File Systems

Parallel Storage Systems

2024-04-22

Prof. Dr. Michael Kuhn

michael.kuhn@ovgu.de

Parallel Computing and I/O
Institute for Intelligent Cooperating Systems
Faculty of Computer Science
Otto von Guericke University Magdeburg
https://parcio.ovgu.de

Outline

- Which hard-disk drive parameter is increasing at the slowest rate?
  1. Capacity
  2. Throughput
  3. Latency
  4. Density

- Which RAID level does not provide redundancy?
  1. RAID 0
  2. RAID 1
  3. RAID 5
  4. RAID 6

- Which problem is called write hole?
  1. Inconsistency due to non-atomic data/parity update
  2. Incorrect parity calculation
  3. Storage device failure during reconstruction
  4. Partial stripe update

Outline

Motivation

1. File systems provide structure
   - File systems typically use a hierarchical organization
   - Hierarchy is built from files and directories
   - Access is handled via file and directory names
   - Other approaches: Tagging, queries etc.
2. File systems manage data and metadata
   - They are responsible for block allocation and management
   - Metadata includes access permissions, time stamps etc.
   - File systems use underlying storage devices
   - Devices can also be provided by storage arrays such as RAID

Examples

- Linux: tmpfs, ext4, XFS, btrfs, ZFS
  - File systems (more or less) conform to POSIX
- Windows: FAT, exFAT, NTFS
- OS X: HFS+, APFS
- Universal: ISO9660, UDF
  - Can be used on arbitrary media, mostly used on optical ones
- Pseudo: sysfs, proc
  - Allow changing system settings etc.

Examples...

- Network: NFS, AFS, Samba
  - Usually provide access to an underlying file system via the network
- Cryptographic: EncFS, eCryptfs
  - Typically make use of an underlying file system
- Parallel distributed: Spectrum Scale, Lustre, OrangeFS, CephFS, GlusterFS
  - Distribute data across multiple servers

I/O Interfaces

- I/O operations are realized using I/O interfaces
  - Interfaces are available for different abstraction levels
  - Interfaces forward operations to the actual file system
- Low-level interfaces provide basic functionality
  - POSIX, MPI-IO
- High-level interfaces provide more convenience
  - HDF, NetCDF, ADIOS

I/O Operations

- open can be used to open and create files
  - Features many different flags and modes
  - O_RDWR: Open for reading and writing
  - O_CREAT: Create file if necessary
  - O_TRUNC: Truncate if is exists already
- Initial access happens via a path
  - Afterwards, file descriptors can be used (with a few exceptions)
- All functions provide a return value
  - errno should be checked in case of errors

```c
fd = open("/path/to/file",
          O_RDWR | O_CREAT |
          O_TRUNC,
          S_IRUSR | S_IWUSR);

rv = close(fd);
rv = unlink("/path/to/file");

if (rv != 0) {
    ...
}
```

I/O Operations...

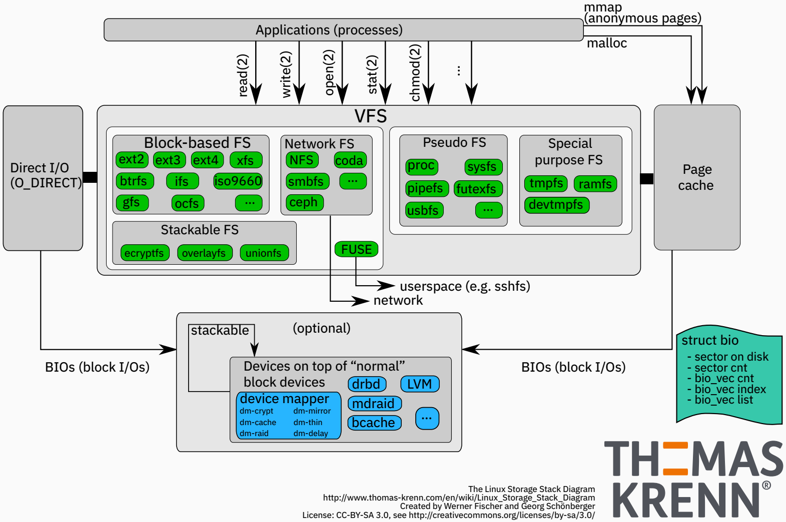
```c
nb = write(fd, data, sizeof(data));
```

- `write` returns the number of written bytes
  - Does not necessarily correspond to the given size (error handling!)
  - `write` updates the file pointer internally
  - `pwrite` is a thread-safe alternative to `write`
- Functions are provided by `libc`
  - Interaction with the file system happens in the kernel
  - System calls can be used to pass requests to the kernel
  - `libc` performs system calls transparently

Virtual File System (Switch)

- VFS is a central file system component in the kernel
  - Provides a standardized interface for all file systems (POSIX)
  - Defines file system structure and interface for the most part
- Forwards operations performed by applications to the corresponding file system
  - File system is selected based on the mount point
- Enables supporting a wide range of different file systems
  - Applications are still portable due to POSIX

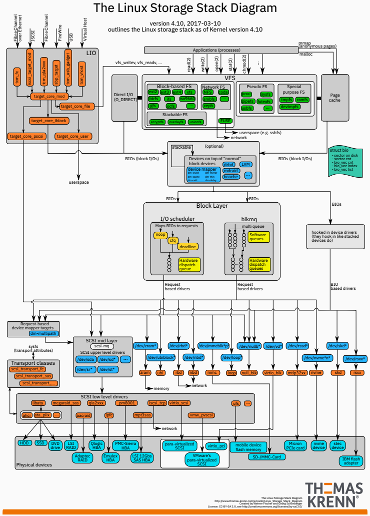
Virtual File System (Switch)…



[Fischer and Schönberger, 2017]

Virtual File System (Switch)...

- Applications call functions in `libc`
- `libc` performs system calls
- System calls are handled by VFS
- VFS determines correct file system instance
- Data is read/written via page cache or directly
- Block layer handles communication with devices



[Fischer and Schönberger, 2017]

Outline

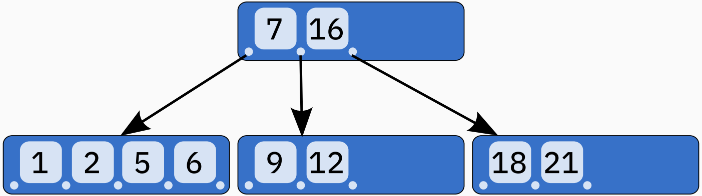
File System Objects

- Differences from user and system point of view
  - Users deal with files and directories that contain data and metadata
    - Files consist of bytes, directories contain files and further directories
  - The system manages all internals
    - Combines individual blocks into files etc.
- Inodes
  - The most basic data structure in POSIX file systems
    - Each file and directory is represented by an inode (see `stat`)
  - Inodes contain mostly metadata
    - Some of the metadata is visible for users, some is internal
  - Inodes are typically referenced by ID and have a fixed size

File System Objects…

- Files
  - Files contain data in the form of a byte array
    - POSIX specifies that data is a byte stream
  - Data can be read/written using explicit functions
  - Data can also be mapped into memory for implicit access
- Directories
  - Directories organize the file system's namespace
    - They can contain files and further directories
    - Directories within directories lead to a hierarchical namespace
  - From a user's point of view, directories are a list of entries
    - Internally, file systems often use tree structures

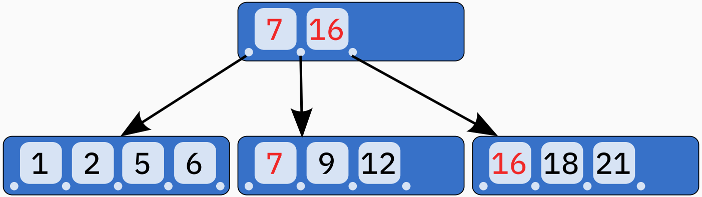
B-Tree vs. B+-Tree



[CyHawk, 2010]

- B-trees are generalized binary trees
- It is optimized for systems that read/write large blocks
    - Pointers and data are mixed in the tree

B-Tree vs. B+-Tree...



[CyHawk, 2010]

- B+-trees are a modification of B-trees
- Data is only stored in leaf nodes
  - Advantageous for caching since internal nodes are easier to cache
- Used in NTFS, XFS etc.

Alternative Trees

- H-trees
  - Based on B-trees
  - Has different handling of hash collisions
  - Used in ext3 and ext4
- B^{ε}-trees
  - Optimized for write operations
  - Operations are buffered in nodes
  - Improved performance for insert, range query and update operations

- pwrite and pread behave like write and read
  - They allow specifying the offset and do not modify the file pointer
  - File pointer is shared per file descriptor
  - Both functions are therefore thread-safe
- Access is done via an open file descriptor
  - Can be used in parallel by multiple threads

```
nb = write(fd, data,
           sizeof(data));
nb = read(fd, data,
          sizeof(data));

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

- mmap allows mapping a file into memory
  - The file will be mapped at address pt
  - There are several visibility settings (shared vs. private)
  - File can be larger than main memory
- Mapped files can be accessed like other objects in memory
  - Can be used in memcpy or assignments
  - Operating system takes care of reading and writing

```
char* pt;
pt = mmap(NULL, file_size,
          PROT_READ | PROT_WRITE,
          MAP_SHARED, fd, offset);
memcpy(pt + 42, data,
       sizeof(data));
memcpy(data, pt + 42,
       sizeof(data));
munmap(pt, FILE_SIZE);
```

- Both access models have advantages and disadvantages
    - Both modes benefit from the operating system's cache and optimizations
- Explicit access
    - Advantages: high level of control, can be used for direct I/O
    - Disadvantages: separate buffers are necessary, copies between kernel and user space
- Implicit access
    - Advantages: no separate buffers are necessary, efficient handling by the operating system, no copies necessary, large files can be mapped completely
    - Disadvantages: less control, complicated error handling via signals

Quiz

- What do you expect pread to return?
    1. 0
    2. 23
    3. 42
    4. 4,096

```c
int fd;

fd = open("newfile",
    O_RDWR | O_CREAT | O_TRUNC,
    0666);

pwrite(fd, data, 23, 0);
pread(fd, data, 42, 0);

close(fd);
```

Directories

- Traditionally managed as an array
  - Provides low performance since whole array has to be scanned
- Nowadays, tree structures are used
  - More complex but faster
- Name is not stored in inode
  - Multiple names can reference the same inode

Inode	Size	Length	Type	Name
23	10	2	2	.
24	11	3	2	..
⋮	⋮	⋮	⋮	⋮
42	14	6	1	hello
42	14	6	1	world

[djwong, 2018]

Inodes

- Inode structure can become complex due to backwards compatibility
  - Hard to change the on-disk format
- In ext4, many fields are split up due to backwards compatibility reasons
  - Time stamps: 4 bytes for seconds since 1970, 4 bytes for nanoseconds
  - Size: Upper and lower 4 bytes
- Fields are overloaded
  - Block pointers, extent tree or inline data (if file is smaller than 60 bytes)
  - 100 bytes for extended attributes

Field Size	Content
2 Bytes	Permissions
2 Bytes	User ID
4 Bytes	File Size
4 Bytes	Access Time
4 Bytes	Change Time (Inode)
4 Bytes	Modification Time (Data)
4 Bytes	Delete Time
2 Bytes	Group ID
2 Bytes	Link Count
⋮	⋮
60 Bytes	Block Pointers, Extent Tree or Inline Data
⋮	⋮
4 Bytes	Version Number
100 Bytes	Free Space

[djwong, 2018]

Inodes...

- Inodes are reference counted
  1. Inode is created for `foo`
  2. Reference is added for `bar`
- `ls` shows link count
  - Number of links to same inode
- `stat` shows internals
  - Including the inode ID
- `rm` removes a reference
  - Inode is freed if there are no references left

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

- Syntax describes available operations and their parameters
  - open, close, creat
  - read, write, lseek
  - chmod, chown, stat
  - link, unlink
  - (f)truncate, fallocate
- Semantics specifies how I/O operations should behave
  - `write`: *"POSIX requires that a read(2) which can be proved to occur after a write() has returned returns the new data. Note that not all filesystems are POSIX conforming."*

Sparse Files and Preallocation

- Sparse files are files with holes
  - Can be created using `lseek` or `truncate`
  - Allows efficiently storing files with many `0` bytes
- Files have correct logical size
  - Size is stored in the inode
- No space is actually allocated
  - du shows allocated size

```
$ truncate --size=1G dummy

$ ls -lh dummy
-rw-r--r--. 1 usr grp 1.0G Apr 18 23:49 dummy

$ du -h dummy
0    dummy
```

- Preallocation makes sure blocks are allocated
  - Can be done using `fallocate` or `posix_fallocate`
- Can prevent fragmentation
  - Repeatedly appending data can fragment file

```
$ fallocate --length 1G dummy

$ ls -lh dummy
-rw-r--r--. 1 usr grp 1.0G Apr 19 19:14 dummy

$ du -h dummy
1,0G    dummy
```

Outline

ext4

- ext4 is the default file system in many Linux distributions
  - It has been introduced in 2006 and marked stable in 2008
  - Predecessors: ext, ext2, ext3
- Many parameters have to be defined statically when creating the file system
  - Block size, file system size, inode count etc.
  - Some of them can be tuned afterwards
- ext4 is a traditional file system
  - Data is changed in-place (that is, no copy-on-write)
  - It does not support snapshots or checksums for data
  - It does not provide any other convenience features

ext

- ext was the first file system specifically designed for Linux
  - First file system to use the VFS layer
- Inspired by the Unix File System (UFS)
- Got rid of limitations within the MINIX file system
  - File sizes up to 2 GB
  - File names up to 255 characters

ext2

- ext2 introduced several new features and enhancements
    - Separate time stamps for access, change and modification
    - Data structures were set up for future extensions
- Test environment for new VFS functions
    - Access Control Lists (ACLs)
    - Extended Attributes

ext3

- ext3 introduced journaling to the file system
  - Will be explained later
- The file system can be resized at runtime
  - Useful for LVM environments
- Large directories can use H-trees
  - Reduces lookup times

ext4

- ext4 further improved the file system
  - Larger file systems, files and directories
  - Extents
  - Preallocation, delayed allocation and improved multi-block allocation
  - Journal checksums
  - Faster file system checks
  - Nanosecond time stamps
  - Support for TRIM (SSDs)

- The storage device is separated into multiple block groups for management reasons
  - Flexible block groups merge multiple groups
- Block size determines the number of inodes and data blocks per block group

Content	**Size**
Padding (Block Group 0)	1,024 Bytes
Superblock	1 Block
Group Descriptions	m Blocks
Reserved GDT Blocks	n Blocks
Data Bitmap	1 Block
Inode Bitmap	1 Block
Inode Table	k Blocks
Data Blocks	l Blocks

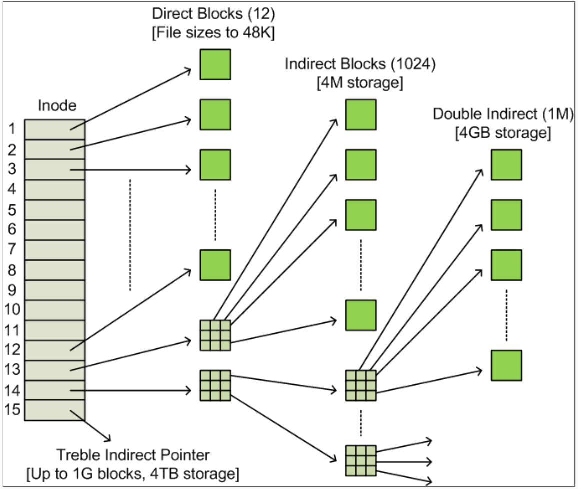
[djwong, 2018]

Block Size	1 KiB	2 KiB	4 KiB	64 KiB
Blocks	2^{64}	2^{64}	2^{64}	2^{64}
Inodes	2^{32}	2^{32}	2^{32}	2^{32}
File System Size	16 ZiB	32 ZiB	64 ZiB	1 YiB
File Size (Extents)	4 TiB	8 TiB	16 TiB	256 TiB
File Size (Blocks)	16 GiB	256 GiB	4 TiB	256 PiB

[djwong, 2018]

- Default block size is typically 4 KiB
  - Block size should not be larger than the system's page size
- There are different maximum file sizes when using extents and blocks

1. Block-based
    - Files are a collection of many same-sized blocks (typically 4 KiB)
    - The inode contains pointers to all blocks of a file
        - Direct, indirect, double indirect and triple indirect
    - Significant overhead for large files due to amount of pointers
        - Example: 1 TiB large size requires 268,435,456 pointers
    - The pointer structure also limits the maximum file size



[Pomeranz, 2008]

2. Extent-based
    - The goal is to have as few extents that are as large as possible
        - The addresses of four extents can be stored in the inode
        - Additional extents are stored in a tree structure using blocks
    - An extent is a pointer to a start block and length
        - Maximum length: 32,768 blocks
        - Results in a maximum extent size of 128 MiB when using 4 KiB blocks
    - Extents allow larger files when using common block sizes

Allocation...

- Block allocation
  - Try to allocate contiguous blocks for faster access
  - Try to allocate blocks within the same block group
- Multi-block allocation
  - Speculatively allocate 8 KiB when creating a file
- Delayed allocation
  - Blocks are only allocated when they have to be written to the storage device

Allocation…

- Files and directories
  - Blocks are allocated in the inode's block group if possible
  - Files' blocks are allocated in the directory's block group if possible
- Goals of allocation strategies
  - Try to allow large accesses
    - HDDs can only deliver low IOPS values due to high seek times
  - Accesses should be close to each other
    - Reduces head movements when using HDDs
    - The block group's metadata might already be cached
- These optimizations are less relevant for SSDs

Journaling

- Problem: File system operations typically require multiple steps
  - Example: Deleting a file
    1. Removing the directory entry
    2. Freeing the data blocks
    3. Freeing the inode
  - This is problematic in case of a crash
- Journaling can be used to ensure the file system's consistency

Journaling…

- Planned changes are first written to the journal
  - They are removed again when an operation is successful
- In case of a crash, the journal is checked for outstanding operations
  - Changes are repeated or discarded
- There are different modes with different performance characteristics
  - Metadata journaling or full journaling

Journaling…

- Journal: All changes are written to the journal
  - Deactivates delayed allocation and `O_DIRECT`
- Ordered: Metadata is written to the journal
  - Corresponding data is written before the metadata
  - Might be problematic with delayed allocation
  - This is the default journaling mode
- Writeback: Metadata is written to the journal
  - Allows data to be written after metadata has been committed
  - Can result in old data appearing after a recovery
  - Offers the highest performance but the lowest safety

Performance Considerations

- File system performance is often hard to assess
  - There are many factors and many involved components
  - Depending on the use case, data or metadata performance might be more important
  - The used functions and access patterns heavily influence achievable performance
  - It is important to always measure for concrete workloads
- Data safety typically decreases performance
  - Full journaling requires data copies, checksums require computing power etc.

Outline

Object Stores

- Object stores can be seen as lightweight file systems
  - They provide a thin abstraction layer above storage devices
  - Data is accessed using an object-based interface
- Object stores only provide some basic functions
  - Create, open, close, read, write of objects
  - Sometimes it is only possible to read or write complete objects
- Some object stores support so-called object sets
  - Can be used to group related objects

Object Stores...

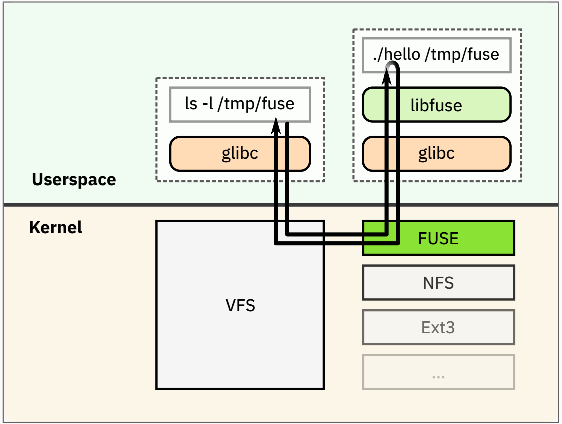
- Object stores typically do not use paths
  - Access is handled via unique IDs
  - There is no overhead caused by path traversal and resolution
  - The resulting namespace is very flat
- Block/extent allocation is performed by the object store
  - Block/extent management is one of the most complex aspects
- Object store concepts are available on different layers of abstraction
  - HDD, file system, cloud storage etc.

Object Stores…

- Object stores can be used as an underlying technology for file systems
  - Allows concentrating on file system functionality
  - Storage management is then handled by a separate layer
- Separation is often not useful for local file systems
  - Functionality and structure mostly determined by POSIX
  - One main difference of file systems is block allocation
- Separation can make sense for parallel distributed file systems
  - Eliminates redundancy caused by underlying local file systems

Kernel vs. User Space

- File systems are typically implemented within the kernel
  - High maintenance cost
  - Implementation is also more complex and error-prone
- Filesystem in Userspace (FUSE)
  - Kernel module and user space library
  - Development using library and run as normal processes
  - VFS and kernel module forward I/O operations to user space
  - Requires mode/context switches and therefore has a lower performance



[Sven, 2007]

Outline

Summary

- File systems manage data and metadata using standardized interfaces
  - The main object are files and directories, inodes are used internally
- Specialized data structures and algorithms are used for efficiency and safety
  - Journaling is used to ensure consistency
  - Extents and tree structures decrease overhead
- Local file systems are often used for parallel distributed file systems
  - They have highly-optimized block allocation schemes etc.
  - Object stores can often be an alternative for file systems
- Modern file systems integrate additional functionality
  - Volume management, checksums, snapshots etc.
  - Both convenience and safety are increasingly important

References

[CyHawk, 2010] CyHawk (2010). **B-tree.** https://en.wikipedia.org/wiki/File:B-tree.svg. License: CC BY-SA 3.0.

[djwong, 2018] djwong (2018). **ext4 Data Structures and Algorithms.** https://www.kernel.org/doc/html/latest/filesystems/ext4/index.html.

[Fischer and Schönberger, 2017] Fischer, W. and Schönberger, G. (2017). **Linux Storage Stack Diagramm.** https://www.thomas-krenn.com/de/wiki/Linux_Storage_Stack_Diagramm.

[Pomeranz, 2008] Pomeranz, H. (2008). **Understanding Indirect Blocks in Unix File Systems.** https://www.sans.org/blog/understanding-indirect-blocks-in-unix-file-systems/.

[Sven, 2007] Sven (2007). **Filesystem in Userspace.** https://en.wikipedia.org/wiki/File:FUSE_structure.svg. License: CC BY-SA 3.0.