi
2^{30}	gibi	Gi
2^{40}	tebi	Ti
2^{50}	pebi	Pi
2^{60}	exbi	Ei

aber: Netzwerkbereich: Basis 10 gemäß dem SI-Standard

5.5 Windows 2000: NTFS

- seit Windows NT
- Journaling-Filesystem
- organisiert Platte in „Volumes“,
  Blockgröße kann **pro Volume** festgelegt werden:
  512 Bytes–64 KByte, üblich sind 4 KByte
- statt inodes gibt es: Master File Table (MFT)
  Jedes Volume besitzt eine MFT.
- Einträge (engl. Records) in der MFT à 1KB $\gg$ | inode |[23]
- Anfangsadresse der MFT-Datei steht im Boot-Block.

[23] BSD inode ist 128 Byte groß.

Die ersten 16 MFT-Einträge sind Metadaten des Dateisystems:

Record		
0	:	beschreibt MFT, z.B. Anfangsadresse
1	:	Kopie von 0
2	:	Logfile: Aktion (z.B. Create Directory, wird erst geloggt und dann ausgeführt)
3	:	Volume-Beschreibung
4	:	Definition der Datei-Attribute
5	:	Root Directory
6	:	Eintrag für die Datei, in der die Bitmap für Blöcke
	:	gespeichert wird.
...	:	...
ab 16	:	User File 1
...	:	...
...	:	...

Ein MFT-Record für eine Benutzerdatei oder ein Directory enthält alle beschreibenden Informationen (analog zum inode):

- 13 mögliche Attribute
- Attribute werden als Tupel (type, length, value) gespeichert
- Standard-Attribute: Besitzer, Zeitstempel, ...
- Attribut: Dateiname (!)
- Attribut: Blocknummern der Blöcke, auf denen die Daten der Datei gespeichert sind
  Abspeichern in *Serien*: (Startadresse, Länge der Serie)

Beispiel: Die Datei `foo` ist auf 9 Blöcken verteilt auf der Platte gespeichert: 20-23, 64-65, 80-82

- UNIX merkt sich im inode die 9 Blockadressen: 20, 21, 22, 23, 64, 65, 80, 81, 82
- NTFS merkt sich im MFT-Record die drei zusammenhängenden Serien: (20,4), (64,2), (80, 3)

Ein MFT-Record für eine Benutzerdatei oder ein Directory enthält alle beschreibenden Informationen (analog zum inode):

- 13 mögliche Attribute
- Attribute werden als Tupel (type, length, value) gespeichert
- Standard-Attribute: Besitzer, Zeitstempel, ...
- Attribut: Dateiname (!)
- Attribut: Blocknummern der Blöcke, auf denen die Daten der Datei gespeichert sind
  Abspeichern in *Serien*: (Startadresse, Länge der Serie)
- Dateien und Dateiverzeichnisse unter 1 KB können direkt auf dem MFT-Eintrag gespeichert werden!!!

Literatur: Andrew S. Tanenbaum: *Moderne Betriebssysteme*, Pearson Studium, Addison Wesley, 3. Auflage, 2009.

Performanzvergleich ANNO 2017

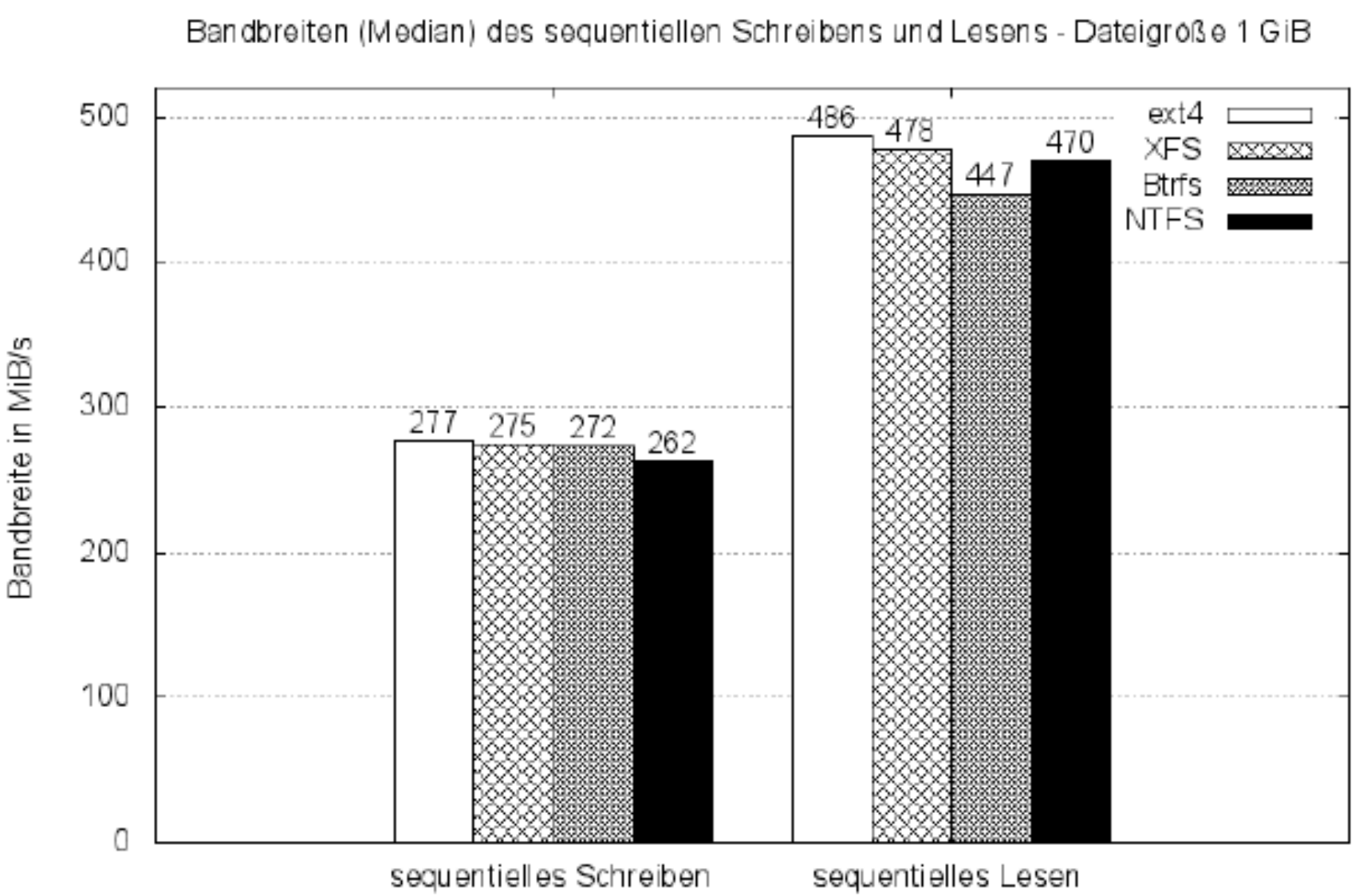
Eckehart Synnatzschke: *Benchmarking der Dateisysteme ext4, XFS, Btrfs und NTFS*, Bachelorarbeit, Uni Potsdam, 2017.

- Messungen mit deaktiviertem Pagecache
- Benchmark: `fio` (Flexible I/O Tester) ist sowohl auf Linux als auch Windows lauffähig
- Blockgröße 4 KB
- Desktop-PC mit Debian GNU/Linux 8.8 Jessie, Windows 10Pro
- vollständige Versuchsbeschreibung siehe Bachelorarbeit

Experimente:

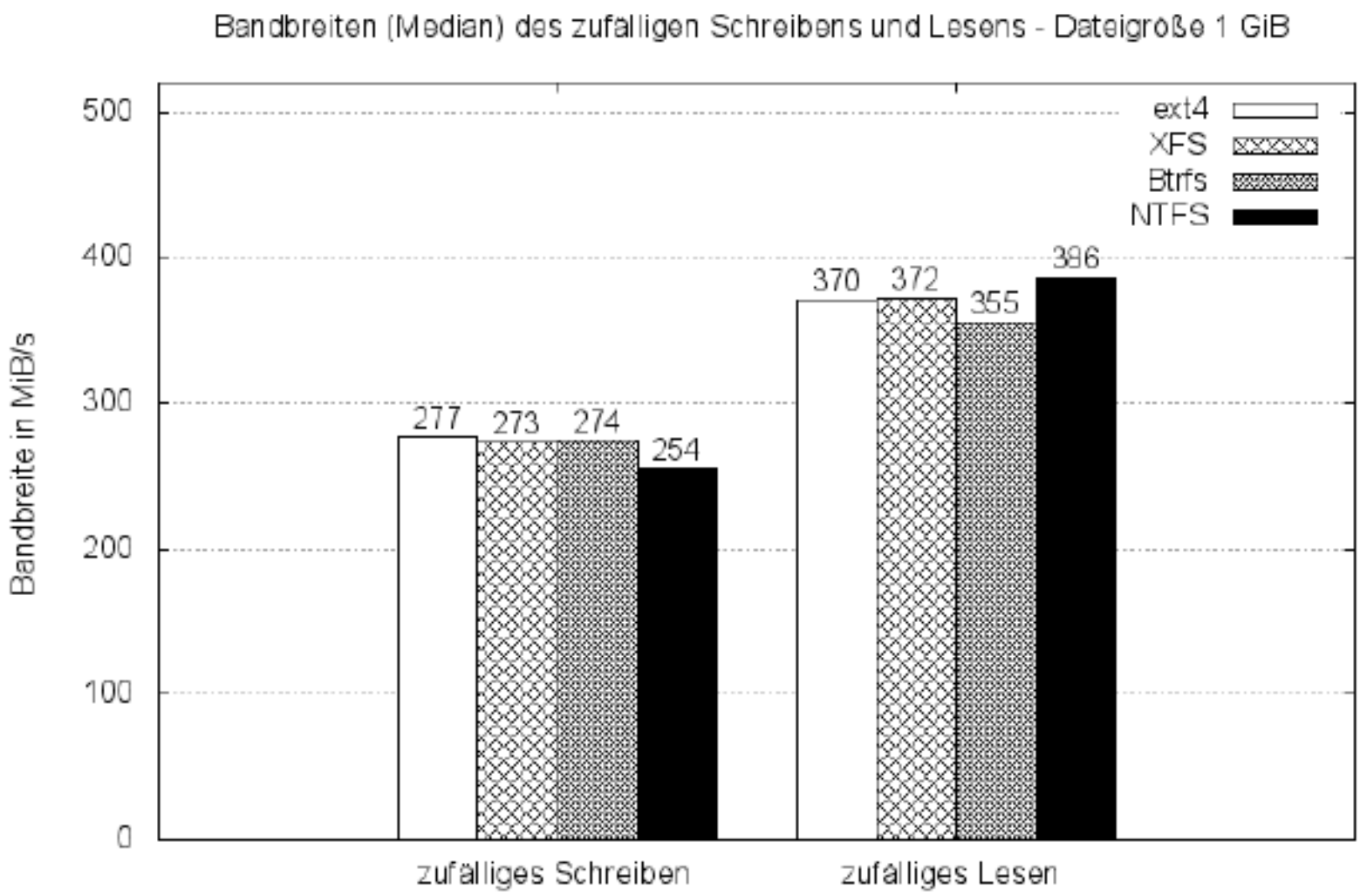
1. Das Schreiben und Lesen erfolgt für die Dateigröße 1 GiB sowohl sequentiell als auch zufällig.
2. Die Dateien der Größen 512 bis 1032 Byte werden nur sequentiell gelesen und geschrieben.

Diagramm 15 (Dell):



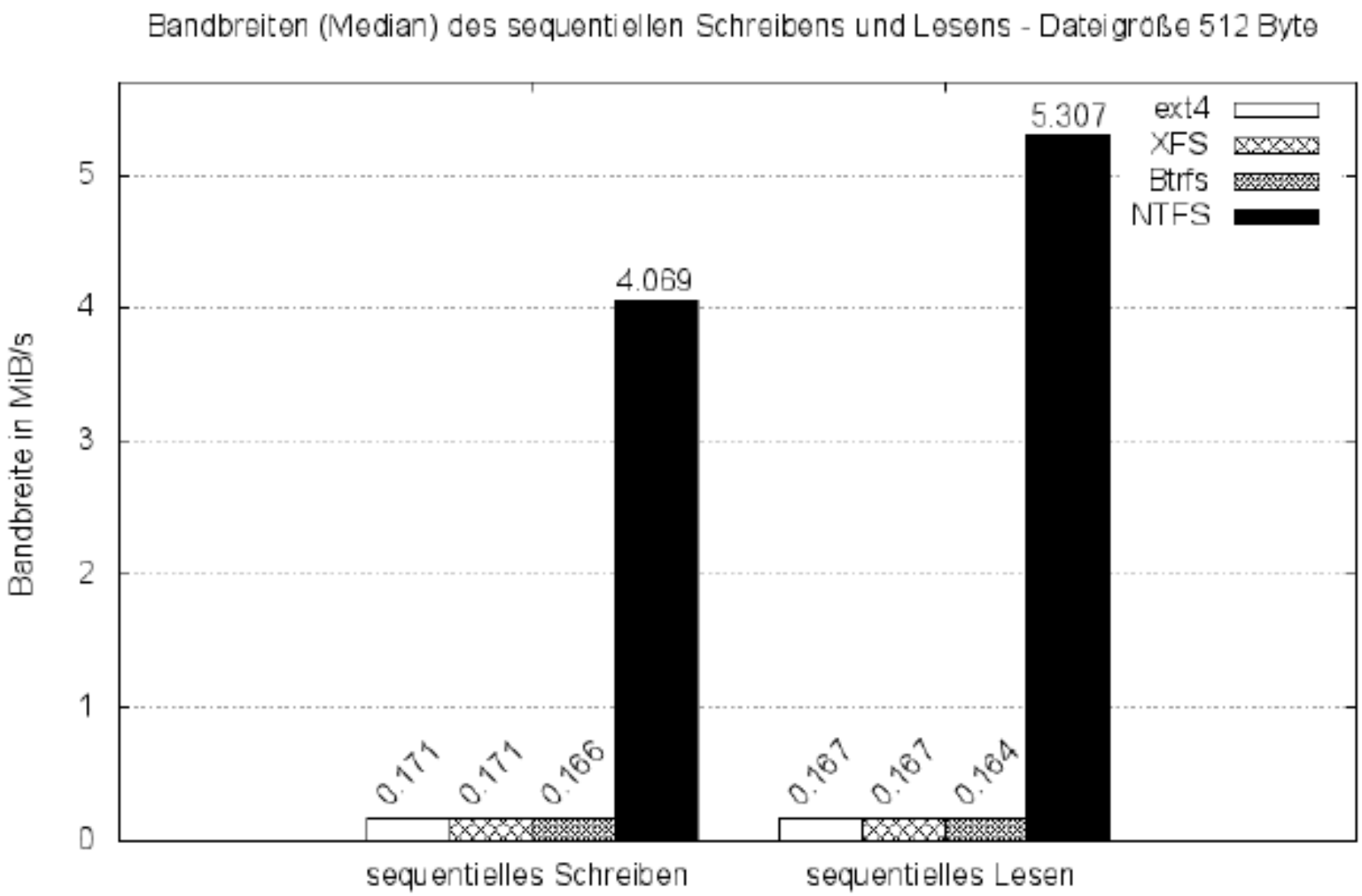
1 GiB File - sequentielles Schreiben und Lesen. - Higher is better.

Diagramm 16 (Dell):



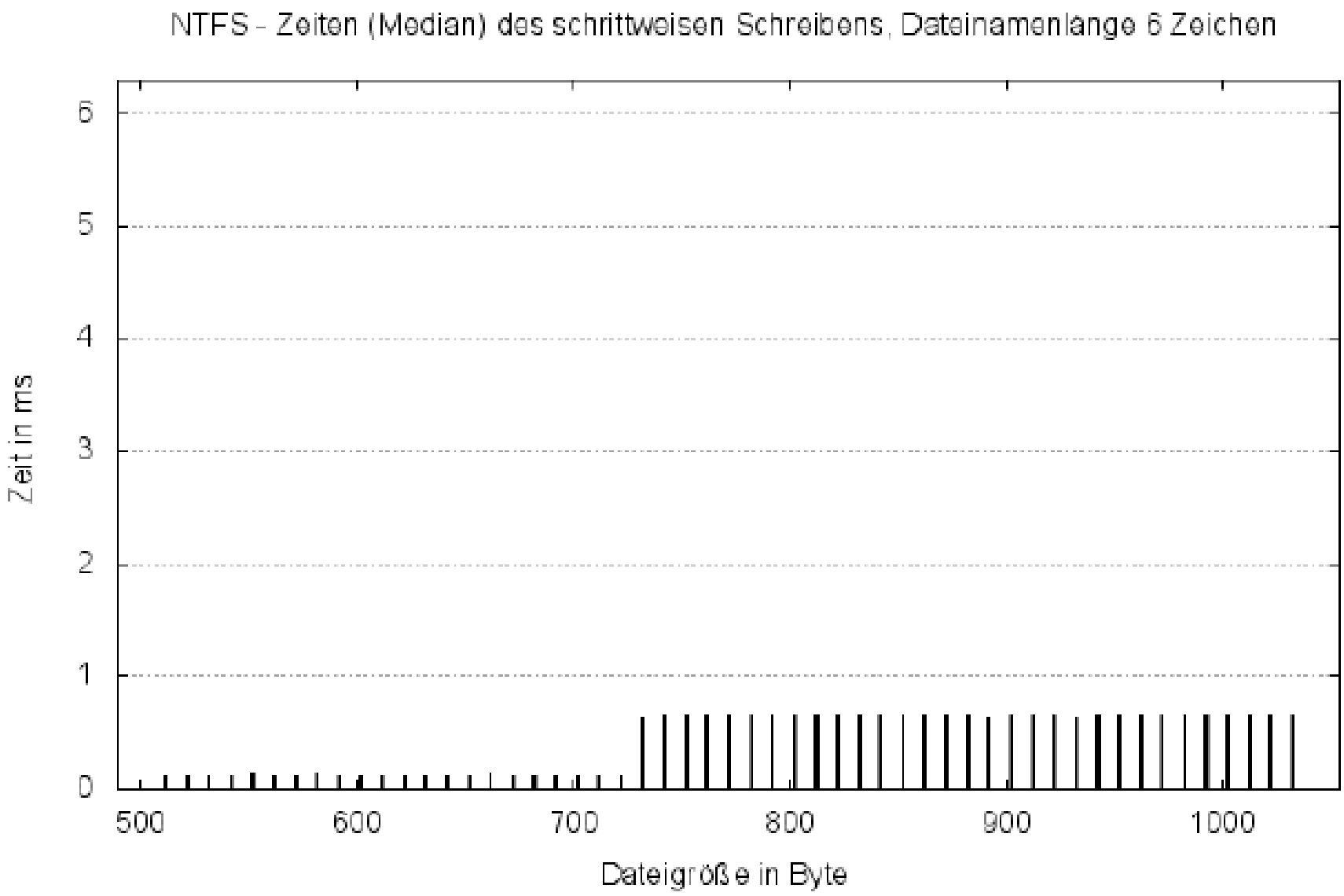
1 GiB File - zufälliges Schreiben und Lesen.

Diagramm 4 (Dell):



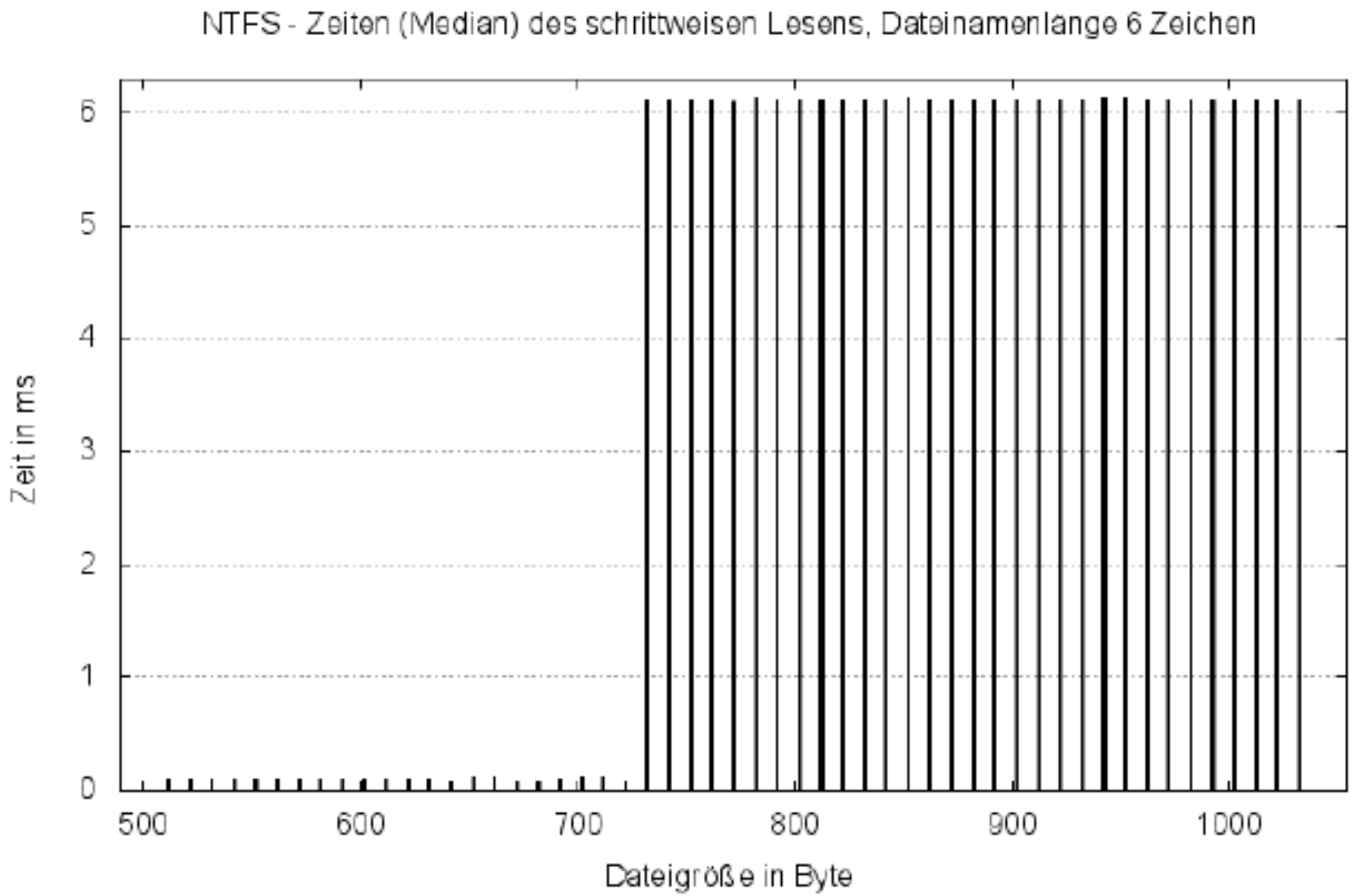
Small Files (512 Bytes)

Diagramm 21 (Dell):



NTFS - Schreiben kleiner Dateien (Lower is better.)

Diagramm 22 (Dell):



NTFS - Lesen kleiner Dateien (Lower is better.)

Ergebnisse:

- Ergebnisse für Dateigröße 1 GiB sind sehr ähnlich, BTRFS beim Lesen etwas langsamer (ca. 8% im Vergleich zum Sieger)
- sequentielles Lesen ist deutlich schneller als zufälliges Lesen,
- Das MFT-Konzept von NTFS ist für kleine Dateien signifikant besser als das inode-Konzept (über 2000 % besser auf dem Dellrechner).
- Zusätzlich wurden Messungen auf einem Notebook durchgeführt (andere Hard- und Software) mit abweichenden Ergebnissen: Auf dem Notebook war NTFS bei kleinen Dateien ebenfalls deutlich besser als die Linux-Dateisysteme, aber nicht so drastisch (ca. 200%).

(ext4 (Release 2016) hat ein optionales Feature *inline data*. Das Feature ermöglicht das Abspeichern von Dateien kleiner als 128 Bytes direkt im inode. Kann nicht auf einem schon existierenden Dateisystem aktiviert werden.)

Definition 31: Eine Protection Domain (Zugriffsschutzbereich) besteht aus einer Menge von Paaren (Objekt, Rechte), wobei Objekte die verschiedenen Dateien sind und Rechte die Zugriffsrechte auf das Objekt beschreiben (lesend, schreibend, ausführbar).

Protection Matrix:

Domain	File1	File2	File3	File4	File5	File6	Printer1
1	r	r, w					
2		r	r, w, x	r, w			w
3						r, w, x	w
...							

Beispiel: Jeder Nutzer könnte eine eigene Domain darstellen.

1. Ansatz: Die Protection Matrix wird **spaltenweise** gespeichert:

Domain	File1	File2	File3	File4	File5	File6	Printer1
1	r	r, w					
2		r	r, w, x	r, w			w
3						r, w, x	w
...							

- Zu jedem Objekt wird eine Liste, die sogenannte Access Control List, abgespeichert, die zu jeder Domain die Zugriffsrechte auf das Objekt enthält **(nur nichtleere Einträge der Protection Matrix)**.
- Access Control Lists wurden in MULTICS implementiert.

Beispiel: Unix benutzt ACLs mit drei Einträgen: Owner, Group, others

Domain	File1	File2	File3	File4	File5	File6	Printer1
Owner	r	r, w					
Group		r	r, w, x	r, w			w
others						r, w, x	w

2. Ansatz: Die Protection Matrix wird zeilenweise gespeichert.

Domain	File1	File2	File3	File4	File5	File6	Printer1
1	r	r, w					
2		r	r, w, x	r, w			w
3						r, w, x	w

Zu jeder Domain wird eine Liste von Objekten, die sogenannte Capability List, mit Zugriffsrechten gespeichert.

Problem: Datei wird gelöscht $\Longrightarrow$ Capabilities für die Datei können auf mehreren Listen auftauchen.

Beispiele:

- Multiprozessorsystem HYDRA (1974),
- Verteilte Betriebssystem Amoeba (80er/90er Jahre),
- der verteilte Authentifizierungs- und Authorisierungsdienst Kerberos (RFC 4120, 2005) (*Objekte* sind in diesem Beispiel verteilte Dienste (Fileserver, LDAP-Server, ...))

Beispiel UNIX

Darf ein Prozeß der Nutzerin `schnor` bzw. `pvogel` die folgende Datei öffnen?

```
> ls -al folien.tex
-rw-r--r-- 1 schnor users 10316 Dec 11 16:19 folien.tex
```

Unix Access Control Algorithm

1. Passt die **effektive** User-ID `euid` des Prozesses auf die Owner-ID?
   Falls ja, Owner-Rechte anwenden!
2. Passt die **effektive** Group-ID `egid` des Prozesses auf die Group-ID?
   Falls ja, Group-Rechte anwenden!
3. Ansonsten gelten die `others`-Rechte.

euid und egid werden vom Eltern-Prozeß geerbt.

Ausnahme:

```
bettina@petzi 25 > ll /usr/bin/passwd
-r-sr-xr-x  2 root  wheel  26564 Nov 20 13:02 /usr/bin/passwd
```

SUID-Bit (s-Bit): Führt ein Benutzer eine Programmdatei mit gesetztem Suid-Bit (set user id) aus, so wird die effektive `uid` (euid) des ausführenden Prozesses durch die `uid` desjenigen Benutzers ersetzt, der Eigentümer der Programmdatei ist.

Bsp: `/bin/passwd, sendmail`

Merke: In UNIX legen euid und egid die Protection Domain eines Prozesses fest.

Beispiel: Wie weit kommt man mit 9 Bits?

uid	gid
Jan	system
Jelle	student
Maike	student

1. Nur Jan kann die Datei `accounting.log` unter jeder gid lesen und schreiben:

   ```
   -rw- --- --- 1 jan system 10889 Jan 12 14:10 accounting.l
   ```

2. Jan will seine Datei `userinfo` für alle Studenten lesbar machen:

   ```
   -rw- r-- --- 1 jan student 10889 Jan 12 14:11 userinfo
   ```

3. Maikes Datei `surprise` dürfen alle Studenten außer Jelle lesen.

   ```
   -rw- r-- --- 1 maike  ??? 10889 Jan 12 14:12 surprise
   ```

   und eine neue Gruppe mit allen Studenten ohne jelle anlegen

Beispiel: Alle Mitglieder der Gruppe `developer` dürfen die Datei `kernel.c` lesen und schreiben, alle Mitglieder der Gruppe `test` nur lesen.

Problem: 2 Gruppen nötig!

Lösung: Mache alle *developer* auch zu Mitgliedern der Gruppe `test`, und erzeuge einen Stellvertreteraccount `linus`

```
> ls -al Linux
dr-x r-x --- 18 linus test      4096 Jan 12 18:09 .
...
-r-- rw- r-- 1 linus developer 10889 Jan 12 14:12 kernel.c
```

besser: Einsatz eines Versionsverwaltungssystems: cvs, svn, git, ...

Erweiterte Access Control Lists unter Linux

- XFS unterstützt ACL, bei ext2, ext3 und ReiserFS optional,
- Samba ist ACL-fähig
- Jedes Objekt besitzt wie bisher die Zugriffsrechte für owner, group und others (Minimal-ACL).
- Der Minimal-ACL lassen sich weitere Einträge (Access Control Entries (ACE)) für Named Users und Named Groups hinzufügen. Es handelt sich dann um eine sogenannte erweiterte ACL.

  Beispiel:

  ```
  setfacl -m g:students:rw foo
  ```

  gibt der Named Group `students` für die Datei `foo` Lese- und Schreibrechte.

- Eine erweiterte ACL benötigt eine Rechtemaske. Sie beschränkt die Zugriffsrechte für alle Gruppen (Owning Group und Named Groups) und Named Users, indem Maske und jeweilige Rechte bitweise mit der logischen Und-Operation verknüpft werden.

Beispiel:

```
setfacl -m m::r-x foo
```

entzieht allen Named Users und Named Groups das Schreibrecht.

Welcher ACE greift? - Matching ACE

1. Passt die effektive User-ID des Prozesses auf die Owner-ID der Datei?
   Falls ja, Owner-ACE anwenden!
2. Passt die effektive User-ID des Prozesses auf die User-Id eines User-ACE (Named User)?
   Falls ja, User-ACE anwenden!
3. Gibt es Group-ACEs, die auf den Benutzer/Prozeß passen?
   Falls ja, gibt es darunter mindestens einen Group-ACE, der die angeforderten Rechte erlaubt?
   - Falls ja: Zugriff ist erlaubt!
   - Falls nein: Zugriff verboten.
4. Passen weder Nutzer- noch Gruppeneinträge, gelten die `others`-Rechte.

siehe auch Stefan Kania - Rechte im Dateisystem

Angriff auch bei perfektem Zugriffsschutz: “Morsen” (Covert Channels, Lampson 1973)

Fazit - Dateisystem

- Systemaufrufe read/write versus `mmap`
- Bitmap: effiziente Datenstruktur, um freie Blöcke zu bestimmen
- UNIX merkt sich Blockadressen auf dem inode.
- NTFS merkt sich Serien von Blockadressen im MFT Eintrag.
- Ein Dateisystem hat eine Vielzahl kleiner Dateien zu verwalten. IFI-Server ANNO 2022: 56 % der Dateien sind kleiner gleich 4 KB groß.
- In UNIX legen die effektive uid und gid die Protection Domain eines Prozesses fest.