Dateisysteme

Hochleistungs-Ein-/Ausgabe

Michael Kuhn

Wissenschaftliches Rechnen
Fachbereich Informatik
Universität Hamburg

2015-04-20

E/A-Schichten

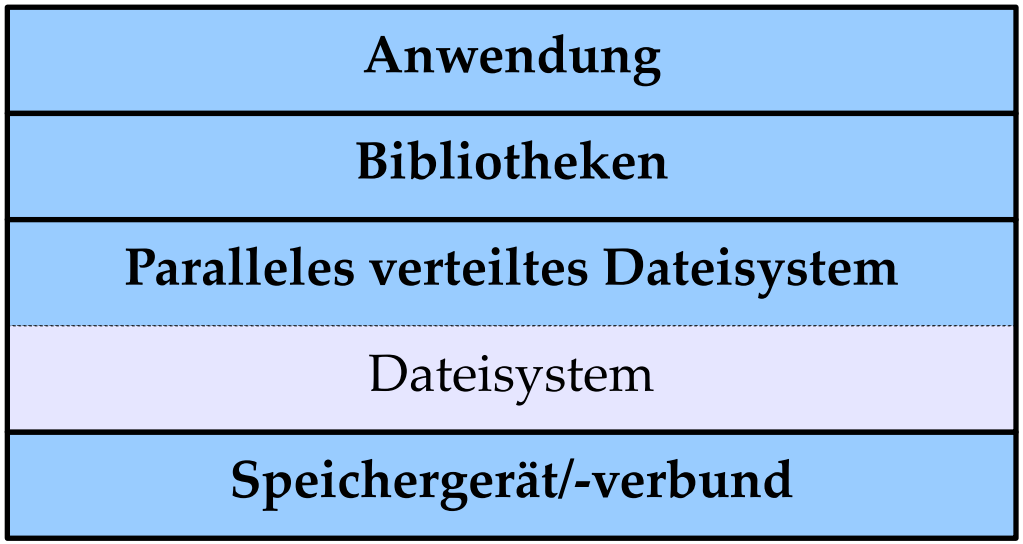


Abbildung: E/A-Schichten

Aufgabe

- Strukturierung
  - Üblicherweise Dateien und Verzeichnisse
  - Hierarchische Organisation
  - Andere Ansätze: Tagging
- Verwaltung von Daten und Metadaten
  - Blockallokation
  - Zugriffsrechte, Zeitstempel etc.
- Dateisysteme nutzen ein darunter liegendes Speichergerät
  - Oder einen Speicherverbund
  - Logical Volume Manager (LVM) und/oder mdadm

Beispiele

- Linux: ext4, XFS, ...
- Windows: FAT, exFAT, NTFS
- OS X: HFS+
- Universal: ISO9660, ...

Beispiele...

- Netzwerk: NFS, AFS, Samba
- Kryptographisch: EncFS, eCryptfs
- Parallel verteilt: GPFS, Lustre, ...
- Pseudo: procfs, ...
- Setzen häufig auf darunterliegenden Dateisystemen auf

E/A-Operationen

```
1 fd = open("/path/to/file", O_RDWR | O_CREAT |
      ↪ O_TRUNC, S_IRUSR | S_IWUSR);
2 nb = write(fd, data, sizeof(data));
3 rv = close(fd);
4 rv = unlink("/path/to/file");
```

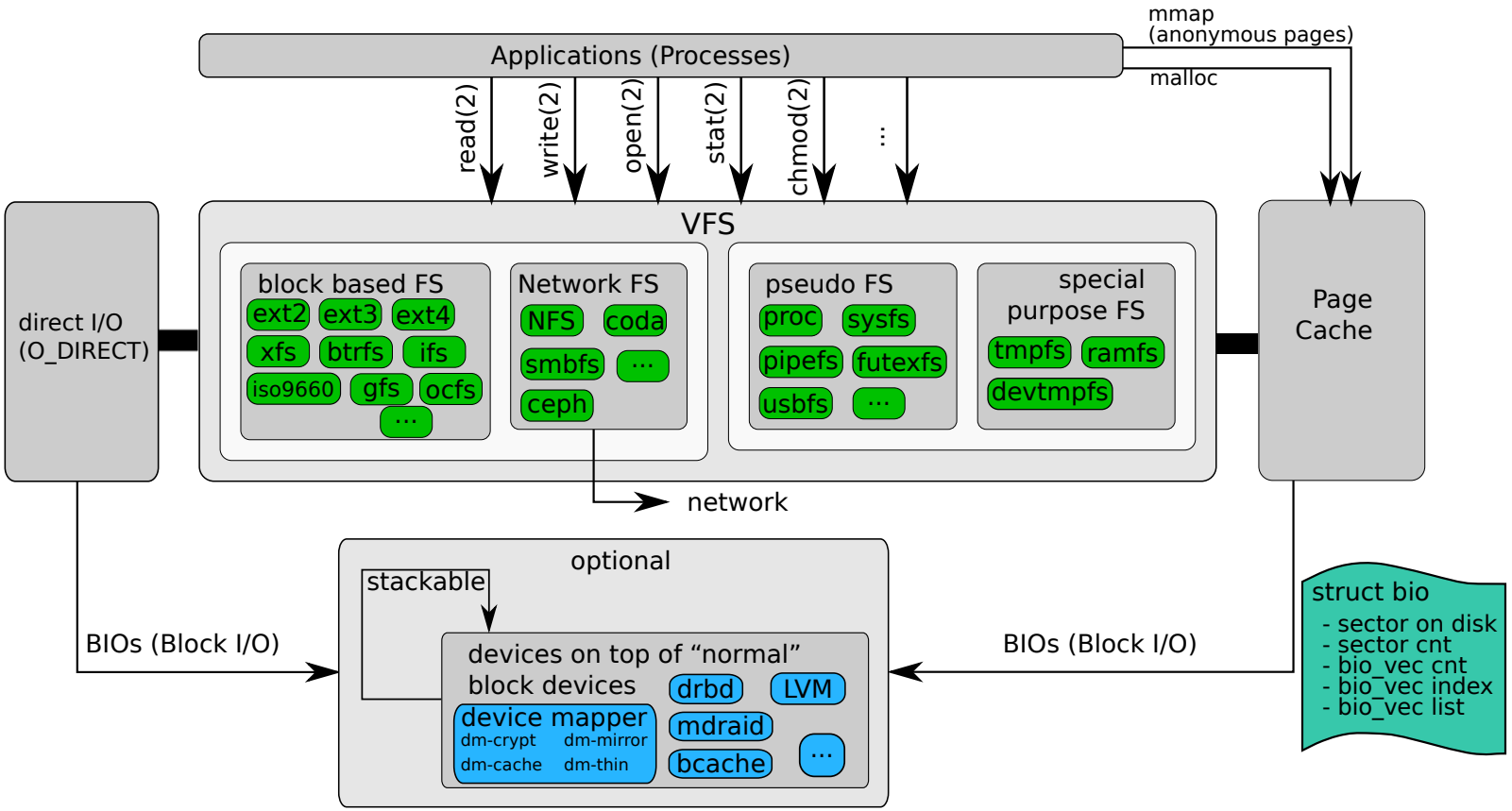
Listing 1: E/A über Low-Level-Funktionen

- Initialer Zugriff über Pfad
- Danach über File Descriptor (FD)

VFS

- Virtual File System (Switch)
- Zentrale Dateisystemkomponente im Kernel
  - Standardisiertes Interface für alle Dateisysteme (POSIX)
  - Gibt Dateisystemstruktur größtenteils vor
- Leitet Anfragen der Anwendungen weiter
  - Basierend auf dem Mountpoint

VFS…



The Linux Storage Stack Diagram
http://www.thomas-krenn.com/en/wiki/Linux_Storage_Stack_Diagram
Created by Werner Fischer and Georg Schönberger
License: CC-BY-SA 3.0, see http://creativecommons.org/licenses/by-sa/3.0/

Abbildung: Virtual File System [3]

VFS…

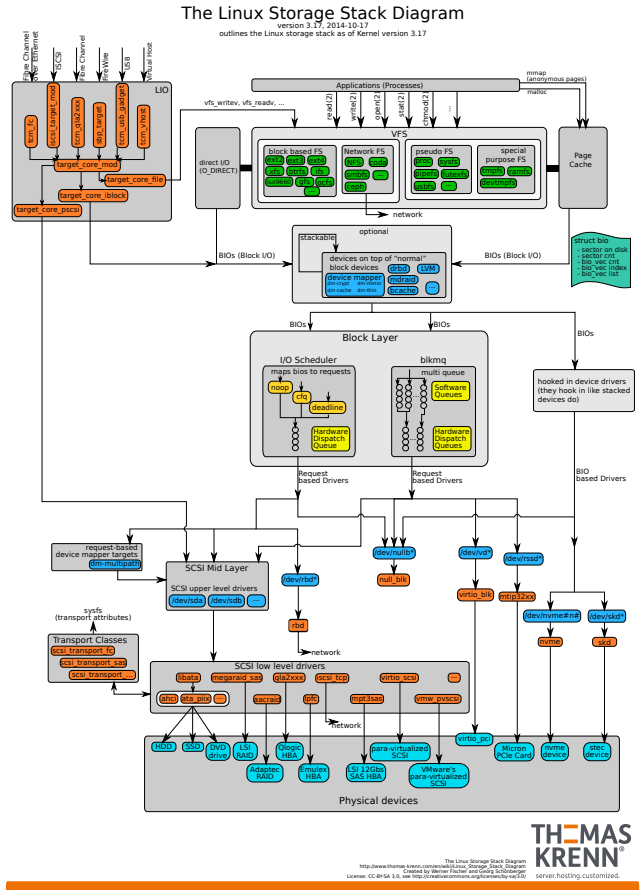


Abbildung: E/A-Stack in Linux 3.17 [3]

Dateisystemobjekte

- Dateien
  - Enthalten Daten in Form eines Byte-Arrays
  - Können gelesen/geschrieben werden (explizit)
  - Können in den Speicher gemappt werden (implizit)
- Verzeichnisse
  - Enthalten Dateien und Verzeichnisse
  - Zur Organisation des Namensraumes

Dateisystemobjekte...

- Inodes
  - Enhalten Metadaten
  - Eigentliche Basisobjekte des Dateisystems
  - Üblicherweise eindeutige IDs

Dateien

```
1 nb = pwrite(fd, data, sizeof(data), 42);
2 nb = pread(fd, data, sizeof(data), 42);
```

Listing 2: Expliziter Zugriff

- `pwrite` und `pread` verhalten sich wie `write` bzw. `read`
  - Explizite Angabe des Offsets
- Zugriff über File Descriptor

Dateien...

```
1 char* pt = mmap(NULL, FILE_SIZE, PROT_READ |
      ↪ PROT_WRITE, MAP_SHARED, fd, 0);
2 memcpy(pt + 42, data, sizeof(data));
3 memcpy(data, pt + 42, sizeof(data));
4 munmap(pt, sizeof(data));
```

Listing 3: Impliziter Zugriff

- `mmap` erlaubt es eine Datei in den Speicher einzublenden
  - Datei beginnt an Adresse `pt`
- Zugriff wie auf andere Speicherobjekte
  - Z.B. via `memcpy` oder direkte Zuweisung

Verzeichnisse

Inode	Größe	Namenslänge	Dateityp	Name
23	10	2	2	`.\0`
24	11	3	2	`..\0`
⋮	⋮	⋮	⋮	⋮
42	14	6	1	`hello\0`
43	14	6	2	`world\0`

Abbildung: ext4-Verzeichniseintrag [1]

- Traditionell lineares Array
  - Langsam, da über das komplette Array iteriert werden muss
- Heutzutage eher Baumstrukturen

Inodes

Feldgröße	Inhalt
2 Bytes	Berechtigungen
2 Bytes	Benutzer-ID
4 Bytes	Dateigröße
4 Bytes	Zugriffszeit
4 Bytes	Inode-Änderungszeit
4 Bytes	Datenänderungszeit
4 Bytes	Löschzeit
2 Bytes	Gruppen-ID
2 Bytes	Linkzahl
⋮	⋮
60 Bytes	Blockzeiger, Extent-Baum oder Inline-Daten
⋮	⋮
4 Bytes	Versionsnummer
100 Bytes	Freier Speicher

Abbildung: ext4-Inode (256 Bytes) [1]

Inodes…

- Kompliziert durch Rückwärtskompatibilität
- Viele Felder sind aufgeteilt
  - Zeitstempel: 4 Bytes für Sekunden seit 1970, 4 Bytes für Nanosekundenauflösung
  - Größe: Obere und untere 4 Bytes
- Felder mehrfach überladen
  - Blockzeiger, Extent-Baum oder Inline-Daten (falls Datei kleiner als 60 Bytes)
  - 100 Bytes am Inode-Ende für erweiterte Attribute

Inodes...

```
$ touch foo
$ ls -l foo
-rw-r--r--. 1 u g 0 19. Apr 18:48 foo
$ ln foo bar
$ ls -l foo bar
-rw-r--r--. 2 u g 0 19. Apr 18:48 bar
-rw-r--r--. 2 u g 0 19. Apr 18:48 foo
$ stat --format=%i foo bar
641174
641174
$ rm foo
$ ls -l bar
-rw-r--r--. 1 u g 0 19. Apr 18:48 bar
```

Listing 4: Inode vs. Datei

POSIX-Schnittstelle

- open, close, creat
- read, write, lseek
- chmod, chown, stat
- link, unlink
- ...

ext4

- Standard-Dateisystem in vielen Linux-Distributionen
  - Eingeführt 2006, stabil 2008
  - Vorgänger: ext, ext2, ext3
- Statische Festlegung bei Dateisystemerzeugung
  - Inode-Zahl
  - Blockgröße

ext

- Erstes Dateisystem speziell für Linux
  - Nutzte als erstes Dateisystem die VFS-Schicht
- Inspiriert vom Unix File System (UFS)
- Beseitigte Beschränkungen des MINIX-Dateisystems
  - Dateigrößen bis 2 GB
  - Dateinamen bis 255 Zeichen

ext2

- Separate Zeitstempel für Zugriff und Inode-/Datenänderung
- Datenstrukturen für zukünftige Erweiterungen ausgelegt
- Testumgebung für neue VFS-Funktionen
  - Access Control Lists (ACLs)
  - Erweiterte Attribute

ext3

- Journaling
- Dateisystemvergrößerung zur Laufzeit
- H-Baum für größere Verzeichnisse

ext4

- Größere Dateisysteme, Dateien und Verzeichnisse
- Extents
- Preallokation, verzögerte Allokation und verbesserte Multiblockallokation
- Journal-Prüfsummen
- Schnellere Dateisystemüberprüfung
- Nanosekunden-Zeitstempel
- Unterstützung für TRIM

ext4...

Inhalt	**Größe**
Padding (Blockgruppe 0)	1.024 Bytes
Superblock	1 Block
Gruppenbeschreibung	n Blöcke
Reservierte GDT-Blöcke	m Blöcke
Daten-Bitmap	1 Block
Inode-Bitmap	1 Block
Inode-Tabelle	k Blöcke
Daten-Blöcke	l Blöcke

Abbildung: ext4-Blockgruppe [1]

ext4...

Blockgröße	**1 KiB**	**2 KiB**	**4 KiB**	**64 KiB**
Blöcke	2^{64}	2^{64}	2^{64}	2^{64}
Inodes	2^{32}	2^{32}	2^{32}	2^{32}
Dateisystemgröße	16 ZiB	32 ZiB	64 ZiB	1 YiB
Dateigröße (Extents)	4 TiB	8 TiB	16 TiB	256 TiB
Dateigröße (Blöcke)	16 GiB	256 GiB	4 TiB	256 PiB

Abbildung: ext4-Limits im 64-Bit-Modus [1]

Allokation

- Blockbasiert
  - Viele Blöcke gleicher Größe (üblicherweise 4 KiB)
  - Zeiger auf Blöcke
    - Direkt, indirekt, doppelt indirekt, dreifach indirekt

Allokation…

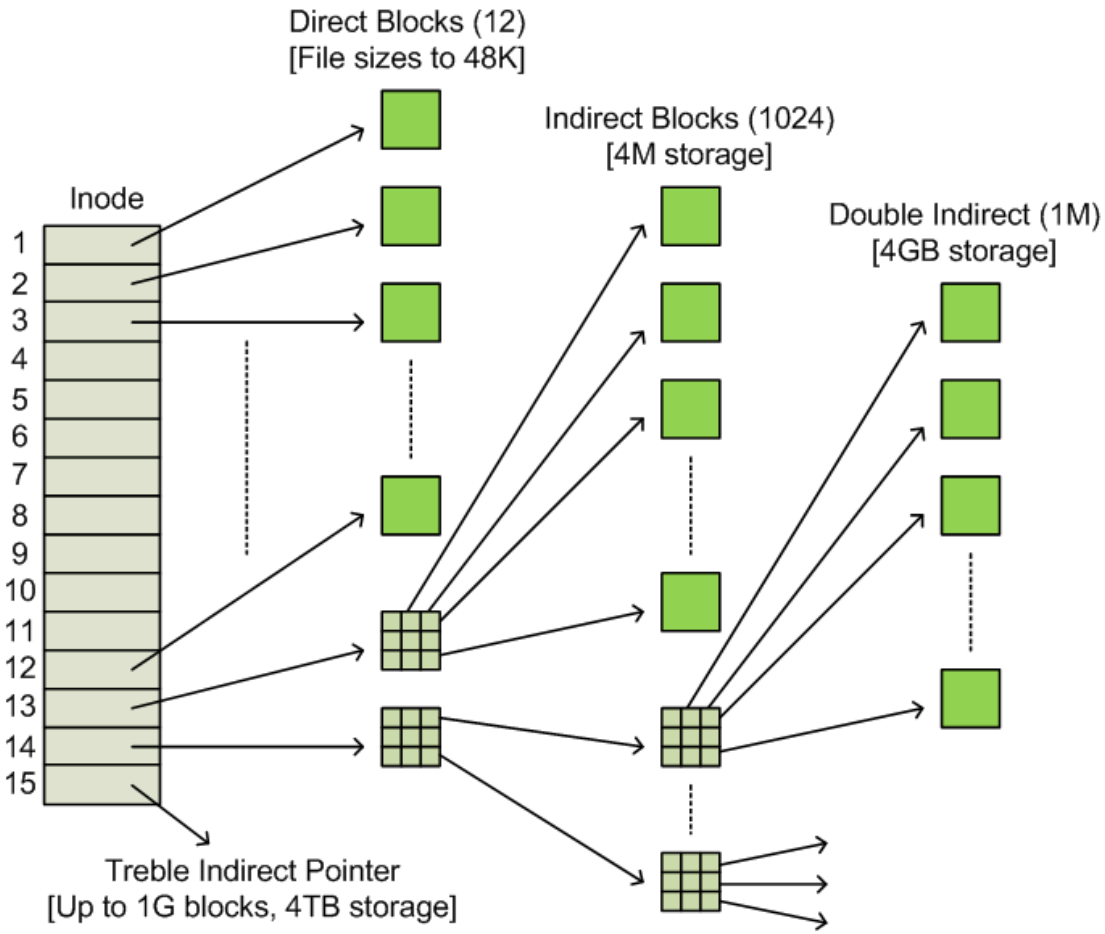


Abbildung: Block-Zeiger [2]

Allokation…

- Extentbasiert
  - Wenige möglichst große Extents
    - Vier Extents können im Inode gespeichert werden
    - Mehr in einer Baumstruktur und zusätzlichen Blöcken
  - Zeiger auf Startblock und Länge
    - Maximale Länge: 32.768 Blöcke
    - Entspricht 128 MiB bei einer Blockgröße von 4 KiB

Allokation…

- Blockallokation
  - Versuche zusammenhängende Blöcke zu allokieren
  - Versuche Blöcke in derselben Blockgruppe zu allokieren
- Multiblockallokation und verzögerte Allokation
  - Spekulativ 8 KiB bei Dateierzeugung allokieren
  - Allokation wird erst durchgeführt, wenn Blöcke auf das Speichergerät geschrieben werden müssen

Allokation...

- Dateien und Verzeichnisse
  - Blöcke möglichst in der Blockgruppe des Inodes allokieren
  - Dateien möglichst in der Blockgruppe des Verzeichnisses allokieren

Sparse-Dateien und Preallokation

- Sparse-Dateien: Dateien mit „Löchern“
  - Z.B. mit `lseek` oder `truncate`
  - Effiziente Speicherung von Dateien mit vielen 0-Bytes

```
1 $ truncate --size=1G dummy
2 $ ls -lh dummy
3 -rw-r--r--. 1 u g 1,0G 18. Apr 23:49 dummy
4 $ du -h dummy
5 0 dummy
```

Listing 5: Erzeugung einer Sparse-Datei

Sparse-Dateien und Preallokation...

- Preallokation: Speicher vorallokieren
  - Mit `fallocate` bzw. `posix_fallocate`
  - Verhindert Fragmentierung bei vielen Dateivergrößerungen

```
1 $ fallocate --length $((1024 * 1024 * 1024)) dummy
2 $ ls -lh dummy
3 -rw-r--r--. 1 u g 1,0G 19. Apr 19:14 dummy
4 $ du -h dummy
5 1,1G dummy
```

Listing 6: Preallokation einer Datei

Journaling

- Journaling zur Sicherung der Konsistenz des Dateisystems
- Dateisystemoperationen benötigen mehrere Schritte
- Z.B. das Löschen einer Datei
  1. Entfernen des Verzeichniseintrags
  2. Freigeben des Inodes
  3. Freigeben der Datenblöcke
- Problematisch im Fall eines Absturzes

Journaling…

- Geplante Änderungen werden ins Journal eingetragen
  - Entfernen wenn Operation vollständig durchgeführt
- Bei der anschließenden Dateisystemüberprüfung
  - Änderungen wiederholen oder
  - Änderungen verwerfen
- Unterschiedliche Modi
  - Metadaten-Journaling und volles Journaling

Journaling…

- Journal: Alle Änderungen werden ins Journal geschrieben
- Ordered: Metadaten werden ins Journal geschrieben
  - Daten vor Metadaten
  - Problematisch mit verzögerter Allokation
- Writeback: Metadaten werden ins Journal geschrieben
  - Reihenfolge beliebig

Funktionen

- „Dateisystem light“
  - Dünne Abstraktionsschicht über Speichergeräten
  - Objektbasierter Zugriff auf Daten
- Nur Grundoperationen
  - Erstellen, Öffnen, Schließen, Lesen, Schreiben
- Manchmal Object Sets
  - Können benutzt werden um verwandte Objekte zu gruppieren

Funktionen…

- Üblicherweise keine Pfade
  - Zugriff über eindeutige IDs
  - Kein Overhead durch Pfadauflösung
- Block-/Extent-Allokation
- Auf unterschiedlichen Abstraktionsebenen verfügbar
  - Cloudspeicher, Festplatte

Schichtung

- Object Stores können als Unterbau für Dateisysteme genutzt werden
  - Erlaubt Konzentration auf Dateisystemfunktionalität
  - Speicherverwaltung durch separate Schicht
- Bei lokalen Dateisystemen nicht sinnvoll
  - Funktionalität größtenteils durch POSIX vorgegeben
  - Hauptunterschied ist Blockallokation
- Sehr sinnvoll für parallele verteilte Dateisysteme
  - Kein redundanter Dateisystem-Overhead

B-Baum vs. B+-Baum

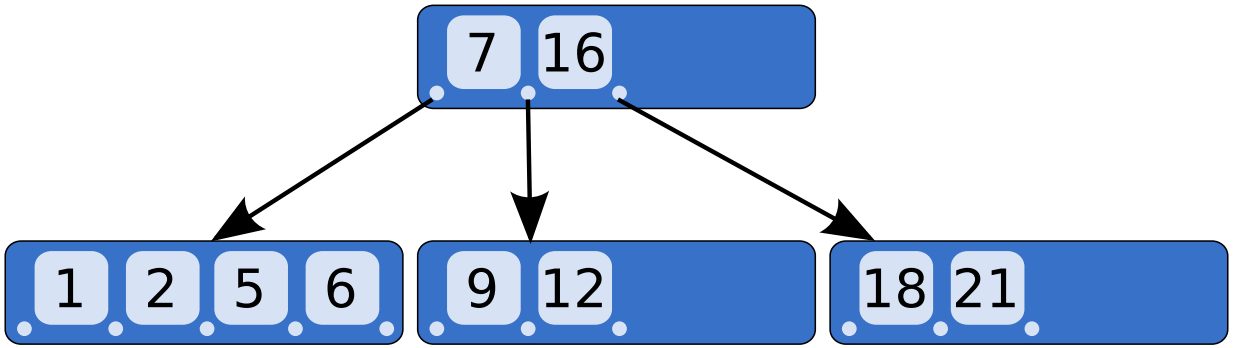


Abbildung: B-Baum [4]

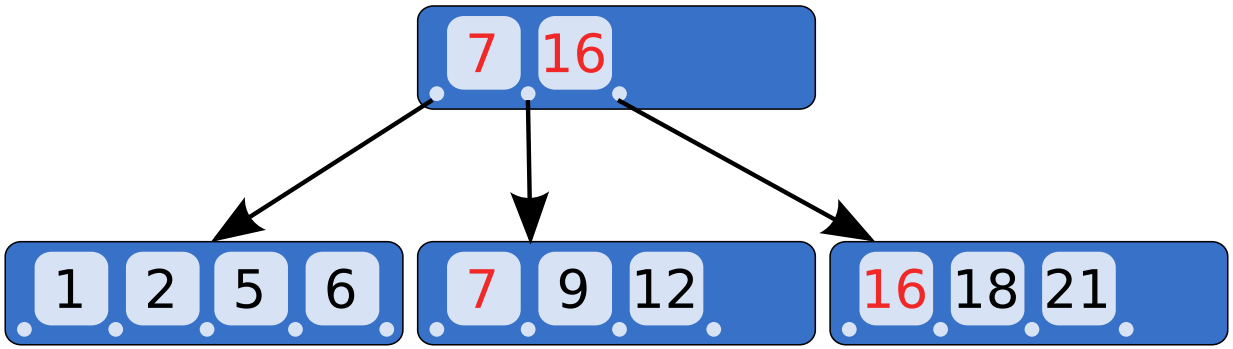


Abbildung: B+-Baum [4]

B-Baum vs. B+-Baum…

- B-Baum
  - Verallgemeinerter Binärbaum
  - Optimiert für Systeme, die große Blöcke lesen/schreiben
  - Zeiger und Daten gemischt
- B+-Baum
  - Daten nur in Blättern
  - Vorteilhaft für Caching, da einfacher alle Knoten zu cachen
  - Benutzt in NTFS, XFS, …

H-Baum

- Basiert auf B-Baum
- Andere Behandlung von Hash-Kollisionen
- Benutzt in ext3 und ext4

Leistungsbewertung

- Dateisystemleistung ist schwierig zu bewerten
  - Viele unterschiedliche Faktoren
  - Daten- vs. Metadatenleistung
  - Leistung unterschiedlicher Funktionen
  - Leistung für spezifische Anforderungen messen
- Datensicherheit kostet üblicherweise Leistung
  - Volles Journaling, Prüfsummen etc.

Kernel- vs. Userspace

- Dateisysteme üblicherweise direkt im Kernel implementiert
  - Hoher Wartungsaufwand
- Alternative: Filesystem in Userspace (FUSE)
  - Besteht aus Kernelmodul und Bibliothek
  - Entwicklung von Dateisystemen als normale Prozesse

Kernel- vs. Userspace…

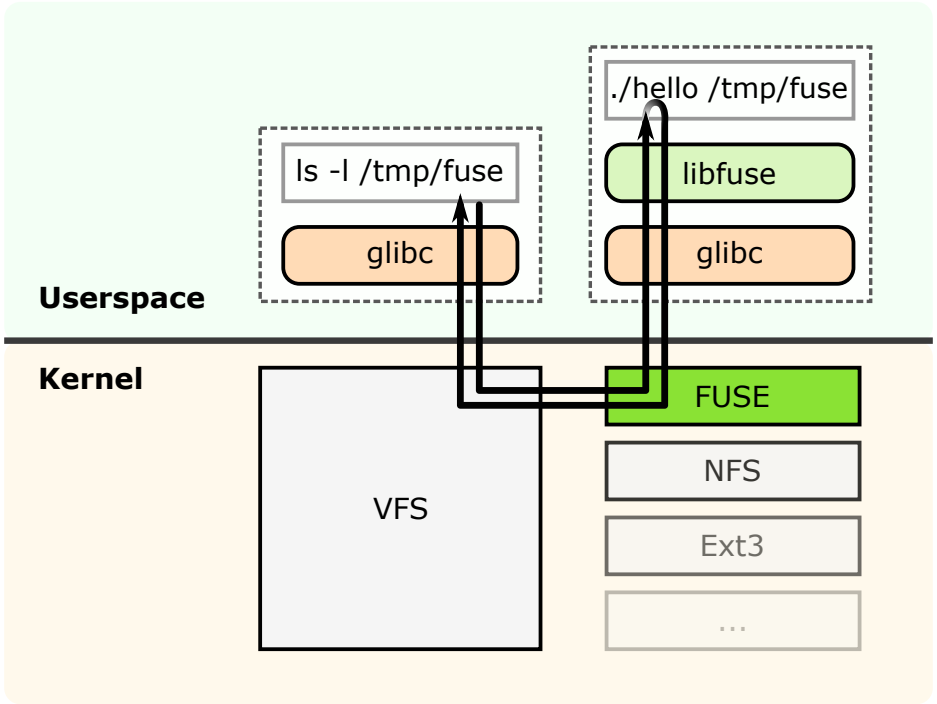


Abbildung: FUSE [5]

- Umleitung in Userspace durch VFS und FUSE-Modul
  - Geringere Leistung durch Kontextwechsel

Ausblick

- Moderne Dateisysteme integrieren zusätzliche Funktionen
  - Volumenverwaltung, Prüfsummen, Schnappschüsse, ...
- Basis für parallele verteilte Dateisysteme
  - Object Stores besser geeignet

Zusammenfassung

- Dateisysteme organisieren Daten und Metadaten
  - Üblicherweise standardisierte Schnittstelle
- Hauptobjekte sind Dateien und Verzeichnisse
  - Inodes speichern Metadaten
- Neue Techniken zur Effizienzsteigerung
  - Journaling um Konsistenz sicherzustellen
  - Speicherallokation mit Hilfe von Extents
  - Baumstrukturen für skalierbaren Zugriff

Quellen I

[1] djwong. Ext4 Disk Layout. https://ext4.wiki.kernel.org/index.php/Ext4_Disk_Layout.

[2] Hal Pomeranz. Understanding Indirect Blocks in Unix File Systems. http://digital-forensics.sans.org/blog/2008/12/24/understanding-indirect-blocks-in-unix-file-systems.

[3] Werner Fischer and Georg Schönberger. Linux Storage Stack Diagramm. https://www.thomas-krenn.com/de/wiki/Linux_Storage_Stack_Diagramm.

[4] Wikipedia. B-tree. http://en.wikipedia.org/wiki/B-tree.

Quellen II

[5] Wikipedia. Filesystem in Userspace. http://en.wikipedia.org/wiki/Filesystem_in_Userspace.