CE5045
Embedded System Design

Embedded Storage and Filesystems

https://github.com/tychen-NCU/EMBS-NCU

Instructor: Dr. Chen, Tseng-Yi

Computer Science & Information Engineering

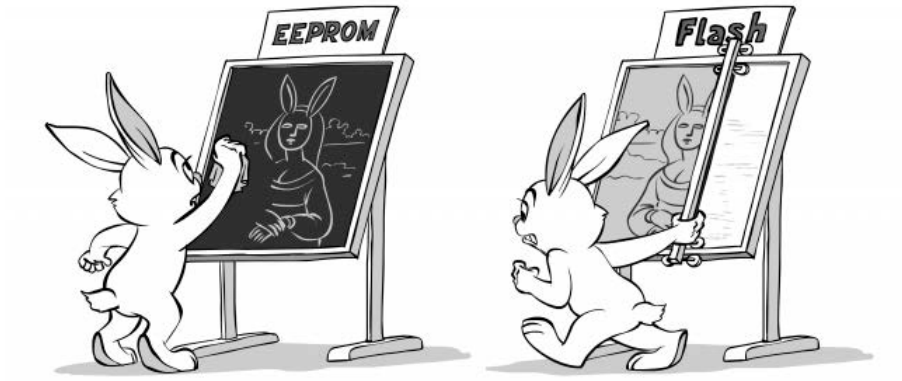
Outline

- What is Embedded Storage?
    - ✓ Flash Memory Introduction
    - ✓ NAND Flash Memory: How They Work?
    - ✓ Flash Organization
    - ✓ Some Issues with Flash Memory
    - ✓ Memory Technology Device: Architecture and Driver
- Embedded Filesystems Introduction
    - ✓ Journaling Flash Filesystem (JFFS2)
    - ✓ Unsorted Block Images: UBI
    - ✓ UBI File System
- Some Research Issues with Embedded Storage

Outline

What is Embedded Storage?

- ➢ The requirements for embedded storage devices
  - ✓ Fast read/write performance and low energy consumption.
- ➢ Traditional v.s. Modern embedded storage architecture
  - ✓ Traditional embedded storage: A ROM and a NVRAM (e.g., EEPROM)
  - ✓ Modern embedded storage: A ROM for booting codes and a flash memory
- ➢ Why flash memory?
  - ✓ Non-volatility: A floating gate
  - ✓ High density: Multi-level cells
  - ✓ Low cost: A large R/W/E unit



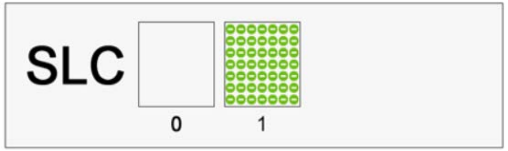
Block Devices v.s. Flash Devices

- Block devices
  - ✓ Allow for **random** data access using fixed size blocks
  - ✓ **Do not require** special care when writing on the media
  - ✓ Block size is **relatively small** (minimum 512 bytes, can be increased for performance reasons)
  - ✓ Considered as **reliable** (if the storage media is not, some hardware or software parts are supposed to make it reliable)
- Flash devices
  - ✓ Allow for **random** data access too
  - ✓ **Require** special care before writing on the media
  - ✓ Erase, write and read operation might **not** use **the same** block size
  - ✓ Reliability depends on the flash technology

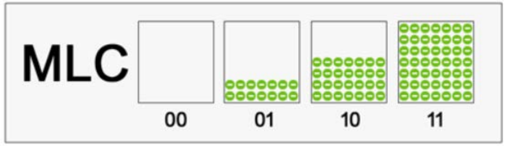
Flash Memory Technologies

- Single-level cell (SLC) NAND
  - ✓ SLC provides faster write speed, lower error rate and longer write endurance
    - Usually used in the applications require high reliability, performance, and viability
- Multi-level cell (MLC) NAND
  - ✓ MLC allows each memory cell to store multiple bits of information
    - Cost and reliability are relatively low
    - Usually used in the cell phone, digital cameras, USB drives and memory cards

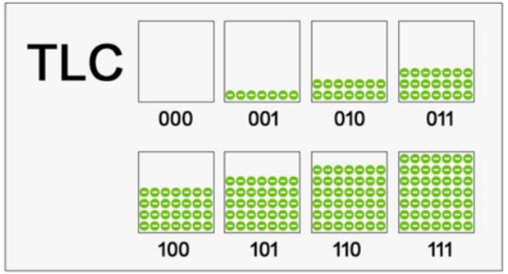
1-bit



2-bit

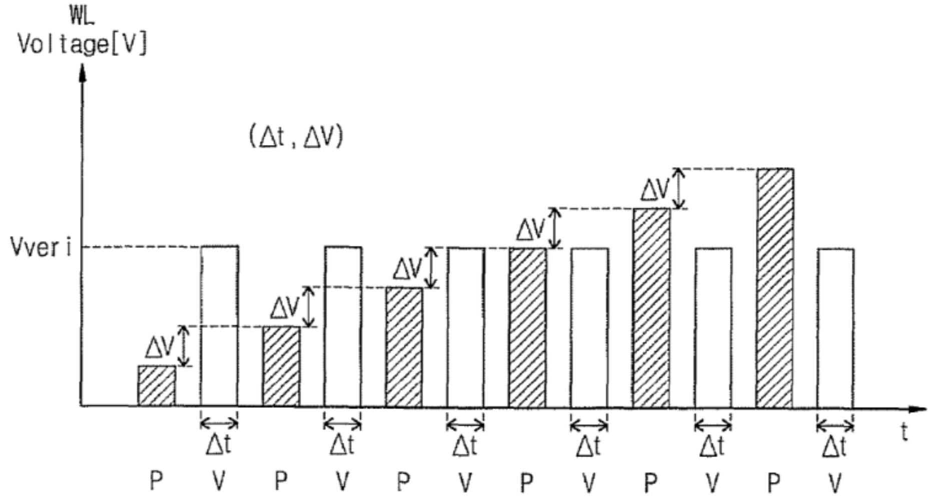


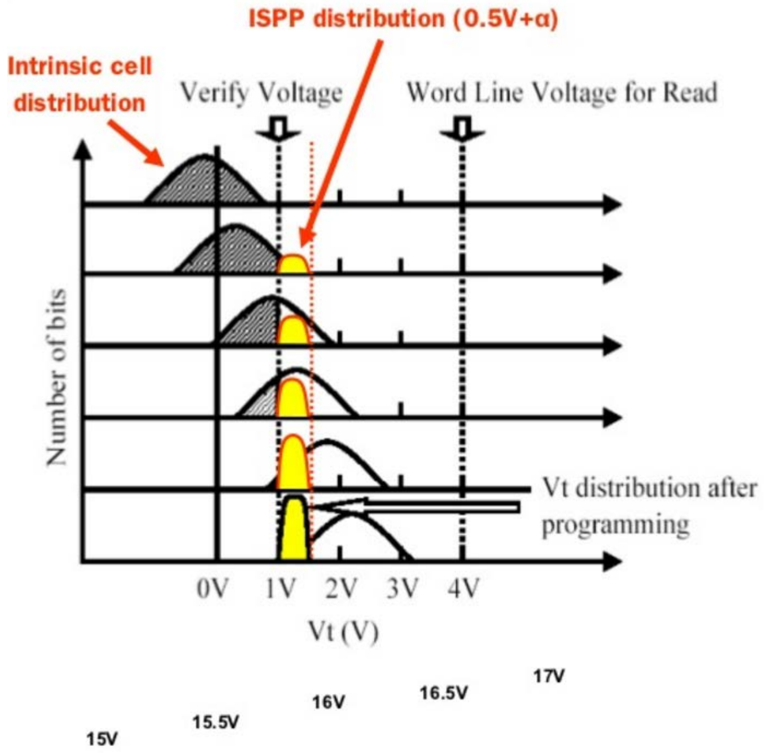
3-bit



NAND Flash Memory: How They Work?

- Encode bits with voltage levels

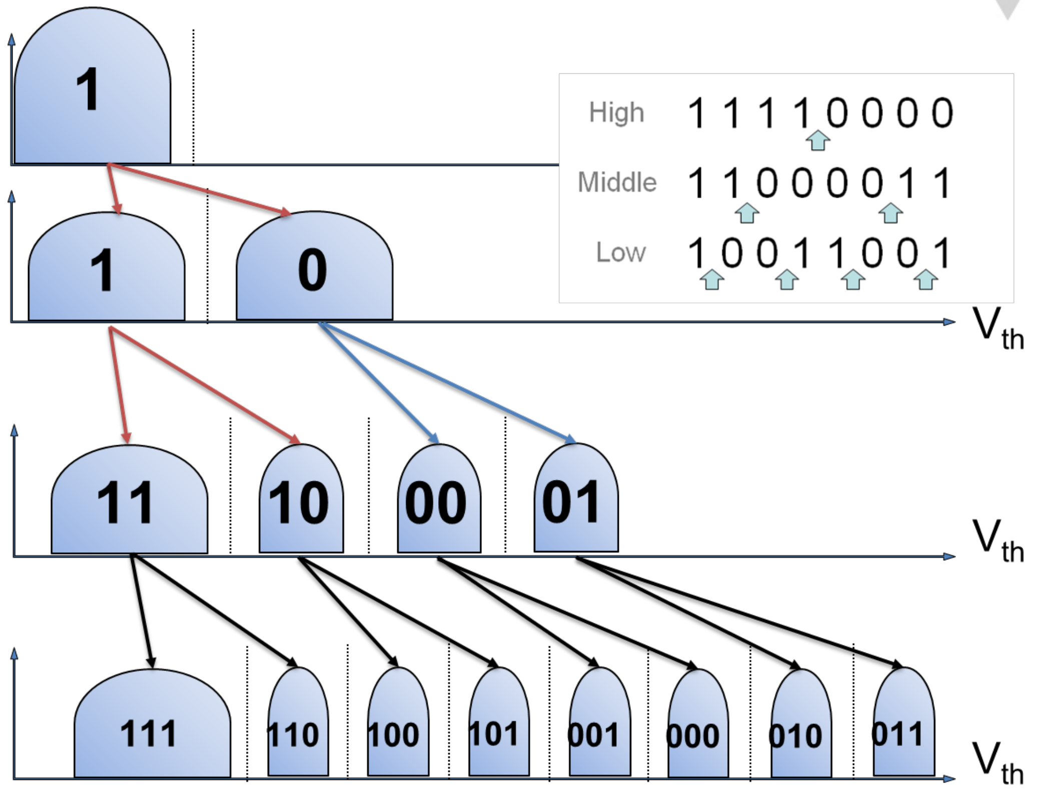




ISPP distribution (0.5V+α)
Intrinsic cell distribution
Verify Voltage
Word Line Voltage for Read
Number of bits
Vt distribution after programming
0V
1V
2V
3V
4V
Vt (V)
15V
15.5V
16V
16.5V
17V

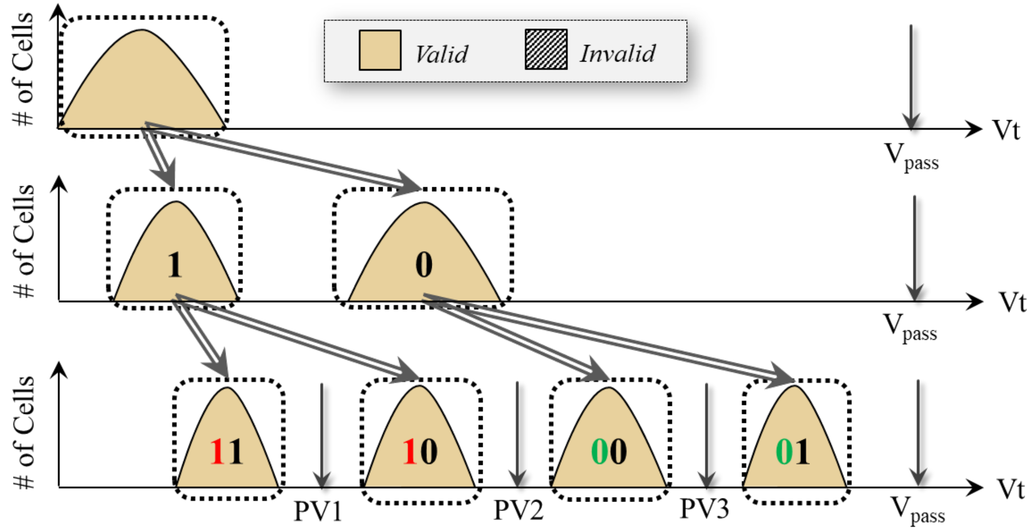
NAND Flash Memory: How They Work?

- Encode bits with voltage levels
- Start with all bits set to 1



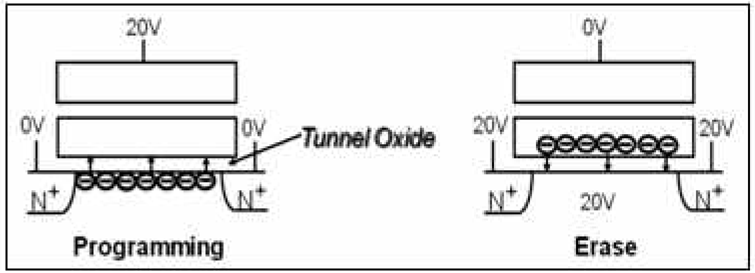
NAND Flash Memory: How They Work?

- Encode bits with voltage levels
- Start with all bits set to 1
- Programming implies changing some bits from 1 to 0



NAND Flash Memory: How They Work?

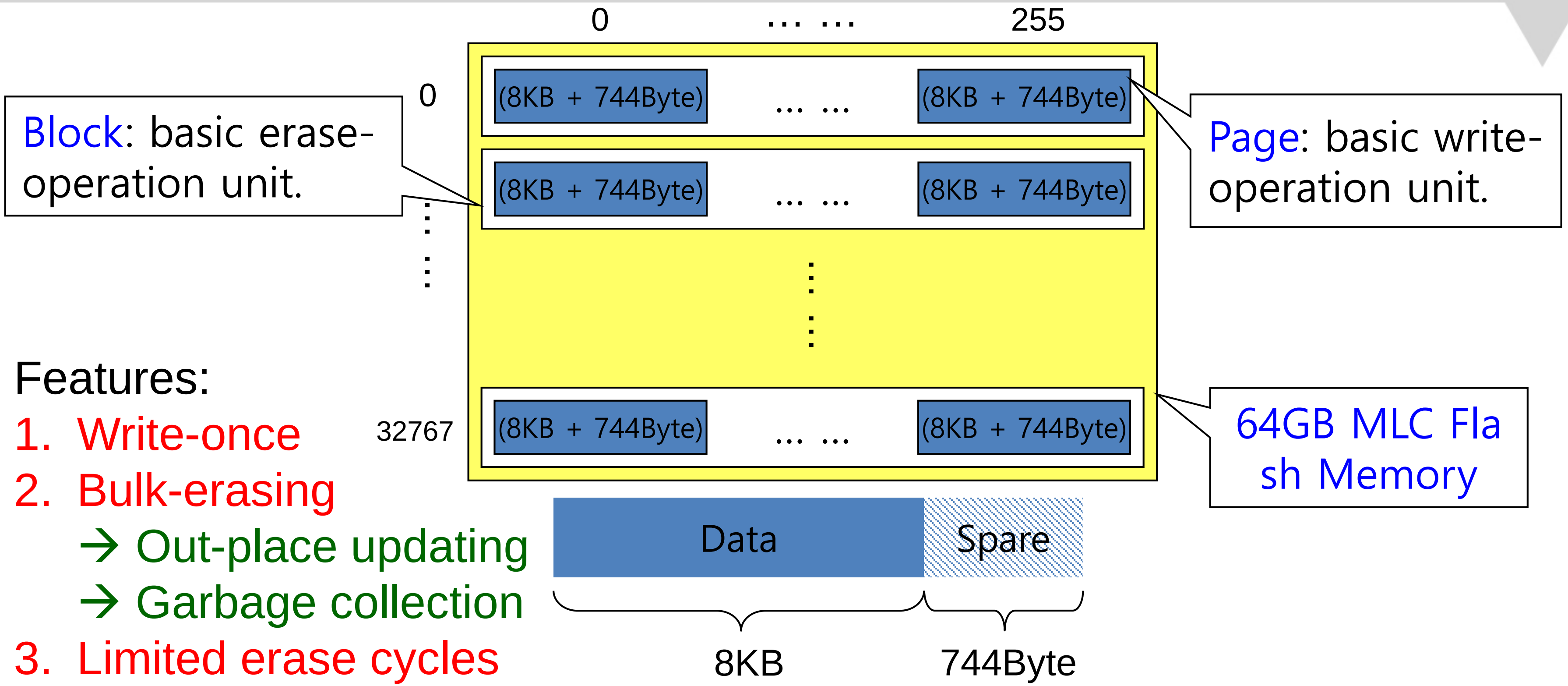
- Encode bits with voltage levels
- Start with all bits set to 1
- Programming implies changing some bits from 1 to 0
- Restoring bits to 1 is done via the ERASE operation



NAND Flash Memory: How They Work?

- ➢ Encode bits with voltage levels
- ➢ Start with all bits set to 1
- ➢ Programming implies changing some bits from 1 to 0
- ➢ Restoring bits to 1 is done via the ERASE operation
- ➢ Programming and erasing is not done on a per bit or per byte basis
    - ✓ **Page**: minimum unit for ‘<u>PROGRAM’</u> operation
    - ✓ **Block**: minimum unit for ‘ERASE’ operation

Flash Memory Organization

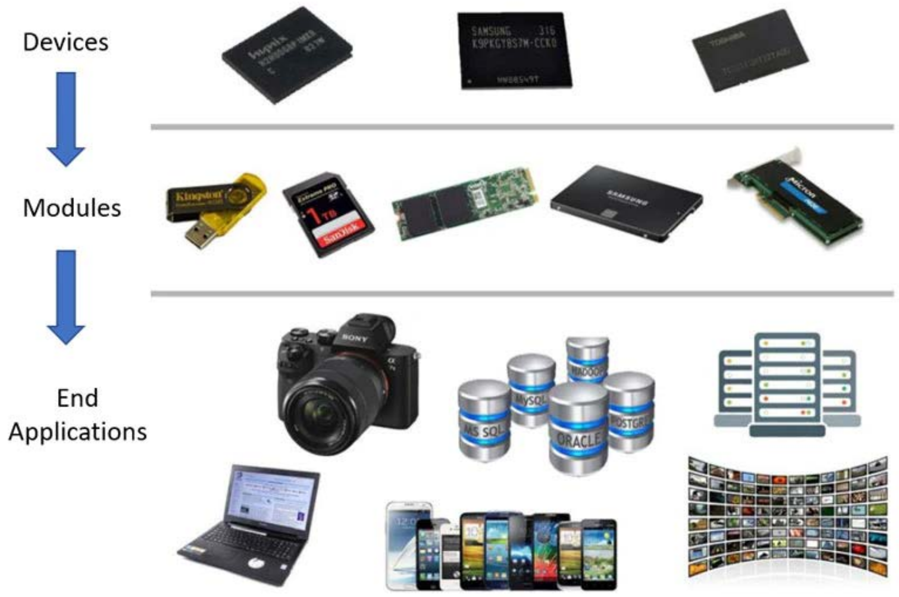


Features:

1. Write-once
2. Bulk-erasing
   → Out-place updating
   → Garbage collection
3. Limited erase cycles
   → Wear leveling

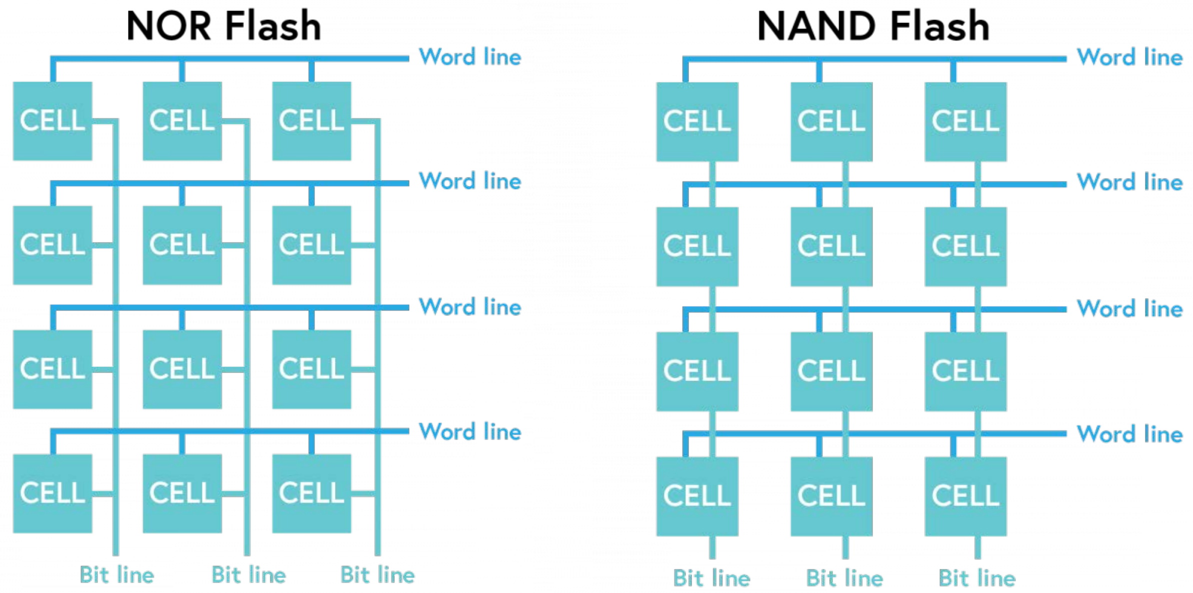
Some Issues with Flash Memory

- Flash Memory: Constraints
  - ✓ Reliability
  - ✓ Lifetime
  - ✓ Despite the number of constraints brought by NAND they are widely used in **embedded systems** for several reasons:
    - Cheaper than other flash technologies
    - Provide high capacity storage
    - Provide good performance (both in read and write access)



Flash Memory: Constraint I

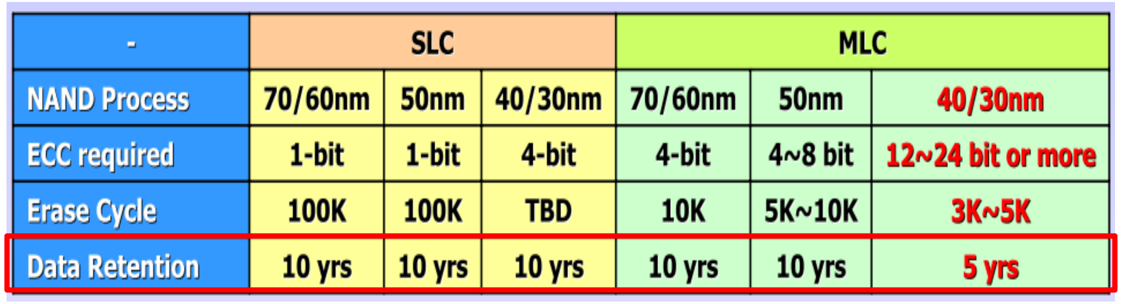
- ➢ Reliability
  - ✓ Far **less reliable** than NOR flash



Flash Memory: Constraint I

- Reliability
  - ✓ Far **less reliable** than NOR flash
  - ✓ Reliability depends on the **NAND flash technology** (SLC, MLC)

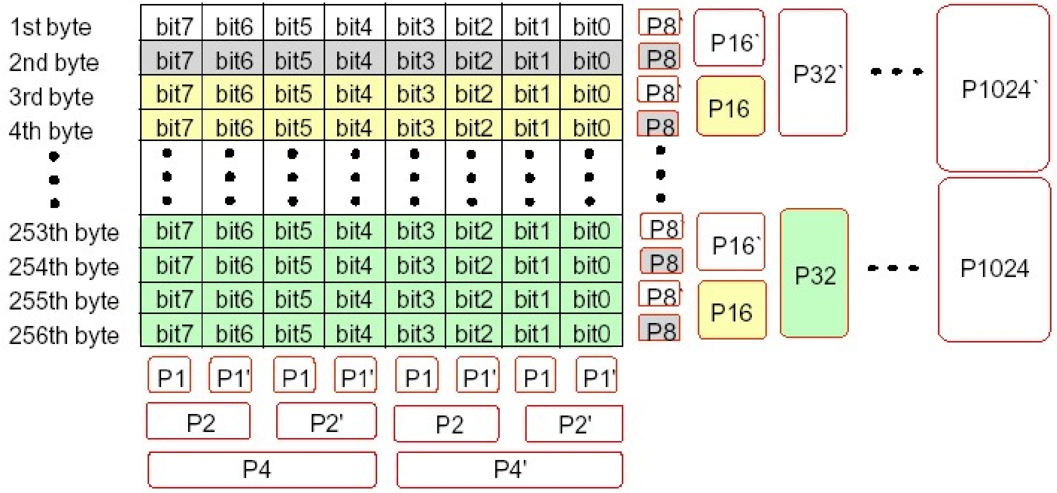
(Source: Hynix Marketing)



-	SLC			MLC		
NAND Process	70/60nm	50nm	40/30nm	70/60nm	50nm	40/30nm
ECC required	1-bit	1-bit	4-bit	4-bit	4~8 bit	12~24 bit or more
Erase Cycle	100K	100K	TBD	10K	5K~10K	3K~5K
Data Retention	10 yrs	10 yrs	10 yrs	10 yrs	10 yrs	5 yrs

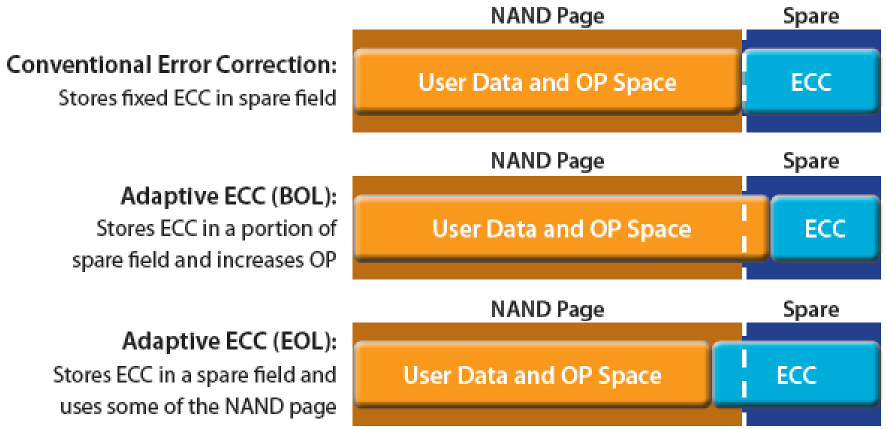
Flash Memory: Constraint I

- Reliability
  - ✓ Far **less reliable** than NOR flash
  - ✓ Reliability depends on the **NAND flash technology** (SLC, MLC)
  - ✓ Require additional mechanisms to recover from bit flips: **ECC** (Error Correcting Code)



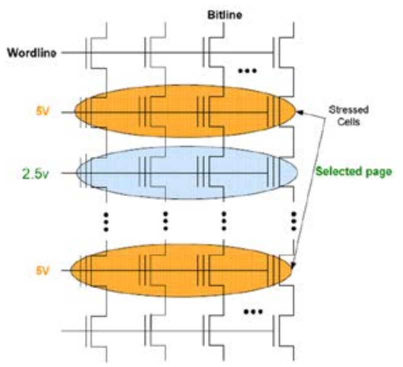
Flash Memory: Constraint I

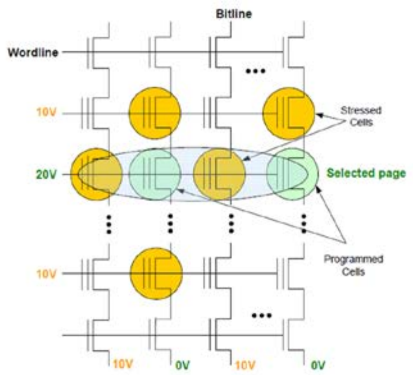
- Reliability
  - ✓ Far **less reliable** than NOR flash
  - ✓ Reliability depends on the **NAND flash technology** (SLC, MLC)
  - ✓ Require additional mechanisms to recover from bit flips: **ECC** (Error Correcting Code)
  - ✓ ECC information stored in the **OOB (Out-of-band area)**

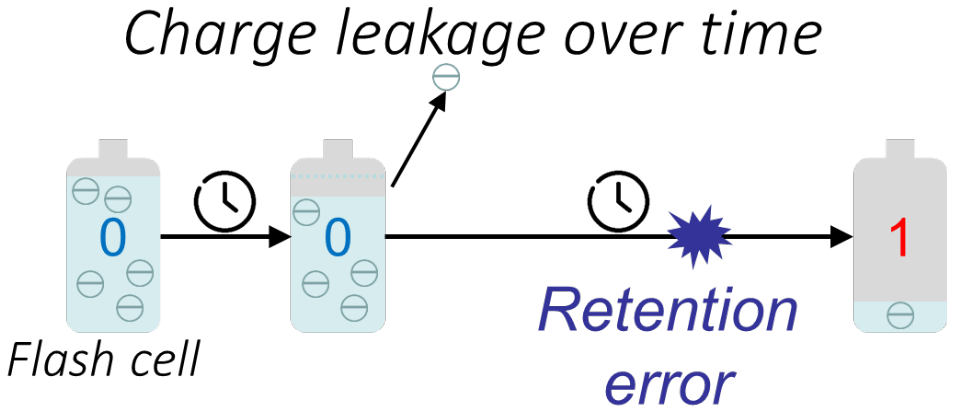

NAND Page
Spare
Conventional Error Correction:
Stores fixed ECC in spare field
User Data and OP Space
ECC
NAND Page
Spare
Adaptive ECC (BOL):
Stores ECC in a portion of spare field and increases OP
User Data and OP Space
ECC
NAND Page
Spare
Adaptive ECC (EOL):
Stores ECC in a spare field and uses some of the NAND page
User Data and OP Space
ECC

Flash Memory: Constraint I

- Reliability
  - ✓ Far **less reliable** than NOR flash
  - ✓ Reliability depends on the **NAND flash technology** (SLC, MLC)
  - ✓ Require additional mechanisms to recover from bit flips: **ECC** (Error Correcting Code)
  - ✓ ECC information stored in the **OOB (Out-of-band area)**
  - ✓ Disturbances and retention error

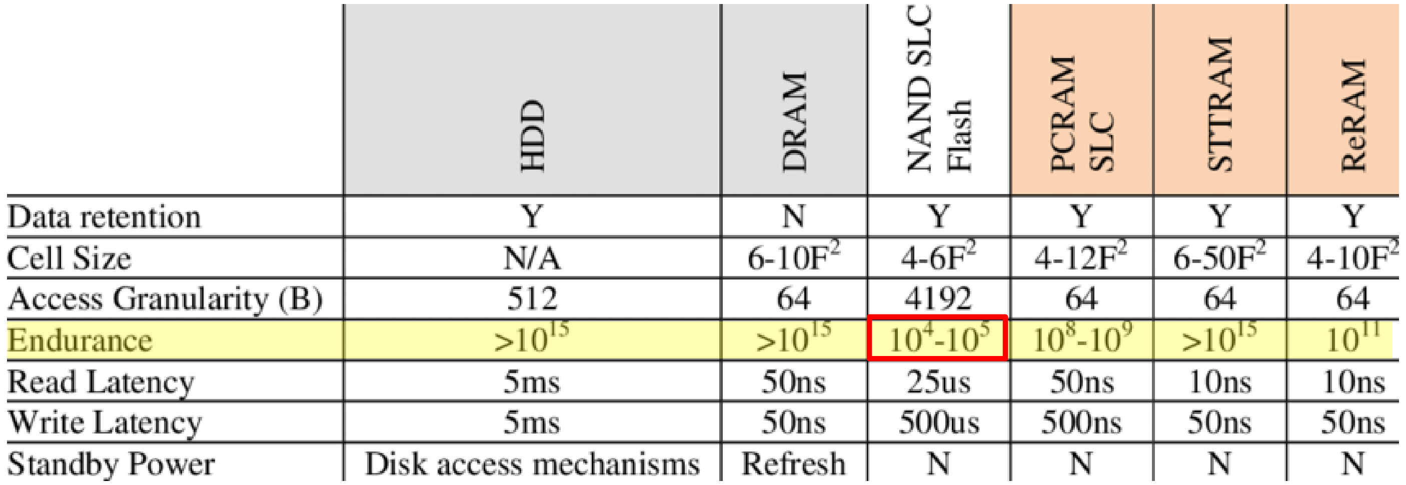






Flash Memory: Constraint II

- Lifetime
  - ✓ **Short lifetime** compared to other storage media

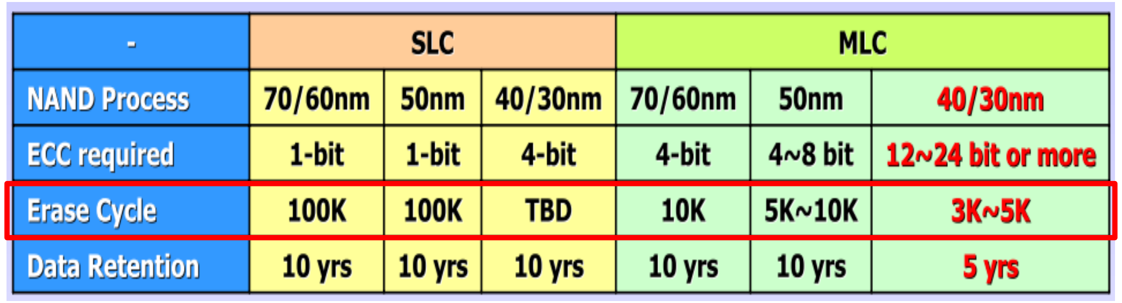


	HDD	DRAM	NAND SLC Flash	PCRAM SLC	STTRAM	ReRAM
Data retention	Y	N	Y	Y	Y	Y
Cell Size	N/A	6-10F^2	4-6F^2	4-12F^2	6-50F^2	4-10F^2
Access Granularity (B)	512	64	4192	64	64	64
Endurance	$>10^{15}$	$>10^{15}$	10^4-10^5	10^8-10^9	$>10^{15}$	10^{11}
Read Latency	5ms	50ns	25us	50ns	10ns	10ns
Write Latency	5ms	50ns	500us	500ns	50ns	50ns
Standby Power	Disk access mechanisms	Refresh	N	N	N	N

Flash Memory: Constraint II

- Lifetime
  - ✓ **Short lifetime** compared to other storage media
  - ✓ Lifetime depends on the **NAND flash technology** (SLC, MLC): between 1000000 and 1000 erase cycles per block

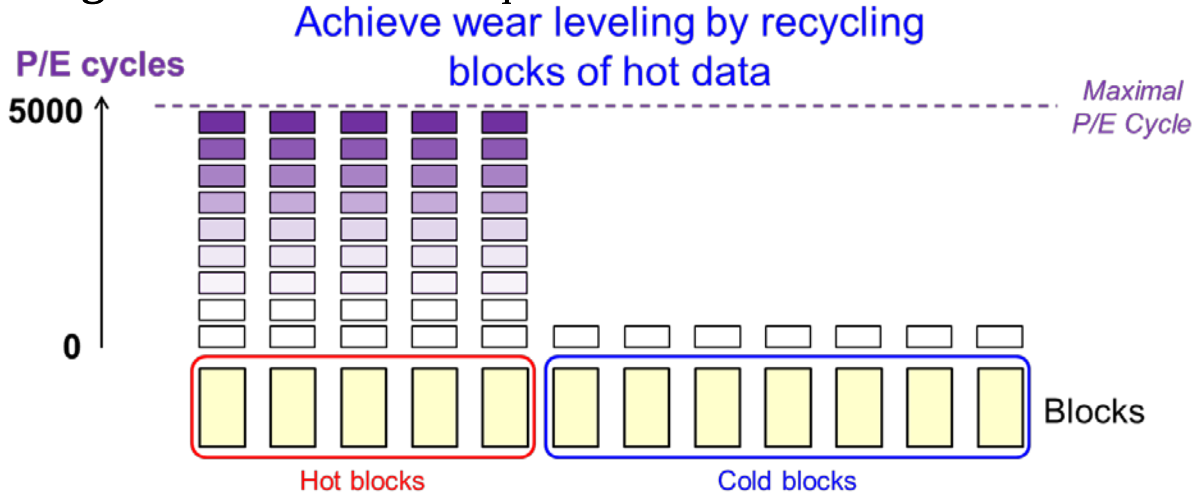
(Source: Hynix Marketing)



-	SLC			MLC		
NAND Process	70/60nm	50nm	40/30nm	70/60nm	50nm	40/30nm
ECC required	1-bit	1-bit	4-bit	4-bit	4~8 bit	12~24 bit or more
Erase Cycle	100K	100K	TBD	10K	5K~10K	3K~5K
Data Retention	10 yrs	10 yrs	10 yrs	10 yrs	10 yrs	5 yrs

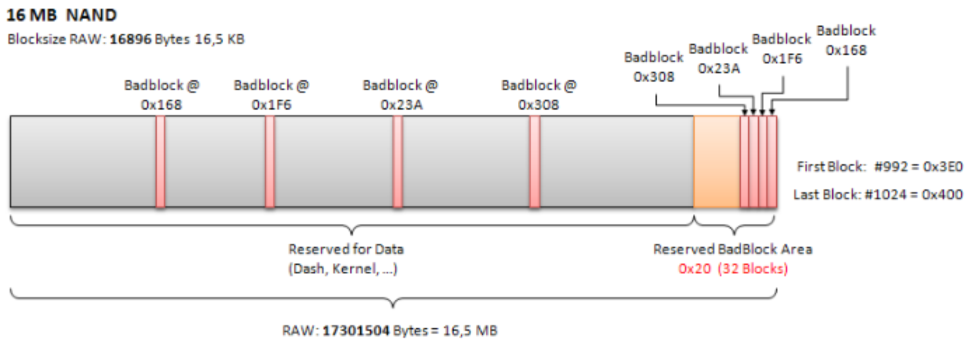
Flash Memory: Constraint II

- Lifetime
  - ✓ **Short lifetime** compared to other storage media
  - ✓ Lifetime depends on the **NAND flash technology** (SLC, MLC): between 1000000 and 1000 erase cycles per block
  - ✓ **Wear leveling mechanisms** are required



Flash Memory: Constraint II

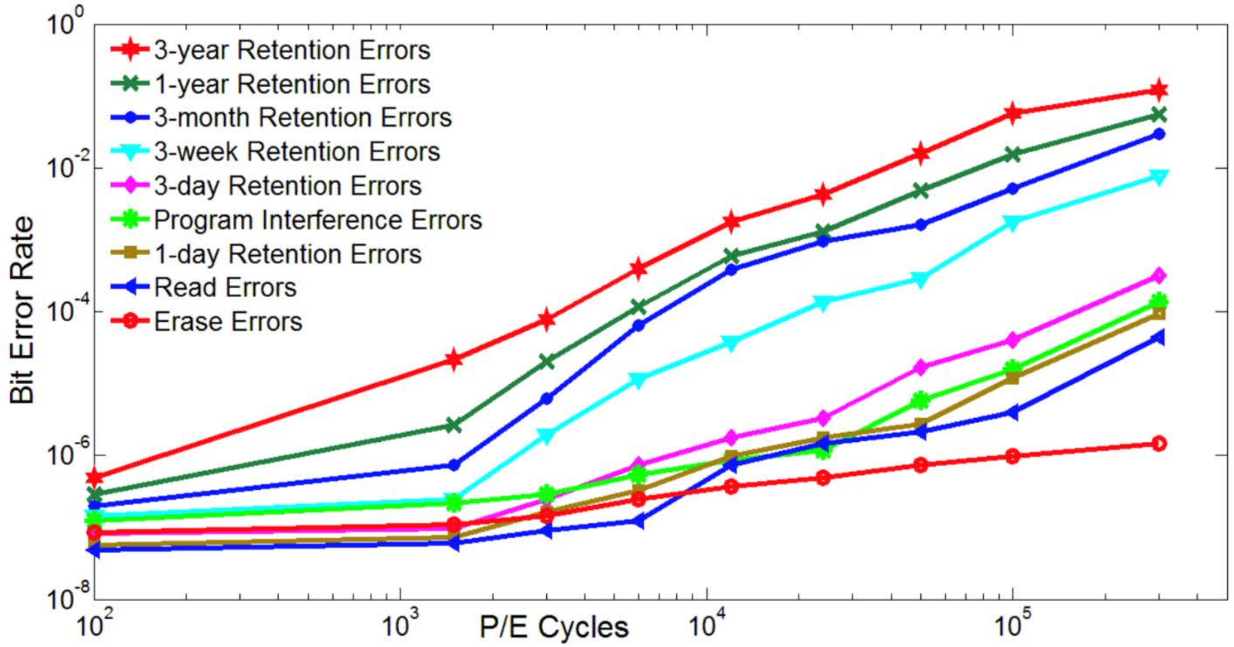
- Lifetime
  - ✓ **Short lifetime** compared to other storage media
  - ✓ Lifetime depends on the **NAND flash technology** (SLC, MLC): between 1000000 and 1000 erase cycles per block
  - ✓ **Wear leveling mechanisms** are required
  - ✓ **Bad block** detection/handling required too



16 MB NAND
Blocksize RAW: 16896 Bytes 16,5 KB
Badblock @ 0x168
Badblock @ 0x1F6
Badblock @ 0x23A
Badblock @ 0x308
Badblock 0x308
Badblock 0x23A
Badblock 0x1F6
Badblock 0x168
First Block: #992 = 0x3E0
Last Block: #1024 = 0x400
Reserved for Data (Dash, Kernel, ...)
Reserved BadBlock Area 0x20 (32 Blocks)
RAW: 17301504 Bytes = 16,5 MB

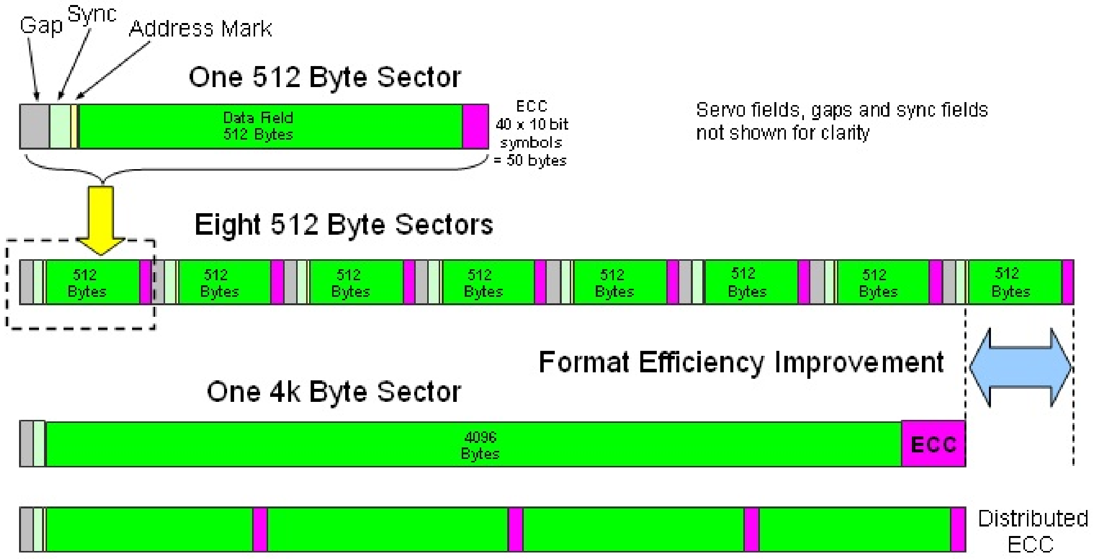
NAND Flash: ECC

- ➢ Error correction in flash memory
    - ✓ ECC partly addresses the **reliability problem** on NAND flash



NAND Flash: ECC

- Error correction in flash memory
  - ✓ ECC partly addresses the **reliability problem** on NAND flash
  - ✓ Operates on **blocks** (usually 512 or 1024 bytes)



NAND Flash: ECC

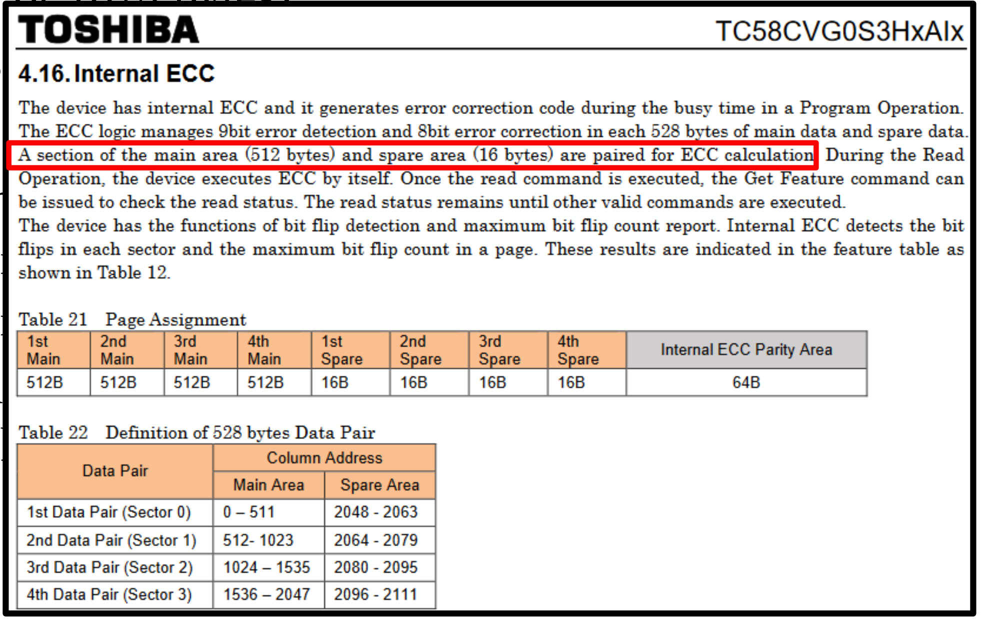
- Error correction in flash memory
  - ECC partly addresses the **reliability problem** on NAND flash
  - Operates on **blocks** (usually 512 or 1024 bytes)
  - ECC data are stored in the **OOB area**
  - Three common algorithms:
    - LDPC: The most popular coding method developed in 1962
    - Reed-Solomon: can fixup several bits per block
    - BCH: can fixup several bits per block

NAND Flash: ECC

- Error correction in flash memory
  - ✓ ECC partly addresses the **reliability problem** on NAND flash
  - ✓ Operates on **blocks** (usually 512 or 1024 bytes)
  - ✓ ECC data are stored in the **OOB area**
  - ✓ Three common algorithms:
    - LDPC: The most popular coding method developed in 1962
    - Reed-Solomon: can fixup several bits per block
    - BCH: can fixup several bits per block
  - ✓ NAND manufacturers specify the **required ECC strength** in their datasheets: ignoring these requirements might compromise data integrity

NAND Flash: ECC

- Error correction in flash memory
  - ✓ ECC partly addresses the **reliability problem** on NAND flash
  - ✓ Operates on **blocks** (usually 512 or 1024 bytes)
  - ✓ ECC data are stored in the **OOB**
  - ✓ Three common algorithms:
    - LDPC: The most popular codi
    - Reed-Solomon: can fixup seve
    - BCH: can fixup several bits pe
  - ✓ NAND manufacturers specify th
    ignoring these requirements mig



TOSHIBA TC58CVG0S3HxAIx

4.16. Internal ECC

The device has internal ECC and it generates error correction code during the busy time in a Program Operation. The ECC logic manages 9bit error detection and 8bit error correction in each 528 bytes of main data and spare data. A section of the main area (512 bytes) and spare area (16 bytes) are paired for ECC calculation. During the Read Operation, the device executes ECC by itself. Once the read command is executed, the Get Feature command can be issued to check the read status. The read status remains until other valid commands are executed.
The device has the functions of bit flip detection and maximum bit flip count report. Internal ECC detects the bit flips in each sector and the maximum bit flip count in a page. These results are indicated in the feature table as shown in Table 12.

Table 21 Page Assignment

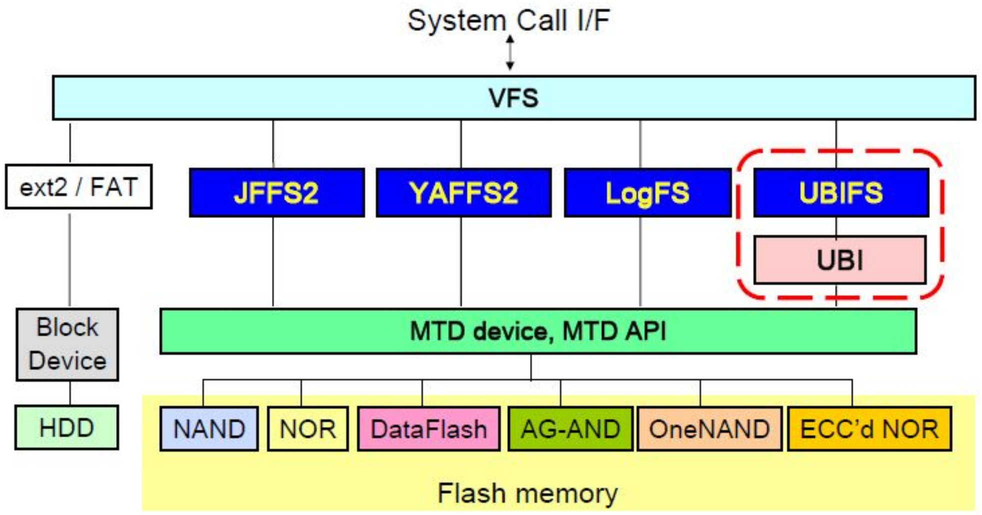
1st Main	2nd Main	3rd Main	4th Main	1st Spare	2nd Spare	3rd Spare	4th Spare	Internal ECC Parity Area
512B	512B	512B	512B	16B	16B	16B	16B	64B

Table 22 Definition of 528 bytes Data Pair

Data Pair	Column Address	
	Main Area	Spare Area
1st Data Pair (Sector 0)	0 – 511	2048 - 2063
2nd Data Pair (Sector 1)	512- 1023	2064 - 2079
3rd Data Pair (Sector 2)	1024 – 1535	2080 - 2095
4th Data Pair (Sector 3)	1536 – 2047	2096 - 2111

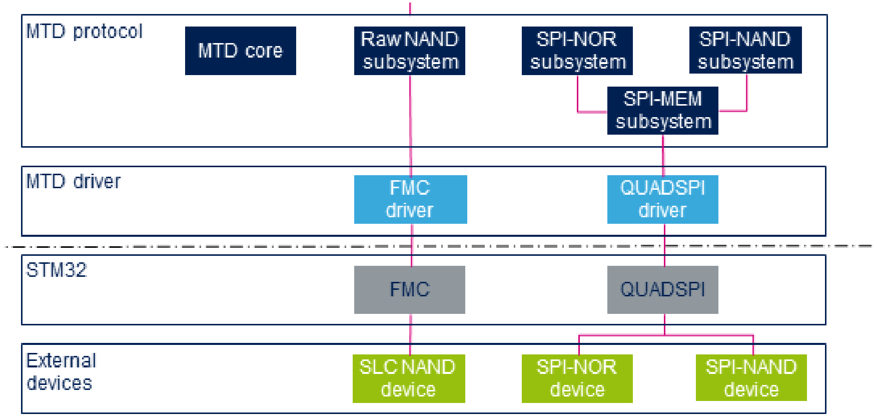
The MTD Subsystem

- What does MTD stand for?
  - ✓ MTD stands for <u>Memory Technology Devices</u>
  - ✓ Generic subsystem in Linux dealing with all types of storage media that are **not fitting in the block subsystem**



The MTD Subsystem

- What does MTD stand for?
  - ✓ MTD stands for Memory Technology Devices
  - ✓ Generic subsystem in Linux dealing with all types of storage media that are **not fitting in the block subsystem**
  - ✓ Supported media types: RAM, ROM, NOR flash, NAND flash, Dataflash
  - ✓ **Independent** of the communication interface (drivers available for parallel, SPI, direct memory mapping, ...)

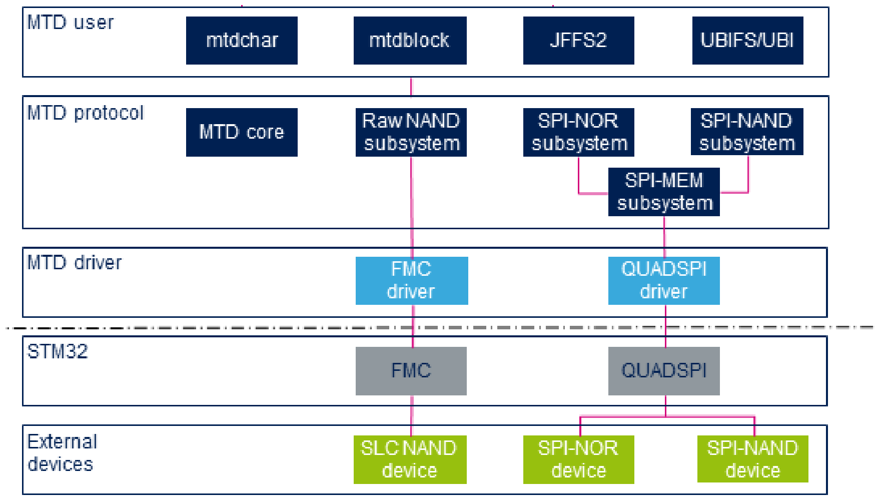


The MTD Subsystem

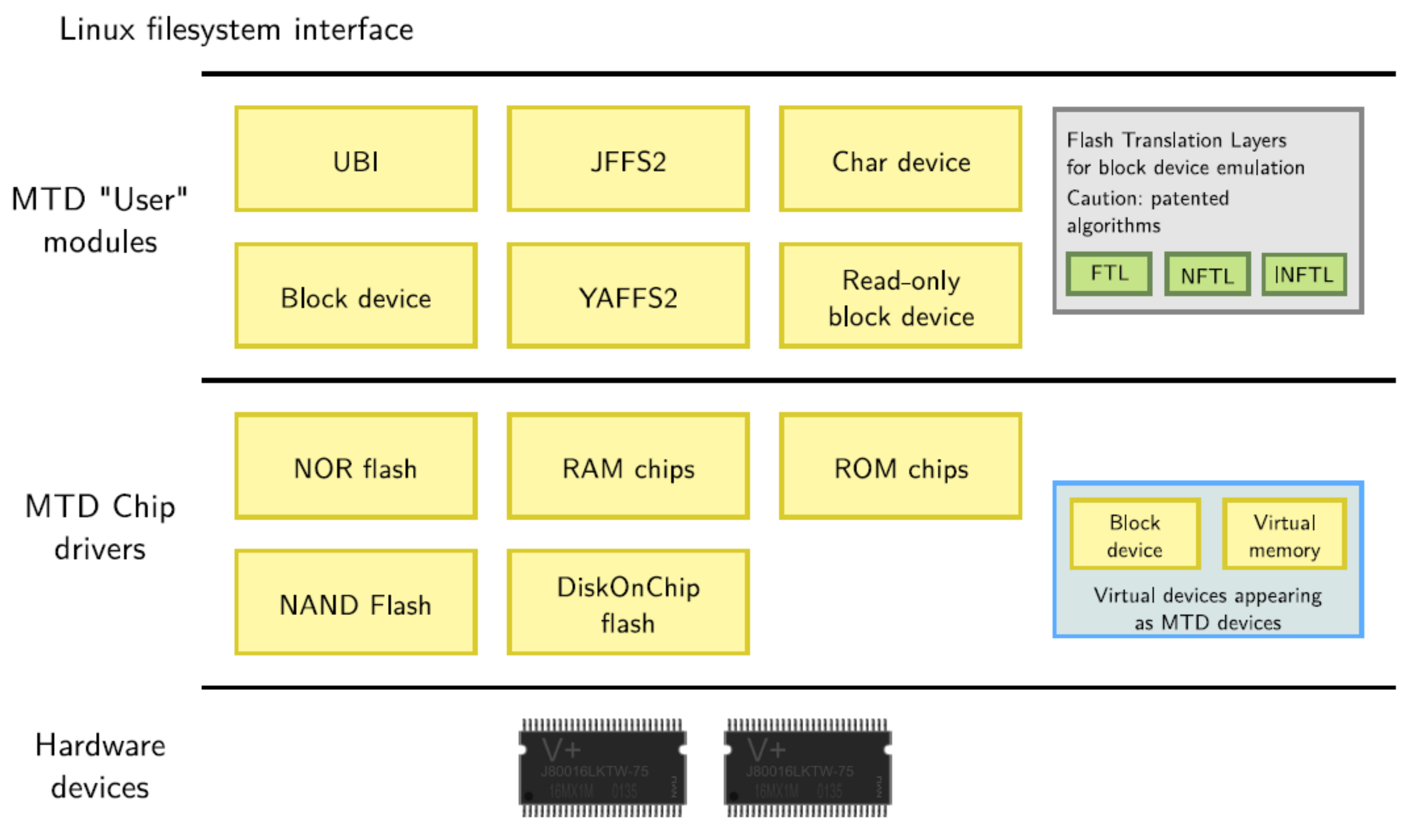
- What does MTD stand for?
  - ✓ **Abstract storage media characteristics** and provide a simple API to access MTD devices
  - ✓ MTD device characteristics exposed to users:
    - **`erasesize`**: minimum erase size unit
    - **`writesize`**: minimum write size unit
    - **`oobsize`**: extra size to store metadata or ECC data
    - **`size`**: device size
    - **`flags`**: information about device type and capabilities

The MTD Subsystem

- What does MTD stand for?
  - ✓ **Various kinds of MTD ”users”** in the kernel: file-systems, block device emulation layers, user space interfaces

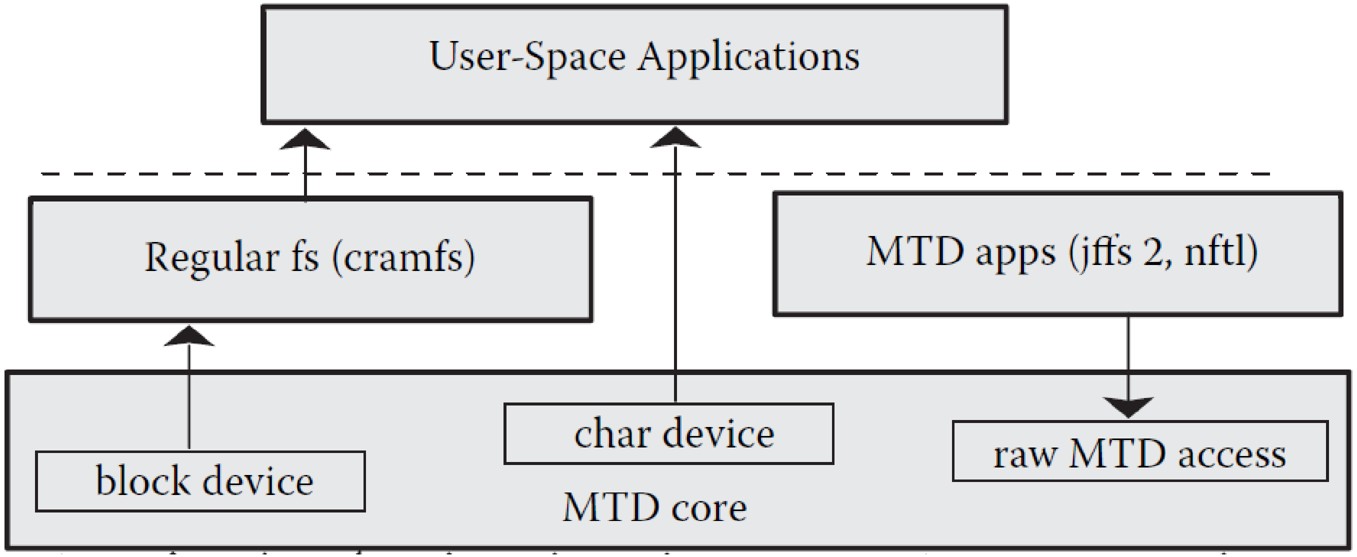


The Architecture of MTD Subsystem



MTD Architecture

- ➢ The MTD architecture is divided into the following components.
  - ✓ **MTD core**: This provides the interface between the low-level flash drivers and the applications. It implements the **character** and **block** device mode.

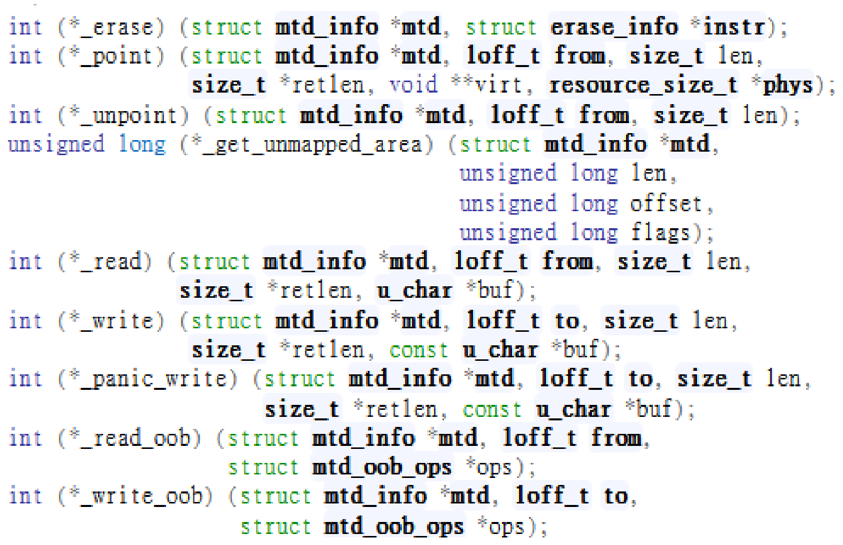


MTD Architecture

- The MTD architecture is divided into the following components.
    - ✓ **MTD core**: This provides the <u>interface</u> between the low-level flash drivers and the applications. It implements the **character** and **block** device mode.3
    - ✓ **Low-level flash drivers**: This section talks about NOR- and NAND-based flash chips only.
    - ✓ **BSP for flash**: A flash can be uniquely connected on a board. For example, a NOR flash can be <u>connected directly on the processor bus</u> or may be connected to an <u>external PCI bus</u>.
    - ✓ **MTD applications**: This can be either kernel submodules such as JFFS2 or NFTL, or user-space applications such as upgrade manager

MTD Operations

- **`mtd_info`** data structure
  - ✓ The heart of the MTD software
  - ✓ Defined in the file include/linux/mtd/mtd.h
  - ✓ With the pointers to all the required routines (such as `erase`, `read`, `write`, etc.)



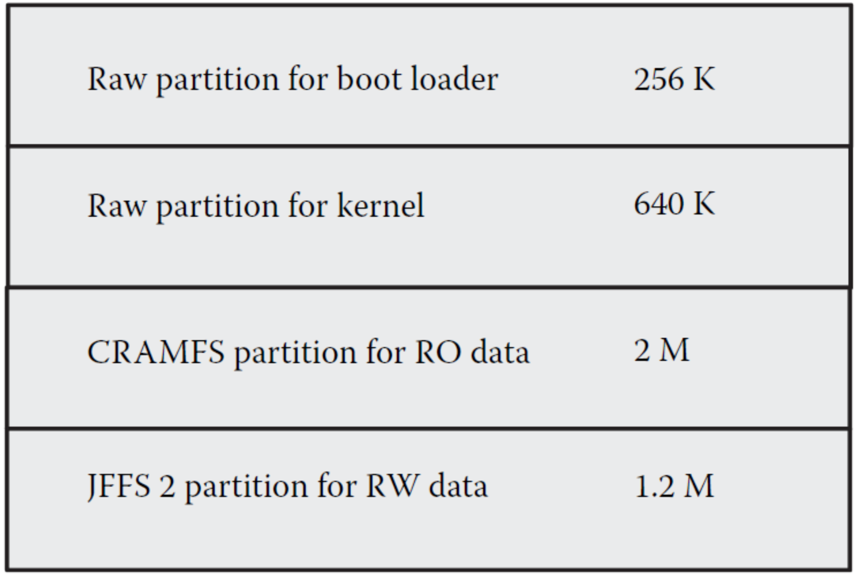
```
int (*_erase) (struct mtd_info *mtd, struct erase_info *instr);
int (*_point) (struct mtd_info *mtd, loff_t from, size_t len,
               size_t *retlen, void **virt, resource_size_t *phys);
int (*_unpoint) (struct mtd_info *mtd, loff_t from, size_t len);
unsigned long (*_get_unmapped_area) (struct mtd_info *mtd,
                                     unsigned long len,
                                     unsigned long offset,
                                     unsigned long flags);
int (*_read) (struct mtd_info *mtd, loff_t from, size_t len,
              size_t *retlen, u_char *buf);
int (*_write) (struct mtd_info *mtd, loff_t to, size_t len,
               size_t *retlen, const u_char *buf);
int (*_panic_write) (struct mtd_info *mtd, loff_t to, size_t len,
                     size_t *retlen, const u_char *buf);
int (*_read_oob) (struct mtd_info *mtd, loff_t from,
                  struct mtd_oob_ops *ops);
int (*_write_oob) (struct mtd_info *mtd, loff_t to,
                   struct mtd_oob_ops *ops);
```

MTD Operations

- **`mtd_info`** data structure
  - ✓ The heart of the MTD software
  - ✓ Defined in the file include/linux/mtd/mtd.h
  - ✓ With the pointers to all the required routines (such as `erase`, `read`, `write`, etc.)
- Interface between MTD core and raw flash drivers
  - ✓ Functions common to both the NAND and NOR flash chips
    - `read()/write()`
    - `erase()`
    - `lock()/unlock()`
    - `sync()`
    - `suspend()/resume()`
    - `read_ecc()/write_ecc()`(NAND chips only)
    - `read_oob()/write_oob()(`NAND chips only)

MTD Partitioning (I/II)

- MTD devices are usually partitioned
  - ✓ It allows to use different areas of the flash for different purposes: read-only filesystem, read-write filesystem, backup areas, bootloader area, kernel area, etc.

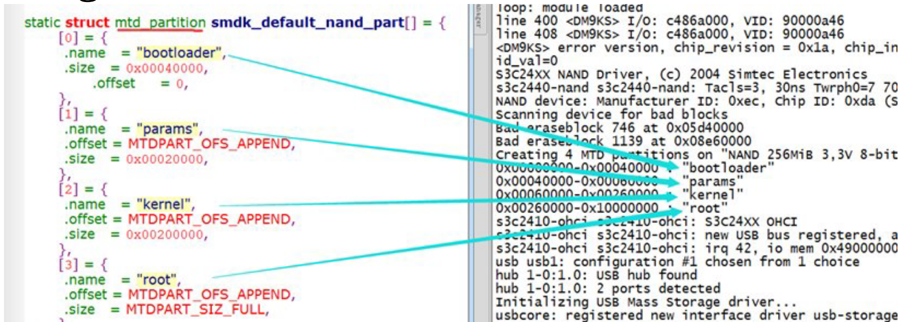


Raw partition for boot loader	256 K
Raw partition for kernel	640 K
CRAMFS partition for RO data	2 M
JFFS 2 partition for RW data	1.2 M

MTD Partitioning (I/II)

- MTD devices are usually partitioned
  - ✓ It allows to use different areas of the flash for different purposes: read-only filesystem, read-write filesystem, backup areas, bootloader area, kernel area, etc.

- The partitioning of MTD devices is **described externally**
  - ✓ Specified in the board device tree
  - ✓ Hard-coded into the kernel code (if no Device Tree)
  - ✓ Specified through the kernel command line

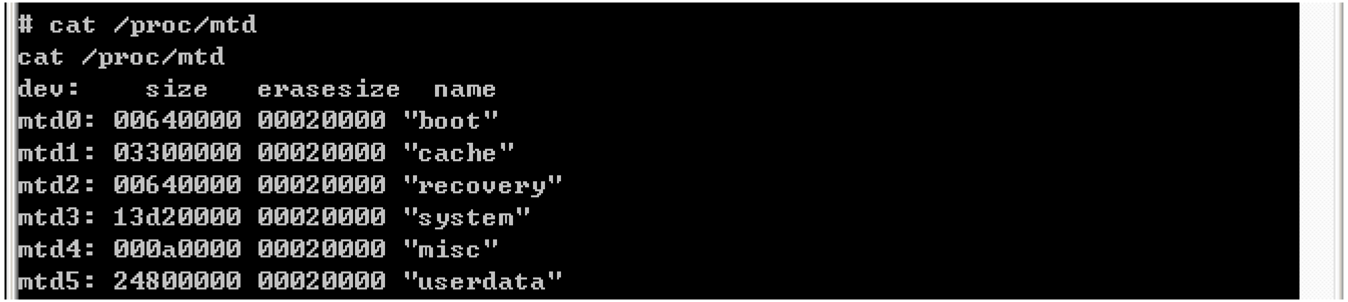


```
static struct mtd_partition smdk_default_nand_part[] = {
	[0] = {
	 .name	= "bootloader",
	 .size	= 0x00040000,
		.offset	= 0,
	},
	[1] = {
	 .name	= "params",
	 .offset = MTDPART_OFS_APPEND,
	 .size	= 0x00020000,
	},
	[2] = {
	 .name	= "kernel",
	 .offset = MTDPART_OFS_APPEND,
	 .size	= 0x00200000,
	},
	[3] = {
	 .name	= "root",
	 .offset = MTDPART_OFS_APPEND,
	 .size	= MTDPART_SIZ_FULL,
```

```
loop: module loaded
line 400 <DM9KS> I/O: c486a000, VID: 90000a46
line 408 <DM9KS> I/O: c486a000, VID: 90000a46
<DM9KS> error version, chip_revision = 0x1a, chip_in
id_val=0
S3C24XX NAND Driver, (c) 2004 Simtec Electronics
s3c2440-nand s3c2440-nand: Tacls=3, 30ns Twrph0=7 70
NAND device: Manufacturer ID: 0xec, Chip ID: 0xda (S
Scanning device for bad blocks
Bad eraseblock 746 at 0x05d40000
Bad eraseblock 1139 at 0x08e60000
Creating 4 MTD partitions on "NAND 256MiB 3,3V 8-bit
0x00000000-0x00040000 : "bootloader"
0x00040000-0x00060000 : "params"
0x00060000-0x00260000 : "kernel"
0x00260000-0x10000000 : "root"
s3c2410-ohci s3c2410-ohci: S3C24XX OHCI
s3c2410-ohci s3c2410-ohci: new USB bus registered, a
s3c2410-ohci s3c2410-ohci: irq 42, io mem 0x49000000
usb usb1: configuration #1 chosen from 1 choice
hub 1-0:1.0: USB hub found
hub 1-0:1.0: 2 ports detected
Initializing USB Mass Storage driver...
usbcore: registered new interface driver usb-storage
```

MTD Partitioning (II/II)

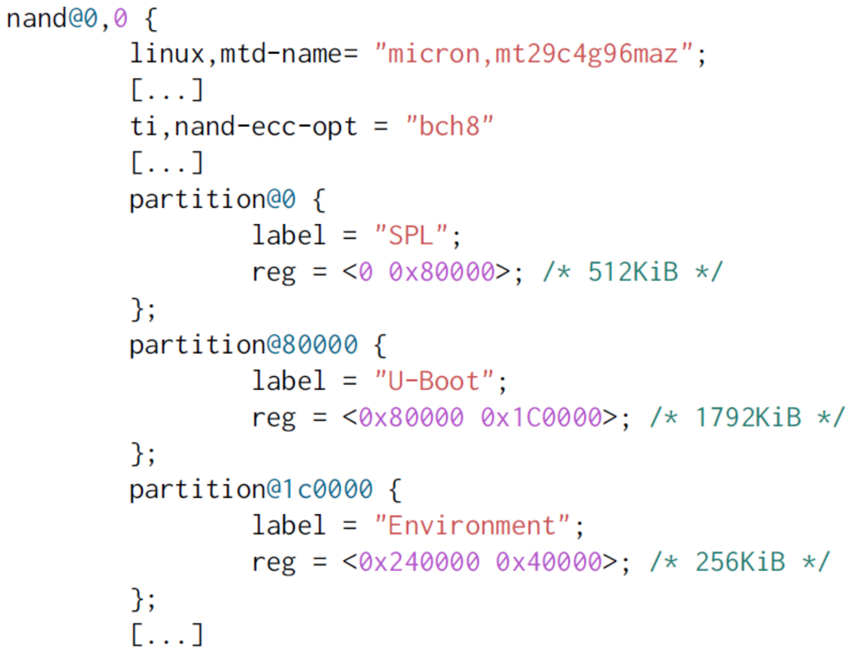
- Each partition becomes a separate MTD device
  - ✓ Different from block device labeling (`hda3`, `sda2`)
  - ✓ `/dev/mtd1` is **either** the second partition of the first flash device, or the first partition of the second flash device



```
# cat /proc/mtd
cat /proc/mtd
dev:    size   erasesize  name
mtd0: 00640000 00020000 "boot"
mtd1: 03300000 00020000 "cache"
mtd2: 00640000 00020000 "recovery"
mtd3: 13d20000 00020000 "system"
mtd4: 000a0000 00020000 "misc"
mtd5: 24800000 00020000 "userdata"
```

MTD Partition: Device Tree

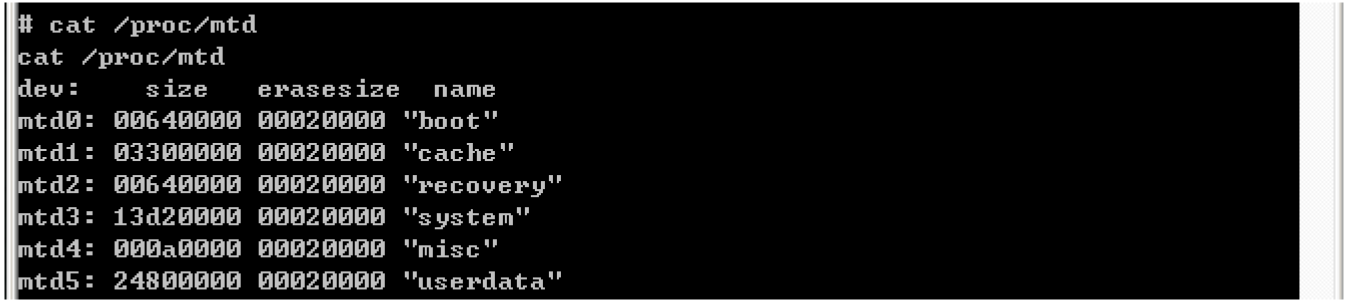
- The Device Tree is the **standard place** to define default MTD partitions for platforms with Device Tree support.
  - ✓ Example from `arch/arm/boot/dts/omap3-overo-base.dtsi:`



```
nand@0,0 {
        linux,mtd-name= "micron,mt29c4g96maz";
        [...]
        ti,nand-ecc-opt = "bch8"
        [...]
        partition@0 {
                label = "SPL";
                reg = <0 0x80000>; /* 512KiB */
        };
        partition@80000 {
                label = "U-Boot";
                reg = <0x80000 0x1C0000>; /* 1792KiB */
        };
        partition@1c0000 {
                label = "Environment";
                reg = <0x240000 0x40000>; /* 256KiB */
        };
        [...]
```

MTD Interface with User Space

- MTD devices are visible in `/proc/mtd`
- User space only sees **MTD partitions**, not the flash device under those partitions



```
# cat /proc/mtd
cat /proc/mtd
dev:    size   erasesize  name
mtd0: 00640000 00020000 "boot"
mtd1: 03300000 00020000 "cache"
mtd2: 00640000 00020000 "recovery"
mtd3: 13d20000 00020000 "system"
mtd4: 000a0000 00020000 "misc"
mtd5: 24800000 00020000 "userdata"
```

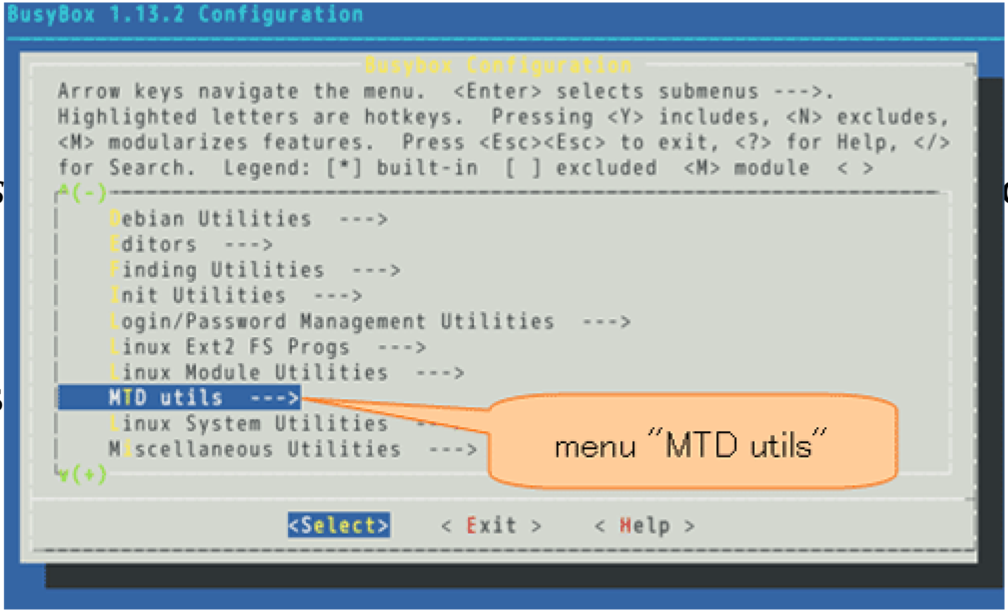
- The **mtdchar** driver creates a character device for each MTD device/partition of the system
  - ✓ Provide `ioctl()` to **erase** and **manage** the flash
  - ✓ Used by the *mtd-utils* utilities

User Space Flash Management Tools

- *mtd-utils* is a set of utilities to manipulate MTD devices
  - ✓ *mtdinfo* to get detailed information about an MTD device
  - ✓ *flash_erase* to partially or completely erase a given MTD device
  - ✓ *flashcp* to write to NOR flash
  - ✓ *nandwrite* to write to NAND flash
  - ✓ Flash filesystem image creation tools: **mkfs.jffs2**, **mkfs.ubifs**, **ubinize**, etc.

User Space Flash Management Tools

- *mtd-utils* is a set
  - ✓ *mtdinfo* to
  - ✓ *flash_eras* ce
  - ✓ *flashcp* to
  - ✓ *nandwrite*
  - ✓ Flash files **ubinize**, etc.



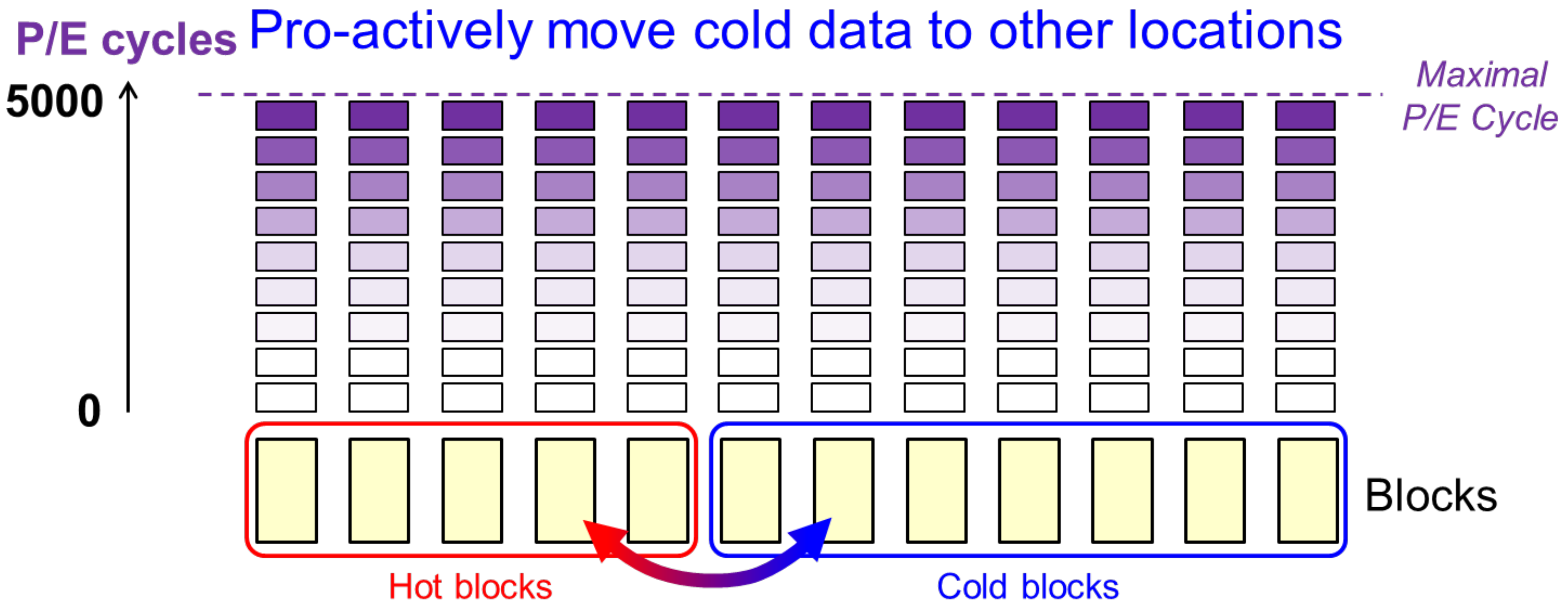
```
BusyBox 1.13.2 Configuration
                         Busybox Configuration
  Arrow keys navigate the menu.  <Enter> selects submenus --->.
  Highlighted letters are hotkeys.  Pressing <Y> includes, <N> excludes,
  <M> modularizes features.  Press <Esc><Esc> to exit, <?> for Help, </>
  for Search.  Legend: [*] built-in  [ ] excluded  <M> module  < >
 ^(-)
        Debian Utilities  --->
        Editors  --->
        Finding Utilities  --->
        Init Utilities  --->
        Login/Password Management Utilities  --->
        Linux Ext2 FS Progs  --->
        Linux Module Utilities  --->
        MTD utils  --->
        Linux System Utilities  --->
        Miscellaneous Utilities  --->
 v(+)

              <Select>    < Exit >    < Help >
```

- On your workstation: usually available as the *mtd-utils package* in your distribution
- On your embedded target: most commands now also available in **BusyBox**

Flash Wear-leveling (I/II)

- Wear leveling consists in **<u>distributing erases over the whole flash device </u>**to avoid quickly reaching the maximum number of erase cycles on blocks that are written really often



Flash Wear-leveling (I/II)

- Wear leveling consists in **distributing erases over the whole flash device** to avoid quickly reaching the maximum number of erase cycles on blocks that are written really often

- Can be done in
    - ✓ The filesystem layer (JFFS2, YAFFS2, ...)
    - ✓ An intermediate layer dedicated to wear leveling (UBI)

- The wear leveling implementation is what makes your flash **lifetime** good or not

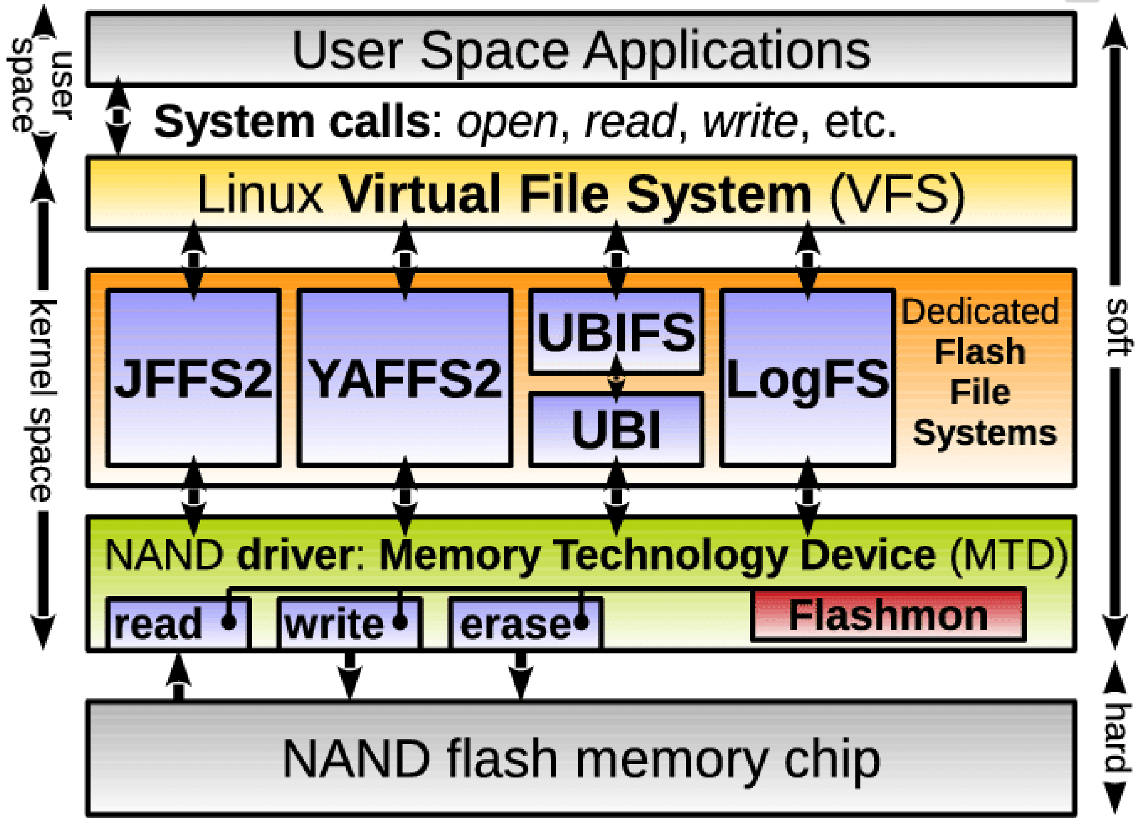
Flash Wear-leveling (II/II)

- Flash users should also take the **limited lifetime** of flash devices into account by taking additional precautions
  - ✓ **Do not** use your flash storage as swap area (rare in embedded systems anyway)
  - ✓ Mount your filesystems as **read-only**, or use read-only filesystems (SquashFS), whenever possible
  - ✓ Keep volatile files in RAM (tmpfs)
  - ✓ Don't use the **`sync`** mount option (commits writes immediately). Use the **`fsync`**() system call for per-file synchronization.

Embedded File Systems

- Specific file systems have been developed to deal flash constraints

- These file systems are relying on the MTD layer to access flash chips

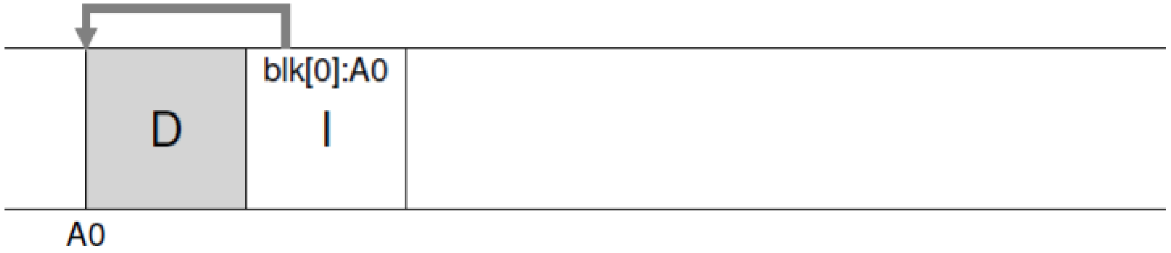

user space
kernel space
soft
hard
User Space Applications
System calls: open, read, write, etc.
Linux Virtual File System (VFS)
JFFS2
YAFFS2
UBIFS
UBI
LogFS
Dedicated Flash File Systems
NAND driver: Memory Technology Device (MTD)
read
write
erase
Flashmon
NAND flash memory chip

Legacy Flash Filesystems: JFFS2

- What is JFFS2?
    - ✓ A **<u>log-structured file system</u>** designed for use on flash devices in embedded systems.
- Features
    - ✓ Supports on the fly compression
    - ✓ Available in the official Linux kernel
    - ✓ Power down no crash
    - ✓ If full slow down performance

Log-structured File System (LFS)

- Log-structured File System (LFS)
  - ✓ **Writes everything** (including data blocks and inodes, etc.) to the disk **sequentially**
  - ✓ E.g., Writing a data block **D** and updated inode **I** to the disk.

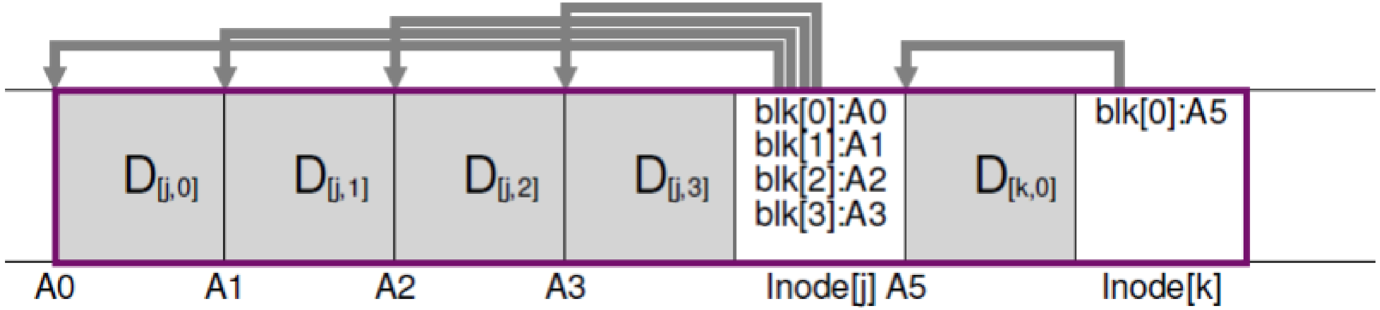


- *Note: in most systems, data blocks are 4 KB in size, whereas an inode is much smaller (e.g., 128 B).*

Writing Sequentially, and Effectively

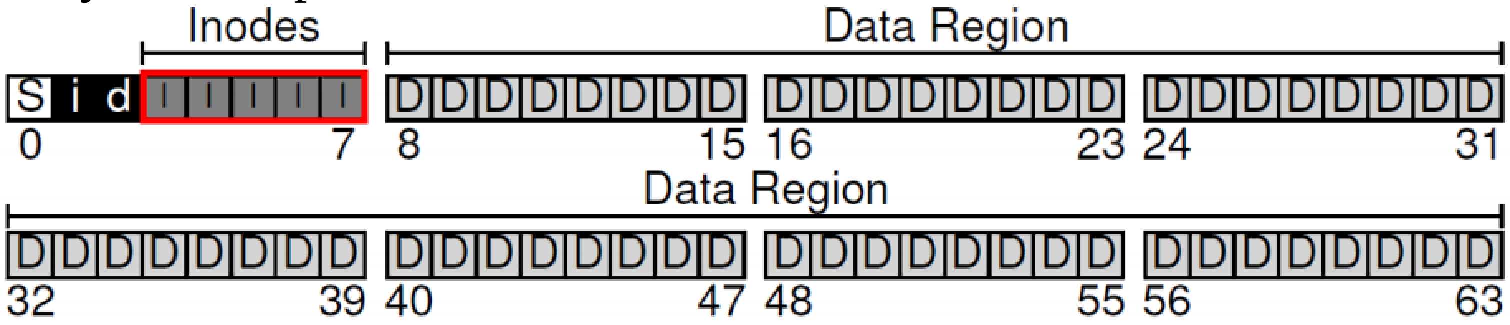
- Writing to disk sequentially is **not (alone) enough** to guarantee efficient writes.
    - ✓ In-between the first and second writes, the disk has rotated

- LFS first **buffers** all writes in an **in-memory segment**; when the segment is large enough, LFS **commits** the segment to disk as a single **large write**.
    - ✓ This technique is well known as write buffering
    - ✓ It is possible to buffer writes to different files in a segment.

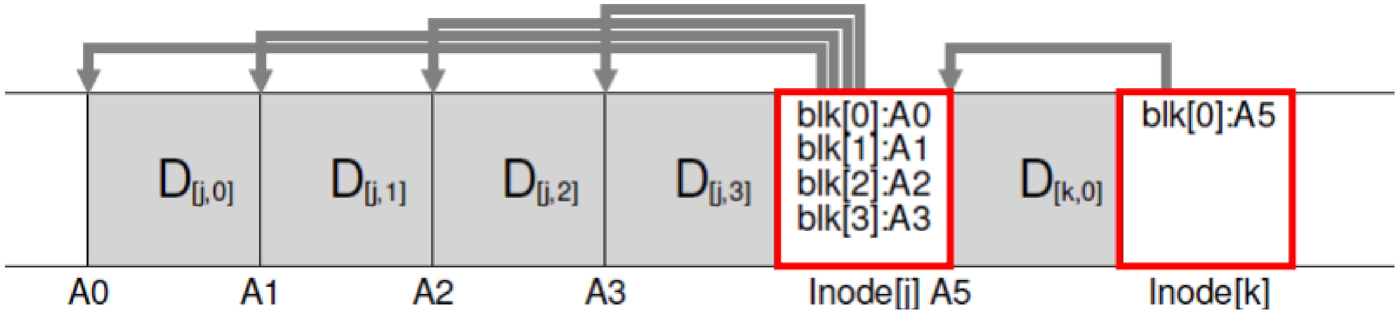


How to Find Inodes? (1/3)

- UNIX file system keeps inodes at **fixed locations**.

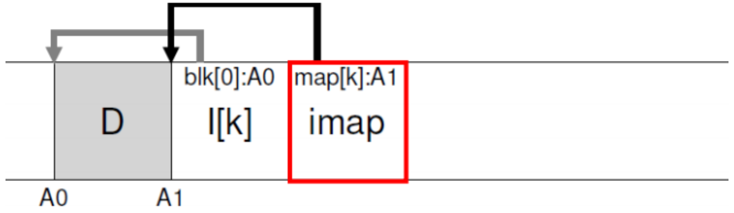


- In LFS, inodes are **scattered** throughout disk..



How to Find Inodes? (2/3)

- Solution through Indirection: The Inode Map (imap)
  - ✓ Maps from `an inode-number` to `the disk-address of the most recent version` of the inode (i.e., one more mapping!).
  - ✓ Implemented as an array of 4 bytes (disk pointer) per entry
  - ✓ Updated whenever an **inode** is written to disk.
- LFS places the `imap` right next to where it is writing.
  - ✓ E.g., when appending a data block, the new data block (D), its node (I[k]), and `imap` are written to disk together:

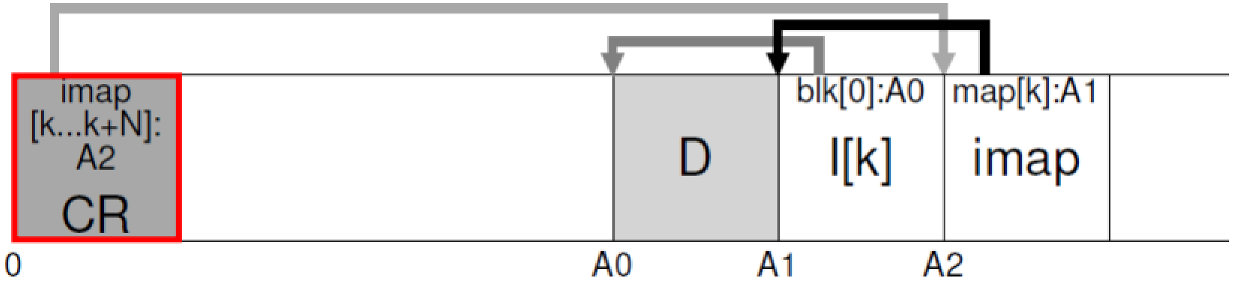


- Now we can find inodes: **But how to find the imap**?

How to Find Inodes? (3/3)

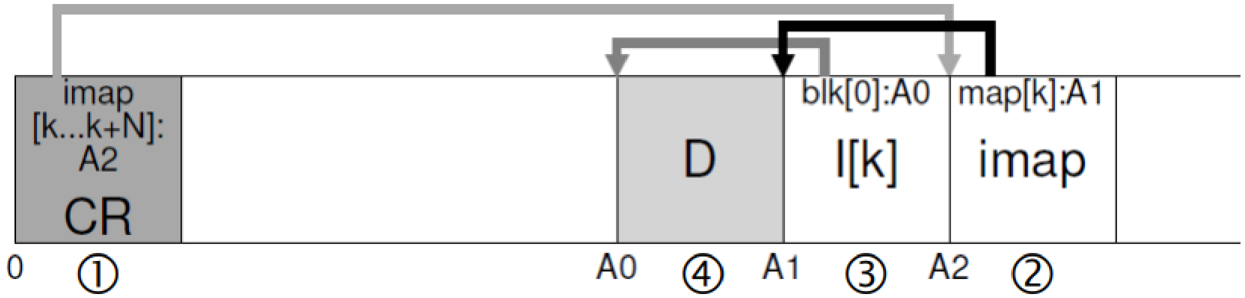
- The pieces of imap are also **spread** across the disk.
- Every file system must have some **fixed and known location** on disk to being a file lookup.

- Complete Solution: The **Checkpoint Region (CR)** records disk pointers to all **latest pieces of imap**.
  - ✓ Flushed to disk periodically (e.g., every 30 seconds)



Example: Reading a File

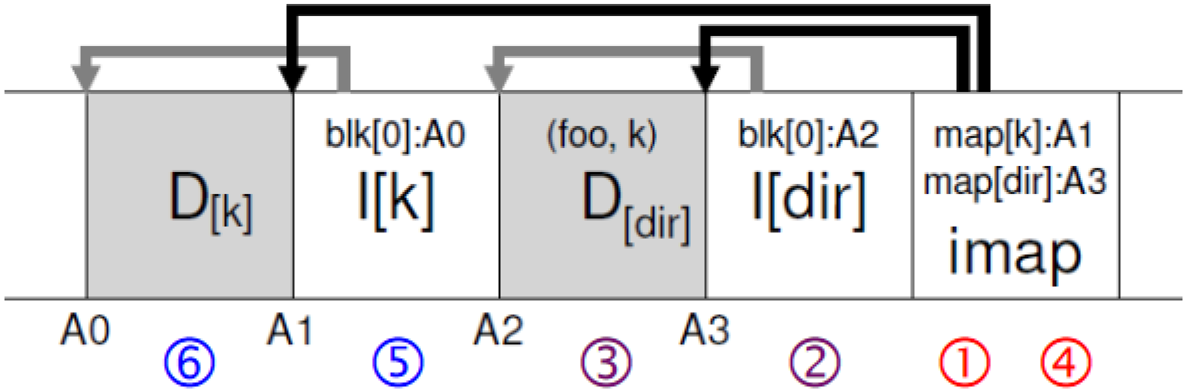
- To read a file from disk, LFS needs to
  - ① Read the checkpoint region to find the latest imap;
  - ② Read the latest imap to have the disk location of the inode;
  - ③ Read the most recent version of the inode (I[k]);
  - ④ Read data blocks using direct/indirect pointer as usual.



- To perform the same number of I/Os as UNIX FS, LFS must cache the checkpoint region (CR) and the entire imap in the system memory.

What about Directories? (1/2)

- The directory structure of LFS is **identical** to UNIX FS.
  - ✓ The directory is a collection of (name, inode-num) entries.
- When creating a file, LFS writes **the data and the new inode**, the directory and its inode, and the latest imap.
  - ✓ LFS will do so sequentially on the disk as follows:



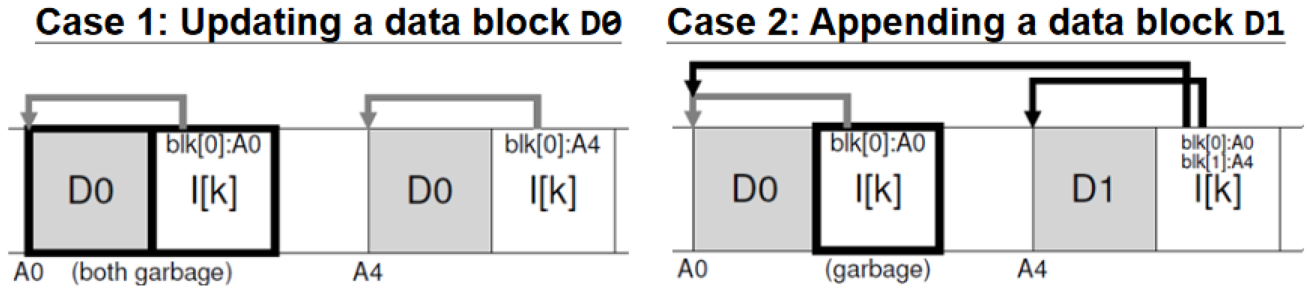
What about Directories? (2/2)

- **Recursive Update Problem:** A serious problem arisen in any file system that never updates in place.
    - ✓ Whenever an inode is updated, its location on disk changes.
    - ✓ This would have also entailed **recursive updates** to the directory that points to this file, the parent of that directory, ..., all the way up the file system tree.

- LFS cleverly avoids this problem with **`imap`**.
    - ✓ The directory is a collection of (`name`, `inode-num`) entries
    - ✓ The `imap` keeps `inode-num` to `inode-location` mappings.

Garbage Collection (1/3)

- LFS **never overwrites** but writes to free locations
  - ✓ **Multiple versions** of data may co-exist across the disk.



- One could keep older versions and allow accessing.
  - ✓ Such a file system is known as a **versioning file system**.
- LFS keeps only the latest *live* versions of data, and periodically cleans old *dead* versions of data..
  - ✓ The process of cleaning is called **garbage collection (GC)**.

Garbage Collection (2/3)

- LFS adopts a **segment-based cleaning** as follows:
    - ✓ Reads in *M* partially-used segments;
    - ✓ Determines which blocks are live within these segments;
    - ✓ Compacts only live contents into *N* new segments ($N<M$);
    - ✓ Writes out *N* segments to disk in new locations;
    - ✓ Frees old *M* segments for subsequent writing

- Two more problems:
    - ✓ How to determine if a block is live (or dead)?
    - ✓ How often, and which segments to clean?

Garbage Collection (3/3)

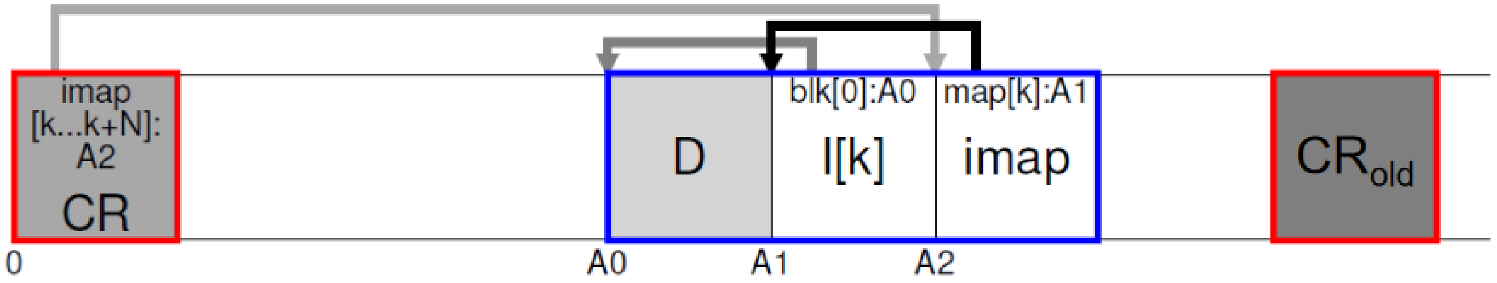
- LFS adds extra information, at the head of each segment, called the **segment summary block (SS)**.
  - ✓ It records, for each data block D in the segment, its inode number N and its offset T (e.g., (k, 0)).
  - ✓ The *liveness* for a block D of address **A** can be determined:

```
(N, T) = SegmentSummary[A];
inode = Read(imap[N]);
if (inode[T] == A)
    // block D is alive
else
    // block D is garbage
```

- Optimization:
  - ✓ Keeping a version number in both **imap** and **SS**, extra reads of inodes can be further avoided.
  - ✓ The version number should be incremented whenever the file is truncated or deleted.

Crash Recovery

- ➢ Crashes when writing to the **checkpoint region**:
  - ✓ Solution: Keeps **two CRs** (e.g., one at the head and one at the end) and writes to them alternately.
    - It first writes a **header (with a timestamp)**, then the body of CR, and then an **end marker (with a timestamp)**
    - Inconsistent pair of timestamps implies an **error**



- ➢ Crashes when writing to a segment:
  - ✓ **Roll Forwarding**: Starts with the last checkpoint region and rebuilds all "**non-checkpointed**" but "**committed**" segments

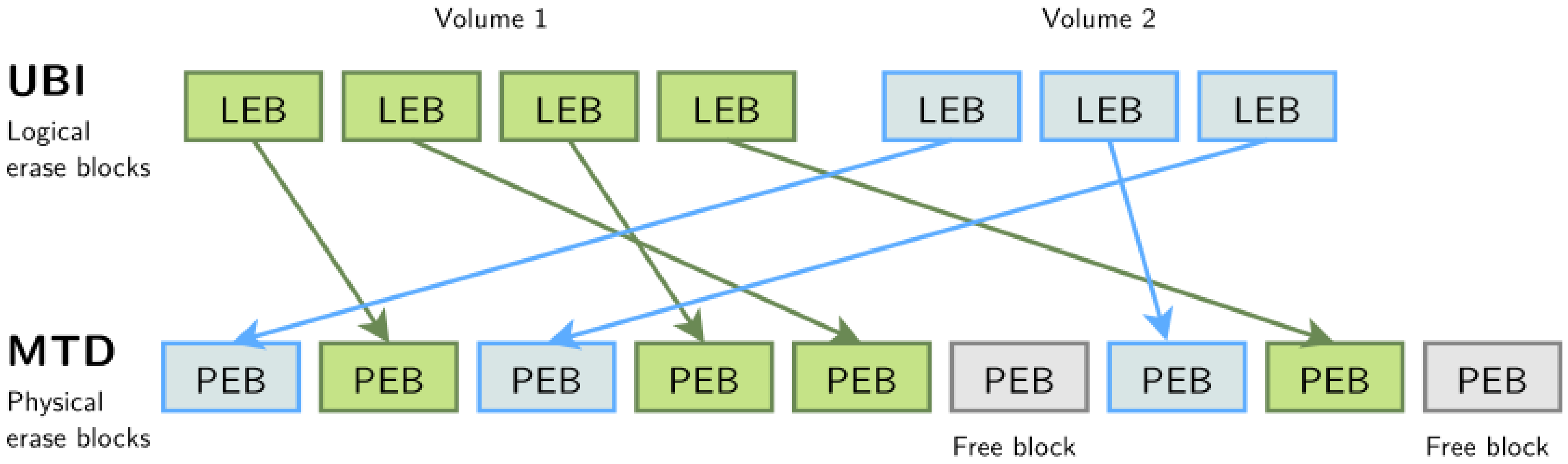
UBI/UBIFS

- The purpose of UBI/UBIFS
  - ✓ Aimed at replacing JFFS2 by addressing its limitations
  - ✓ Design choices:
    - Split the wear leveling and filesystem layers
    - Add some flexibility
    - Focus on scalability, performance and reliability
  - ✓ Drawback: introduces noticeable **space overhead**, especially when used on small <u>devices</u> or <u>partitions</u>.

Unsorted Block Images

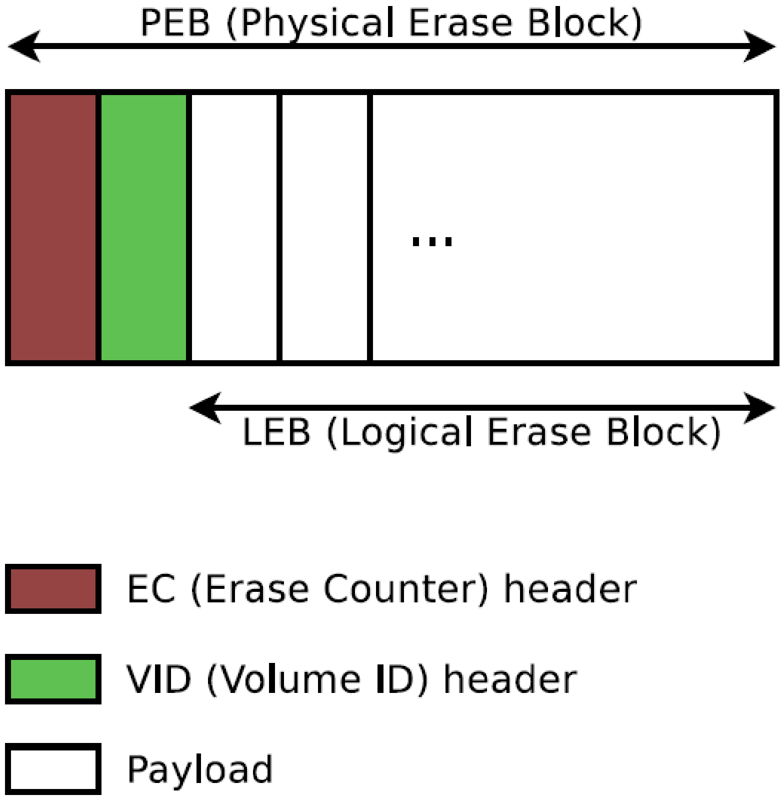
- The information of UBI
  - ✓ http://www.linux-mtd.infradead.org/doc/ubi.html
  - ✓ **Volume management system** on top of MTD devices (similar to what LVM provides for block devices)
  - ✓ Allows to **create multiple logical volumes** and spread writes across all physical blocks
  - ✓ Takes care of managing the **erase blocks** and **wear leveling**. Makes filesystems easier to implement
  - ✓ Wear leveling can operate **on the whole storage**, not only on individual partitions (strong advantage)
  - ✓ Volumes can be **dynamically** resized or, on the opposite, can be read-only (static)

UBI and MTD

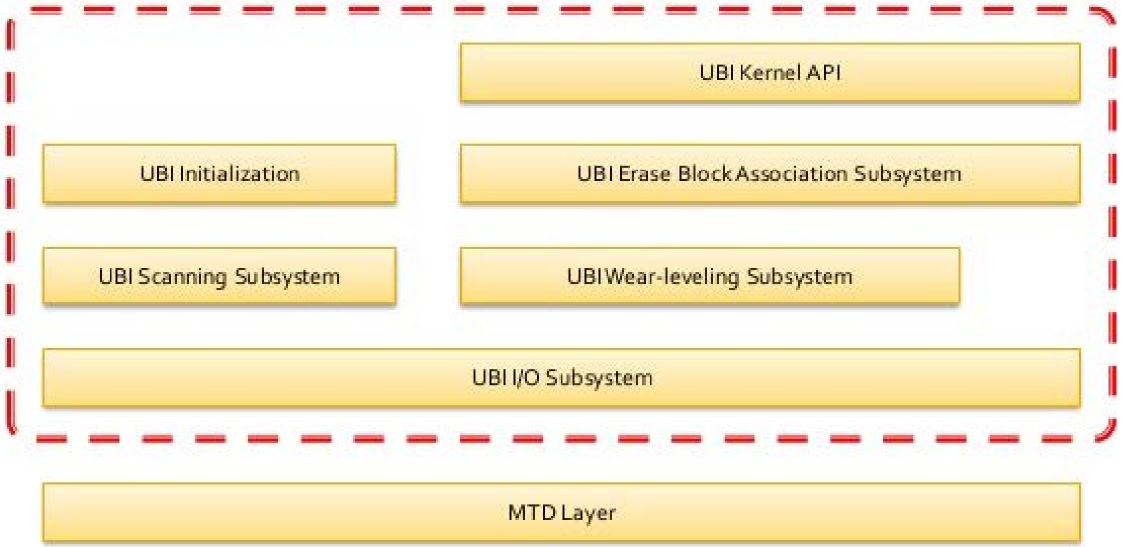

Volume 1
Volume 2
UBI
Logical erase blocks
LEB
LEB
LEB
LEB
LEB
LEB
LEB
MTD
Physical erase blocks
PEB
PEB
PEB
PEB
PEB
PEB
PEB
PEB
PEB
Free block
Free block

UBI: Internals

- UBI is storing its metadata **in-band**
- In each MTD erase block
    - ✓ One page is reserved to count **the number of erase cycles**
    - ✓ Another page is reserved to attach the erase block to a UBI volume
    - ✓ The remaining pages are used to store **payload data**
- If the device supports **subpage write**, the EC and VID headers can be stored on the same page.

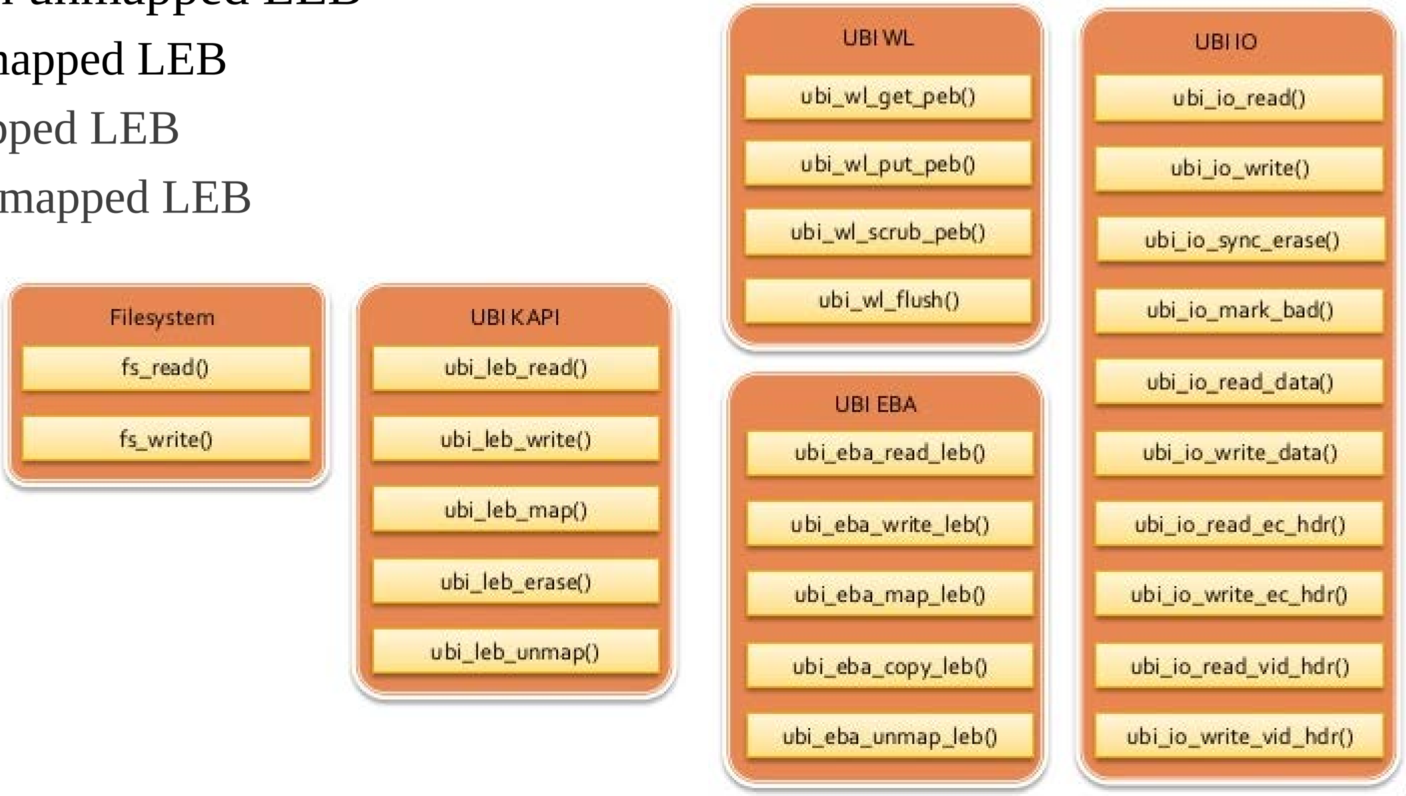

PEB (Physical Erase Block)
...
LEB (Logical Erase Block)
EC (Erase Counter) header
VID (Volume ID) header
Payload

UBI Subsystem



Kernel API & EBA Subsystem

- Read from an unmapped LEB
- Read from a mapped LEB
- Write to a mapped LEB
- Write to an unmapped LEB
- Map a LEB
- Unmap a LEB
- Erase a LEB



UBI: Good Practice

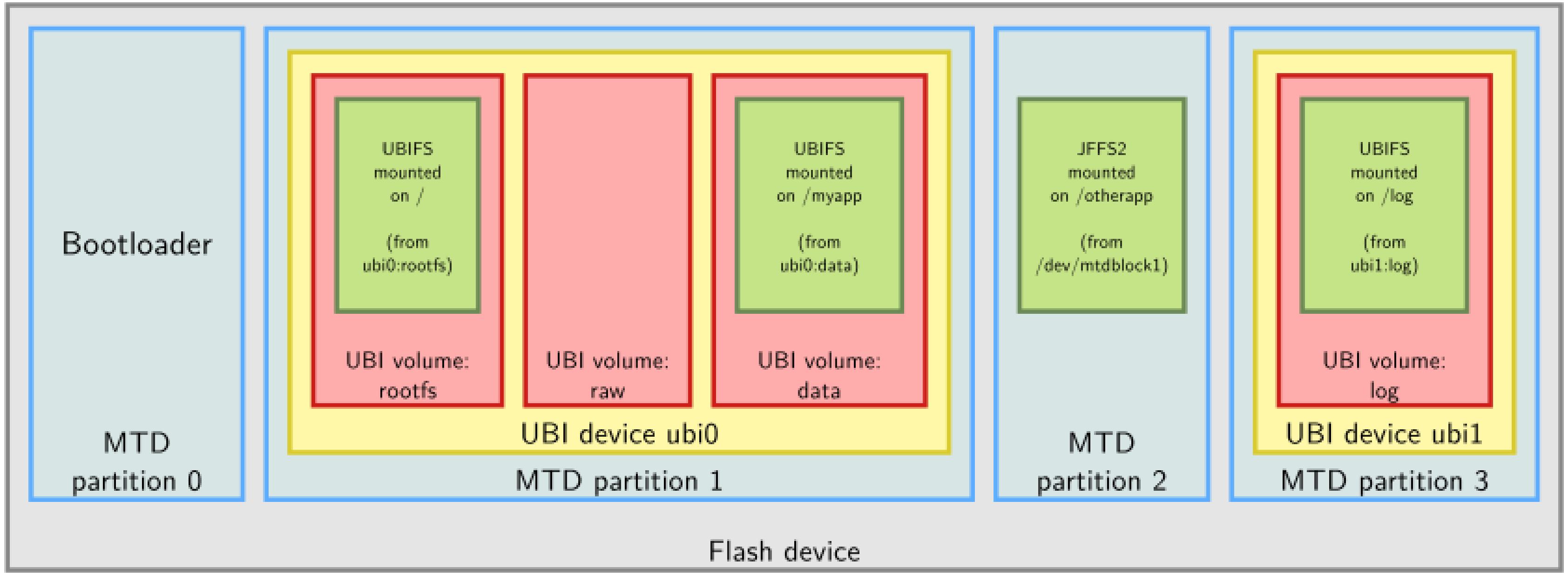
- UBI is responsible for **distributing writes** all over the flash device
  - ✓ The **more space** you assign to a partition attached to the UBI layer the **more efficient** the wear leveling will be

- If you need partitioning, **use UBI volumes** not MTD partitions

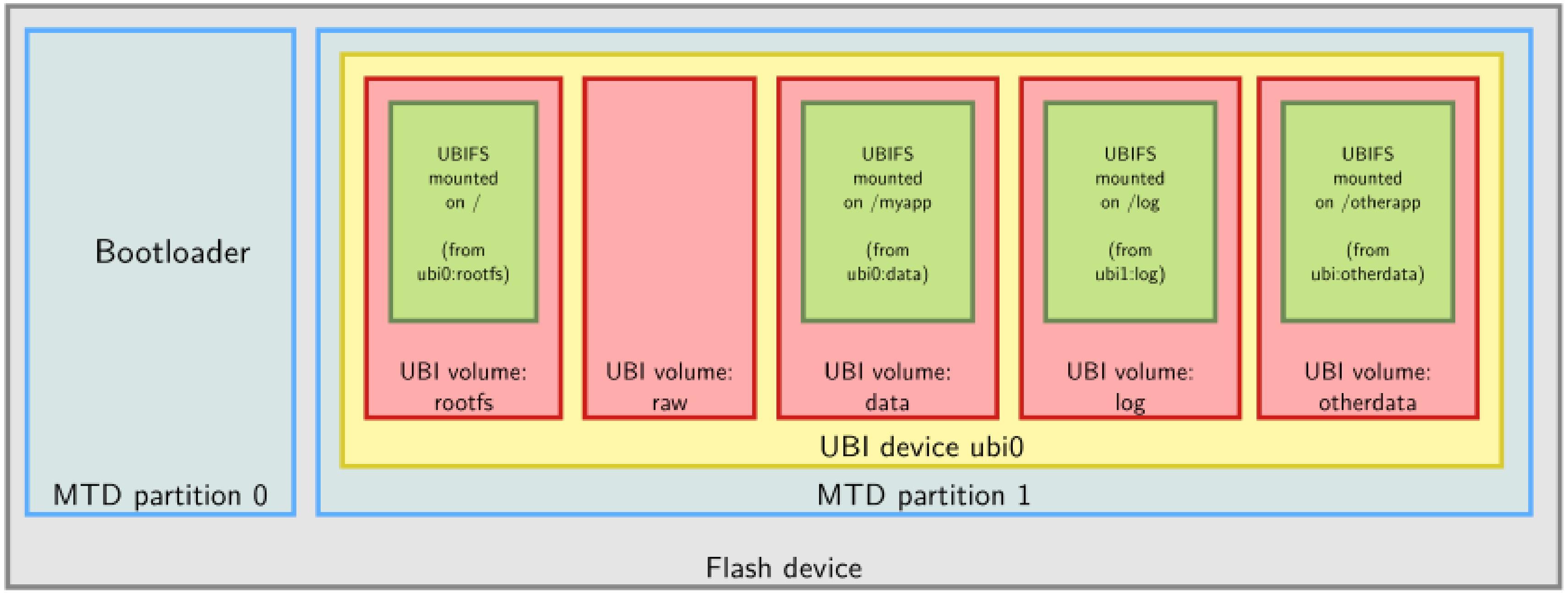
- Some partitions will still have to be MTD partitions: e.g. the **bootloaders** and **bootloader environments**

- If you need extra MTD partitions, try to group them **at the end** or the **beginning** of the flash device

UBI Layout: Bad Example



UBI Layout: Good Example


Bootloader
MTD partition 0
UBIFS mounted on / (from ubi0:rootfs)
UBI volume: rootfs
UBI volume: raw
UBIFS mounted on /myapp (from ubi0:data)
UBI volume: data
UBIFS mounted on /log (from ubi1:log)
UBI volume: log
UBIFS mounted on /otherapp (from ubi:otherdata)
UBI volume: otherdata
UBI device ubi0
MTD partition 1
Flash device

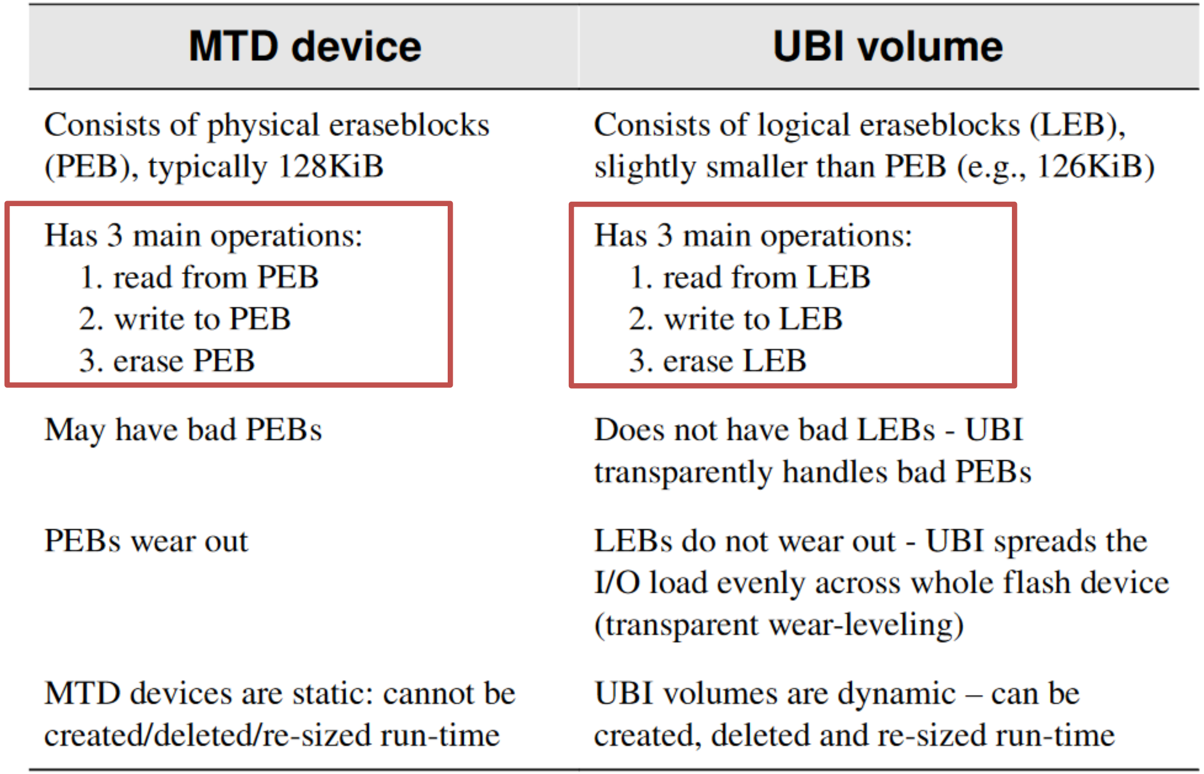
WL Internal Data Structure

- All good PEBs are maintained in four RB-trees and one queue

```
struct ubi_device{
    …
    struct rb_root used;
    struct rb_root errorneous;
    struct rb_root free;
    struct rb_root scrub;
    struct list_head pq[UBI_PROT_QUEUE_LEN];
    …
}
```

UBI Volume vs. MTD Device

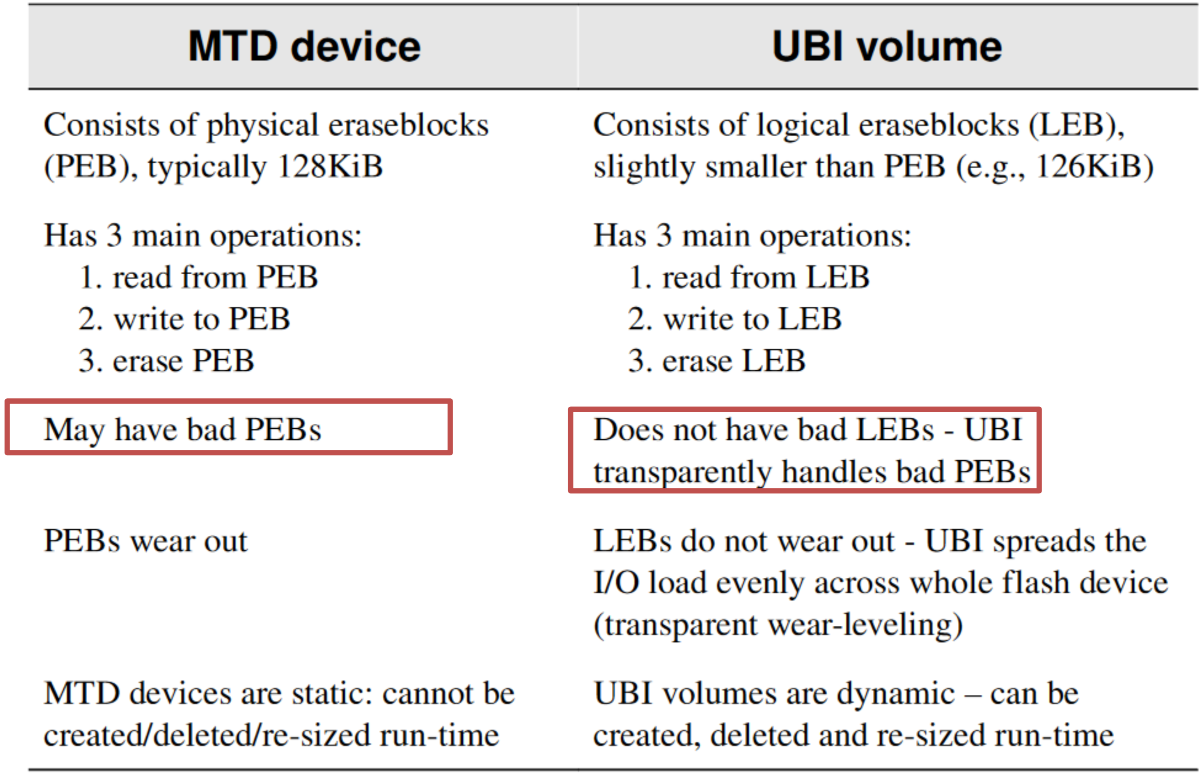
- All good PEBs are maintained in four RB-trees and one queue



MTD device	UBI volume
Consists of physical eraseblocks (PEB), typically 128KiB	Consists of logical eraseblocks (LEB), slightly smaller than PEB (e.g., 126KiB)
Has 3 main operations: 1. read from PEB 2. write to PEB 3. erase PEB	Has 3 main operations: 1. read from LEB 2. write to LEB 3. erase LEB
May have bad PEBs	Does not have bad LEBs - UBI transparently handles bad PEBs
PEBs wear out	LEBs do not wear out - UBI spreads the I/O load evenly across whole flash device (transparent wear-leveling)
MTD devices are static: cannot be created/deleted/re-sized run-time	UBI volumes are dynamic – can be created, deleted and re-sized run-time

UBI Volume vs. MTD Device

- All good PEBs are maintained in four RB-trees and one queue

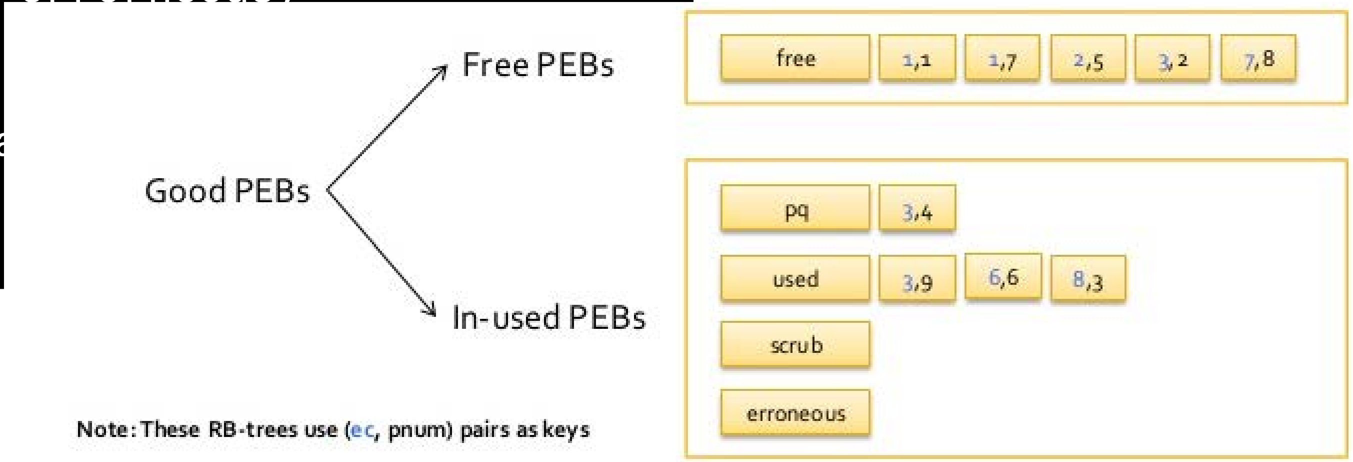


MTD device	UBI volume
Consists of physical eraseblocks (PEB), typically 128KiB	Consists of logical eraseblocks (LEB), slightly smaller than PEB (e.g., 126KiB)
Has 3 main operations: 1. read from PEB 2. write to PEB 3. erase PEB	Has 3 main operations: 1. read from LEB 2. write to LEB 3. erase LEB
May have bad PEBs	Does not have bad LEBs - UBI transparently handles bad PEBs
PEBs wear out	LEBs do not wear out - UBI spreads the I/O load evenly across whole flash device (transparent wear-leveling)
MTD devices are static: cannot be created/deleted/re-sized run-time	UBI volumes are dynamic – can be created, deleted and re-sized run-time

WL Internal Data Structure

- All good PEBs are maintained in four RB-trees and one queue

```
struct ubi_device{
    …
    struct rb_root used;
    struct rb_root errorneous;
    struct rb_root
    struct rb_root
    struct list_hea
    …
}
```



UBIFS (I/II)

- Unsorted Block Images File System
  - ✓ http://www.linux-mtd.infradead.org/doc/ubifs.html
  - ✓ The filesystem part of the UBI/UBIFS couple
  - ✓ Works on **top** of UBI volumes
  - ✓ **Journaling file system** providing better performance than JFFS2 and addressing its scalability issues

UBIFS (II/II)

- UBIFS relies on UBI
  - ✓ UBIFS **does not care** about bad eraseblocks and relies on UBI
  - ✓ UBIFS **does not care** about wear-leveling and relies on UBI
  - ✓ UBIFS exploits the **atomic LEB change** feature

UBIFS (II/II)

- UBIFS relies on UBI
  - ✓ UBIFS **does not care** about bad eraseblocks and relies on UBI
  - ✓ UBIFS **does not care** about wear-leveling and relies on UBI
  - ✓ UBIFS exploits the **atomic LEB change** feature

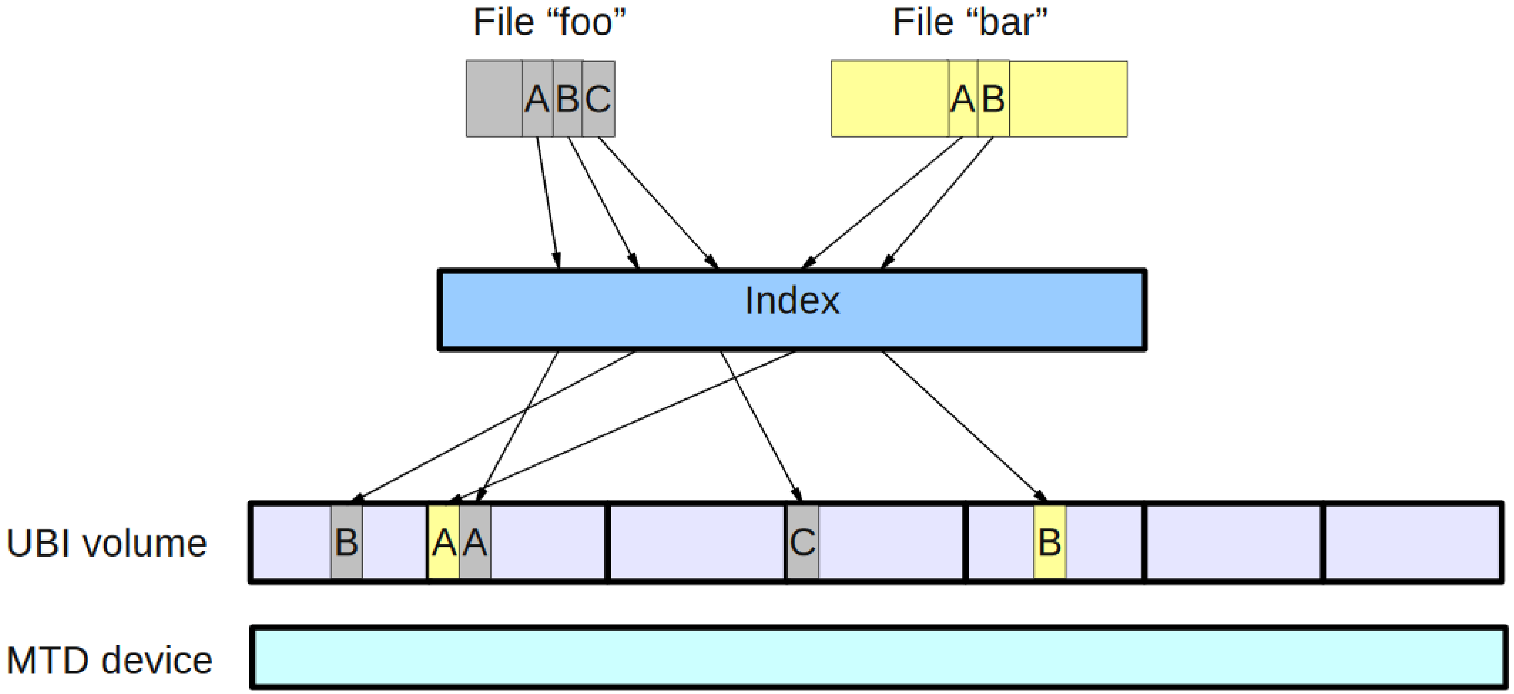
- The Requirements of UBI
  - ✓ Good scalability
  - ✓ High performance
  - ✓ On-the-flight compression
  - ✓ Power-cut tolerance
  - ✓ High reliability
  - ✓ Recoverability

LEB Properties

- UBIFS stores per LEB information of flash
  - ✓ LEB type (**indexing** or **data**)
  - ✓ Amount of free and dirty space
- Overall space accounting information is maintained on the media
  - ✓ Total amount of free space
  - ✓ Total amount of dirty space
  - ✓ Etc.
- How to use LEB information
  - ✓ A free LEB should be found for new data
  - ✓ A dirty LEB should be found for GC

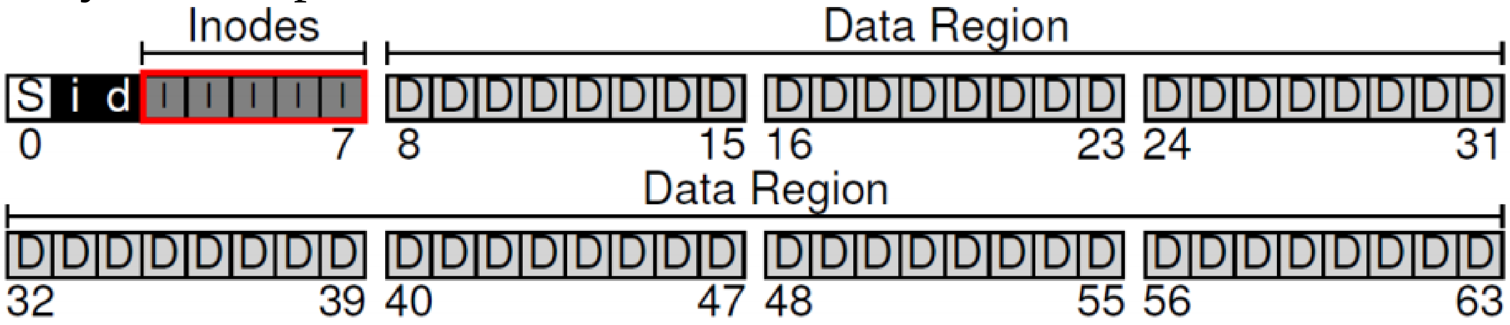
File System Index

- Index allows to lookup physical address for any piece of FS data

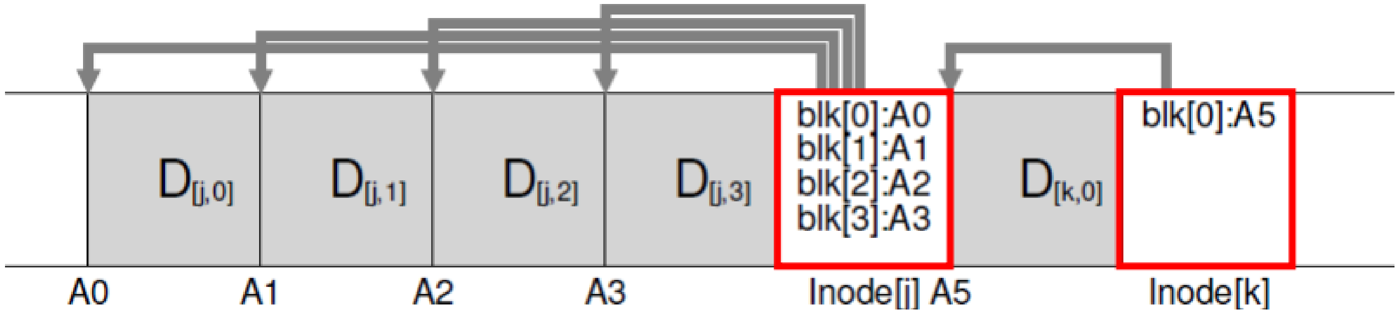


Recall: Index in JFFS2

- UNIX file system keeps inodes at **fixed locations**.

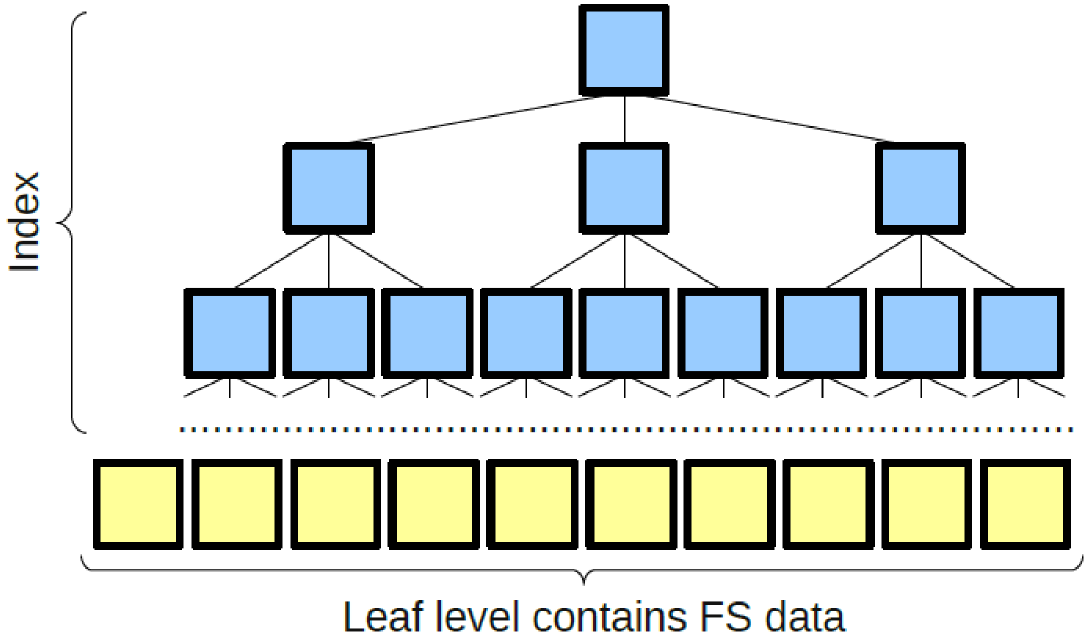


- In LFS, inodes are **scattered** throughout disk..



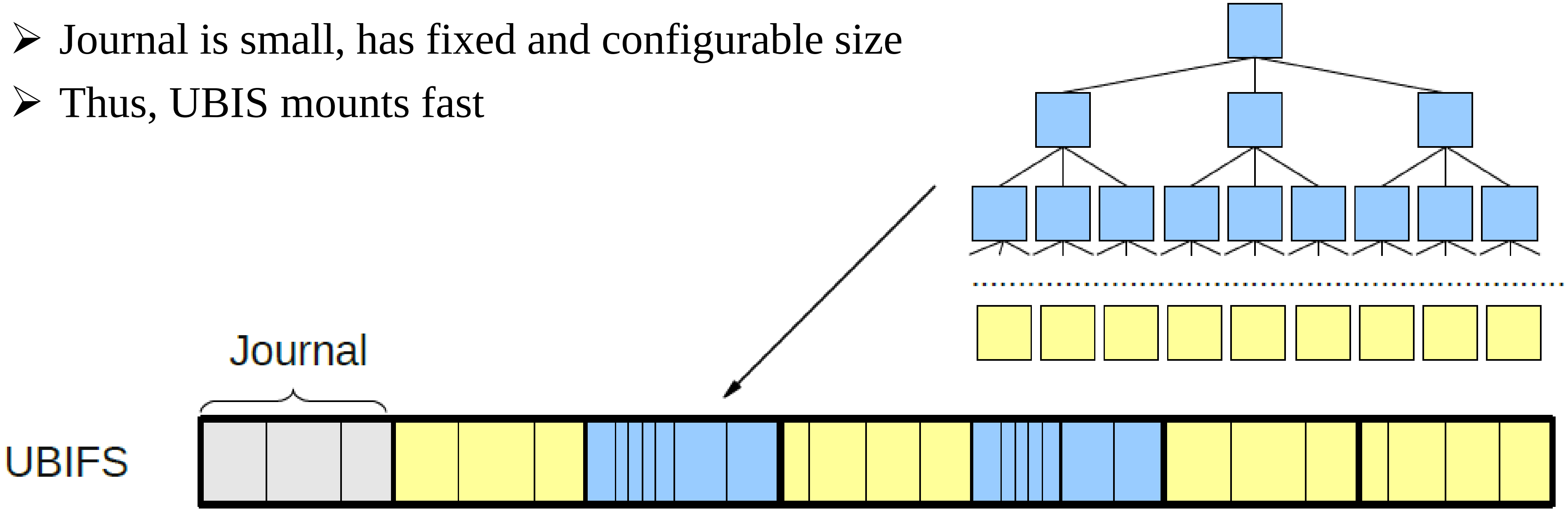
UBIFS Index (I/II)

- UBIFS index is a B+ tree
- Leaf level contains data
- Tree fanout is configurable, default is 8



UBIFS Index (II/II)

- UBIFS index is stored and maintained on **flash**
- Full flash media scanning **is not needed**
- **Only the journal** is scanned in case of power cut
- Journal is small, has fixed and configurable size
- Thus, UBIS mounts fast



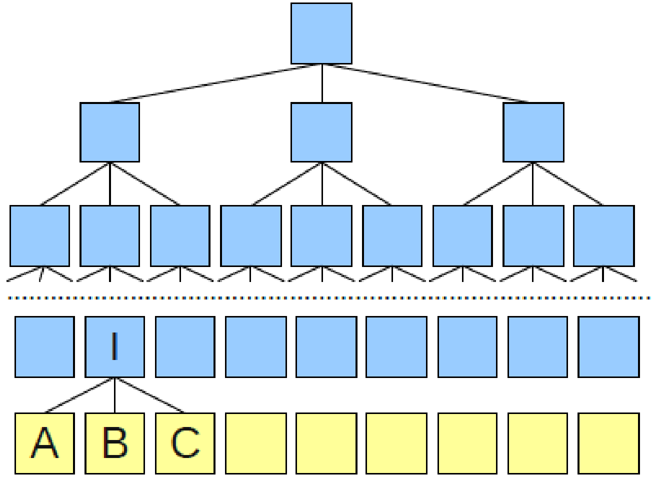
Out-of-place Updates

➢ Flash technology and power-cut-tolerance require out-of-place updates

Change “foo” (overwrite A, B, C)

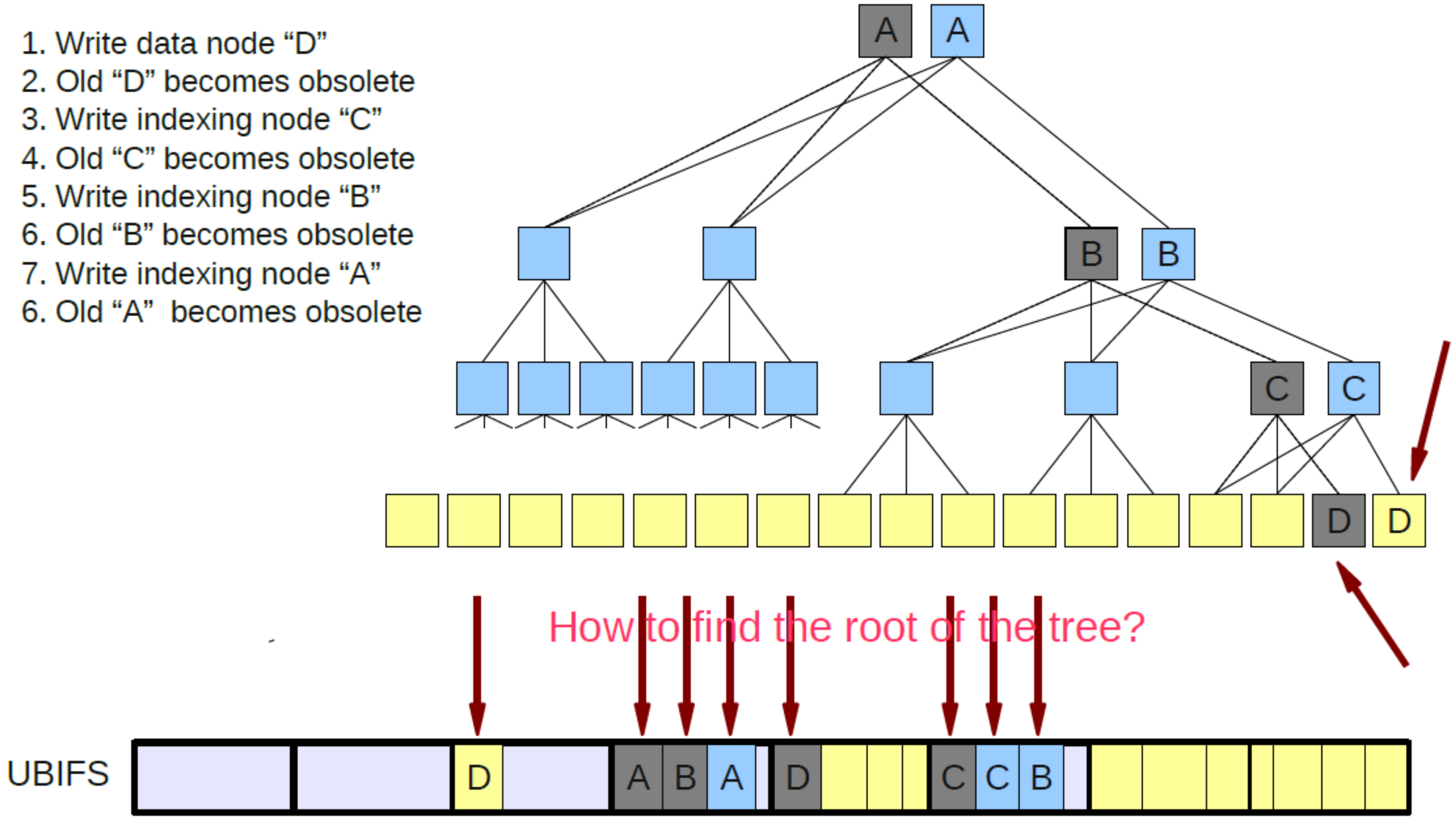
File “/foo”: A B C

1. Write “A” to a free space
2. Old “A” becomes obsolete
3. Write “B” to free space
4. Old “B” becomes obsolete
5. Write “C” to free space
6. Old “C” becomes obsolete



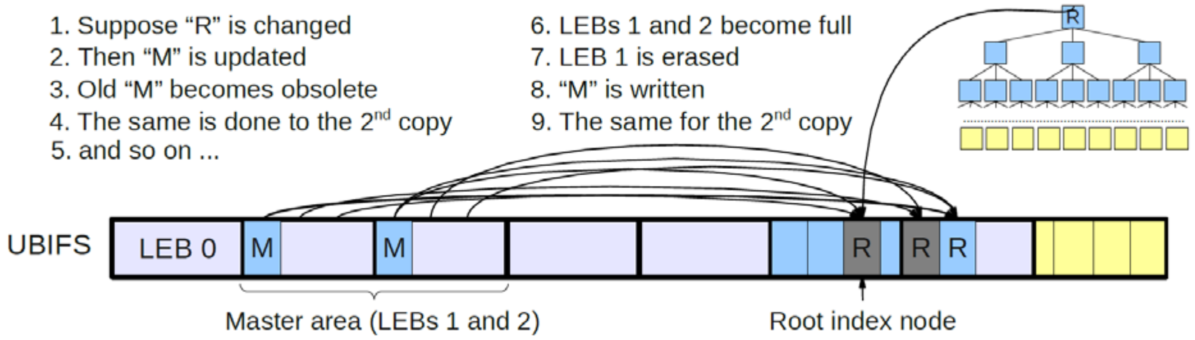
Wandering Trees

1. Write data node “D”
2. Old “D” becomes obsolete
3. Write indexing node “C”
4. Old “C” becomes obsolete
5. Write indexing node “B”
6. Old “B” becomes obsolete
7. Write indexing node “A”
6. Old “A” becomes obsolete



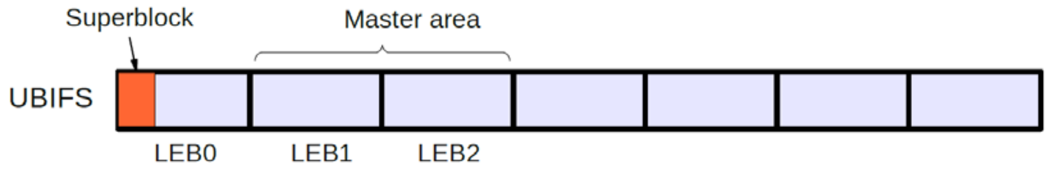
Master Node

- Stored at the master area (**LEBs 1** and **2**)
- Points to the **root index node**
- **2 copies** of master node exist for **recoverability**
- Master area may be **quickly** found on mount
- Valid master node is found by **scanning master area**


1. Suppose "R" is changed
2. Then "M" is updated
3. Old "M" becomes obsolete
4. The same is done to the 2nd copy
5. and so on ...
6. LEBs 1 and 2 become full
7. LEB 1 is erased
8. "M" is written
9. The same for the 2nd copy
R
UBIFS
LEB 0
M
M
R
R
R
Master area (LEBs 1 and 2)
Root index node

Superblock

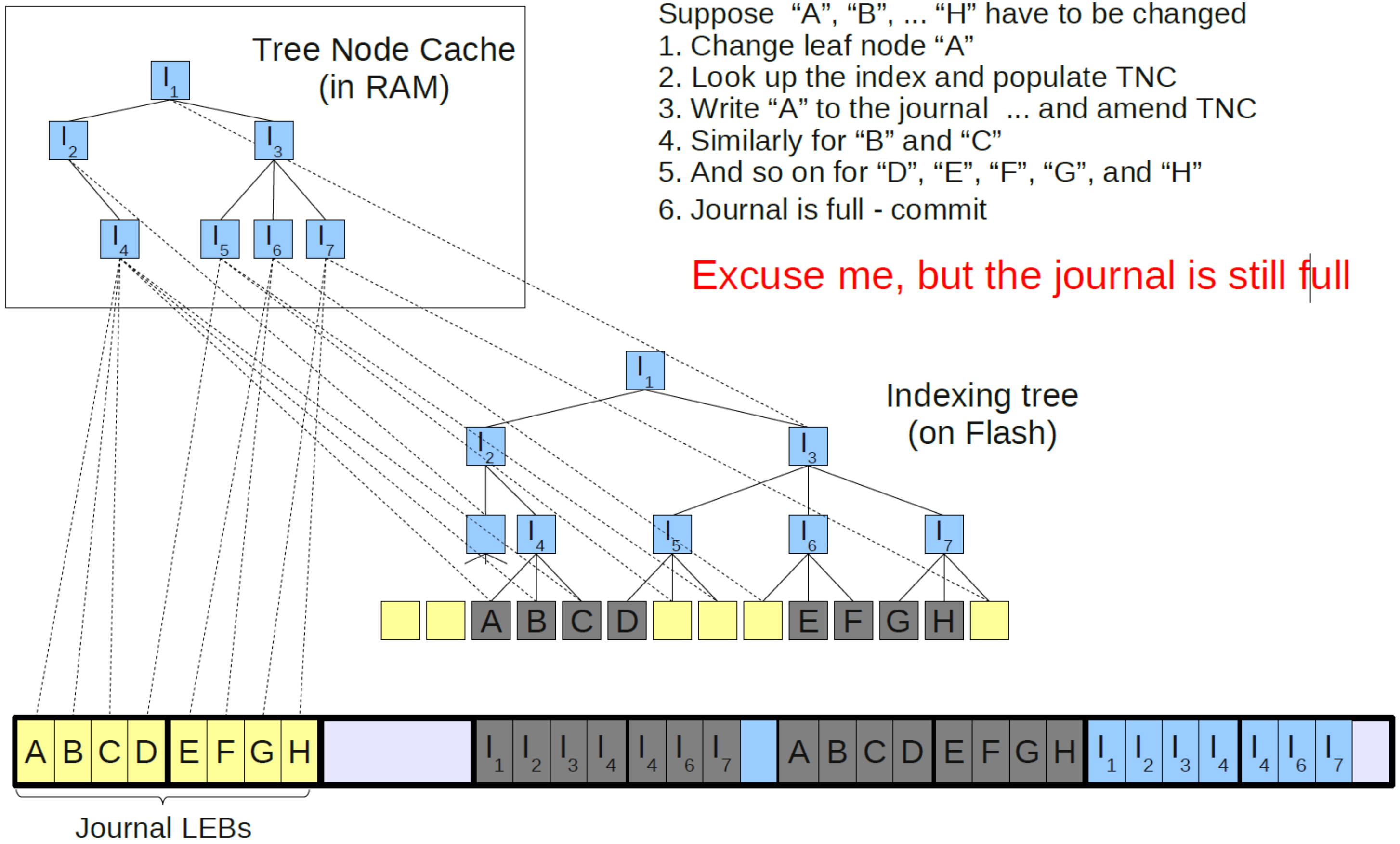
- Situated at **LEB0**
- **Read-only** for UBIFS
- May be changed by **user-space tools**
- Stores configuration information like **indexing tree fanout**, **default compression** type (zlib or LZO), etc
- Superblock is read on mount



Journal

- All FS changes go to the **journal**
- Indexing information is changed in RAM, but not on the flash
- Journal **greatly increases** FS write performance
- When mounting, journal is scanned and replayed
- Journal is roughly **like** a small JFFS2 inside UBIFS
- Journal size is **configurable** and is stored in **superblock**

Journal Process (I/II)



Suppose "A", "B", ... "H" have to be changed

1. Change leaf node "A"
2. Look up the index and populate TNC
3. Write "A" to the journal ... and amend TNC
4. Similarly for "B" and "C"
5. And so on for "D", "E", "F", "G", and "H"
6. Journal is full - commit

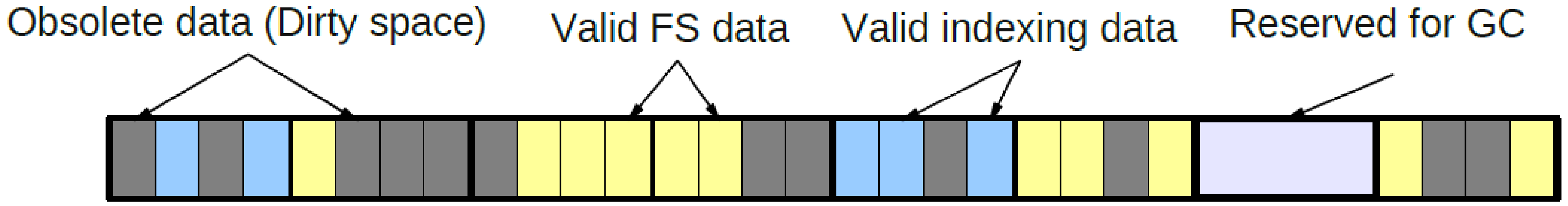
Excuse me, but the journal is still full

Journal Process (II/II)

- Journal is also wandering
  - ✓ **After the commit** we pick different LEBs for the journal
  - ✓ **Do not** move data out of the journal
  - ✓ Instead, we **move the journal**
  - ✓ Journal **changes** the position all the time
- More about the Journal
  - ✓ Journal has **multiple heads**
  - ✓ Journal LEBs may have **random** addresses
  - ✓ **LEBs do not have to be empty** to be used for journal
  - ✓ This makes journal very **flexible** and **efficient**

Garbage Collection

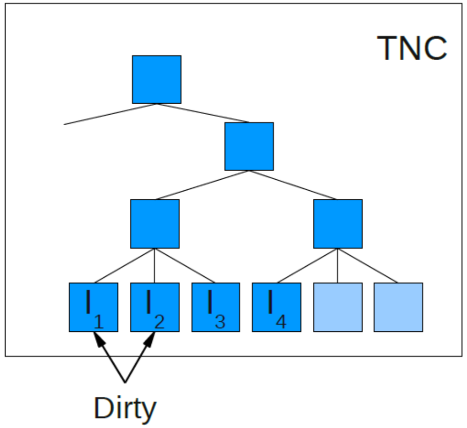
- At some point UBIFS runs **out of free space**
- Garbage Collector (GC) is responsible to turn dirty space to free space
- **One empty LEB** is always reserved for GC

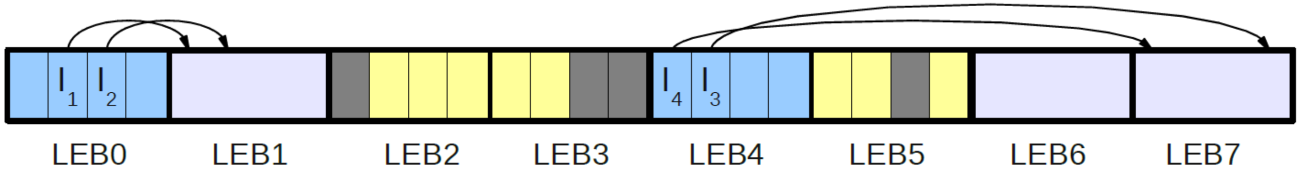


How Does GC Work?

- GC copies valid data to the journal (GC head)

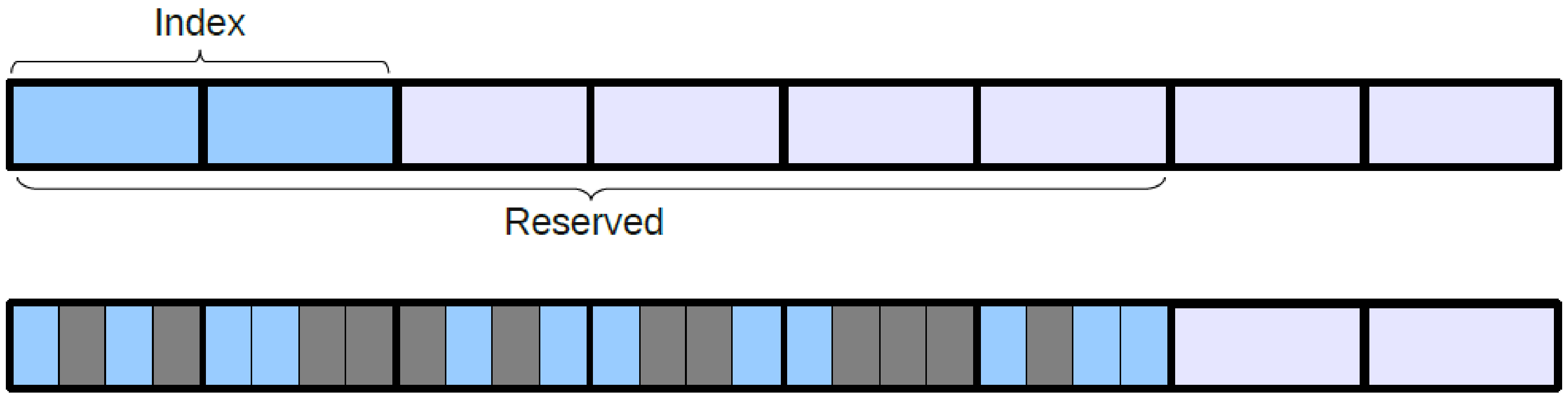
1. Pick a dirty LEB ... LEB1
2. Copy valid data to LEB6
3. LEB1 may now be erased
4. Pick another dirty LEB ... LEB7
5. Copy valid data to LEB 6
6. LEB 7 now may be erased
8. LEB 1 is reserved for GC, LEB7 is available
9. How about the index?
10. Indexing nodes are just marked as dirty in TNC
11. But what if there is no free space for commit?





Commit

- Commit operation is always guaranteed to succeed
- For the index UBIFS reserves 3x as much space
- Atomic LEB change UBI feature is used


Index
Reserved