CS632/SEP564: Embedded Operating Systems (Fall 2008)

File Systems

Storage: A Logical View

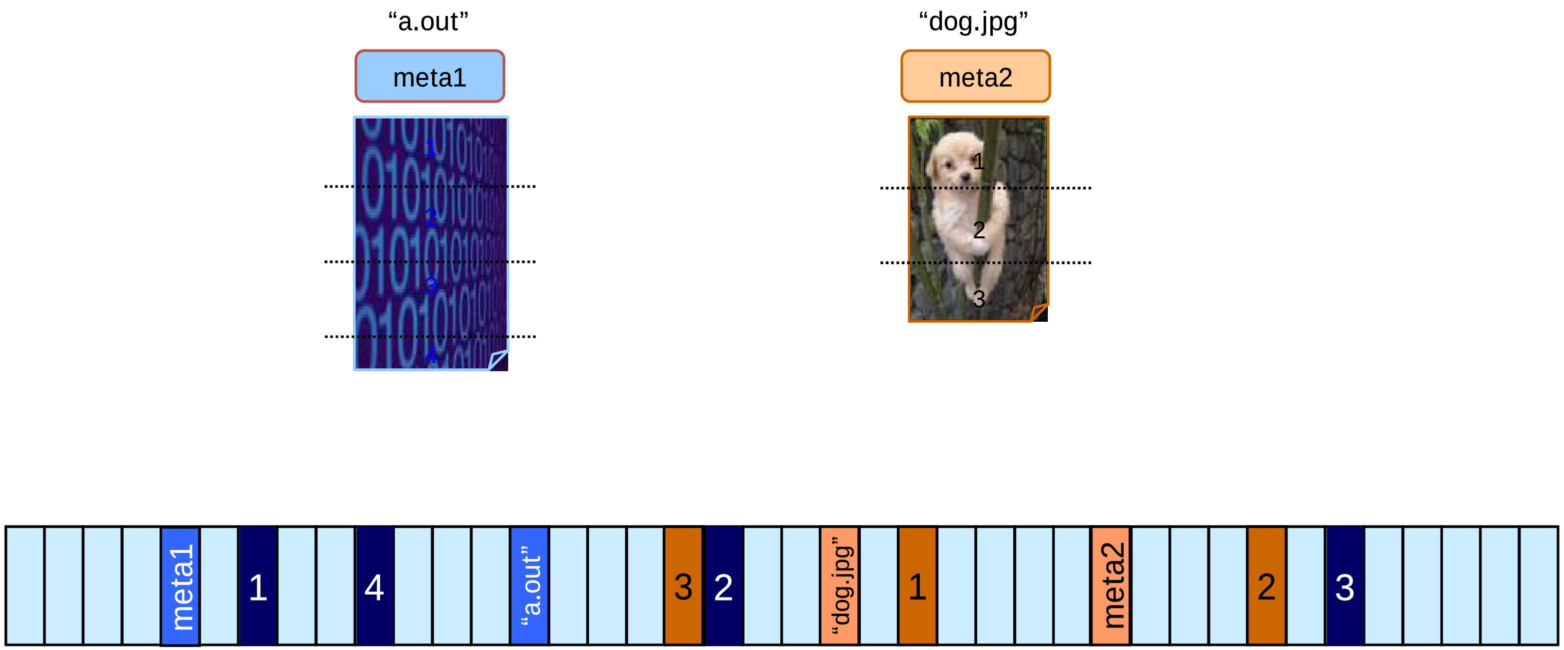
- **Abstraction given by block device drivers:**

- **Operations**
  - Identify(): returns N
  - Read(start sector #, # of sectors)
  - Write(start sector #, # of sectors)

Source: Sang Lyul Min (Seoul National Univ.)

File System Basics

- **File system: A mapping problem**
  - <filename, data, metadata> → <a set of blocks>



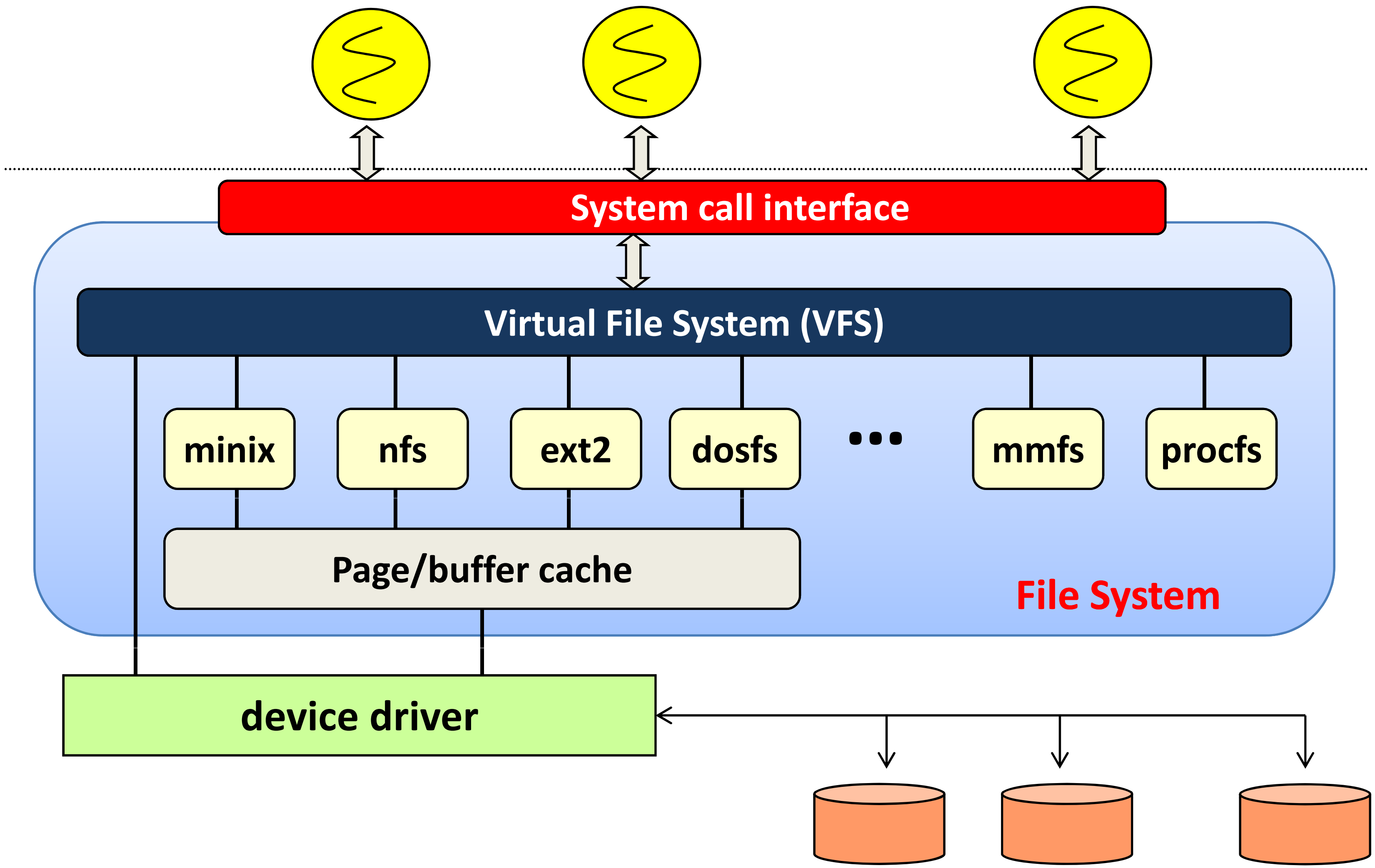
Filesystems in Linux (1)

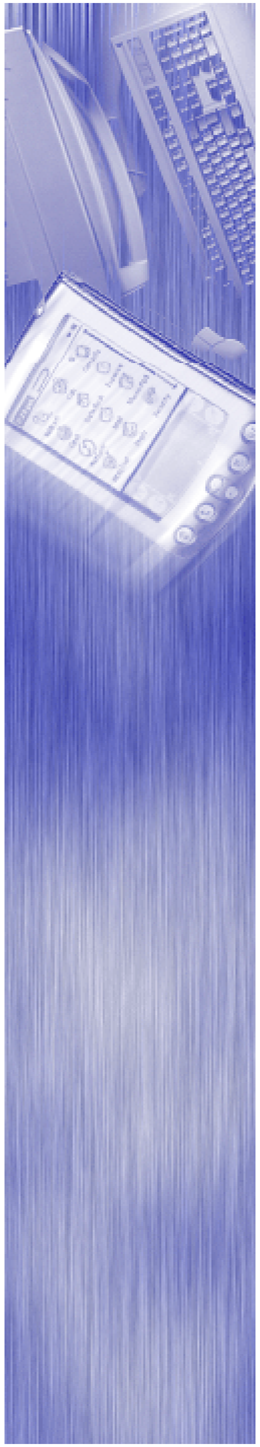
- **Disk-based filesystems**
  - Ext2/3, ReiserFS, JFS (IBM), XFS (SGI)
  - Unix variants: SYSV (System V, Coherent, Xenix), UFS (BSD, Solaris, NEXTSTEP), MINIX, Veritas VxFS (SCO Unixware)
  - Microsoft: FAT (MS-DOS), VFAT (Windows 95 and later), NTFS (Windows NT 4 and later)
  - HPFS (IBM OS/2), HFS (Apple Macintosh), AFFS (Amiga), ADFS (Acorn)
  - ISO9660 CD-ROM filesystem, UDF (Universal Disk Format) DVD filesystem

Filesystems in Linux (2)

- **Network filesystems**
  - NFS (Network File System)
  - CIFS (Common Internet File System)
  - AFS (Andrew File System)
  - Coda
  - NCP (Novell's NetWare Core Protocol)
- **Special filesystems**
  - /proc
  - /dev
  - ...
- **Flash filesystems**

File System Internals





CS632/SEP564: Embedded Operating Systems (Fall 2008)

Virtual File System (VFS)

VFS (1)

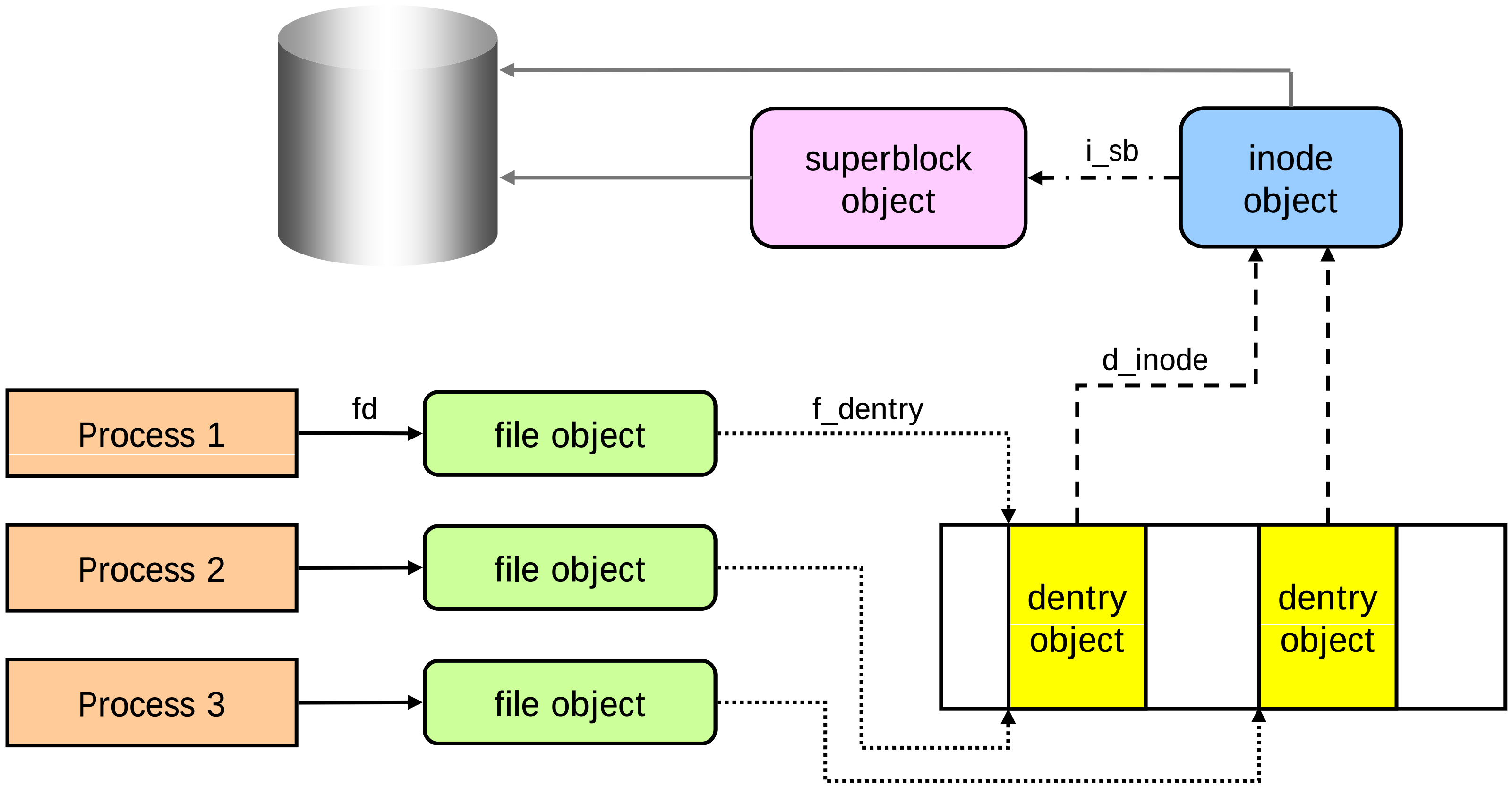
- **Virtual File System**
  - Manages kernel-level file abstractions in one format for all file systems.
  - Receives system call requests from user-level (e.g., open, write, stat, etc.)
  - Interacts with a specific file system based on mount point traversal.
  - Receives requests from other parts of the kernel, mostly from memory management.
  - Translates file descriptors to VFS data structures (such as vnode).

VFS (2)

- **Linux: VFS common file model**
  - The superblock object
    - stores information concerning a mounted file system.
  - The inode object
    - stores general information about a specific file.
  - The file object
    - stores information about the interaction between an open file and a process.
  - The dentry object
    - stores information about the linking of a directory entry with the corresponding file.
  - In order to stick to the VFS common file model, in-kernel structures may be constructed on the fly.

VFS (3)

- **Interaction b/w processes and VFS objects**



Superblock Object (1)

- **The superblock object**
  - Store information describing the specific filesystem
  - Usually corresponds to the filesystem superblock or filesystem control block, which is stored in a special sector on disk.
  - Non disk-based filesystems generate the superblock on the fly and store it in memory
    - /proc

Superblock Object (2)

- **struct super_block** <linux/fs.h>
  - The global variable super_blocks points to the list of superblock objects.

struct list_head	s_list;	Superblock objects are doubly linked
dev_t	s_dev;	Device identifier
unsigned long	s_blocksize;	Block size in bytes
struct super_operations *	s_op;	Superblock methods
struct dentry *	s_root;	Dentry object of the filesystem’s root directory
void *	s_fs_info;	Filesystem-specific superblock information
unsigned char	s_dirt;	Flag indicating whether the superblock is updated
struct list_head	s_inodes;	List of all inodes
struct list_head	s_dirty;	List of modified inodes
struct list_head	s_files;	List of file objects
...		

Superblock Object (3)

- **struct super_operations** <linux/fs.h>
  - struct inode *alloc_inode(struct super_block *sb);
    - – Allocate space for an inode object including the space required for filesystem-specific data
  - void destroy_inode(struct inode *inode);
  - void read_inode(struct inode *inode);
    - – Fill the fields of the inode object with the data on disk.
  - void write_inode(struct inode *inode, int sync);
    - – Update a filesystem inode synchronously or asynchronously
  - void put_inode(struct inode *inode);
    - – Invoked when the inode is released -- its reference counter is decreased.

Superblock Object (4)

- **struct super_operations (cont'd)**
  - void drop_inode(struct inode *inode);
    - Invoked when the last user releases the inode.
  - void delete_inode(struct inode *inode);
    - Invoked when the inode must be destroyed.
  - void put_super(struct super_block *sb);
    - Release the superblock object (on unmount)
  - void write_super(struct super_block *sb);
    - Update a filesystem superblock
  - void statfs(struct super_block *sb, struct kstatfs *buf);
    - Return statistics on a filesystem
  - …

Inode Object (1)

- **The inode object**
  - Represent all the information needed by the kernel to manipulate a file or directory.
    - For UNIX filesystems, it is read from the on-disk inode.
    - Otherwise, the filesystem must fill the inode object from whatever stored on the disk. (e.g., FAT)
  - The inode is unique to the file and remains the same as long as the file exists.

Inode Object (2)

- **struct inode** <linux/fs.h>

unsigned long	i_ino;	Inode number
umode_t	i_mode	File type and access rights
unsigned int	i_nlink;	The number of hard links
uid_t	i_uid;	Owner ID
gid_t	i_gid;	Group ID
loff_t	i_size;	File length in bytes
struct timespec	i_atime;	Time of last file access
struct timespec	i_mtime;	Time of last file write
struct timespec	i_ctime;	Time of last inode change
struct inode_operations *	i_op;	Inode operations
struct file_operations *	i_fops;	Default file operations
struct super_block *	i_sb;	Pointer to superblock object
struct list_head	i_sb_list;	Pointers for the list of inodes of the superblock
struct list_head	i_dentry;	The head of the list of dentry objects referencing this
...		

Inode Object (3)

- **struct inode_operations** <linux/fs.h>
  - int create(struct inode *dir, struct dentry *dentry,
    int mode, struct nameidata *nd);
  - struct dentry *lookup(struct inode *dir, struct dentry *dentry,
    struct nameidata *nd);
  - int link(struct dentry *old, struct inode *dir, struct dentry *dentry);
  - int unlink(struct inode *dir, struct dentry *dentry);
  - int symlink(struct inode *dir, struct dentry *dentry,
    const char *symname);
  - int mkdir(struct inode *dir, struct dentry *dentry, int mode);
  - int rmdir(struct inode *dir, struct dentry *dentry);

Inode Object (4)

- **struct inode_operations (cont'd)**
  - int mknod(struct inode *dir, struct dentry *dentry, int mode, dev_t rdev);
  - int rename(struct inode *old_dir, struct dentry *old_dentry, struct inode *new_dir, struct dentry *new_dentry);
  - ...

File Object (1)

- **The file object**
  - In-memory representation of a file opened by a process
  - File objects have no corresponding image on disk.
  - Multiple file objects can exist for the same file.
- **struct file** <linux/fs.h>

struct list_head	f_list;	Pointers for generic file object list
struct dentry *	f_dentry;	dentry object associated with this file
struct file_operations *	f_op;	Pointer to file operation table
atomic_t	f_count;	File object's reference counter
unsigned int	f_flags;	Flags specified when opening the file
mode_t	f_mode;	Process access mode
loff_t	f_pos;	Current file offset (file pointer)
struct address_space *	f_mapping;	Pointer to file's address space object
...		

File Object (2)

- **struct file_operations** <linux/fs.h>
  - int open(struct inode *inode, struct file *file);
  - int llseek(struct file *file, loff_t offset, int origin);
  - ssize_t read(struct file *file, char *buf, size_t count, loff_t *pos);
  - ssize_t write(struct file *file, const char *buf, size_t count,
    loff_t *pos);
  - int readdir(struct file *file, void *dirent, filldir_t filldir);
  - int mmap(struct file *file, struct vm_area_struct *vma);
  - int flush(struct file *file);
  - int fsync(struct file *file, struct dentry *dentry, int datasync);
  - int release(struct inode *inode, struct file *file);
  - …

Dentry Object (1)

- **The dentry object**
  - Directory implementations differ among filesystems.
  - Once a directory entry is read into memory, it is transformed by VFS into a dentry object.
  - The kernel creates a dentry object for every component to its corresponding inode.
    - /tmp/test: three dentry objects (/, /tmp, /tmp/test)
  - Dentry objects have no corresponding image on disk.
  - Dentry objects are stored in a slab allocator cache.
    - kmem_cache_alloc(), kmem_cache_free()

Dentry Object (2)

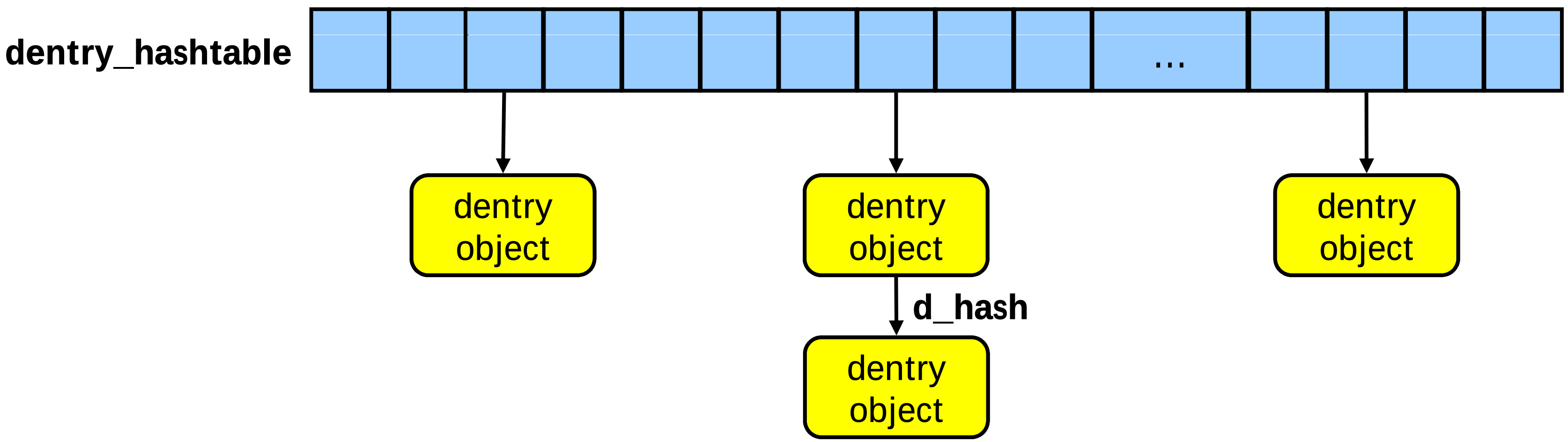
- **struct dentry** <linux/dcache.h>

atomic_t	d_count;	Dentry object usage counter
unsigned int	d_flag;	Dentry cache flags
struct qstr	d_name;	File name
struct inode *	d_inode;	Inode associated with the dentry object
struct dentry *	d_parent;	Dentry object of parent directory
struct list_head	d_lru;	Pointers for the list of unused dentries
struct list_head	d_child;	For directories, pointers for the list of dentries in the same parent directory
struct list_head	d_subdirs;	For directories, head of the list of subdirectory dentries
struct list_head	d_alias;	Pointers for the list of dentries associated with the same inode (alias)
struct dentry_operations *	d_op;	Dentry methods
struct super_block *	d_sb;	Superblock object of the file
struct hlist_node	d_hash;	Pointer for list in hash table entry
int	d_mounted;	The number of file systems mounted on this dentry
...		

Dentry Object (3)

▪ The dentry cache

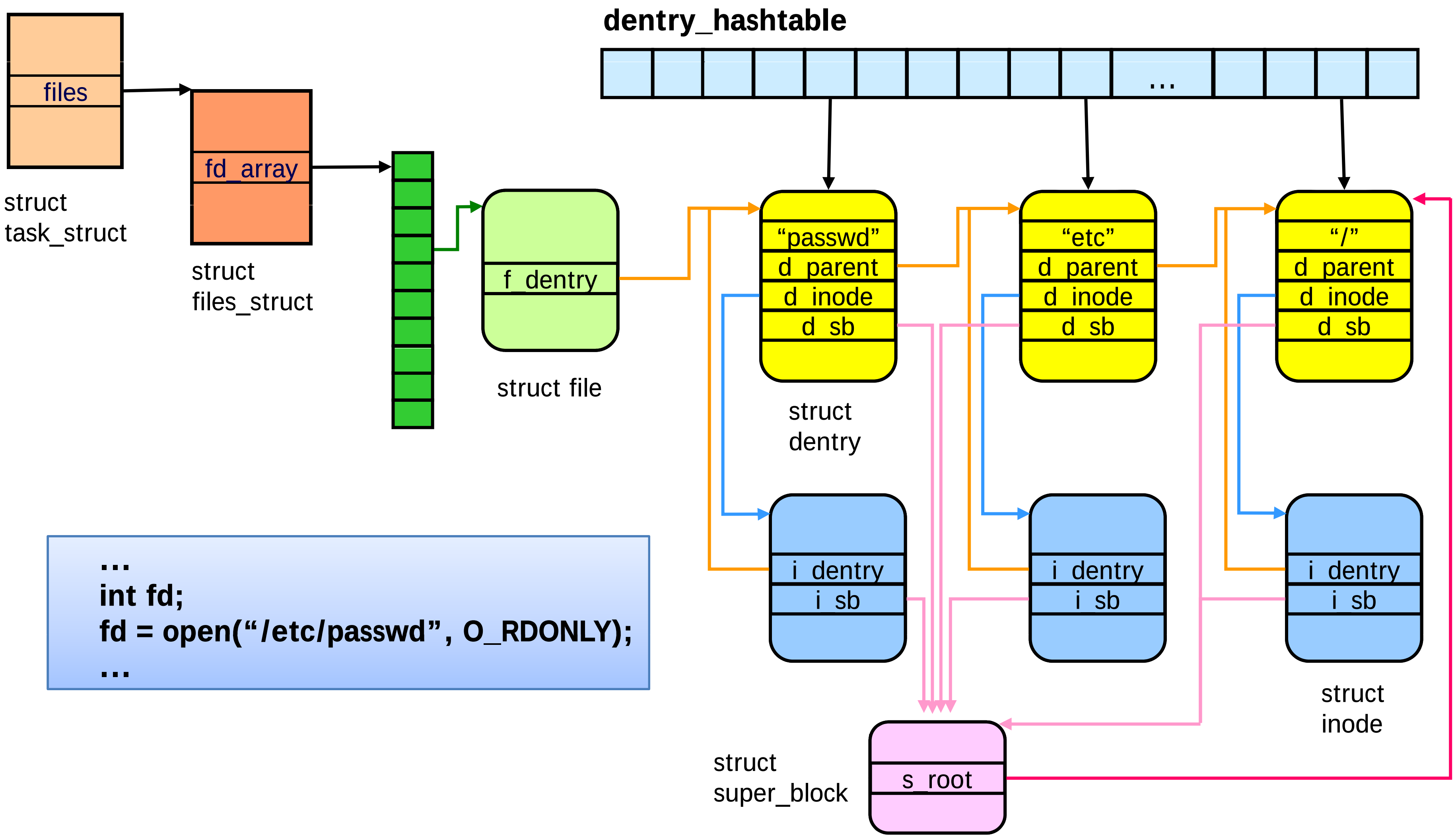
- Hash table: **dentry_hashtable** array
  - Derive the dentry object associated with a given filename and a given directory quickly.
  - Hash table size: 256 entries per megabyte of RAM
  - The adjacent elements associated with a single hash value are linked via **d_hash** field of the dentry object.



Dentry Object (4)

- **struct dentry_operations** <linux/dcache.h>
  - int d_revalidate(struct dentry *dentry, struct nameidata *nd);
  - int d_hash(struct dentry *dentry, struct qstr *name);
    - Create a filesystem-specific hash value for the dentry hash table
  - int d_compare(struct dentry *dentry, struct qstr *name1,
    struct qstr *name2);
  - int d_delete(struct dentry *dentry);
  - void d_release(struct dentry *dentry);
  - void d_iput(struct dentry *dentry, struct inode *inode);
    - Called when a dentry object becomes negative (losing inode).
    - The default VFS function invokes iput() to release the inode object.

The Big Picture




```
…
int fd;
fd = open("/etc/passwd", O_RDONLY);
…
```

CS632/SEP564: Embedded Operating Systems (Fall 2008)

FAT FS

KAIST

FAT FS (1)

- **FAT filesystem**
  - Used in MS-DOS based OSes
    - MS-DOS, Windows 3.1, 95, 98, ...
  - Originally developed as a simple file system suitable for floppy disk drives less than 500KB in size.
  - FAT stands for File Allocation Table.
    - Each FAT entry contains a pointer to a region on the disk
  - Currently there are three FAT file system types: FAT12, FAT16, FAT32

FAT FS (2)

▪ FAT filesystem organization

- FAT file system on disk data structure is all “little endian.”
- The data area is divided into clusters.
  - Used for subdirectories and files.
- Root directory region doesn’t exist on FAT32.

FAT FS (3)

- **Boot sector**
  - The first sector on the disk.
  - Contains BPB (BIOS Parameter Block).
    - Sectors per cluster
    - The number of sectors on the volume.
    - Volume label.
    - The number of root directory entries.
    - File system type (FAT12, FAT16, FAT32)
    - and many more.
  - If the volume is bootable, the first sector also contains the code required to boot the OS.

FAT FS (4)

- **FAT (File Allocation Table)**
  - Starts at sector 1 (after the boot sector)
  - The FAT defines a singly linked list of the clusters of a file.
  - The first two entries in the FAT can be ignored.
    - The first entry available is entry 2.
  - The individual entries in the FAT table define the “chains” of clusters that make up a file.
  - There are two copies so that corruption of the FAT can be detected and repaired.

FAT FS (5)

- **FAT12 example**
  - Each FAT12 entry is 12bits.
    - When designed, space was tight.
    - Pack 2 entries into 3 bytes.
    - 4096 possible clusters.
    - If a sector is 512bytes and cluster = 1 sector, can represent 2MB of data.
  - FAT12 entry values:
    - 0 Unused cluster
    - 0xFF0-0xFF6 Reserved cluster
    - 0xFF7 Bad cluster
    - 0xFF8-0xFFF End of Clusterchain mark
    - Other Next cluster in file

FAT FS (6)

- **Maximum partition size allowed**

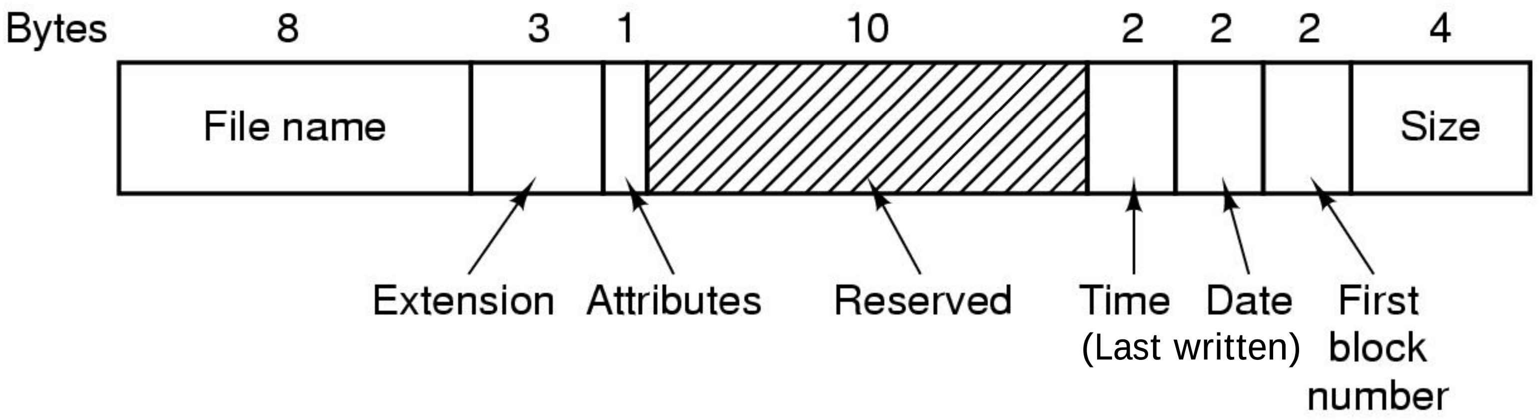
Block size	FAT-12	FAT-16	FAT-32
512 B	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1 GB	2 TB
32 KB		2 GB	2 TB

FAT FS (7)

- **Directories**
  - The root directory is fixed in length and always located at the start of the volume (after the FAT).
    - FAT32 treats the root directory as just another cluster chain in the data area.
  - A subdirectory is nothing but a regular file that has a special attribute indicating it is a directory.
    - No size restriction
  - The data or contents of the "file" is a series of 32byte FAT directory entries.
    - Filename's first character is usage indicator:
      - » 0x00 Never been used.
      - » 0xe5 Used before but entry has been released.

FAT FS (8)

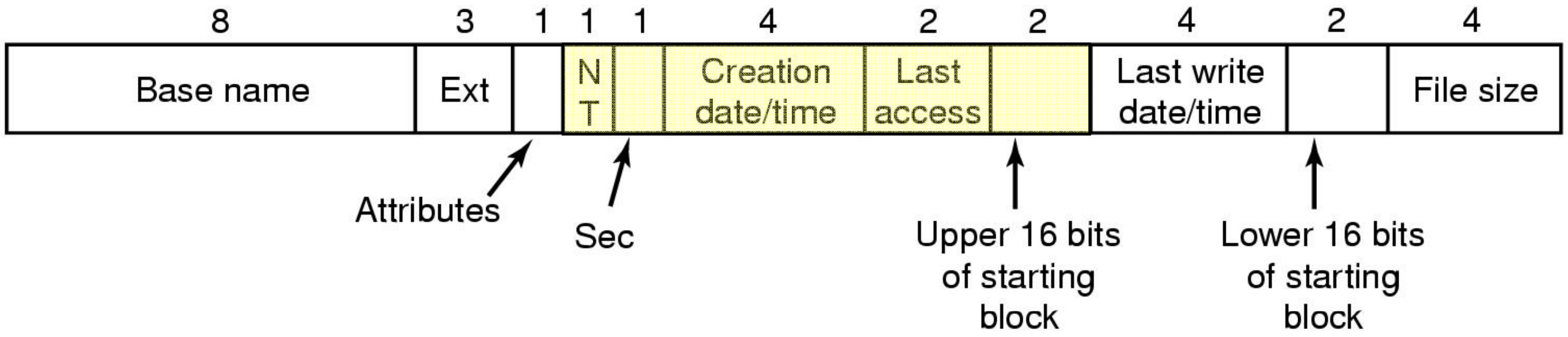
- **FAT directory entry**



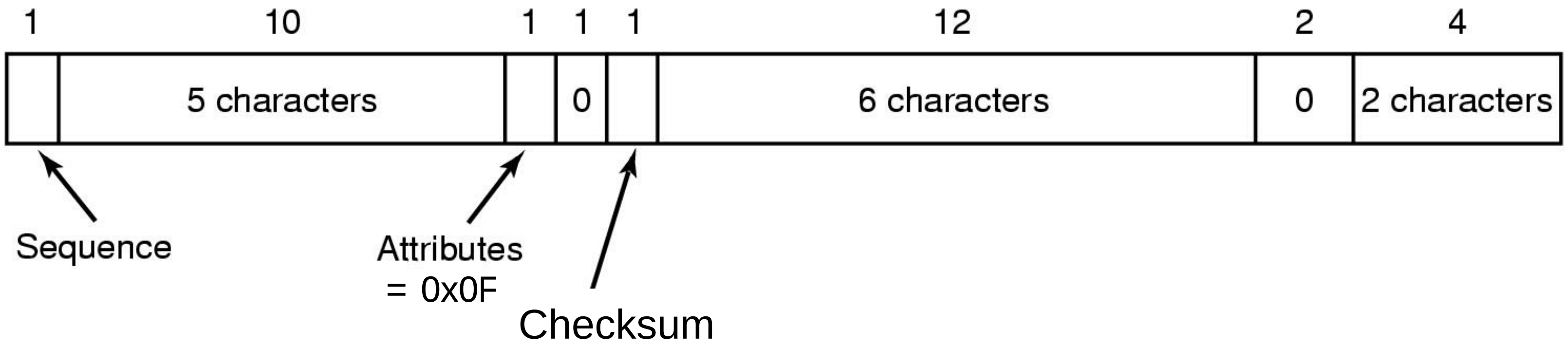
- Attributes:
  - Read Only, Hidden, System, Volume Label, Subdirectory, Archive

FAT FS (9)

- **FAT32 directory entry**



- An entry for a long file name

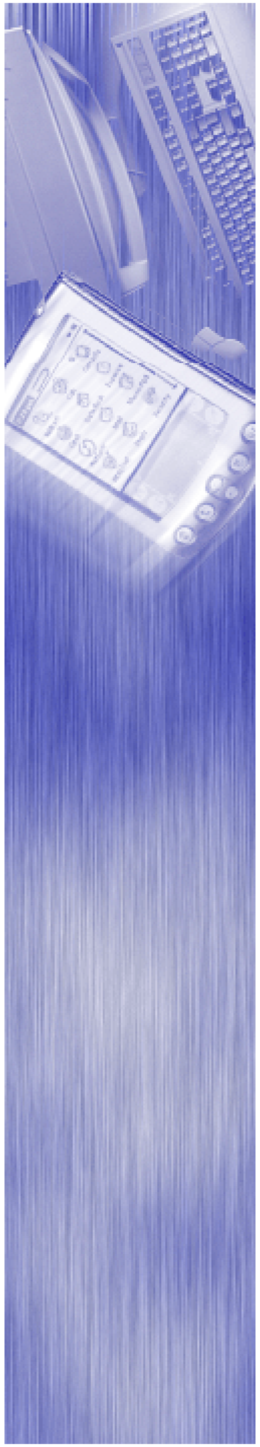


FAT FS (10)

- **Representing long file name in FAT32**
  - The quick brown fox jumps over the lazy dog

68	d o g	A	0	CK		0	
3	o v e r	A	0	CK	t h e l a	0	z y
2	w n f o	A	0	CK	x j u m p	0	s
1	T h e q	A	0	CK	u i c k b	0	r o
T	H E Q U I ~ 1	A	NT	S	Creation time | Last acc | Upp | Last write	Low	Size

Bytes



CS632/SEP564: Embedded Operating Systems (Fall 2008)

CRAMFS

KAIST

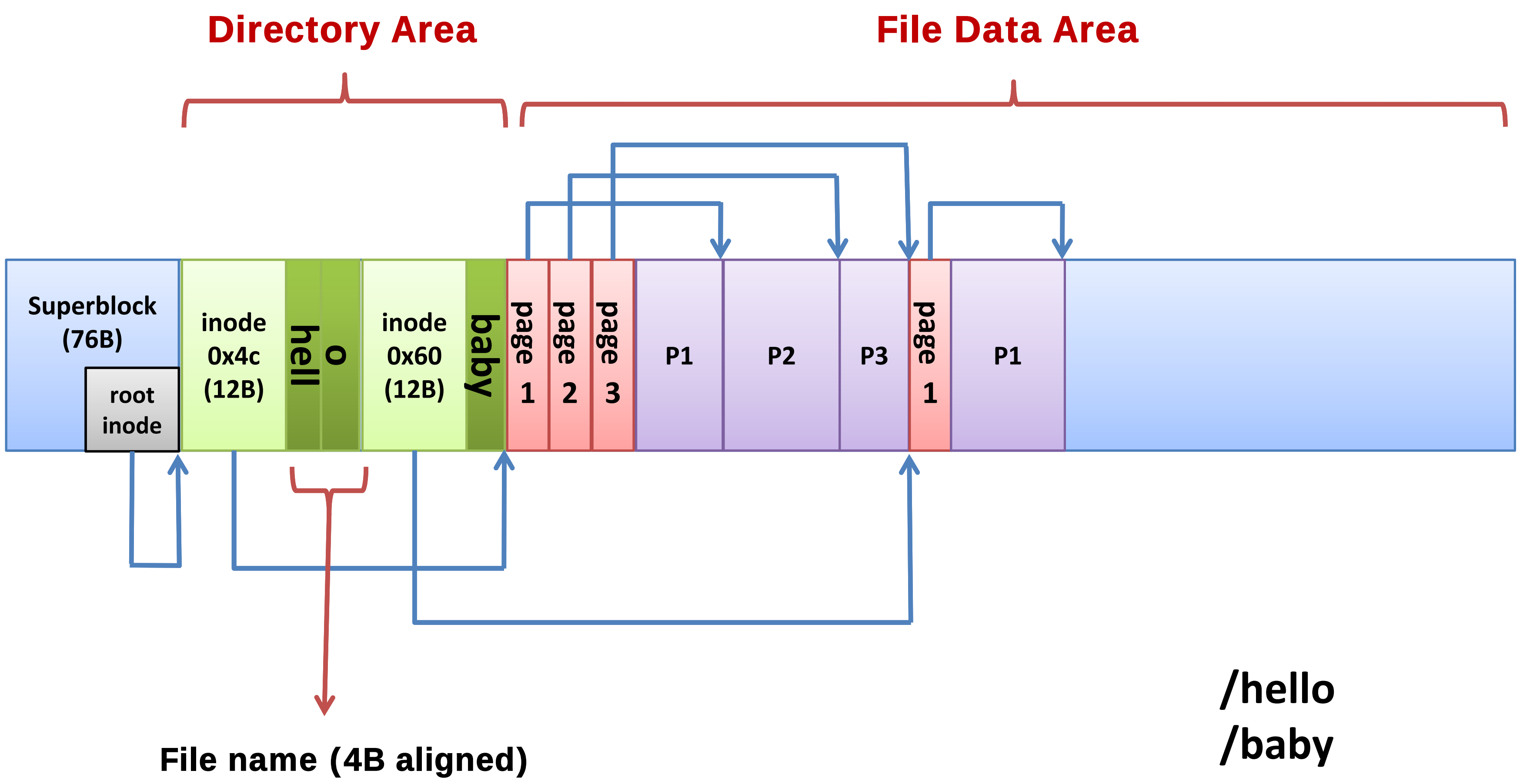
CRAMFS (1)

- **Compressed ROM Filesystem**
  - Designed to be simple and small
  - Uses the zlib routines to compress a file one page at a time
  - Allows random page access
  - The metadata is not compressed, but is expressed in a very terse representation to make it use much less disk space
  - File size is limited to less than 16MB
  - Maximum filesystem size is a little over 256MB
  - Limited metadata: no timestamps, 8-bit gid, …

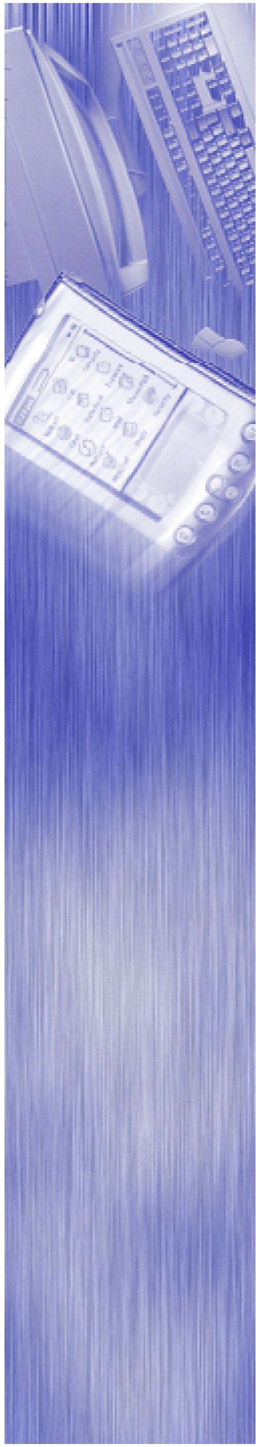
CRAMFS (2)

- **File system layout**
  - Superblock
    - 76 bytes, fixed
  - Directory area
    - One entry for each file or directory
    - Consists of inode (fixed size) + file name (variable size)
    - Offset of the directory entry is used as the inode number
  - File data area
    - Offsets + compressed data chunks
    - All data blocks of a file are stored sequentially

CRAMFS (3)



/hello
/baby



CS632/SEP564: Embedded Operating Systems (Fall 2008)

JFFS/JFFS2

KAIST

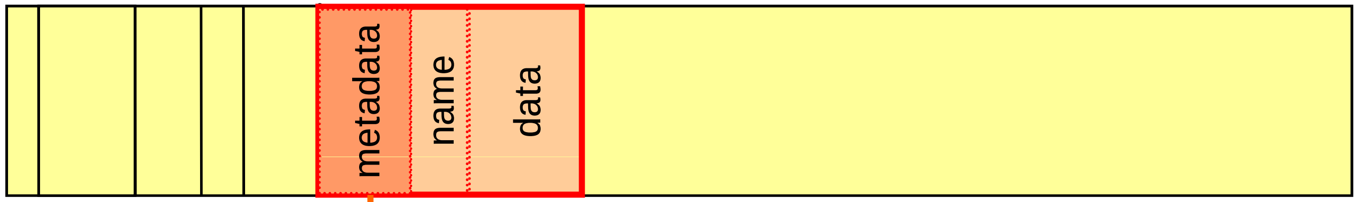
JFFS (1)

- **JFFS (Journaling Flash File Systems)**
  - Developed by Axis Communications, Sweden in 1999.
  - Released under GNU GPL
  - Designed for small NOR flashes
  - A log-structured file system
    - Any file system modification is appended to the log.
    - The log is the only data structure on the flash media.

      Log = <metadata, (name), (data)>
    - A file is obsoleted by a later log in whole or in part.
    - Obsoleted logs are reclaimed via garbage collection.
  - Rely on special in-core data structures for filename→metadata, metadata→data mappings.

JFFS (2)

- **JFFS architecture**

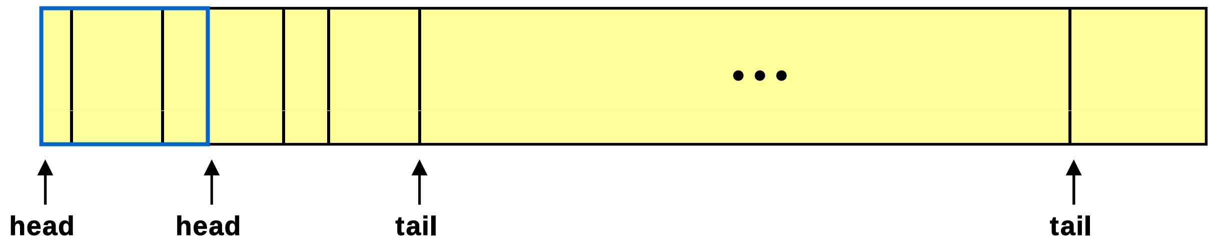


jffs_raw_inode

magic	: magic number
ino	: inode number
pino	: parent inode number
version	: version number
mode	: file's type or mode
uid, gid	: file's owner and group
atime	: last access time
mtime	: last modification time
ctime	: creation time
offset	: where to begin to write
dsize	: size of the node's data
rsize	: how much are going to be replaced?
nsize, nlink, flags	: name length, number of links, flags for rename/deleted/accurate
dchksum	: checksum for the data
nchksum, chksum	: checksums for the name and the raw inode

JFFS (3)

- **Garbage collection**
  - The free space is eventually exhausted. Now what?
  - Erase the oldest block in the log.



  - Live nodes should be moved.
  - Perfectly wear-leveled.

JFFS2 (1)

- **JFFS limitations**
  - Poor garbage collection performance
    - A block is garbage collected even if it contains only clean nodes.
    - In many cases, there are static data. (libraries, program executables, etc.)
  - No compression support
    - Flash memories are expensive.
  - No support for hard links
    - File name and parent i-node are stored in each i-node.
  - No support for NAND flashes

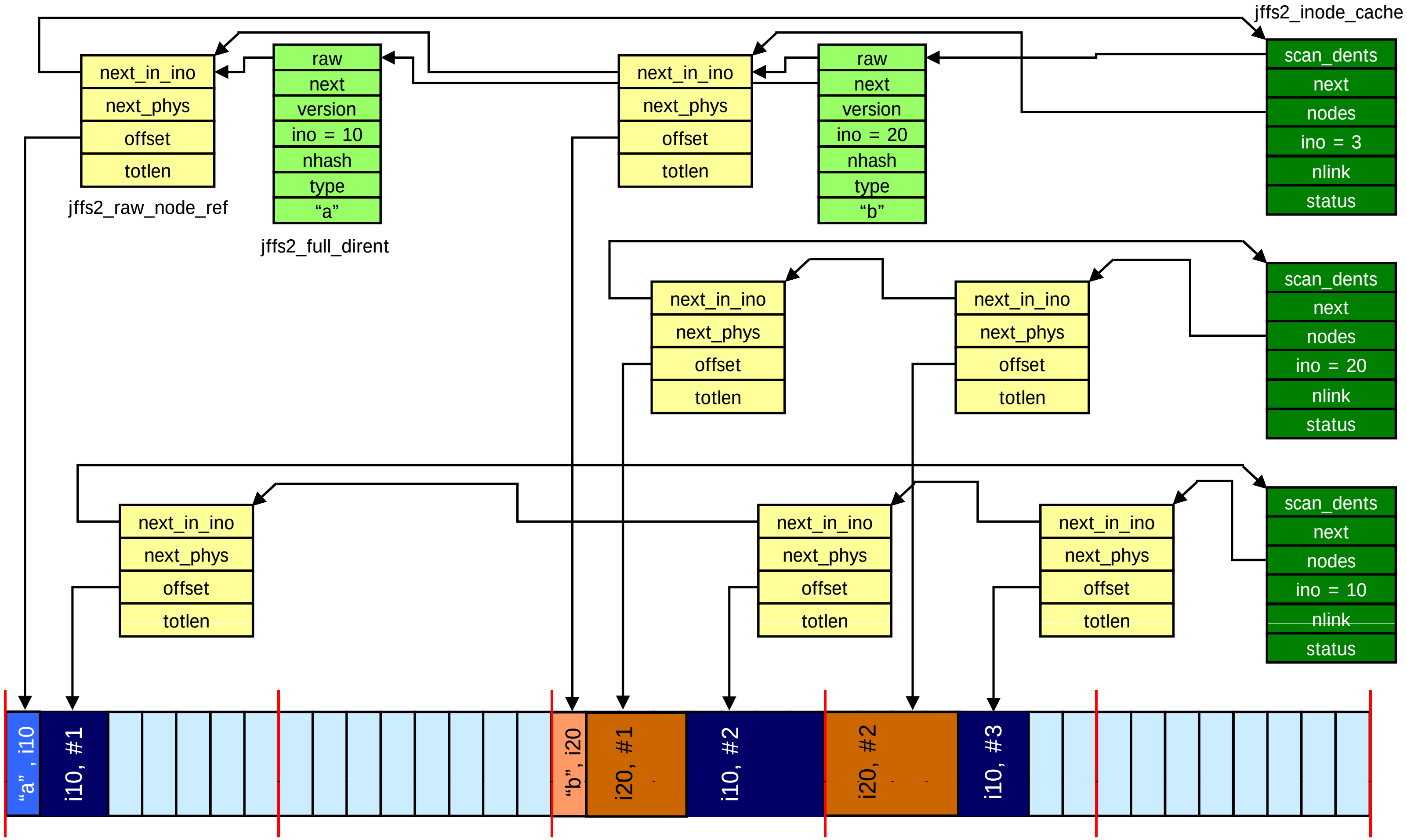
JFFS2 (2)

Node types

- JFFS2_NODETYPE_INODE
  - Similar to jffs_raw_inode
  - No filename, no parent i-node number
  - Compression support
- JFFS2_NODETYPE_DIRENT
  - Represent a directory entry, or a link
  - File name, i-node, parent i-node (directory's i-node), etc.
  - File name with i-node = 0: deleted file
- JFFS2_NODETYPE_CLEANMARKER
  - To deal with the problem of partially-erased blocks due to the power failure during erase operation

JFFS2 (3)

- **JFFS2 architecture**



JFFS2 (4)

- **Block lists (old)**
  - free_list: empty blocks
  - clean_list: blocks full of valid nodes
  - dirty_list: blocks containing at least one obsoleted node
- **Block lists (now)**
  - free_list: empty blocks
  - clean_list: blocks full of valid nodes
  - very_dirty_list: blocks with lots (>50%) of dirty nodes
  - dirty_list: blocks with some (<50%) dirty nodes
  - erasable_list: blocks which are completely dirty
  - …

JFFS2 (5)

- **Garbage collection**
  - Invoked if the size of free_list is less than the threshold.
  - Small nodes can be merged by GC.
  - Which blocks? (old)
    - 99% from dirty_list (jiffies % 100 != 0)
    - 1% from clean_list (for wear-leveling)
  - Which blocks? (now)
    - n = jiffies % 128
    - If (n < 50) GC from erasable_list
    - else if (n < 110) GC from very_dirty_list
    - else if (n < 126) GC from dirty_list
    - else if GC from clean_list

JFFS2 (6)

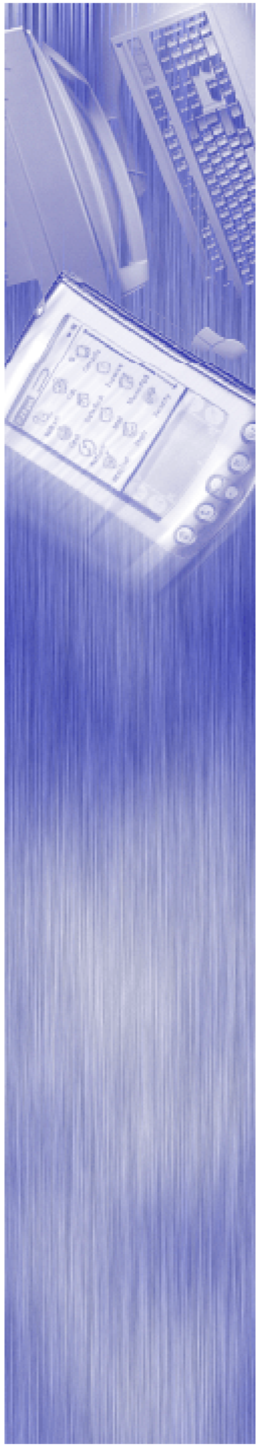
JFFS2 limitations

- Large memory consumption
  - In-core data structures
    - jffs2_raw_node_ref (16bytes/node), jffs2_inode_cache
- Slow mount time
  - 4 sec for 4MB!
- Runtime overheads (space & time)
  - Build child directory entries from flash on directory access
  - Build node fragments on file access
  - All the inode's nodes should be examined (with CRC checked)
- Do not utilize NAND OOB area

JFFS2 (7)

- **JFFS2 memory consumption example**
  - JFFS2 with 64MB NAND flash
    - Typical Linux root FS: 2.2MB
      (719 directories, 2995 regular files)
    - 64MB file with 512bytes/node: 6.7MB
    - 64MB file with 10bytes/node: 47.6MB
  - JFFS2 with 1GB NAND flash (estimated)
    - Typical Linux root FS: 34.7MB
    - 64MB file with 512bytes/node: 104.2MB
    - 64MB file with 10bytes/node: 743.6MB

(Source: JFFS3 Design Issues, June 4, 2005)



CS632/SEP564: Embedded Operating Systems (Fall 2008)

Linux MTD Layer

MTD (1)

- **What is MTD (Memory Technology Device)?**
  - in <linux/drivers/mtd/Kconfig>

```
config MTD
    tristate "Memory Technology Device (MTD) support"
    help
      Memory Technology Devices are flash, RAM and similar chips, often
      used for solid state file systems on embedded devices. This option
      will provide the generic support for MTD drivers to register
      themselves with the kernel and for potential users of MTD devices
      to enumerate the devices which are present and obtain a handle on
      them. It will also allow you to select individual drivers for
      particular hardware and users of MTD devices.
```

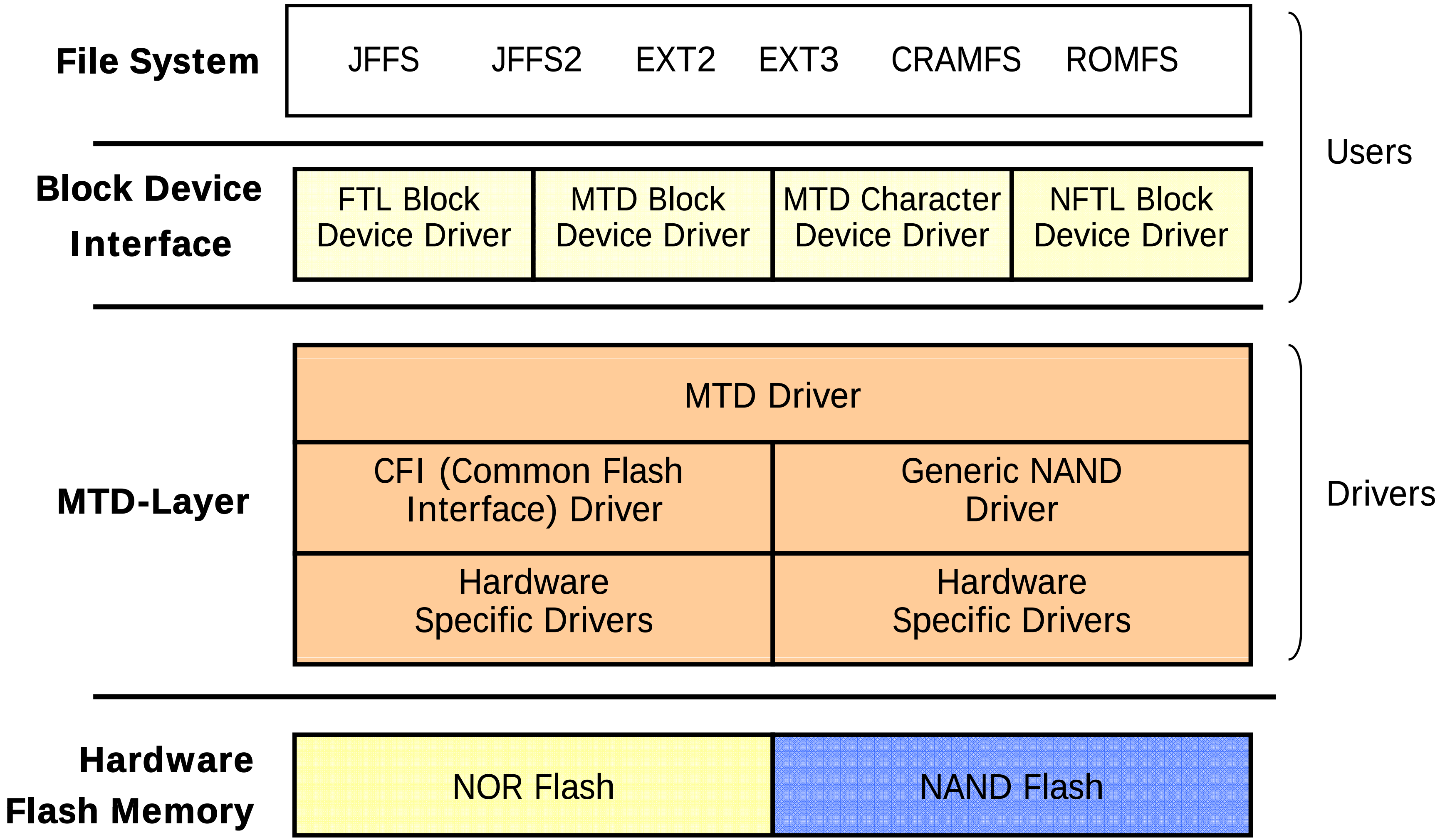
MTD (2)

- **What is MTD? (cont'd)**
  - Generic device driver of memory mapped device
    - Flash memory, ROM, etc.
  - Goals
    - To make it simple to provide a driver for new hardware by providing a generic interface between the hardware drivers and the upper layers of the system.
  - MTD provides an "MTD device" abstraction.
  - Added in Linux kernel 2.4.0
    - /usr/src/linux/drivers/mtd
  - http://www.linux-mtd.infradead.org

MTD (3)

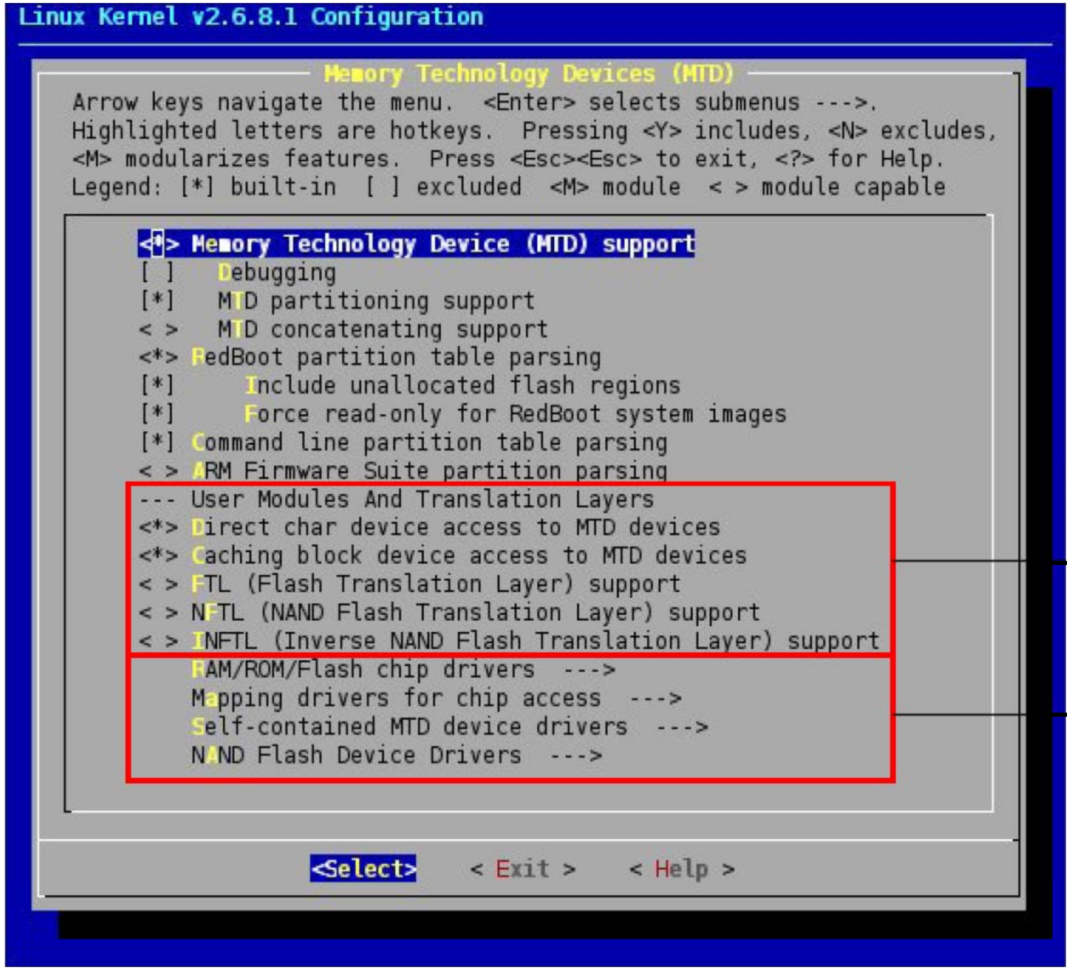
- **MTD user module**
  - Interfaces that can be used directly from userspace
    - Raw character access
    - Raw block access
    - Flash translation layer (FTL, NFTL, INFTL)
    - Flash aware file systems (JFFS2, YAFFS, ...)
- **MTD driver module**
  - Provide physical access to memory device and accessed through the user modules
    - On-board memory, On-board NAND flash
    - Common Flash Interface (CFI) on-board NOR flash
    - M-Systems' DiskOnChip 2000 and Millennium

MTD (4)


File System
JFFS
JFFS2
EXT2
EXT3
CRAMFS
ROMFS
Users
Block Device Interface
FTL Block Device Driver
MTD Block Device Driver
MTD Character Device Driver
NFTL Block Device Driver
MTD-Layer
MTD Driver
CFI (Common Flash Interface) Driver
Generic NAND Driver
Hardware Specific Drivers
Hardware Specific Drivers
Drivers
Hardware Flash Memory
NOR Flash
NAND Flash

Using MTD (1)

- **Kernel configuration**



MTD user module

MTD driver module

Using MTD (2)

- **Module loading**

```
$ modprobe nandsim
$ cat /proc/mtd
dev:    size   erasesize  name
mtd0: 00800000 00002000 "NAND simulator partition"
$ modprobe jffs2
$ lsmod
Module                  Size  Used by
nandsim                22948  0
nand                   32132  2 nandsim
nand_ids                4224  2 nandsim,nand
nand_ecc                2688  1 nand
mtdpart                10496  2 nandsim,nand
jffs2                 100528  1
zlib_deflate           21024  1 jffs2
mtdcore                 7108  4 nand,mtdpart,jffs2
$
$ 
```

Using MTD (3)

- **Mounting**

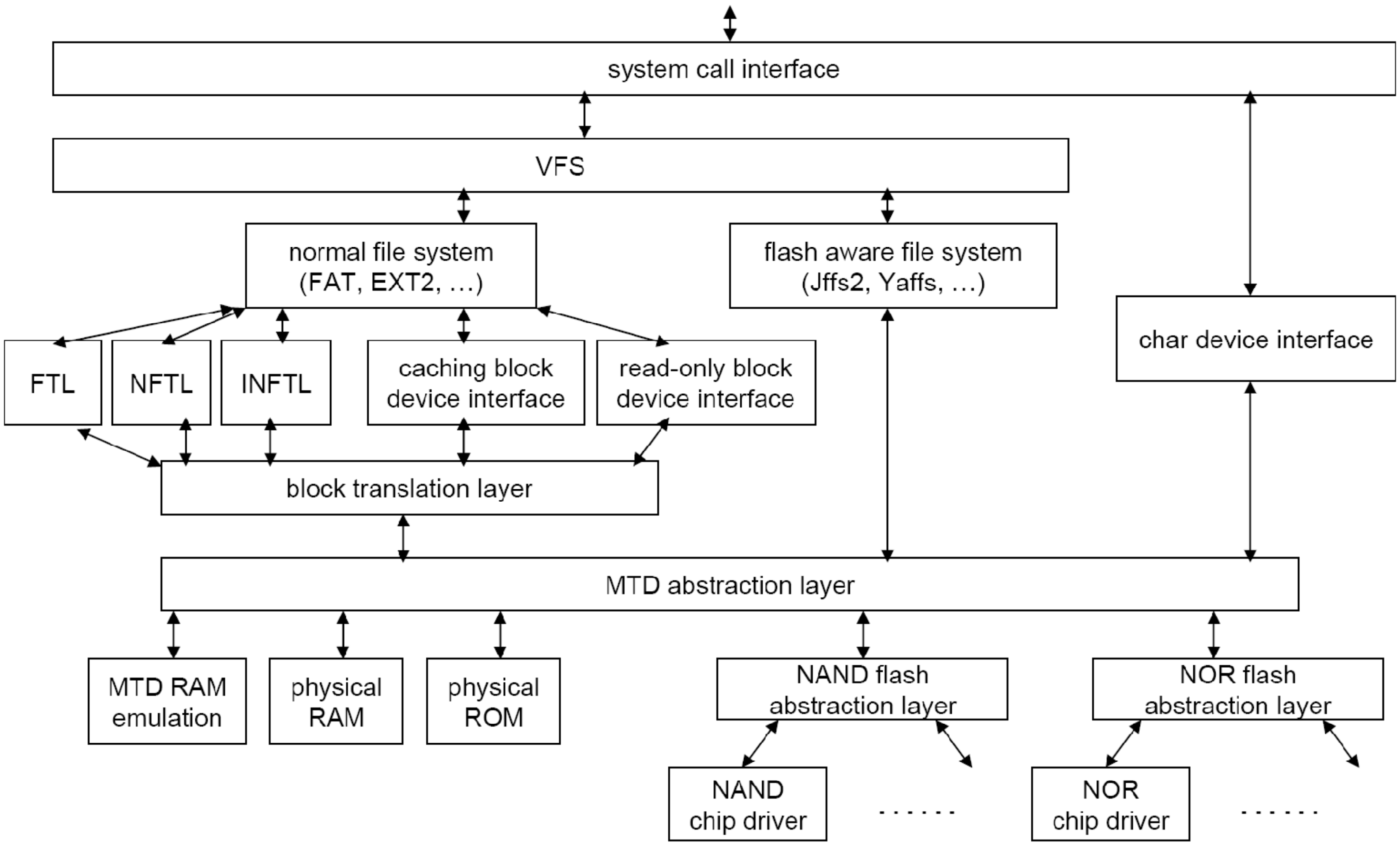
```
$ mknod /dev/mtdblock0 b 31 0
$ mount -t jffs2 /dev/mtdblock0 /mnt/
$ df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/hda2              19G  4.2G   14G  24% /
tmpfs                 506M     0  506M   0% /dev/shm
/dev/hda5              46G   27G   18G  61% /home
/dev/hdc1              22G   21G  1.8G  93% /d
tmpfs                  10M  136K  9.9M   2% /dev
/dev/mtdblock0        8.0M  280K  7.8M   4% /mnt
$ 
```

Using MTD (4)

- **Main interface**
  - 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, u_char *buf);
  - int erase (struct mtd_info *mtd, struct erase_info *instr);

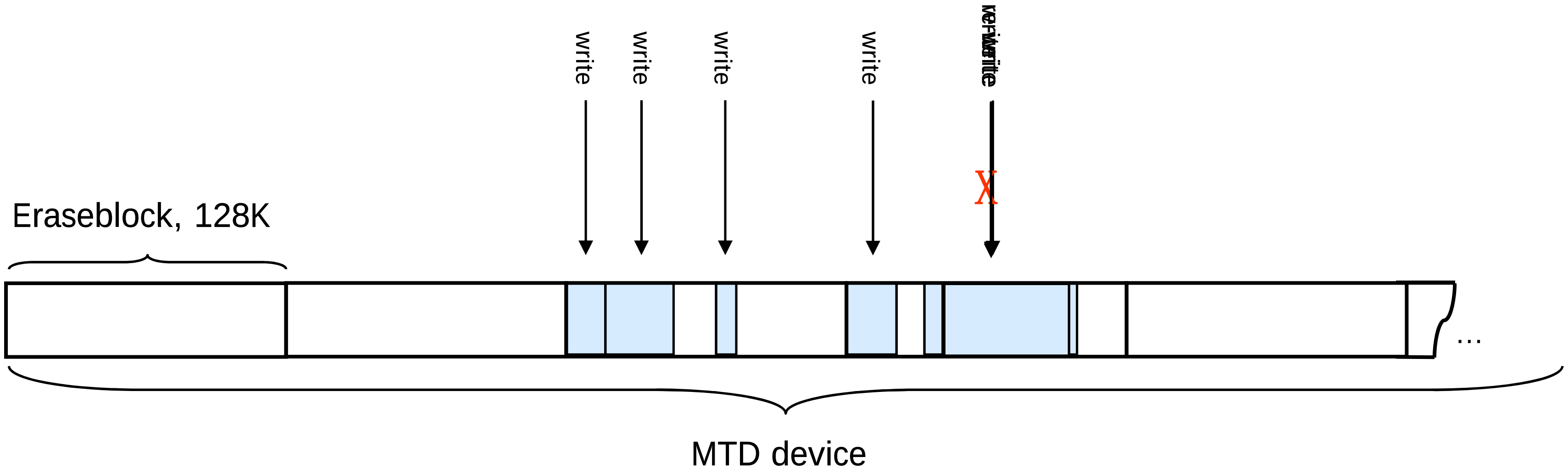
- **Misc. interface**
  - read_ecc(), write_ecc(), read_oob(), write_oob(), ...

MTD Big Picture



MTD Device (1)

- **MTD device**
  - MTD device consists of eraseblocks
  - Eraseblock size varies, typically 32—128 KB
  - Erasesblocks may be written to, but not re-written
  - Whole erase block has to be erased first
  - Then, it is possible to write there



MTD Device (2)

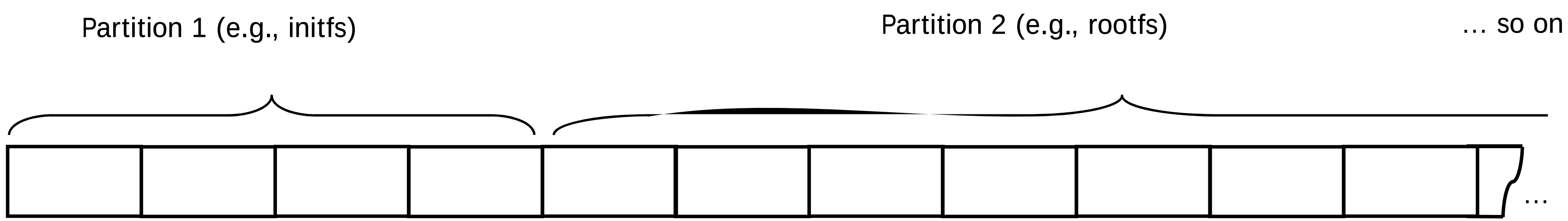
▪ Block device vs. MTD device

Block device	MTD device
• Consists of sectors • Sectors are small (512, 1024 bytes) • 2 operations: **read** and **write** • Bad sectors are hidden by hardware • Sectors do not get worn out	• Consists of eraseblocks • Eraseblocks are larger (32-128Kbytes) • 3 operations: **read**, **write**, and **erase** • Bad eraseblocks are not hidden • Eraseblocks get worn-out after 10^4-10^5 erasures

MTD device is more difficult to handle!

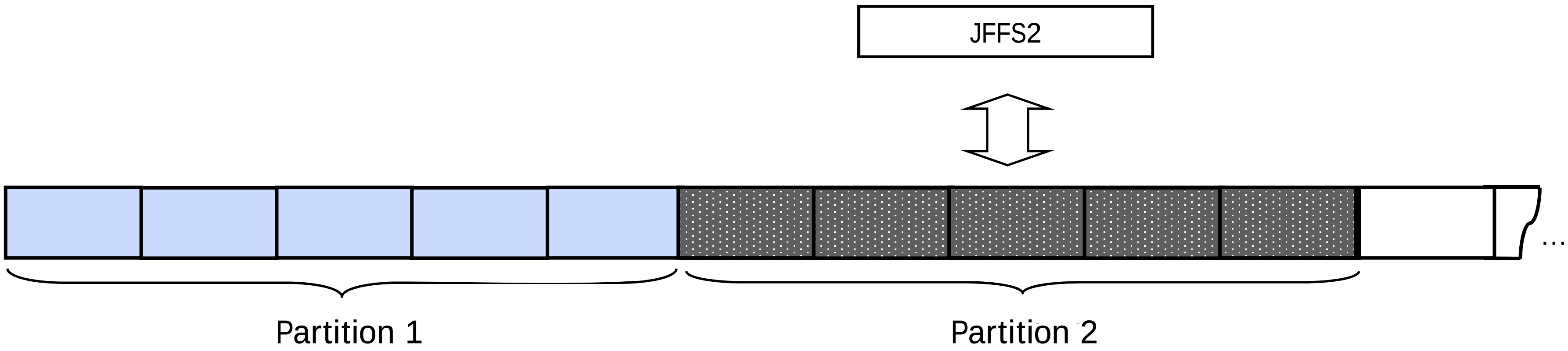
MTD Device (3)

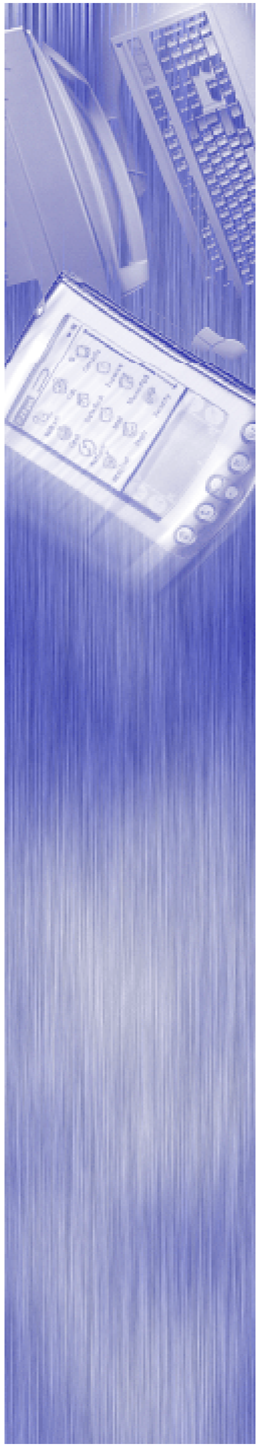
- **MTD partitions**
  - Flash chip may be split on several MTD partitions
  - MTD partition is a set of consecutive eraseblocks
  - MTD partition is a physical flash area



MTD Device (4)

- **Drawbacks of MTD partitions**
  - MTD partitions are static
  - Do not provide wear-leveling for the whole chip





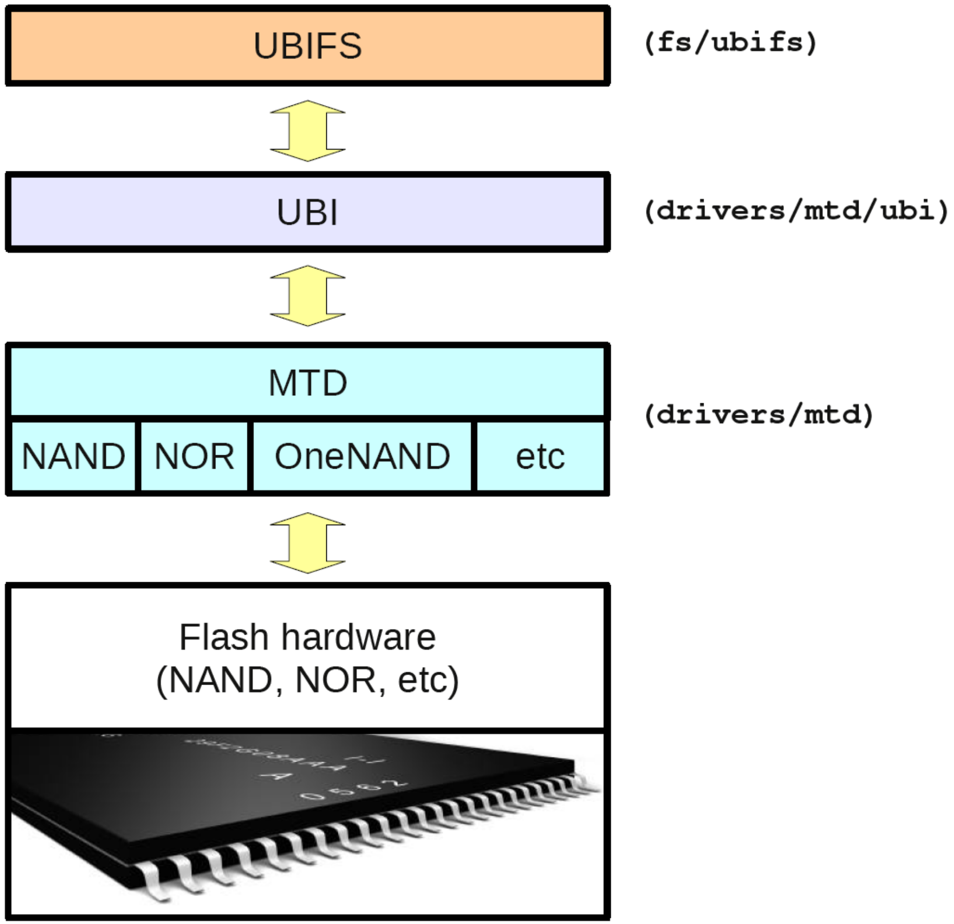
CS632/SEP564: Embedded Operating Systems (Fall 2008)

UBI Layer

Courtesy: Slides borrowed from “UBI - Unsorted Block Images ” by Artem Bityutskiy

UBI (1)

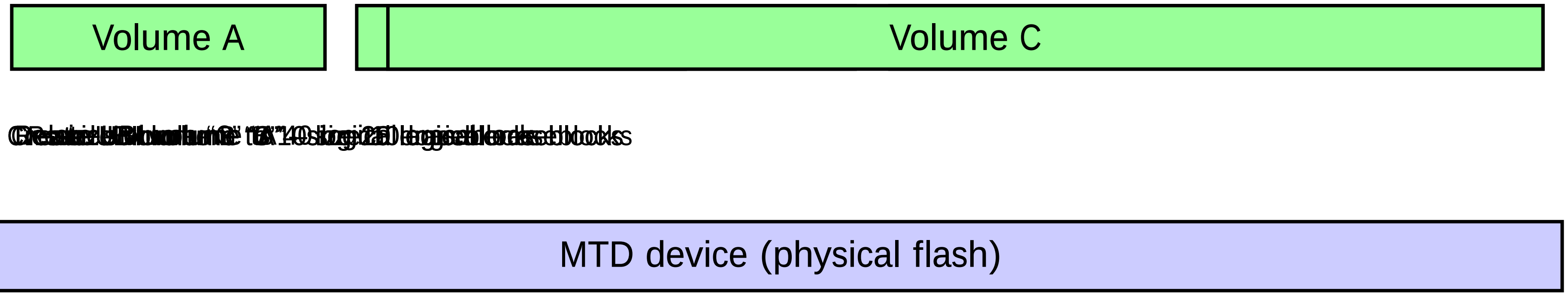
- **Unsorted Block Images**



UBI (2)

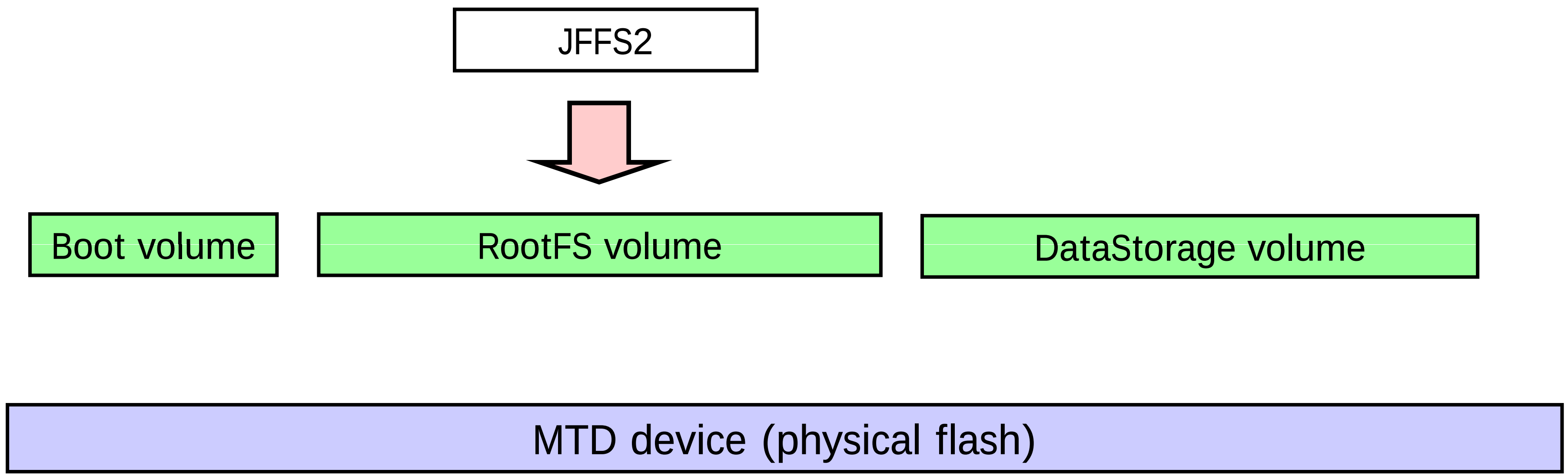
▪ Logical volumes

- UBI provides logical volumes instead of MTD partitions
- UBI volumes are in a way similar to LVM volumes.
- UBI volumes may be dynamically created, deleted and re-sized.



UBI (3)

- **Wear-leveling**
  - UBI does wear-leveling across whole MTD device.
  - Wear-leveling is done by UBI, not by the UBI user.



UBI (4)

▪ UBI volume vs. MTD partition

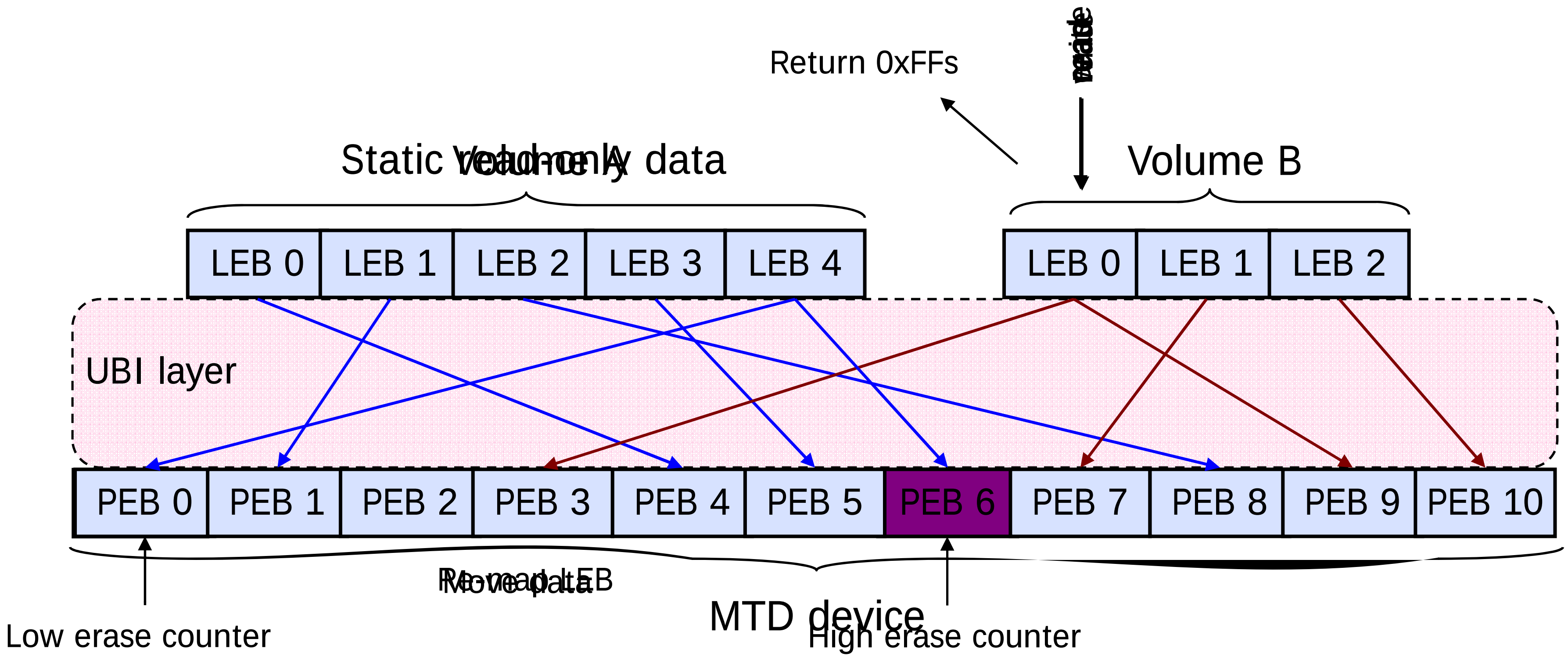
MTD partition	UBI volume
• Consists of physical eraseblocks (PEB) • Does not implement wear-leveling	• Consists of logical eraseblocks (LEB) • Implements wear-leveling
• Admits of bad PEBs	• Devoid of bad LEBs

Advantages of UBI

- Allows dynamic volume creation, deletion and re-sizing ➔ more flexibility
- Eliminates the “wear” problem ➔ simpler software
- Eliminates bad eraseblocks problem ➔ simpler software

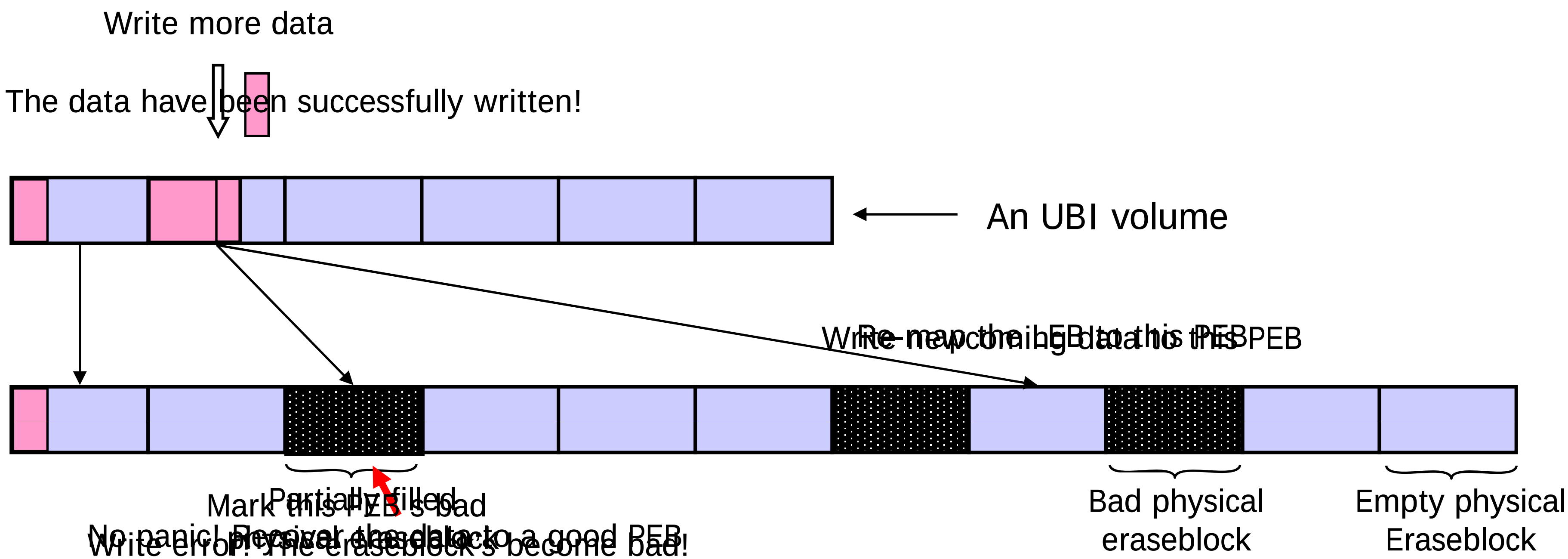
UBI Internals (1)

- **How it works**
  - LEBs are mapped to PEBs
  - Any LEB may be mapped to any PEB



UBI Internals (2)

- **Bad eraseblocks handling**
  - UBI volumes are devoid of bad eraseblocks
  - UBI does proper error recovery transparently



UBI Interfaces

- **UBI character devices**
  - UBI devices: /dev/ubi0, /dev/ubi1, …
    - Volume create, delete, re-size, and get device description operations
  - UBI volumes: /dev/ubi0_0, /dev/ubi0_1, …
    - Read, write, update, and get volume description operations
- **UBI sysfs interface**
  - /sys/class/ubi
- **UBI in-kernel interface**
  - Include/linux/mtd/ubi.h