BTI1341 – Operating Systems

Part 3: Persistence – 12) File Systems 1

Autumn 2025-26 `master@352a46f (20250901-155000)`

P. Mainini / E. Benoist / C. Fuhrer / L. Ith

Berner Fachhochschule | Haute école spécialisée bernoise | Bern University of Applied Sciences

Outline

- File System Structures
- Access Methods (API)
- The Fast File System (FFS)
- Appendix

Introduction

So far, in this part, I/O devices and HDDs in particular have been introduced as underlying hardware components for persistent storage. Furthermore, files and directories where introduced as widely used abstraction, together with the user space APIs for working with them.

The remaining lessons of this course will now investigate file systems (FS) , the part of the kernel responsible for the interaction between storage devices and the file-/directory API.[1]

As opposed to CPU- and memory virtualization, *no new hardware features* are required. FS are pure software, which has also lead to the diversity of different concepts and implementations available today. Still, understanding of the underlying HW is important for FS design (performance, efficiency, etc.)!

[1] ☞ There are also File Systems in User Space (FUSE) which are not part of this discussion.

Structures and Methods

The first file system examined is a simplified version of the old UNIX FS.[2] There are also other basic FS architectures/principles in use, which will be not discussed further (e.g. FAT or NTFS).

When discussing file systems, two important aspects have to be considered:

1. The *data structures* used by a file system: How is data and metadata organized? Which structures are used on-disk?
2. *Access methods:* How is the user space API (i.e. calls like `open()`, `read()`, `write()`, ...) implemented? Which structures are read or written? Is it efficient?

[2] See [uni] for more details.

File System Structures

Blocks and Extents

In general, a FS divides a disk into *discrete units*, either blocks or extents. A block is again a fixed size unit (some FS may use different sizes at the same time),[3] while an extent is a contiguous region reserved for a file.

In the following, *blocks of 4 KiB and a disk of 64 blocks* will be assumed. The blocks are linearly addressable from $0 \ldots N - 1$.

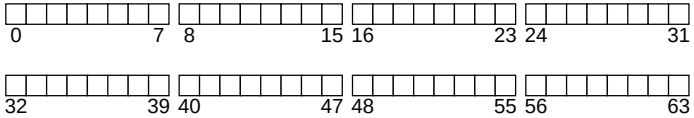

Figure: A Disk With 64 Blocks

Courtesy of [ADAD18]

[3] ⚠ The blocks used here are not the same as the physical blocks of the HDD discussed in 10-persistence-1!

Data Region and Inode Table

The available blocks are divided in two parts, depending on the kind of data stored in them:

- Data region : Effective user data *(i.e. file contents)*
- Inode table : Inodes for the stored user data (next slide)

For each part, a *fixed amount* of blocks is allocated when the FS is created, e.g.:

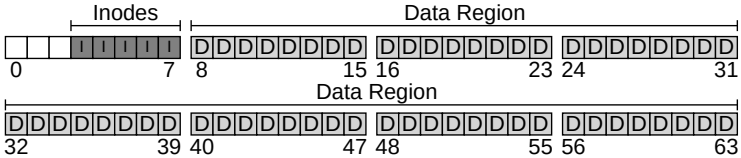

Figure: Data Region (56 Blocks) and Inode Table (5 Blocks)

Courtesy of [ADAD18]

Inode

Definition

The inode is a generic term used for the data structure holding the *metadata* of a given file.

Its contents are FS dependent and typically contain things like the file size, related data blocks, owner, permissions, creation-, access- and modification times etc. The name itself is the short form for “index node” and has been used since the early UNIX days.

Inodes are typically small, e.g. 128 or 256 bytes. For a real world example, see the ext2 FS inode structure in [lin].

Inode Example: xv6

The on-disk inode structure in xv6 is defined in `fs.h`:

```
struct dinode {
  short type;      // File type
  short major;     // Major device number (T_DEV only)
  short minor;     // Minor device number (T_DEV only)
  short nlink;     // Number of links to inode in file system
  uint size;       // Size of file (bytes)
  uint addrs[NDIRECT+1];   // Data block addresses
};
```

Allocation Structures

Similar to free-space management in virtual memory,[4] the FS needs to *track which blocks are free or in use* (both data and inode blocks). For this, some allocation structure like a free list or a bitmap is required.

For the example FS, a separate *bitmap* will be used to track free inode- and data blocks, respectively. To keep it simple, each bitmap is stored in a separate block (blocks 1 and 2):

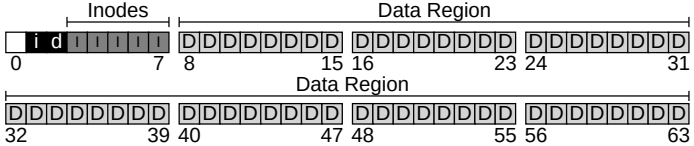

Figure: Allocation Bitmaps for Inodes and Data

Courtesy of [ADAD18]

[4] See *05-memory-3*.

The Superblock

Definition

Every file system has a superblock, which stores additional metadata about the FS itself. It typically contains a magic number (identifying the FS type), the amount of inode- and data blocks, as well as other important parameters.

Example: Block 0 is the superblock. It stores the maximum number of inodes (80, assuming 256 bytes per inode) and data blocks (56):

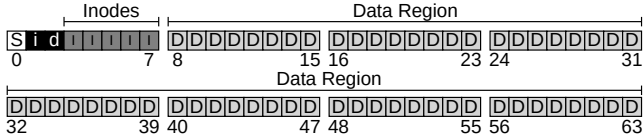

Figure: Complete Example with Superblock

Courtesy of [ADAD18]

⚠ The number of inodes is also the maximum number of files and/or directories!

Addressing Data Blocks I

Given the inode for a file, how can the corresponding data blocks be addressed? Two common options are:

1. *Lists of pointers* with data block addresses: Efficient and flexible, but limits the file size to the number of pointers × block size.
2. *A linked list* consisting of the data blocks: Simple, but certain operations are inefficient for large files (e.g. random block access). A possible solution could be caching the list in memory for open files.

Another option is to use extents : Instead of addressing individual blocks, a contiguous region is addressed using a pointer to the start and its length in blocks (similar to *segments* in virtual memory). Extents are less flexible but more compact.

Addressing Data Blocks II

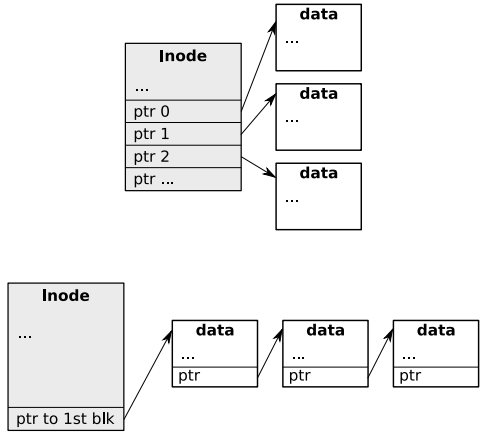

Figure: List of Pointers (top) vs. Linked List (bottom)

The Multi-Level Index

The basic idea of a multi-level index of data blocks is to have *indirect pointers*, i.e. pointers which point to a block of additional pointers. Multiple *levels of indirection* can be used, leading to a large amount of pointers. Typically, *a few direct pointers* are present in the inode itself, such that no indirection is required for small files.

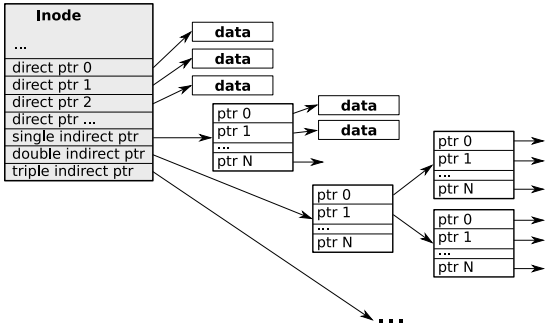

Figure: Inode with 3 Level Indirection

Example: Multi-Level Index

First, assume an inode having 12 direct pointers, allowing for file sizes up to ≈ 49 KiB (using 4 KiB blocks).

Adding a single indirect pointer adds another 1024 (32-bit-) pointers, allowing for file sizes of up to $(12 + 1024) \times 4$ KiB or ≈ 4.2 MiB.

With two indirection levels (i.e. a single and a double indirect pointer), a total of $12 + 1024 + 1024^2 \approx 1$m pointers are available, supporting files of up to ≈ 4 GiB in size.

File Size Distribution

How useful are multi-level indexes? Does it make sense to treat small files differently?

Studies have shown, that in general the *majority of files are small*. According to Tanenbaum, 60-70 % of the files fit in 4 KiB blocks. Similar results are found by Agrawal et al., which note however that large files become more popular (e.g. due to videos and databases).[5]

A measurement on the author's system seems to confirm this:

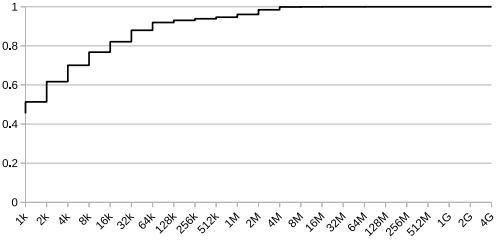

Figure: File Sizes on the Author's System, 2020 (CDF)

Directories in a FS

Recap: Directories are also just files (cf. *11-persistence-2*). Their contents are FS dependent, at the very least they contain *a list/mapping of (inode, name)-pairs*.

Here is the `dirent` structure from xv6 (`fs.h`):

```
// Directory is a file containing a sequence of dirent structures.
#define DIRSIZ 14

struct dirent {
  ushort inum;
  char name[DIRSIZ];
};
```

As directories are also just files, in general *no special treatment* by the FS is required (except for a different file type).

Access Methods (API)

Reading a File

When a file, e.g. "`/foo/bar`" (size: 3 blocks) is *read* from disk, the following steps are performed by the given example FS:

1. The *inode* of "`bar`" must be found by directory traversal :
   - 1.1 Locate the inode of "`/`" (*known* to the FS, often 2).
   - 1.2 It is a *directory* → read corresponding data blocks.
   - 1.3 Locate the inode of "`foo`". Also a directory → go back to 1.2.
2. Read the inode of "`bar`", check permissions, [...], allocate and return a *file descriptor*.
3. Read the 3 *data blocks:*
   - 3.1 Read the inode to find the address of the block.
   - 3.2 Read the data block.
   - 3.3 Update the inode of "`bar`" (e.g. last-accessed time).
   - 3.4 Repeat!

Timeline: Reading a File

	Bitmap		Inode			Data				
	D	I	/	foo	bar	/	foo	bar[0]	bar[1]	bar[2]
open			R			R				
				R			R			
					R					
read					R					
								R		
					W					
read					R					
									R	
					W					
read					R					
										R
					W					

Creating a File

The following happens on *creation* of "`/foo/bar`" (size: 3 blocks):

1. First, the *inode* needs to be created:
   - 1.1 The inode of "`foo`" needs to be found → *directory traversal*.
   - 1.2 A *new inode* is allocated for "`bar`" → read and write of the inode bitmap.
   - 1.3 The new inode of "`bar`" needs to be written.
   - 1.4 The *data of* "*`foo`*" needs to be updated with name and inode number of "`bar`".
   - 1.5 The *inode of* "*`foo`*" needs to be updated (e.g. last-accessed time).
2. The 3 *data blocks* have to be written (5 I/O requests each):
   - 2.1 A *free block* must be found → read and write of the data bitmap.
   - 2.2 A read and a write are required to *update the inode* of "`bar`".
   - 2.3 Finally, the data block is written.

Timeline: Creating a File I

	Bitmap		Inode			Data				
	D	I	/	foo	bar	/	foo	bar[0]	bar[1]	bar[2]
create			R			R				
				R						
							R			
		R								
		W								
					W					
							W			
				W						
write					R					
	R									
	W							W		
					W					

Timeline: Creating a File II

	Bitmap		Inode			Data				
	D	I	/	foo	bar	/	foo	bar[0]	bar[1]	bar[2]
					R					
write	R W									
									W	
					W					
					R					
write	R W									
										W
					W					

Observations

- Allocation structures (the bitmaps) are not accessed when reading. When writing, they are only accessed upon creation of data blocks (for a new file or when appending).
- Creating a file requires additional I/O requests, as the corresponding directory needs to be updated. This gets worse if the directory needs an additional data block for this.
- Closing a file generates no I/O requests, only the relevant in-memory structures need to be freed.
- Due to directory traversal, the amount of I/O requests grows proportionally to the length of the path name. At least two reads are required for each additional path level (or more for directories spanning multiple data blocks).

→ *For efficient FS implementation, some caching is required.*

Caching and Buffering

In general, file systems use caching for frequently used blocks and some replacement policy (cf. *05-memory-3*). Modern systems often have a unified page cache , which manages pages for virtual memory and file systems together.

For writes, caching is also called write buffering and has *not the same benefits* (writes must be done at some point). *Delaying writes* can help *grouping* them, which may reduce *unnecessary or obsolete writes*. It also enables disk scheduling (cf. *10-persistence-1*).

Write buffering is a trade-off, as *data may be lost* in case of power failure. In general, data is written after 5-30 seconds and there are utilities and syscalls to *force pending writes* (e.g. `sync`, `sync()` and `fsync()`).

The Fast File System (FFS)

Performance Issues

The given example FS, as well as the old UNIX FS, have serious performance issues:

- The layout of the on-disk FS structures causes a lot of *random disk accesses* (e.g. as inodes are far away from data blocks).

Figure: Recap: On-disk FS Structures

Courtesy of [ADAD18]

- The usage of a free-list instead of bitmaps leads to increasing *fragmentation* over time (old UNIX FS).
- Due to the *small block size* (512 bytes), a lot of overhead occurs in transfer.

FS Fragmentation

Various usage patterns may lead to FS fragmentation when *re-allocating* previously used blocks. Fragmentation in turn leads to *more random accesses* (reducing throughput):

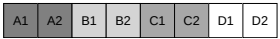

Figure: Four Files of 2 Blocks Each

Courtesy of [ADAD18]

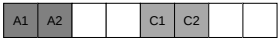

Figure: Free Space After Deleting B and D

Courtesy of [ADAD18]

Figure: Re-allocating Space for a Larger File

Courtesy of [ADAD18]

The Fast File System

Berkeley's Fast File System (FFS) , also called UNIX File System (UFS) , has addressed these issues in two important ways:

1. By being more *"disk aware"*, i.e. by organizing FS structures on-disk in a way which improves access- and read-/write performance.
2. By *using policies*, which try to keep related things together:
   - Inodes and the corresponding data blocks
   - Directories and the files they contain

The mechanisms introduced by FFS are still in use in other, modern file systems!

Grouping Blocks Together

FFS basic idea: to *reduce random accesses* to the disk, the FS is segmented into many block groups :

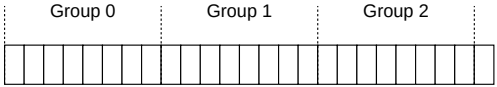

Figure: Three Block Groups

Courtesy of [ADAD18]

Each block group has its *own FS structures:*

- A copy of the FS superblock (for redundancy)
- Per-group allocation structures (inode and data bitmaps)
- Inode table and data region

Allocation Policies

To keep related things together, FFS uses simple *policies for allocating* new files and directories:

- New directories are created in a block group containing *few existing directories and many free inodes*. This helps balancing directories over the disk and ensures enough available inodes for subsequent file allocations.
- New files are placed in the *same block group as their containing directory*. Also, the data blocks belonging to a file are placed in the same group.
- If there is not enough space in a group, the next adjacent group is used.

Example: FFS Allocation

Example: Allocation of 3 directories "`/`", "`/a`" and "`/b`", as well as 4 files "`/a/c`", "`/a/d`", "`/a/e`" and "`/b/f`" (size: 2 blocks per file). Here, the FFS consists of block groups with 8 inodes and 16 data blocks per group.

```
group  inodes  data
  0    /-------  /---------------
  1    acde----  accddee---------
  2    bf------  bff-------------
  3    --------  ----------------
  4    --------  ----------------
  5    --------  ----------------
 ...
```

The Large-File Exception

For *large files*, there is an exception to the rules given on Slide 31: If treated normally, a large file would fill one or more block groups:

```
group  inodes  data
  0    /a------  /aaaaaaaaaaaaaaa
  1    --------  aaaaa-----------
 ...
```

Instead, to preserve room for other files of the group, FFS distributes large files over multiple groups in *chunks* of a given size:

```
group  inodes  data
  0    /a------  /aaaaa----------
  1    --------  aaaaa-----------
  2    --------  aaaaa-----------
  3    --------  aaaaa-----------
 ...
```

Large-File Performance Impact

Distributing a file over multiple groups has a negative impact on *performance for sequential access.* With an *appropriate chunk size*, this effect can be amortized.[6]

Example: Having a disk with an average positioning time of 10 ms and a transfer rate of 40 MiB/s, how big should a chunk be to reduce performance at most by 50%?

Solution: The chunk size must be chosen such that the transfer of a chunk takes at most 10 ms as well. For 40 MiB/s, the chunk size would be 400 KiB.

[6] By reducing positioning time (seek and rotation) vs. transfer time.

Chunk Size (Amortization) I

More precisely, the required chunk size depending on the acceptable performance loss can be calculated as follows:

$$D = \frac{F}{1-F} \cdot R_{\text{peak}} \cdot T_{\text{position}}$$

Where D is the size of the chunk, R_{peak} is the peak transfer rate in MiB/s and T_{position} the average positioning time of the disk in seconds. F is the desired fraction of the peak performance.

For the example on the previous slide:

$$D = \frac{0.5}{0.5} \cdot 40 \text{ MiB/s} \cdot 0.01\text{s} = 400 \text{ KiB}.$$

Note: FFS does not make such calculations, but distributes chunks based on the inode structure (amount of pointers etc.).

Chunk Size (Amortization) II

Amortization is a trade-off in disk space usage vs. sequential bandwith, as can be seen in the following figure:

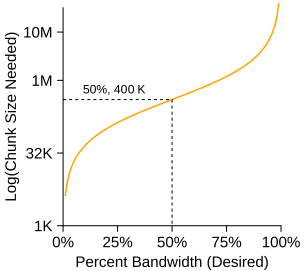

Figure: Amortization for Chunk Size vs. Bandwidth

Courtesy of [ADAD18], Own Modification

Locality Assumptions

The FFS policies (Slide 31) are again based on the assumption of the locality of reference principle.[7] Does the assumption hold?

Effectively, files which are *close to each other* in a directory tree are often also *accessed together!*

Examples:

- Source code and compilation output
- Contents on a web server
- Tracks of a music album
- etc.

[7] See *04-memory-2*.

Measuring Locality

A study on the SEER system traced file accesses and measured locality using a metric: 0 if the same file was accessed again, 1 for another file in the same directory etc.[8] Result: The same file was accessed again $\approx 7\%$ of the time, while accesses to the same file or one in the same directory made up $\approx 40\%$ of the requests:

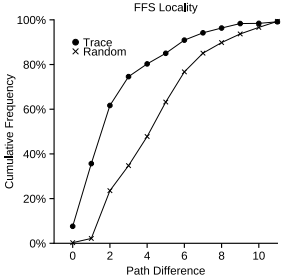

Figure: FFS Locality for SEER Traces

Courtesy of [ADAD18]

[8] See [Kue94] for details.

Appendix

Bibliography I

[ABDL07] Nitin Agrawal, William Bolosky, John Douceur, and Jacob Lorch, *A five-year study of file-system metadata*, TOS 3 (2007).

[ADAD18] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, 1.00 ed., Arpaci-Dusseau Books, August 2018, Available online: `http://ostep.org`.

[Kue94] Geoffrey H. Kuenning, *The Design of the Seer Predictive Caching System*, Proceedings of the Workshop on Mobile Computing Systems and Applications, 1994, pp. 37–43.

[lin] *Linux kernel*, *ext2 FS inode structure*, `https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/fs/ext2/ext2.h?h=v4.19.98#n302`.

[TB14] Andrew S. Tanenbaum and Herbert Bos, *Modern operating systems*, 4th ed., Prentice Hall Press, USA, 2014.

[uni] *Wikipedia, Unix File System, History and evolution*, `https://en.wikipedia.org/wiki/Unix_File_System#History_and_evolution`.