Operating Systems Design
13. File System Implementation

Paul Krzyzanowski
pxk@cs.rutgers.edu

1

Terms

- Disk
  - Non-volatile block-addressable storage.
- Block = sector
  - Smallest chunk of I/O on a disk
  - Most disks have 512-byte blocks
- Partition
  - Subset of all blocks on a disk. A disk has ≥ 1 partitions
- Volume
  - Disk, disks, or partition that contains a file system
  - A volume may span disks

2

More terms

- Superblock
  - Area on the volume that contains key file system information
- Metadata
  - Attributes of a file, not the file contents (data)
  - E.g., modification time, length, permissions, owner
- inode
  - A structure that stores a file's metadata and location of its data

3

Files

- Contents (Data)
  - Unstructured (byte stream) or structured (records)
  - Stored in data blocks
  - Find a way of allocating and tracking the blocks that a file uses
- Metadata
  - Usually stored in an inode … sometimes in a directory entry
  - Except for the name, which is stored in a directory

4

Directories

- A directory is just a file containing names & references
  - Name → (metadata, data) *Unix (UFS) approach*
  - (Name, metadata) → data *MS-DOS (FAT) approach*
- Linear list
  - Search can be slow for large directories.
  - Cache frequently-used entries
- Hash table
  - Linear list but with hash structure
  - Hash(name)
- More exotic structures: B-Tree, HTree

5

Block allocation: Contiguous

- Each file occupies a set of adjacent blocks
- You just need to know the starting block & file length
- We'd love to have contiguous storage for files!
  - Minimize disk seeks when accessing a file

6

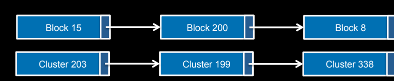
Problems with contiguous allocation

- Storage allocation is a pain (remember main memory?)
  - External fragmentation: free blocks of space scattered throughout
  - vs. Internal fragmentation: unused space within a block (allocation unit)
  - Periodic defragmentation: move files (yuck!)
- Concurrent file creation: how much space do you need?
- Compromise solution: extents
  - Allocate a contiguous chunk of space
  - If the file needs more space, allocate another chunk (extent)
  - Need to keep track of all extents
  - Not all extents will be the same size: it depends how much contiguous space you can allocate

7

Block allocation: Linked Allocation

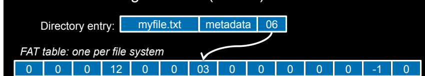
- A file's data is a linked list of disk blocks
  - Directory contains a pointer to the first block of the file
  - Each block contains a pointer to the next block
- Problems
  - Only good for sequential access
  - Each block uses space for the pointer to the next block
- Clusters
  - Multiples of blocks: reduce overhead for block pointer & improve throughput
  - *A cluster is the smallest amount of disk space that can be allocated to a file*
  - Penalty: increased internal fragmentation

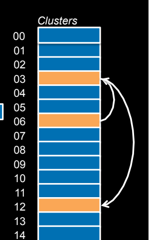


8

File Allocation Table (DOS/Windows FAT)

- Variation of Linked Allocation
- Section of disk at beginning of the volume contains a file allocation table
- The table has one entry per block. Contents contain the next logical block (cluster) in the file.





- FAT-16: 16-bit block pointers
  - 16-bit cluster numbers; up to 64 sectors/cluster
  - Max file system size = 2 GB (with 512 byte sectors)
- FAT-32: 32-bit block pointers
  - 32-bit cluster numbers; up to 64 sectors/cluster
  - Max file system size = 8 TB (with 512 byte sectors)
  - Max file size = 4 GB

9

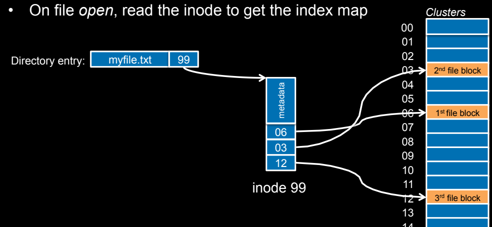
Indexed Allocation (Block map)

- Linked allocation is not efficient for random access
- FAT requires storing the *entire* table in memory for efficient access
- Indexed allocation:
  - Store the entire list of block pointers for a file in one place: the index block (inode)
  - One inode per file
  - We can read this into memory when we open the file

10

Indexed Allocation (block/cluster map)

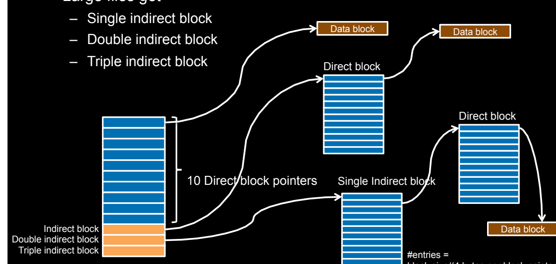
- Directory entry contains name and inode number
- inode contains file metadata (length, timestamps, owner, etc.) *and* a block map
- On file *open*, read the inode to get the index map



11

Combined indexing (Unix File System)

- We want inodes to be a fixed size
- Large files get
  - Single indirect block
  - Double indirect block
  - Triple indirect block



12

Combined Indexing: inside the inode

- Direct block numbers
  - These contain block numbers that contain the file's data
- Indirect block number
  - This is a block number of a block that contains a list of direct block numbers. Each block number is the number of a block that contains the file's data
- Double indirect block number
  - This refers to a block that contains a list of indirect block numbers. Each indirect block number is the number of a block that contains a list of direct block numbers
- Triple indirect block number
  - This refers to a block that contains a list of double indirect block numbers. Each double indirect block number is the number of a block that contains a list of indirect direct block numbers. Each of these contains a list of direct block numbers

13

Example

- Unix File System
  - 1024-byte blocks, 32-bit block pointers
  - inode contains
    - 10 direct blocks, 1 indirect, 1 double-indirect, 1 triple indirect
- Capacity
  - Direct blocks will address: 1K × 10 blocks = 10,240 bytes
  - 1 level of indirect blocks: additional (1K/4)×1K = 256K bytes
  - 1 Double indirect blocks: additional (1K/4) × (1K/4) × 1K = 64M bytes
  - Maximum file size = 10,240 + 256K + 64M = 65,792K bytes

14

Extent lists

- Extents: Instead of listing block addresses
  - Each address represents a range of blocks
  - Contiguous set of blocks
  - E.g., 48-bit block # + 2-byte length (total = 64 bits)
- Why are they attractive?
  - Less block numbers to store if we have lots of contiguous allocation
- Problem: file seek operations
  - Locating a specific location requires traversing a list
  - Extra painful with indirect blocks

15

Directories

- A directory is just a file
  - Name → (metadata, data)
  - or (Name, metadata) → data
- Linear list
  - Search can be slow for large directories
  - Cache frequently-used entries
- Hash table
  - Linear list but with hash structure
  - Hash(name)
- B-Tree
  - Balanced tree (constant depth) with high fan-out
  - Variations include B+ Tree and HTree

16

Implementing File Operations

17

Initialization

- Low-level formatting (file system independent)
  - Define blocks (sectors) on a track
  - Create spare sectors
  - Identify and remap bad blocks
- High-level formatting (file system specific)
  - Define the file system structure
  - Initialize the free block map
  - Initialize sizes of inode and journal areas
  - Create a top-level (root) directory

18

Opening files

Two-step process

1. Pathname Lookup (*namei* function in kernel)
   - Traverse directory structure based on the pathname to find file
   - Return the associated inode
   - (cache frequently-used directory entries)
2. Verify access permissions
   - If OK, allocate in-memory structure to maintain state about access (e.g., that file is open read-only)

19

Writing files

A *write* either overwrites data in a file or adds data to the file, causing it to grow

- Allocate disk blocks to hold data
- Add the blocks to the list of blocks owned by the file
  - Update free block bitmap, the inode, and possibly indirect blocks
  - Write the file data
  - Modify inode metadata (file length)
  - Change current file offset in kernel

20

Deleting files

- Remove name from the directory
  - Prevent future access
- If there are no more links to the inode (disk references)
  - mark the file for deletion
- … and if there are no more programs with open handles to the file (in-memory references)
  - Release the resources used by the file
    - Return data blocks to the free block map
    - Return inode to the free inode list
- Example:
  - Open temp file, delete it, continue to access it
  - OS cleans up the data when the process exits

21

Additional file system operations

- Hard links (aliases)
  - Multiple directory entries (file names) that refer to the same inode
  - inode contains reference count to handle deletion
- Symbolic links
  - File data contains a path name
  - Underlying file can disappear or change
- Access control lists (ACLs)
  - Classic UNIX approach: user, group, world permissions
  - ACL: enumerated list of users and permissions
    - Variable size

22

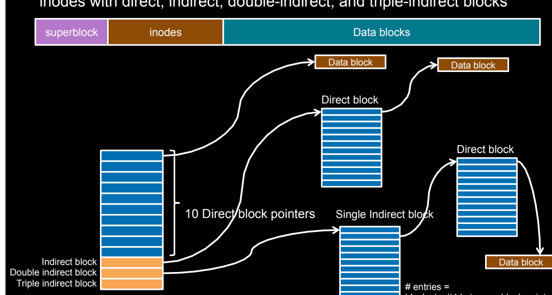
Additional file system operations

- Extended attributes (NTFS, HFS+, XFS, etc.)
  - E.g., store URL from downloaded web/ftp content, app creator, icons
- Indexing
  - Create a database for fast file searches
- Journaling
  - Batch groups of changes. Commit them at once to a transaction log

23

Unix File System (UFS)

inodes with direct, indirect, double-indirect, and triple-indirect blocks



24

Unix File System (UFS)

Superblock contains:

- Size of file system
- # of free blocks
- list of free blocks (+ pointer to free block lists)
- index of the next free block in the free block list
- Size of the inode list
- Number of free inodes in the file system
- Index of the next free inode in the free inode list
- Modified flag (clean/dirty)

25

Unix File System (UFS)

- Free space managed as a linked list of blocks
  - Eventually this list becomes random
  - Every disk block access will require a seek!
- Fragmentation is a big problem
- Typical performance was often:
  2–4% of raw disk bandwidth!

26

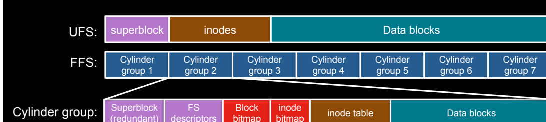
BSD Fast File System (FFS)

- Try to improve UFS
- Improvement #1: Use larger blocks
  - ≥ 4096 bytes instead of UFS's 512-byte or 1024-byte blocks
    - Block size is recorded in the superblock
  - Just doubling the block size resulted in > 2x performance!
  - 4 KB blocks let you have 4 GB files with only two levels of indirection
  - Problem: increased internal fragmentation
    - Lots of files were small
  - Solution: Manage fragments within a block (down to 512 bytes)
    - A file is 0 or more full blocks and possibly one fragmented block
    - Free space bitmap stores fragment data
    - As a file grows, fragments are copied to larger fragments and then to a full block
    - Allow user programs to find the optimal block size
      - Standard I/O library and others use this
  - Also, avoid extra writes by caching in the system buffer cache

27

BSD Fast File System (FFS)

- Improvement #2: Minimize head movement (reduce seek time)
  - Seek latency is usually much higher than rotational latency
  - Keep file data close to its inode to minimize seek time to fetch data
  - Keep related files & directories together
  - Cylinder: collection of all blocks on the same track on all heads of a disk
  - Cylinder group: Collection of blocks on one or more consecutive cylinders



28

How do you find inodes?

- UFS was easy:
  - inodes_per_block = sizeof(block)/sizeof(inode)
  - inode_block = inode / inodes_per_block
  - block_offset = (inode % inodes_per_block) * sizeof(inode)
- FFS
  - We need to know how big each chunk of inodes in a cylinder group is: keep a table

29

BSD Fast File System (FFS)

- Optimize for sequential access

- Allocate data close together
  - Pre-allocate up to 8 adjacent blocks when allocating a block
    - Achieves good performance under heavy loads
    - Speeds sequential reads
- Prefetch
  - If 2 or more logically sequential blocks are read
    - Assume sequential read and request one large I/O on the entire range of sequential blocks
  - Otherwise, schedule a read-ahead

30

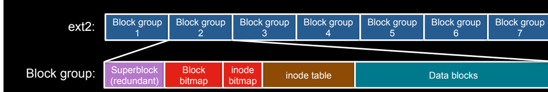
BSD Fast File System (FFS)

- Improve fault tolerance
  - Strict ordering of writes of file system metadata
  - *fsck* still requires up to five passes to repair
  - All metadata writes are synchronous (not buffered)
  - This limits the max # of I/O operations
- Directories
  - Max filename length = 256 bytes (vs. 12 bytes of UFS)
- Symbolic links introduced
  - Hard links could not point to directories and worked only within the FS
- Performance:
  - 14-47% of raw disk bandwidth
  - Better than the 2-5% of UFS

31

Linux ext2

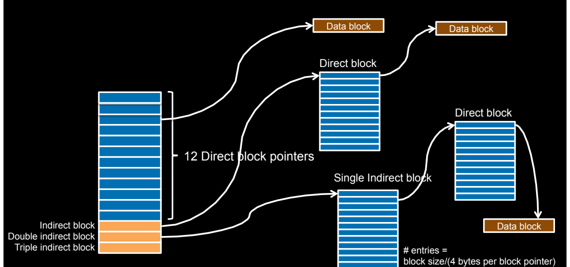
- Similar to BSD FFS
- No fragments
- No cylinder groups (not useful in modern disks) – block groups
- Divides disk into fixed-size block groups
  - Like FFS, somewhat fault tolerant: recover chunks of disk even if some parts are not accessible



32

Linux ext2

inodes with direct, indirect, double-indirect, and triple-indirect blocks



33

Linux ext2

- Improve performance via aggressive caching
  - Reduce fault tolerance because of no synchronous writes
  - Almost all operations are done in memory until the buffer cache gets flushed
- Unlike FFS:
  - No guarantees about the consistency of the file system
    - Don't know the order of operations to the disk: risky if they don't all complete
  - No guarantee on whether a write was written to the disk when a system call completes
- In most cases, ext2 is *much* faster than FFS

34

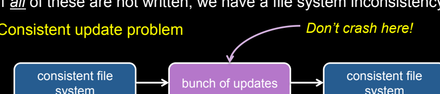
Journaling

35

File system inconsistencies

Example:

- Writing a block to a file may require:
  - inode is
    - updated with a new block pointer
    - Updated with a new file size
  - Data free block bitmap is updated
  - Data block contents written to disk
- If *all* of these are not written, we have a file system inconsistency
- Consistent update problem



36

Journaling

- Journaling = write-ahead logging
- Keep a transaction-oriented journal of changes
  - Record what you are about to do (*along with the data*)

```
Transaction-begin
  New inode 779
  New block bitmap, group 4
  New data block 24120
Transaction-end
```

  - Once this has committed to the disk then overwrite the real data
  - If all goes well, we don't need this transaction entry
  - If a crash happens any time after the log was committed
    Replay the log on reboot (redo logging)
- This is called *full data journaling*

37

Writing the journal

- Writing the journal all at once would be great but is risky
  - We don't know what order the disk will schedule the block writes
  - Don't want to risk having a "transaction-end" written while the contents of the transaction have not been written yet
  - Write all blocks *except* transaction-end
  - Then write transaction-end
- If the log is replayed and a transaction-end is missing, ignore the log entry

38

Cost of journaling

- We're writing everything twice
  …and constantly seeking to the journal area of the disk
- Optimization
  - Do not write user data to the journal
  - Metadata journaling (also called ordered journaling)

```
Transaction-begin
  New inode 779
  New block bitmap, group 4
Transaction-end
```

- What about the data?
  - Write it to the disk first (not in the journal)
  - Then mark the end of the transaction
  - This prevents pointing to garbage after a crash and journal replay

39

Linux ext3

- ext3 = ext2 + journaling (mostly)
- Goal: improved fault recovery
  - Reduce the time spent in checking file system consistency & repairing the file system

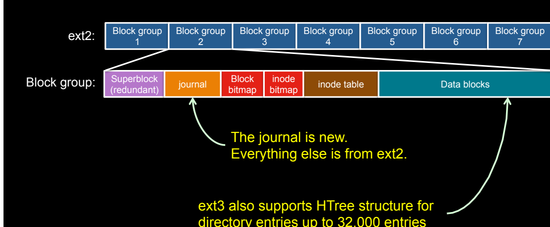
40

ext3 journaling options

- journal
  - full data + metadata journaling
  - [slowest]
- ordered
  - Data blocks written first, then metadata journaling
  - Write a transaction-end only when the other writes have completed
- writeback
  - Metadata journaling with no ordering of data blocks
  - Recent files can get corrupted after a crash
  - [fastest]

41

ext3 layout



42

Linux ext4: extensions to ext3

- Large file system support
  - 1 exabyte (10^{18} bytes); file sizes to 16 TB
- Extents used instead of block maps
  - Range of contiguous blocks
  - 1 extent can map up to 12 MB of space (4 KB block size)
  - 4 extents per inode. Additional ones are stored in an HTree (constant-depth tree similar to a B-tree)
- Ability to pre-allocate space for files
  - Increase chance that it will be contiguous
- Delayed allocation
  - Allocate on flush – only when data is written to disk
  - Improve block allocation decisions because we know the size

43

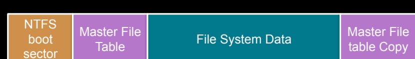
Linux ext4: extensions to ext3

- Over 64,000 directory entries (vs. 32,000 in ext3)
  - HTree structure
- Journal checksums
  - Monitor journal corruption
- Faster file system checking
  - Ignore unallocated block groups
- Interface for multiple-block allocations
  - Increase contiguous storage
- Timestamps in nanoseconds

44

Microsoft NTFS

- Standard file system for Windows; successor to FAT-32
- 64-bit volume sizes, journaling, and data compression
- Cluster-based (file compression not supported on clusters > 4 KB)



Boot Sector: info about layout of the volume & FS structures; Windows bootloader
MFT: contains information about all files in the file system
File system data: all the data that is not in the MFT
MFT Copy: copy of critical part of MFT for recovery (first 4 records)

45

NTFS Master File Table

- The MFT is itself a file (starting at a well-known place)
- It contains file records (inode) for all files, including itself
  - B-Tree structure
- MFT Special files:

MFT record 0	$Mft	Master file table
MFT record 1	$MftMirr	Duplicate of 1st 4 records of MFT
MFT record 2	$LogFile	Metadata journal for recovery
MFT record 3	$Volume	Info about the file system volume
MFT record 4	$AttrDef	Attribute definitions
MFT record 5		Root folder
MFT record 6	$Bitmap	Cluster bitmap (free/used clusters)

And a few more less interesting ones…

- Because the Bitmap is just a file, the volume bitmap is a file, the size of a volume can be easily expanded

46

NTFS MFT & Attributes

- MFT can grow just like any other file
  - To minimize fragmentation, 12.5% of the volume is reserved for use by the MFT ("MFT Zone")
- Each file record is 1, 2, or 4 KB (determined at FS initialization)
- File record info: set of typed attributes
  - Some attributes may have multiple instances (e.g., name & MS-DOS name)
  - Resident attributes: attributes that fit in the MFT record
  - If the attributes take up too much space, additional clusters are allocated
    - an "Attribute List" attribute is added
    - Describes location of all other file records
    - Attributes stored outside of the MFT record are Nonresident attributes

47

NTFS File Data

- File data is an attribute
  - NTFS supports multiple data attributes per file
  - One main, unnamed stream associated with a data file; other named streams are possible
  - Manage related data as a single unit
- Small folders and small data files can fit entirely within the MFT.
  - Large folders are B-tree structures and point to external clusters
- Block allocation: via extents

48

Microsoft NTFS

- Directories
  - Stored as B+ trees in alphabetic order
  - Name, MFT location, size of file, last access & modification times
  - Size & times are duplicated in the file record & directory entry
    - Designed top optimize some directory listings
- Write-ahead logging
  - Writes planned changes to the log, then writes the blocks
- Transparent data compression of files
  - Method 1: compress long ranges of zero-filled data by not allocating them to blocks (sparse files)
  - Method 2: break file into 16-block chunks
    - Compress each chunk
    - If at least one block is not saved then do not compress the chunk

49

The next generation: ReFS for Windows 8

- ReFS = Resilient File System
- Goals
  - Verify & auto-correct data; checksums for metadata
  - Optimize for extreme scale
  - Never take the file system offline – even in case of corruption
  - Allocate-on-write transactional model
  - Shared storage pools for fault tolerance & load balancing
  - Data striping for performance; redundancy for fault tolerance
- General approach
  - Use B+ trees to represent all information on the disk
    - "Table" interface for enumerable sets of key-value pairs
  - Provide a generic key-value interface to implement files, directories, and all other structures

50

Log Structured File Systems

51

NAND flash memory

- Memory arranged in "pages" – similar to disk sectors
  - Unit of allocation and programming
  - Individual bytes cannot be written
- Conventional file systems
  - Modify the same blocks over and over
  - At odds with NAND flash performance
  - Also, optimizations for seek time are useless
- Limited erase-write cycles
  - 100,000 to 1,000,000 cycles
  - Employ wear leveling to distribute writes among all (most) blocks
  - Bad block "retirement"
- Options:
  - Managed NAND = NAND flash + integrated controller
    - Handles block mapping
    - Can use conventional file systems
  - OS file system software optimized for Flash

52

Dynamic wear leveling

- Dynamic wear leveling
  - Monitors high- and low-use areas of the memory
    - At a certain point it will swap high-use blocks with low-use ones
  - Map logical block addresses to flash memory block addresses
  - When a block of data is written to flash memory, it is written to a new location and the map is updated
  - Blocks that are never modified will not get reused
- Static wear leveling
  - Static data is moved periodically to give all blocks a chance to wear evenly

53

File systems designed for wear leveling

- YAFFS2 and JFFS2
  - YAFFS is favored for disks > 64 MB
    - File system of choice for Android
  - JFFS is favored for smaller disks
    - Typically used in embedded systems
  - Only address dynamic wear leveling

54

YAFFS

- Unit of allocation = "chunk"
- Several (32 – 128+) chunks = 1 block
  - Unit of erasure for YAFFS
- Log structure: all updates written sequentially
  - Each log entry is 1 chunk in size:
    - *Data chunk*
    - or *Object header* (describes directory, file, link, etc.)
  - Sequence numbers are used to organize a log chronologically
  - Each chunk contains:
    - Object ID: object the chunk belongs to
    - Chunk ID: where the chunk belongs in the file
    - Byte count: # bytes of valid data in the chunk

55

YAFFS

Create a file

Chunk	ObjectId	ChunkID			
0	500	0	Live	Object header for file (length=0)	Block 1
1					
2					
3					

Adapted from http://www.yaffs.net/files/yaffs.net/HowYaffsWorks.pdf

56

YAFFS

Write some data to the file

Chunk	ObjectId	ChunkID			
0	500	0	Live	Object header for file (length=0)	Block 1
1	500	1	Live	First chunk of data	
2	500	2	Live	Second chunk of data	
3	500	3	Live	Third chunk of data	

Adapted from http://www.yaffs.net/files/yaffs.net/HowYaffsWorks.pdf

57

YAFFS

Close the file: write new header

Chunk	ObjectId	ChunkID			
0	500	0	Deleted	Object header for file (length=0)	Block 1
1	500	1	Live	First chunk of data	
2	500	2	Live	Second chunk of data	
3	500	3	Live	Third chunk of data	
0	500	0	Live	Object header for file (length=*n*)	Block 2

Adapted from http://www.yaffs.net/files/yaffs.net/HowYaffsWorks.pdf

58

YAFFS

Open file; modify first chunk; close file

Chunk	ObjectId	ChunkID			
0	500	0	Deleted	Object header for file (length=0)	Block 1
1	500	1	Deleted	First chunk of data	
2	500	2	Live	Second chunk of data	
3	500	3	Live	Third chunk of data	
0	500	0	Deleted	Object header for file (length=*n*)	Block 2
1	500	1	Live	New first chunk of data	
2	500	0	Live	New object header for file (length=*n*)	

Adapted from http://www.yaffs.net/files/yaffs.net/HowYaffsWorks.pdf

59

YAFFS Garbage Collection

- If all chunks in a block are deleted, the block can be erased & reused
- If blocks have some free chunks
  - We need to do garbage collection
  - Copy active chunks onto other blocks so we can free a block
    - Passive collection: pick blocks with few used chunks
    - Aggressive collection: try harder to consolidate chunks

60

YAFFS in-memory structures

- Construct file system state in memory
  - Map of in-use chunks
  - In-memory object state for each object
  - File tree/directory structure to locate objects
  - Scan the log backwards chronologically
    highest→lowest sequence number
  - Checkpointing: save the state of these structures at unmount time to speed up the next mount

61

YAFFS error detection/correction

- ECC used for error recovery
  - Correct 1 bad bit per 256 bytes
  - Detect 2 bad bits per 256 bytes
  - Bad blocks:
    if read or write fails, ask driver to mark the block as bad

62

JFFS1

- Log-structured file system
  - Nodes containing data & metadata are stored sequentially, progressing through the storage space: circular log
  - Node contains inode number, all metadata, and variable data
  - Each new node contains a version higher than a previous one
  - inode numbers are 32-bits; never reused
  - Old nodes (for data that is covered in later nodes) are "dirty space"
- On mount, the entire file system is scanned to rebuild the directory hierarchy
- Garbage collection: go through dirty space – find unneeded nodes and data blocks that are overwritten

63

JFFS2

- Keep linked lists representing individual erase blocks
  - Most of erase blocks will be on the *clean list* or *dirty list*
- Roughly 99/100 of the time, pick a block from the dirty list. Otherwise, pick a block from the clean list.
  - Optimize garbage collection to rese blocks which are partially obsolete
  - Move data sufficiently so that no one erase block will wear out before the others
- Node types
  - inode: inode metadata and possibly a range of data. Data may be compressed
  - Directory entry: inode number, name, and version
  - Clear marker: newly-erased block may be safely used for storage

64

JFFS2 mounting

- Physical medium is scanned & checked for validity
  - Raw node references are allocated, inode cache structures allocated
- First pass through physical nodes
  - Build full map of data for each inode (and detect obsolete nodes)
- Second pass
  - Find inodes that have no remaining links on the file system and delete them
- Third pass
  - Free temporary info that was cached for each node

65

The End

66

