Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

# Operating Systems

## Part 3: Persistence – 13) File Systems 2

Revision: `master@323240b (20230907-115823)`

**BTI1341 / Fall 2023/24**

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

# Outline

Crash Consistency

Journaling

Log-Structured File Systems (LFS)

Nonvolatile Memory Devices (NVM)

The End!

Appendix

# Crash Consistency

# Motivation

A file system is different compared to other OS data structures seen so far: it has a *long lifetime*.

An important aspect is crash consistency : How to update FS structures despite *power loss* or *system crashes*?

Main problem: Consistency of operations requiring *more than one* disk write.[1]

Think e.g. of an update consisting of two or more separate, on-disk structures which need to be written together. In any case, one of them will be written first. What happens if power fails before the second can be written?

[1] Recall from *10-persistence-1*, that HDDs guarantee only a single block to be written atomically!

# Example

Assume *appending a new block* to an existing file, with the following initial on-disk structures:


Figure: A File With a Single Data Block

Courtesy of [ADAD18]

The simplified inode of the file looks as follows:

```
size     : 1
pointer0 : 4
pointer1 : null
pointer2 : null
pointer3 : null
```

# Example (cont.)

Appending a block requires *three separate writes*: data block, data allocation bitmap and inode of the file.


Figure: The File With a New Block Appended

Courtesy of [ADAD18]

Here is the updated inode:

```
size     : 2
pointer0 : 4
pointer1 : 5
pointer2 : null
pointer3 : null
```

# Failure Scenarios

In the previous example, seven different failure scenarios may occur, depending on which block(s) could have been written before the outage:

| **Block(s) written** | **Result** |
|---|---|
| None | Failure occurs before any write starts. Data loss, but no FS issue. |
| Only data | Data loss, but no FS issue (FS still consistent). |
| Only inode | Data loss, inode points to garbage block. FS is inconsistent (no allocation in bitmap). |
| Only bitmap | Data loss, FS is inconsistent , space leak as allocated block will never be used or freed again. |
| Data and inode | *No data loss*, but FS is inconsistent (no allocation in bitmap). |
| Data and bitmap | Data loss and FS inconsistent . |
| Inode and bitmap | Data loss, inode points to garbage block. FS is *consistent*. |

# File System Issues

As seen on the previous slide, different kinds of *FS issues* can occur:

- **Data loss**: Data is simply lost, and there is nothing that can be done about it. *May require some action*, if at the same time the FS has become inconsistent.
- **File system inconsistency**: There is a contradiction in the FS structures, e.g. between inodes and allocation structures. *Must be repaired*.
- **Space leaks**: A special kind of inconsistency, which reduces the capacity of the file system. *Must be repaired*.

There are different approaches for repairing and ensuring FS crash consistency, which will be explored in the following.

# File System Checkers

A file system checker is a program which normally runs while the file system is *not mounted* (e.g. at boot). It tries to restore FS consistency by running *different checks* and fixing detected issues.

Some of the checks performed include:

- Sanity checks of the superblock
- Sanity checks of inodes (e.g. file type, number of links)
- Restore allocation structures based on the inode contents
- Checks of directory content

With the increase of disk sizes, file system checking became nearly *impractical* (taking minutes to hours). This is even worse for RAIDs and similar systems. Furthermore, running such expensive checks to detect only a few bad blocks is *inefficient*.

# Journaling

# Introduction

Another approach for crash consistency comes from databases: journaling , a variant of so-called write-ahead logging .

Basic idea: *before* performing the effective updates, write a log of *what will be done*, also called a transaction . Then, if a crash occurs during the updates, the log can be consulted and the updates can be *replayed* if necessary.

This is a *trade-off*: Normal writes require a bit more time, but crash recovery is much faster.

The following discussion of FS journaling is based on the *ext3* file system, which extends *ext2* with journaling capabilities.

# Adding a Journal

Recap: This is a basic FS with block groups, like e.g. FFS, introduced in *12-persistence-3*:

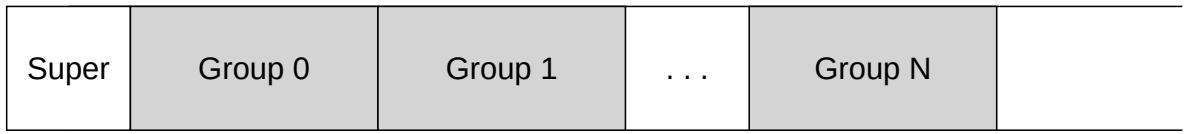



Figure: A File System With Block Groups

Courtesy of [ADAD18]

Journaling now adds an additional FS structure, namely the journal . In general, it is placed somewhere on the same disk, but other options, e.g. a separate journal disk, are possible as well.


Figure: FS With Journal Added

Courtesy of [ADAD18]

# Journaled Writing

Compared to a FS without journaling, writing a file to disk now consists of two steps:

1. Write changes (data and metadata) to the journal.
2. Checkpoint the FS, i.e. write the changes to disk.

In the example on Slide 5, a new data block (Db) is written, requiring updates to the inode and inode bitmap (I and B). The contents of a typical journal for this transaction could look as follows:

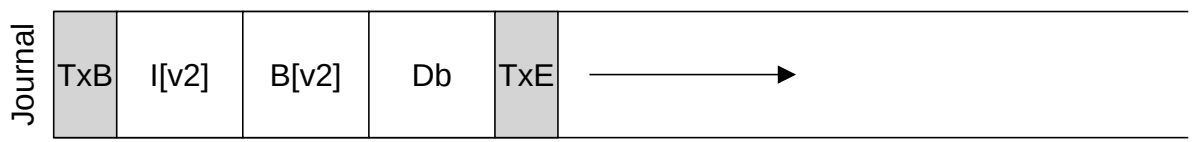



Figure: Journal Contents (Inode, Bitmap, Data Block)

Courtesy of [ADAD18]

Note that each transaction is delimited by some *markers* (TxB and TxE), containing *additional metadata*.

# Crash Consistency

Assuming that the *complete transaction* was written to the journal, a crash during the effective writes *can be recovered*.

*But: How to deal with crashes while writing the journal itself?*

Solution: write the journal in *two steps*:


Figure: Step One: **Journal Write**

Courtesy of [ADAD18]

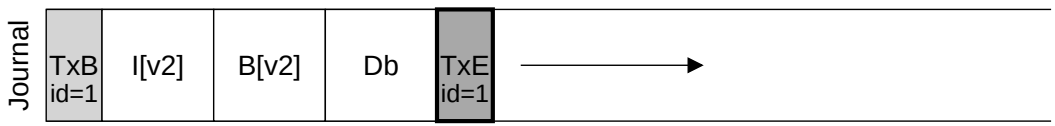



Figure: Step Two: **Journal Commit**

Courtesy of [ADAD18]

⚠ As long as the *commit* only writes a *single disk block*, this works.

# Recovery With Journal

With a journal, there are two possible scenarios for crash recovery:

1. The crash occurred *before journal commit*: There is no state which could be safely recovered. *Data is lost*, the transaction is removed and the FS is still in consistent state.

2. The crash occurred any time *while checkpointing*: The FS can simply *replay the transaction* (i.e. perform all writes) out of the journal. Multiple transactions are simply replayed in order. This is also called  redo logging .

# Managing the Journal

Clearly, the journal may not grow indefinitely. After a transaction has been checkpointed, it needs to be freed again in the journal.For this, a circular data structure can be used:

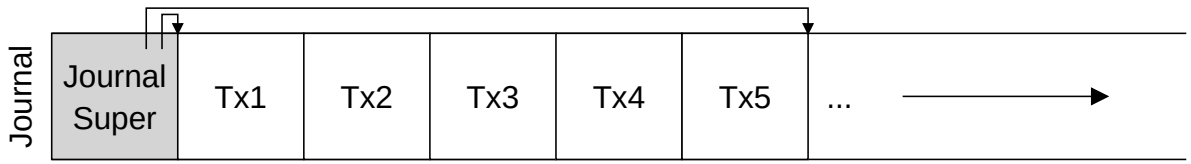



Figure: Circular Journal

Courtesy of [ADAD18]

Thus, a full *journaled write* consists of the following steps:

1. Write to journal
2. Journal commit
3. Checkpoint (write FS changes)
4. Free the checkpointed transaction

# Metadata Journaling

Journaling helps with fast recovery, but *data is written twice*! This is especially bad for sequential write performance.

A possible solution for this is metadata journaling , which only writes the metadata, i.e. the inode and the inode bitmap, to the journal. Data blocks are written directly and only once.

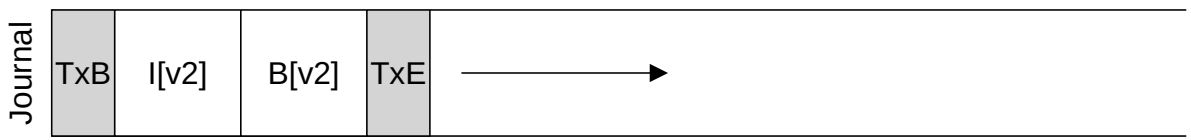



Figure: Metadata Journal

Courtesy of [ADAD18]

⚠ The order is important: Data blocks must have been written before committing the journal!

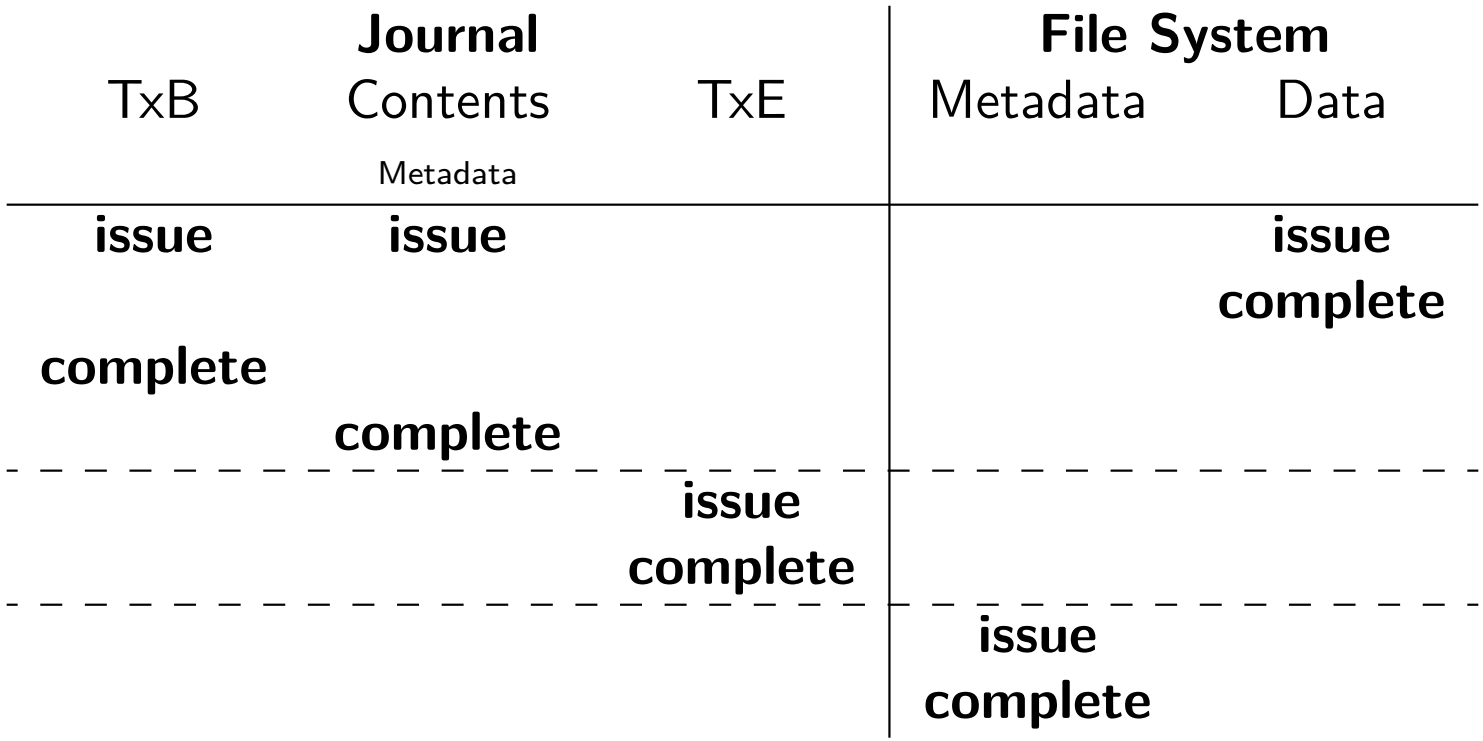
# Timeline: Data Journaling

The following table depicts an exemplary timeline for data journaling (including data blocks). Completion times are arbitrary.

| **Journal** | | | | **File System** | |
|---|---|---|---|---|---|
| TxB | Contents | | TxE | Metadata | Data |
| | Metadata | Data | | | |
| **issue** | **issue** | **issue** | | | |
| **complete** | | | | | |
| | **complete** | | | | |
| | | **complete** | | | |
| | | | **issue** | | |
| | | | **complete** | | |
| | | | | **issue** | **issue** |
| | | | | | **complete** |
| | | | | **complete** | |

# Timeline: Metadata Journaling

The following table depicts an exemplary timeline for metadata journaling (without data blocks). Completion times are arbitrary.



# Other Approaches

File system checking and journaling are two possible approaches to maintain FS consistency. There are other possibilities:

- Log-structured FS , discussed in the next Section. In practice, only used for specific applications, e.g. on media with wear out (cf. Slide 35 ff.).
- FS using soft updates , which perform "dependency-tracking" between writes to ensure that no inconsistency can occur. This is more complicated than journaling but has the advantage that the FS can be directly mounted after a crash (no journal replay).
- FS using Copy-On-Write (COW) , which never overwrite blocks on update and also provide deduplication. Examples are ZFS and Btrfs.

See also [hen].

# Log-Structured File Systems (LFS)

# Motivation

To avoid writing data twice, the
Log-structured File System (LFS) only writes the log to disk and stores data directly as part of the log. It is motivated by the following observations:

- More RAM available $\rightarrow$ more caching. Due to this, disk accesses are *mostly writes*, reads come from the cache.
- Sequential access is much faster than random access.
- Common workloads perform poorly in FS like FFS. As e.g. shown in *12-persistence-3*, creating a new file requires many writes.
- Bad performance on RAID systems, mostly due to the *small write problem* (*11-persistence-2*).

# Basic Idea

The basic idea is simple: perform *all writes sequentially*. Also, writes should either be *contiguous or large* to reduce rotational delays.

To achieve this, LFS buffers writes in memory as a *segment*, and then writes the whole segment to disk when it is full. The following is an example segment, updating two inodes:

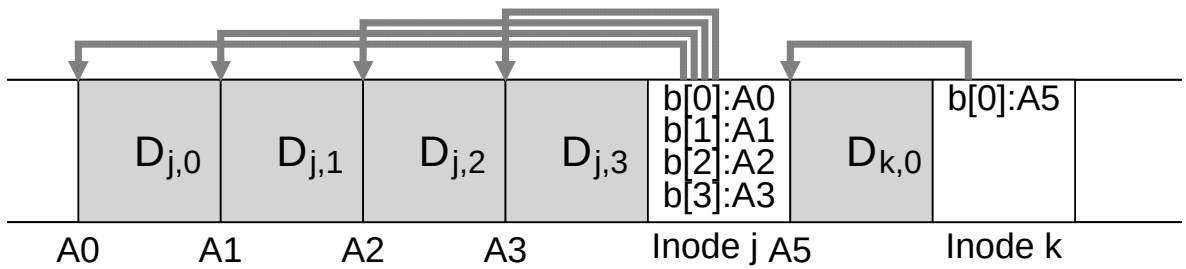



Figure: LFS Segment Updating Two Inodes

Courtesy of [ADAD18]

# Segment Size

LFS segment writes are sequential writes, comparable to FFS large file writes. Performance depends on the size of the segment chosen.

To find the *optimal segment size*, amortization can be used, as for finding FFS chunk sizes.[2]

**Example:**
Assuming a disk with a peak transfer rate of 100 MiB/s and a positioning time of 10 ms, which segment size should be chosen to obtain 90% of the possible peak performance?

**Solution:**

$D = \frac{0.9}{0.1} \cdot 100 \text{ MiB/s} \cdot 0.01\text{s} = 9 \text{ MiB}.$

[2] Recall from *12-persistence-3, Slide 35:* $D = \frac{F}{1-F} \cdot R_{peak} \cdot T_{position}$

# The Inode Map (IMap)

A segment can be written *anywhere on disk* and contains updates to random inodes. Thus, *inodes have no fixed location* anymore and a different mechanism for finding them is required.[3]

For finding inodes, LFS uses an Inode Map (IMap) : basically a list of inode numbers with pointers to their current on-disk location. As it needs to be updated frequently, it is *split in chunks* which are written as *part of the segment* in which the updates occur:

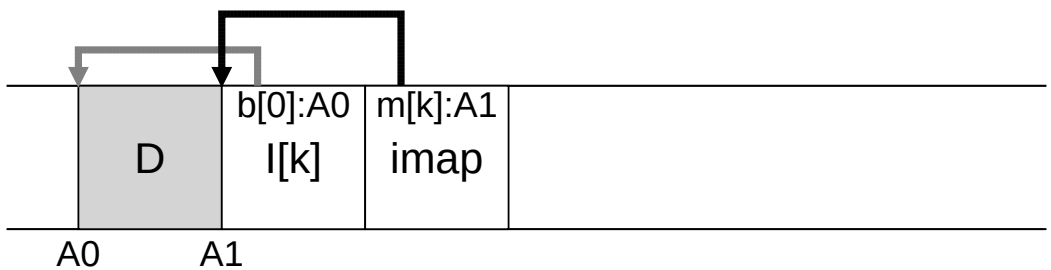



Figure: LFS Segment With Inode Map Chunk

Courtesy of [ADAD18]

[3]Note also, that older segments are not necessarily overwritten and thus multiple copies of an inode may exist independently.

# Finding the Inode Map

Finally, as IMap chunks are written as part of different segments, a mechanism for finding them is required. For this, the checkpoint region contains a pointer to each current chunk of the IMap:

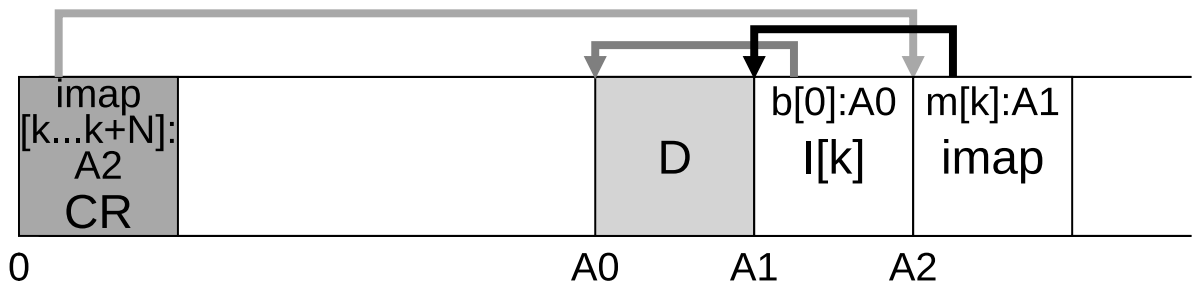



Figure: The Checkpoint Region

Courtesy of [ADAD18]

Clearly, the checkpoint region must reside at a well-known location on disk. It is *cached in memory* and only updated periodically (e.g. every 30 secs) to prevent negative impact on performance.

# Directories

In LFS, a directory is also just a list of *(inode, name)-pairs*:

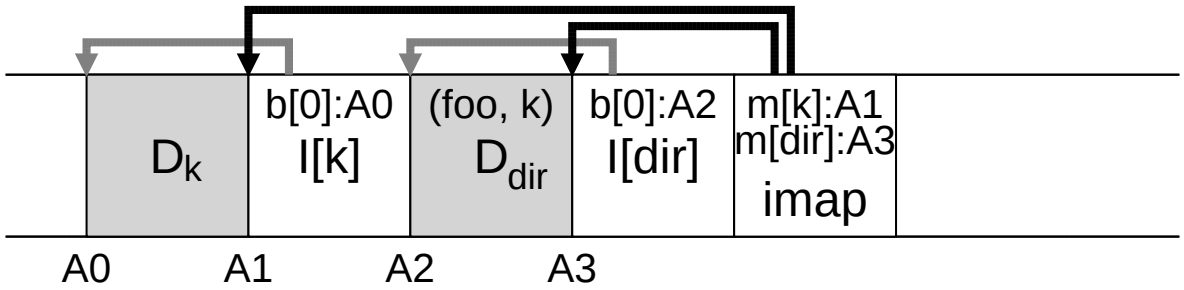



Figure: A Segment Updating a File and its Inode

Courtesy of [ADAD18]

As the *IMap is an indirection*, updates to an inode do not require an update of the directory! This prevents the recursive update problem which could arise otherwise: Updating an inode would lead to an update of the containing directory, which in turn would lead to an update of its containing directory etc.

# Garbage Collection

When updating and appending new blocks to an inode, old versions of the inode- and data blocks remain on the disk:

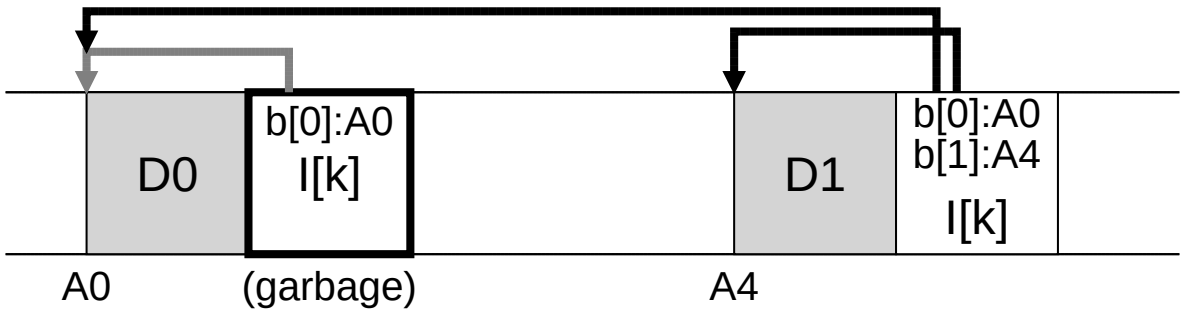



Figure: Garbage After Appending a Block to an Inode

Courtesy of [ADAD18]

Periodically (or when there is no space), garbage collection must take place. To avoid fragmentation and ensure constant write performance, *whole segments must be freed* and their allocated blocks be copied to a new segment.

# Finding Allocated Blocks

To identify allocated blocks, every segment in LFS has a Segment Summary (SS) block . It contains a record for every block in the segment with the corresponding inode and the data block offset within that inode:

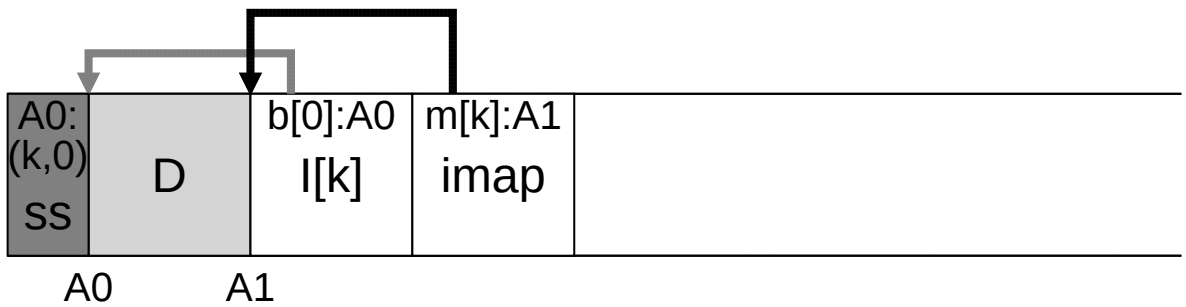



Figure: The Segment Summary Block

Courtesy of [ADAD18]

In the figure above, the SS records that the data block at address `A0` is the first data block (index 0) of the inode `k`.

# Finding Allocated Blocks (cont.)

Using the segment summary, *determining if a block is allocated* is straightforward:

1. Given disk address, find the inode number and offset in the SS.
2. Find inode address in the IMap.
3. Using offset, *compare block address from SS and inode*.

If the inode and the SS point to the same block, the block is still allocated. Otherwise, it may be garbage collected.

To further improve performance, LFS keeps an inode version number in the IMap and SS. If a file is deleted (or truncated), the version is increased in the IMap. If the version from SS and IMap differ, all corresponding data blocks can be considered unallocated.

# Recovery / Crash Consistency

In LFS, the segments are *organized like a journal*, with a pointer in the SS pointing to the next segment. The checkpoint region keeps pointers to the first and last segment of the log. Crashes may occur at two points:

1. During *segment write*: Due to caching of the checkpoint region (cf. Slide 26), many IMap updates can get lost. To prevent this, roll forward retrieves the known end of the log from the last checkpoint region and tries to find further segments from there.

2. During *checkpoint region update*: LFS maintains two copies of the checkpoint region, which are themselves written in journaled fashion to ensure recovery and crash consistency.

# LFS Overall Structure

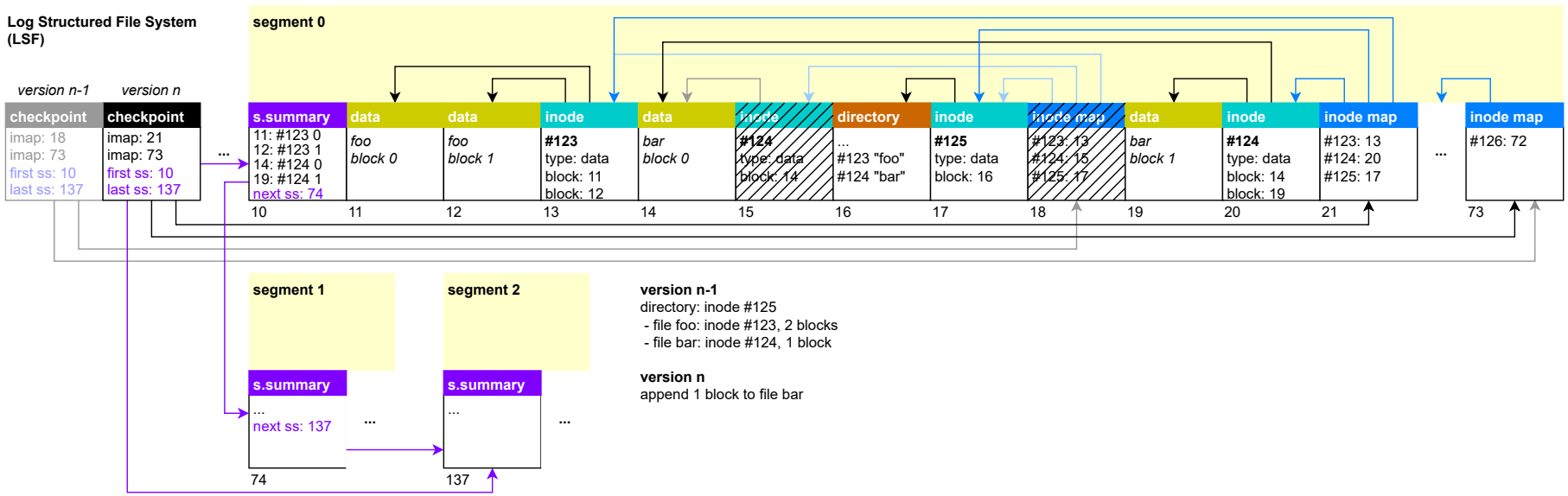



Figure: Combined Example of LFS Structures

# Nonvolatile Memory Devices (NVM)

# Overview

Nonvolatile Memory Devices (NVM) have become an important *alternative to HDDs*.[4] Examples include:

- USB drives (USB sticks), SD cards, etc.
- Solid-state disks (SSD)
- Storage in embedded devices (e.g. included in smartphones)

In general, NVM are based on NAND Flash technology and have no moving parts. Advantages are:

- Increased reliability
- Lower latency / higher troughput
  $\rightarrow$ Especially for random reads and writes!
- Reduced power consumption

On the downside, they are in general *more expensive per MB* (still).

[4] See *10-persistence-1* for an introduction of HDDs.

# NAND Flash Technology

In NAND Flash, bits are stored *electrically* in transistors (vs. magnetically for HDDs). Single-Level Cells (SLC) and Multi-Level Cells (MLC) can be distinguished. SLCs store a single bit per cell and are typically faster but also more expensive.

A large number of *cells are grouped into banks* (planes). Parts of a bank are addressed either as a block (also: erase block) or as a page. *Pages have a few KiB* (up to 8 KiB), *blocks are larger* (up to 2 MiB).

Compared to HDDs, the low level semantics are different:

- Read: Any *page* can be addressed and read.
- Erase: Before writing a page, its *whole block must be erased*, which sets all bits to **1**.
- Write (program): After erasing, individual bits are set to **0**.

# NAND Flash Technology (cont.)

Thus, for writing a page, a whole block must be erased, which requires *caching and restoring* all other pages of the block!

The following table shows the *asymmetry in duration* for the different operations (sources: [ADAD18, ana]).

| | **Read** $[\mu s]$ | **Write** $[\mu s]$ | **Erase** $[\mu s]$ |
|---|---|---|---|
| SLC | 25 | 200-300 | 1500-2000 |
| MLC[5] | 50 | 600-900 | $\sim 3000$ |
| TLC[6] | $\sim 75$ | $\sim 900$-1350 | $\sim 4500$ |

The last important difference compared to HDDs is wear out of cells over time: each cell only supports a limited number of write and erase cycles!

[5] Confusingly, "MLC" is used for cells with two and more levels.

[6] Tripple-Level-Cells

# Solid State Disks (SSD)

Solid State Disks (SSD) are a class of NVM devices providing a *HDD interface* to the host. Internally, they consist of multiple flash chips, some RAM and control logic:

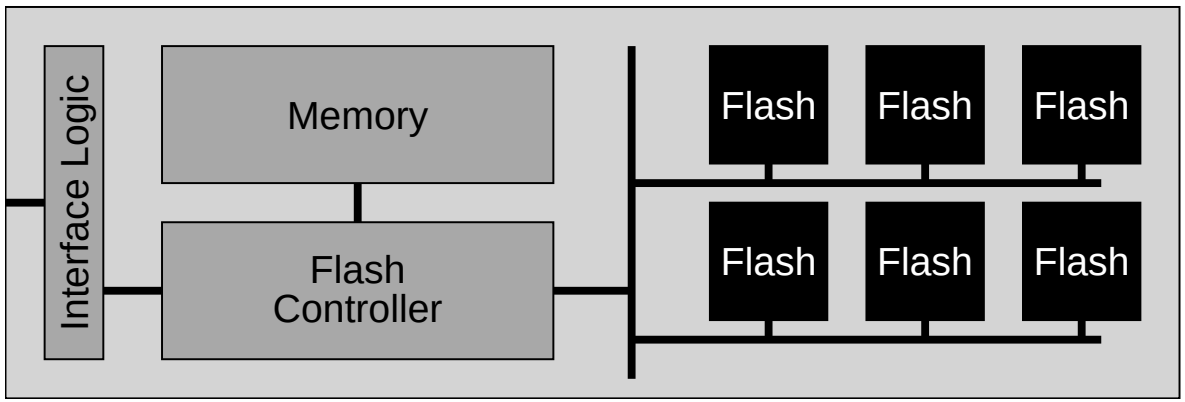



Figure: A Logical View of an SSD

Courtesy of [ADAD18]

In the controller, the Flash Transition Layer (FTL) translates between HDD I/O requests (i.e. reading/writing blocks) and flash access (reading, writing and erasing of pages/blocks).

# Wear Leveling

As cells cannot be written indefinitely, *writes must be distributed* uniformly over all pages. For early SSDs, special file systems like JFFS2 where required to prevent premature failure.[7] With modern devices, the *flash controller* takes care of wear leveling .

A log-structured approach, like LFS, is well suited for implementing wear leveling. In fact, similar methods are typically used by the flash controller:

- Due to the log structure, every write goes to a new block.
- Performance is improved as erasing of blocks is less frequently required.
- Still, additional precautions must be taken to periodically relocate control structures and rarely used pages.

[7] Those might still be required on certain kinds of NVM storage!

# The End!

Congratulations! You have reached the end of this course in operating systems – but hopefully not the end of your personal journey into the subject...

Three major topics where treated: virtualization, concurrency and persistence. The knowledge gained in each of these provides a solid foundation for further exploration. Furthermore, there are many more interesting topics not covered in detail, as for example device drivers and OS security.

Operating systems are fascinating works of computer science and engineering. Knowledge of their first principles leads to a better understanding in almost all areas where IT is involved today.

# Appendix

# Bibliography


[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`.

[ana] *Understanding TLC NAND*, `https://www.anandtech.com/show/5067/understanding-tlc-nand/`.

[hen] *lwn.net, KHB: A Filesystems reading list*, `https://lwn.net/Articles/196292/`.