



National Yang Ming Chiao Tung University
Computer Architecture & System Lab

File System-II

IOC5226 Operating System Capstone

Tsung Tai Yeh

Department of Computer Science

National Yang Ming Chiao Tung University



Acknowledgements and Disclaimer

- Slides were developed in the reference with
 - MIT 6.828 Operating system engineering class, 2018
 - MIT 6.004 Operating system, 2018
 - Remzi H. Arpaci-Dusseau etl. , Operating systems: Three easy pieces. WISC



Outline

- Block devices vs. raw flash devices
- Journaled file system
- Flash file systems



Block vs. raw flash device

- Storage devices: **block devices** and **raw flash devices**
 - They are handled by different subsystems and different filesystems
- **Block devices**
 - Can be **read and written to on a per-block basis, in random order, without erasing**
 - **Hard disks, RAM disks**
 - SSD, SD cards, eMMC: flash-based storage, but have an integrated controller that emulates a block device, managing the flash in a transparent way
- **Raw flash devices** (driven by a controller on the SoC)
 - They can **read, but writing requires prior erasing**
 - **NOR flash, NAND flash**



Block device list

- The list of all block devices available can be found in ‘/proc/partitions’
- **/sys/block**
 - Stores information about each block device

major	minor	#blocks	name
8	0	41943040	sda
8	1	512000	sda1
8	2	512000	sda2
8	3	40916992	sda3
11	0	759172	sr0
253	0	36720640	dm-0
253	1	4194304	dm-1



Partitioning

- Block devices can be partitioned to store different parts of a system
 - **The partition table** is stored inside the device itself, and is read and analyzed automatically by the Linux kernel
 - **mmcblk0** is the entire device
 - **mmcblk0p2** is the second partition of mmcblk0
 - Two partition table formats
 - **MBR (Master Boot Record)**
 - **GPT (GUID Partition Table)** supports disk bigger than 2TB
 - Numerous tools to create and modify partitions on a block device
 - fdisk, cfdisk, sfdisk, parted, etc.



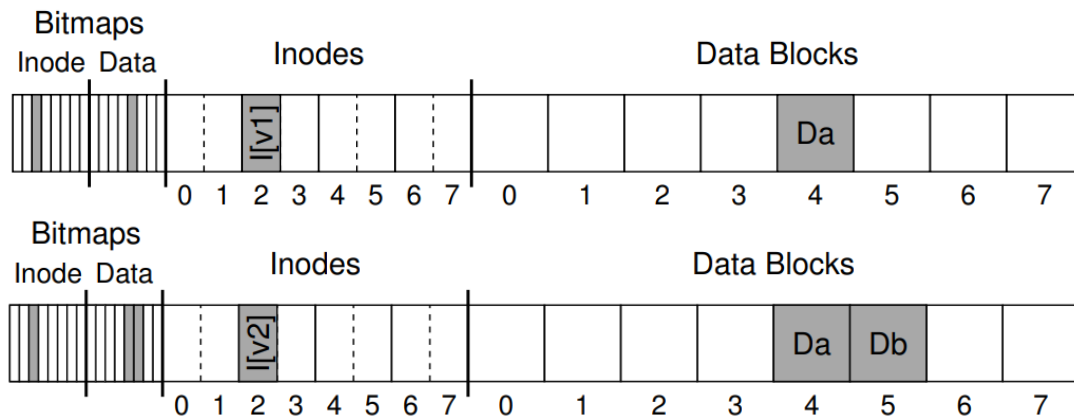
Transfer data to a block device

- Transfer data to or from a block device in a raw way
 - This directly writes to the block device itself, **bypassing any filesystem layer**
 - The block devices in '/dev/' allow such raw access
 - dd (**d**isk **d**uplicate) is the tool of choice for such transfers
 - **dd if=/dev/mmcblk0p1 of=testfile bs=1M count=16**
Transfer 16 blocks of 1 MB from /dev/mmcblk0p1 to testfile
 - **dd if=testfile of=/dev/sda2 bs=1M seek=4**
Transfer the complete contents of testfile to /dev/sda2, by blocks of 1 MB, but starting at offset 4 MB in /dev/sda2



File system in-consistency

- A single inode is allocated (inode number 2) marked in the inode bitmap, and a single allocated data block (data block 4)
- The inode is denoted $I[v1]$, as it is the first version of this inode



- When appending to the file, we add a new block (Db) to it
- Update the inode, new data block, and a new version of the data bitmap $B[V2]$



File system in-consistency

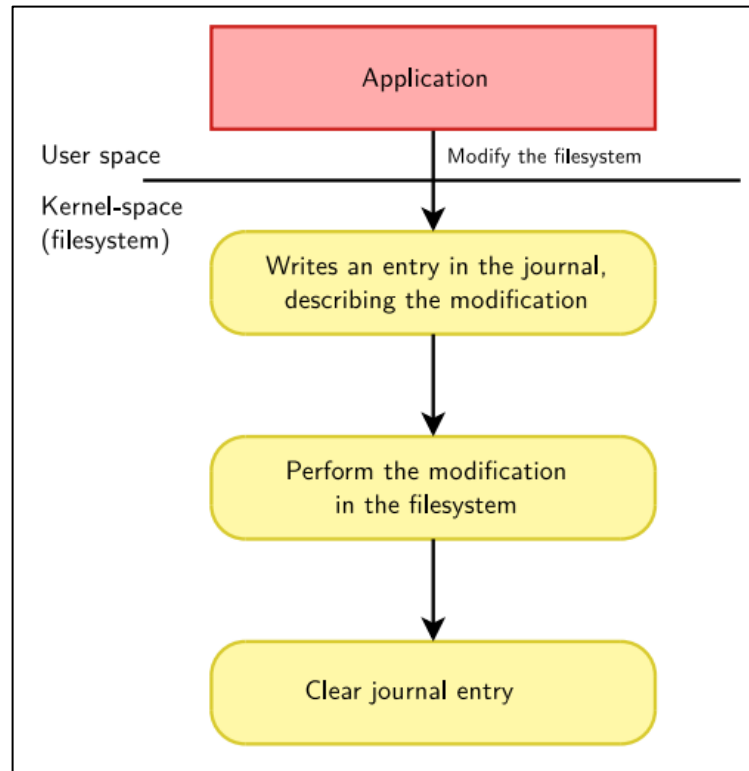
- The **writes** of appending data **don't happen immediately** when the user issues a write() system call
 - The dirty inode, bitmap, and **new data will sit in main memory** (in the buffer cache) **for some time first**
 - Then, the file system will issue the requisite write requests to disk
 - **A crash happens after one or two of these writes -> cause file-system in-consistency**



Journalled filesystems

- **Write-ahead logging**

- When updating the disk
- Before overwriting the structures in place
- First write down a little **note** on the disk
- The note describes what you are about to do
- **By writing the note to disk -> guarantee that if a crash takes place during the update**





Data journaling

- **In ext2 file system**

- The disk is divided into block groups
- Each block group contains an inode bitmap, data bitmap, inodes, and data blocks



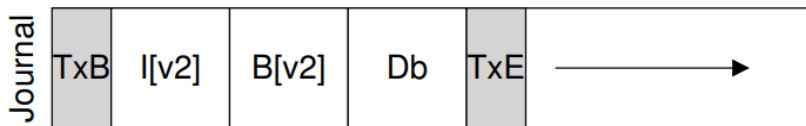
- **In ext 3 file system**

- The journal occupies some small amount of space within the partition or on another device
- Before writing each block group to its final disk location, we are now first going to write them to the log





Data journaling



- **The transaction begin (TxB)**
 - Tells us about the update, including information about the pending update (I[V2], B[V2], and Db) to the file system and transaction identifier (TID)
- **The transaction end (TxE)**
 - TxE is a marker of the end of the transaction, also include TID
- **Checkpoint**
 - Once the transaction is safely on disk, we are ready to overwrite the old structures in the file system
 - We issue the writes I[V2], B[V2], and Db to their disk locations
 - If these write complete successfully, we have done checkpointed



Data journaling

- **Journal write**

- Write **a transaction-begin block** to the log
- Write **all pending data and metadata updates** to the log
- Write **a transaction-end block** to the log
- Wait for these writes to complete

- **Checkpoint**

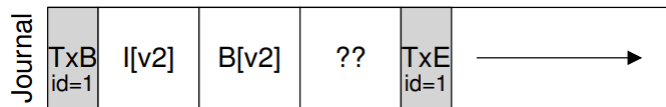
- Write the pending metadata and data updates to their final locations in the file system

- How about a crash occurs during the writes to the journal ?



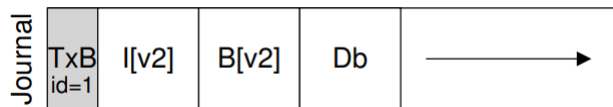
Data journaling

- **How about a crash occurs during the writes to the journal ?**
 - **One simple way to do is to issue each one item (TxB, I[V2], B[V2], Db, TxE) at a time, waiting for each to complete -> too slow**
 - **How about issue all five block writes at once ? (unsafe, why ?)**
 - **Given such a big write, the disk may perform scheduling and complete small pieces of the big write in any order**
 - (1) write TxB, I[v2], B[v2], and TxE
 - (2) write Db
 - How about the disk loses power between (1) and (2) ?



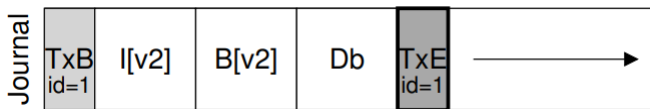


Data journaling



- **How about a crash occurs during the writes to the journal ?**

- The file system issues the transactional write in two steps
- First, write all blocks except the TxE block to the journal
- Second, issue the write of the TxE block
- Why does this two-step method work ?
 - The disk guarantees that any 512-byte write (one block)will either happen or not



- **Three phases on the current protocol to update file system**

- **Journal write:** write TxB, metadata, and data to the log
- **Journal commit:** write TxE to the log, wait for write to complete
- **Checkpoint:** write the contents of the update to their final on-disk location



File system recovery after crashes

- The **crash happens before the transaction is written** safely to the log
 - The pending update is simply skipped
- The **crash happens after the transaction has committed** to the log and before the checkpoint is complete
 - The file system can **recover the update when the system boots**
 - The file system recovery process will **scan the log and look for transactions that have committed to the disk**
 - These transactions are replayed to write blocks to their final on-disk locations (redo-logging)



Batch log updates

- **How to reduce excessive write traffic** during the update of log back to the disk ?
 - To create one file, one has to update several on-disk structures
 - **Inode bitmap** (to allocate a new inode)
 - **The newly-created inode of the file**
 - **The data block of the parent directory**
 - **The parent directory inode**
 - The **Linux ext3 don't commit each update to disk one at a time**
 - **Buffer all updates into a global transaction**
 - **Only marks** the in-memory structures as dirty
 - The signal global transaction is committed when it is finally time to write blocks to disk



Finite size journaling

- **The log is of a finite size.** What happens if the log is full ?
 - The larger the log, the longer recovery will take
 - No further transactions can be committed to the disk
- **Circular log**
 - Journaling file systems **treat the log as a circular data structure, re-using it over and over**
 - Once a transaction has been checkpointed, the file system should free the space it was occupied, allow the log space to be reused
 - E.g. The journal superblock records enough information to know which transactions have not yet been checkpointed

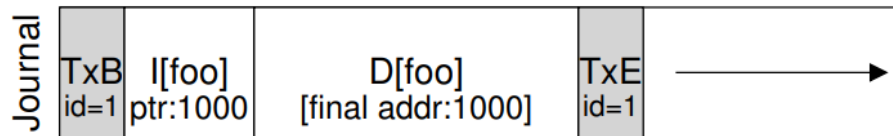


Metadata journaling

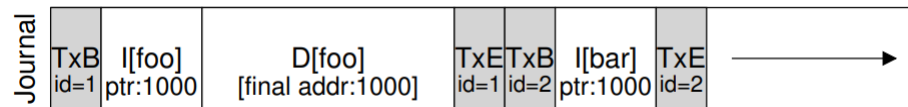
- In **journaling file system**, we are writing to the journal first for each write to disk -> **double write traffic**
 - **One write to the journal, the other writes to the main file system**
- **Data journaling** (ordered journaling in Linux ext3)
 - **The data block (Db) is not written to the journal**
 - The $I[v2]$, $B[v2]$ are both metadata and will be logged and then check-pointed
 - The Db will only be written once to the file system
 - Linux ext3 write data blocks to the disk first before related metadata.
Why ?



Block reuse



- **In some form of metadata journaling**
 - Data blocks for files are not journaled
 - A directory called foo, which contents are written to the log
- **When a user deletes everything in the directory**
 - Freeing up block 1000 for reuse
 - A new file (bar) is created
 - The inode of bar is committed to disk
 - Only the inode of bar is committed to the journal because metadata journaling is in use
 - The newly-written data in block 1000 in the file bar is not journaled





Block reuse

- **Assume a crash occurs**

- The newly-written data in block 1000 in the file bar is not journaled
- The recovery simply replays everything in the log
- Write the directory data in block 1000, which overwrites the 'bar' data with old directory contents !

- **In Linux ext3**

- Add a new type of record to the journal, known as a **revoke** record
- Deletes the directory would cause a revoke record to be written to the journal
- **Any such revoked data is never replayed**



Other approach

- How to keep file system metadata consistent ?
- **Copy-on-write (COW) file system**
 - Sun's ZFS
 - **Never overwrites files or directories in place**
 - Places new updates to previously unused locations on disk
 - After a number of updates are completed, COW file systems flip the root structure of the file system to include pointers to the newly updated structures



Other journaled Linux/UNIX file systems

- **btrfs**
 - Integrates data checksumming, volume management, snapshots, etc.
- **XFS**
 - High-performance file system inherited from SGI IRIX
- **ZFS**
 - Provide standard and advanced file system and volume management (CoW, snapshot, etc.)
- All those file systems provide the necessary functionalities
 - Symbolic links, permissions, ownership, device files, etc.



tmpfs: file system in RAM

- **Not a block file system**
- **Store temporary data in RAM**
 - System log files, connection data, temporary files ...
 - More space-efficient than ramdisks: files are directly in the file cache, grows and shrinks to accommodate stored files
- **How to use ?**
 - `mount -t tmpfs run /var/run`
 - `mount -t tmpfs shm /dev/shm`



Recap: block device vs. raw flash devices

- **Block devices**

- Allow for random data access using fixed size blocks
- Block size is small (minimum 512 bytes, can be increased)
- Considered as reliable (rely on the hardware and software support)

- **Raw flash devices**

- Allow for random data access, too
- Require special care before writing on the media (erasing the region that is about to write on)
- Erase, write and read operations might not use the same block size
- Reliability depends on the flash technology



NAND flash chips: how they work ?

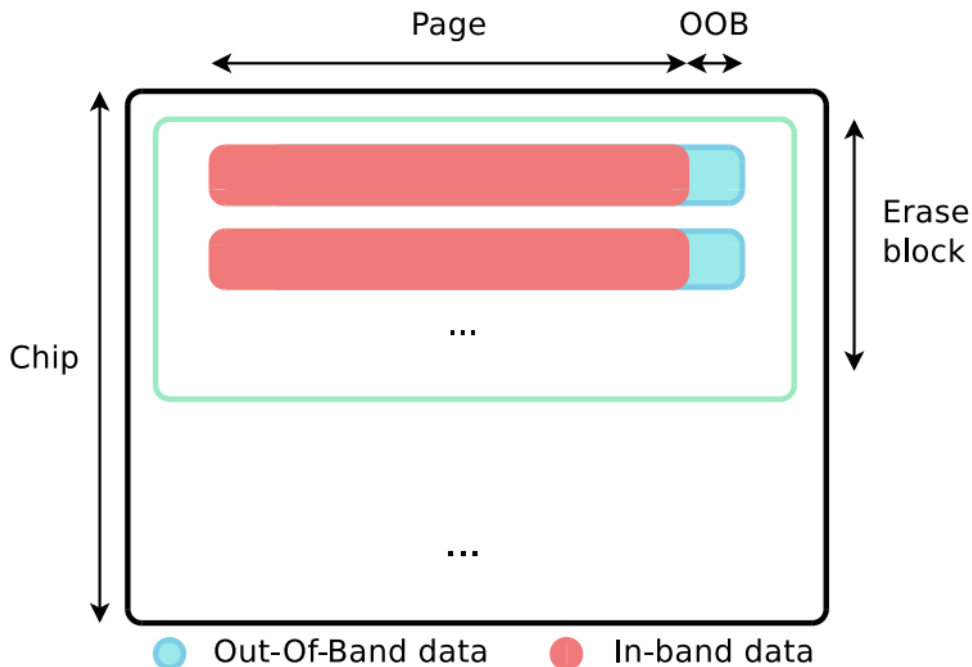
- **Encode bits with voltage levels**
 - **SLC (single level cell)** – 1 bit per memory cell
 - **MLC (multi level cell)** – multiple bits per cell
- **Start with all bits set to 1**
 - Writing implies changing some bits from 1 to 0 (assuming 1 bit per cell)
 - Restore bits to 1 is done via the ERASE operation
 - Writing and erasing are not done on a per bit or per byte basis
- **Organization**
 - **Page:** minimum unit for PROGRAM (write), example size: 4K
 - **Block:** minimum unit for ERASE, example size: 128 K



NAND flash storage: organization

- Microchip SAMA5D3 Xplained

- **Page size**
 - 2048 bytes
- **OOB size**
 - 64 bytes
- **Erase block size**
 - 131072 bytes





NAND flash storage: constraints

- **Reliability**

- Require mechanisms to recover from bit flips: ECC (Error Correcting Code)
- ECC information stored in the OOB (Out-of-band area)

- **Lifetime**

- Short lifetime compared to other storage media (between 1,000,000 and 1,000 erase cycles per block)
- Wear leveling mechanisms are required to erase blocks evenly
- Bad block detection/handling required, too



NAND flash: ECC

- **Error Correcting Code (ECC)**
 - Operates on chunks of usually 512 or 1024 bytes
 - ECC data are stored in the OOB area
- **Three algorithms**
 - Hamming: can fix up a single bit per chunk
 - Reed-Solomon: can fix up several bits per chunk
 - BCH: can fix up several bits per chunk



Memory Technology Devices (MTDs)

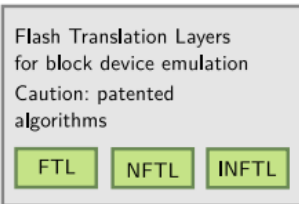
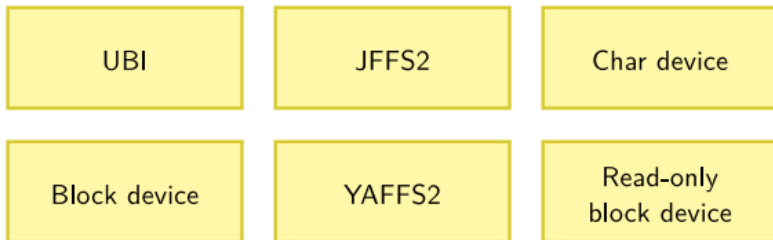
- **Generic subsystem in Linux**
 - **Dealing with all types of storage media that are not fitting in the block subsystem**
 - Support media: RAM, ROM, NOR flash, NAND flash, Dataflash
 - Abstract storage media characteristics and provide a simple API to access MTD devices
 - MTD device characteristics exposed to users
 - *erasesize*: minimum erase size unit
 - *writesize*: minimum write size unit
 - *obbsize*: extra size to store metadata or ECC data
 - *size*: device size
 - *flag*: information about device type and capabilities



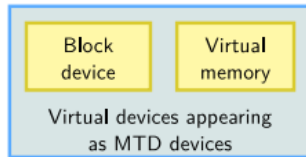
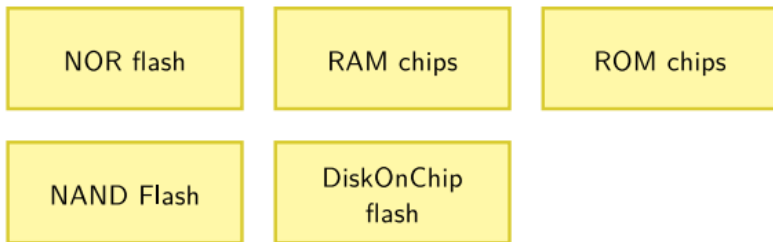
The MTD subsystem

Linux filesystem interface

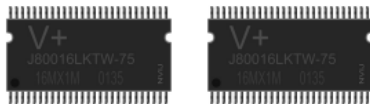
MTD "User"
modules



MTD Chip
drivers



Hardware
devices





Flash wear leveling

- Wear leveling
 - Distributing erases over the whole flash device to avoid quickly reaching the maximum number of erase cycles on blocks
 - The wear leveling implementation affects the life time of the flash memory
- Can be done in
 - The file system (JFFS2, YAFFS2)
 - An intermediate layer dedicated to wear leveling (UBI)



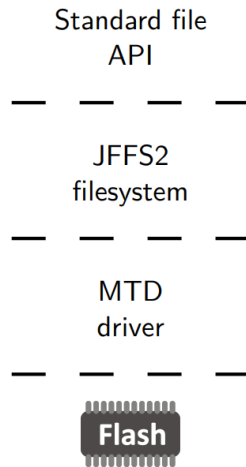
Flash file system: JFFS2

- **Flash file systems**

- Rely on the MTD layer to access flash chips
- Legacy flash file system: JFFS2, YAFFS2

- **Journaling flash file system version 2 (JFFS2)**

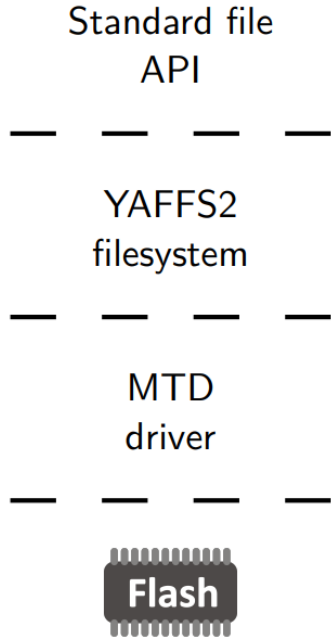
- Supports on-the-fly compression
- Wear leveling, power failure resistant
- Available in the official Linux kernel
- The large partitions affects the boot time
- <http://www.linux-mtd.infradead.org/doc/jffs2.html>





Flash file system: YAFFS2

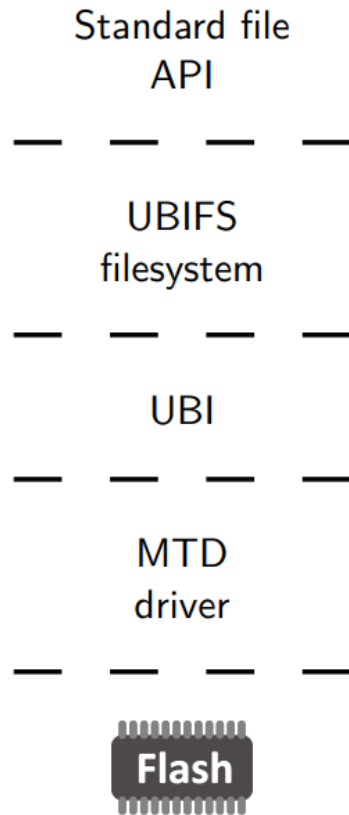
- Yet another flash file system version 2 (YAFFS2)
 - Mainly supports NAND flash
 - No compression
 - Wear leveling, power failure resistant
 - Fast boot time
 - Not part of the official Linux kernel
 - <https://yaffs.net/>





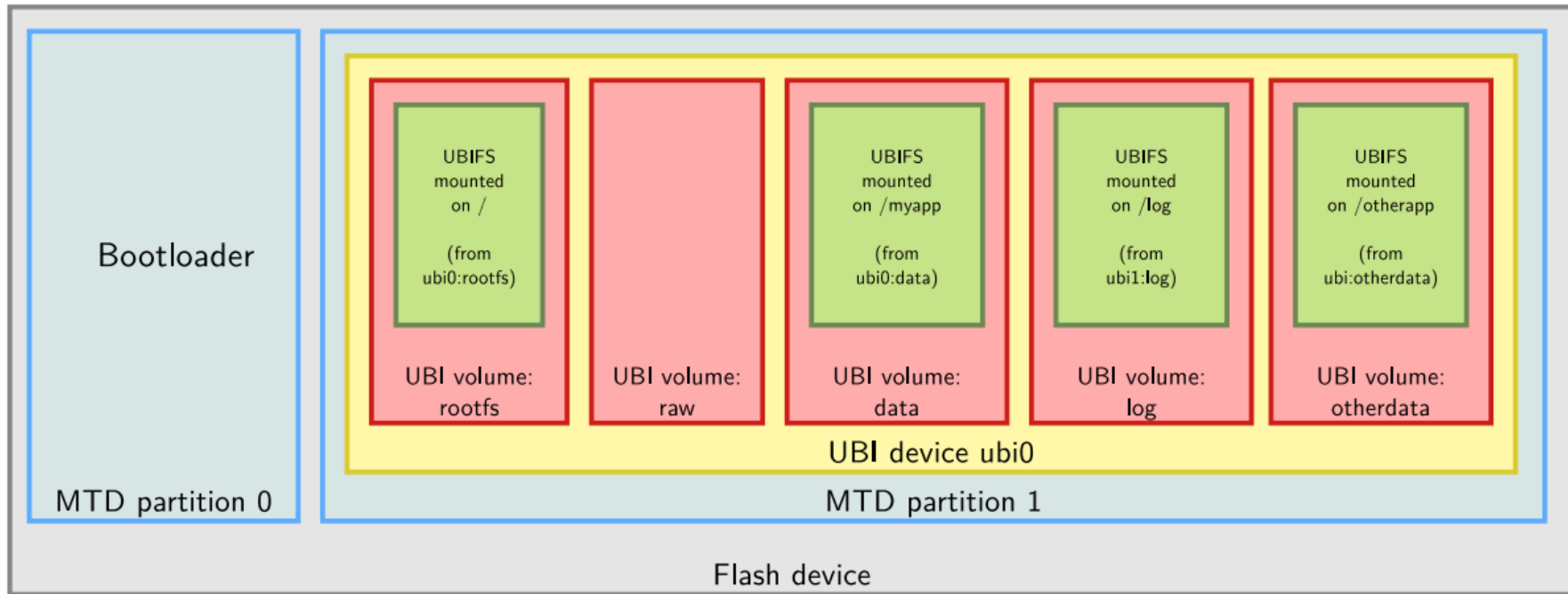
UBI/UBIFS

- Unsorted block images (UBI)
 - Aimed at replacing JFFS2 by addressing its limitations
 - Volume management system on top of MTD devices
 - Allows to create multiple logical volumes and spread writes across all physical blocks
 - Managing the erase blocks and wear leveling
- Drawback
 - Noticeable space overhead





UBI layout





Conclusion

- Journaling reduces recovery time
 - From $O(\text{size-of-the-disk-volume})$ to $O(\text{size-of-the-log})$
 - Speeding recovery substantially after a crash and restart
- The ordered metadata journaling
 - Reduce the amount of traffic to the journal while still preserving reasonable consistency guarantees for both file system metadata and user data
- Flash file systems
 - JAFFS2, YAFFS2, UBI/UBIFS