Operating Systems

14. File System Implementation

Paul Krzyzanowski
Rutgers University
Spring 2015

File System Implementation

File System Design Challenge

How do we organize a hierarchical file system on an array of blocks?

... and make it space efficient & fast?

Directory organization

- 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 complex structures: B-Tree, Htree
  - Balanced tree, constant depth
  - Great for huge directories

Block allocation: Contiguous

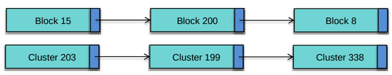
- Each file occupies a set of adjacent blocks
- You just need to know the starting block & file length
- We'd love to have contiguous storage for files!
  - Minimizes disk seeks when accessing a file

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 entire 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

Block allocation: Linked Allocation

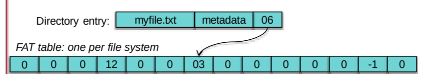
- A file's data is a linked list of disk blocks
  - Directory contains a pointer to the first block of the file
  - Each block contains a pointer to the next block
- Problems
  - Only good for sequential access
  - Each block uses space for the pointer to the next block
- Clusters
  - Multiples of blocks: 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

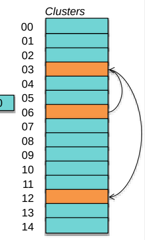


7

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

8

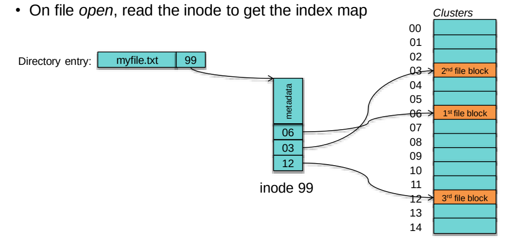
Indexed Allocation (Block map)

- 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

9

Indexed Allocation (block/cluster map)

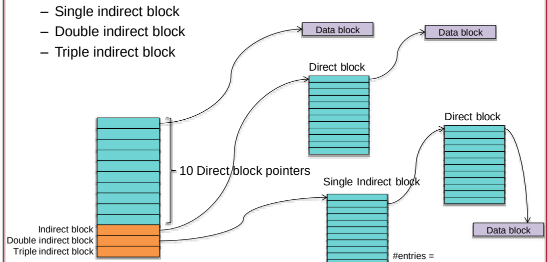
- Directory entry contains name and inode number
- inode contains file metadata (length, timestamps, owner, etc.) *and* a block map
- On file *open*, read the inode to get the index map



10

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



11

Combined Indexing: inside the inode

- Direct block numbers
  - These contain block numbers that contain the file's data. Having these gives us direct access to 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

12

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 Indirect block: additional (1K/4)×1K = 256K bytes
  - 1 Double indirect block: additional (1K/4) × (1K/4) × 1K = 64M bytes
  - 1 Triple indirect block: additional (1K/4) × (1K/4) × (1K/4) × 1K = 16G bytes
  - Maximum file size = 10,240 + 256K + 64M + 16G =
    = 17247250432 bytes ≈ **16G bytes**

13

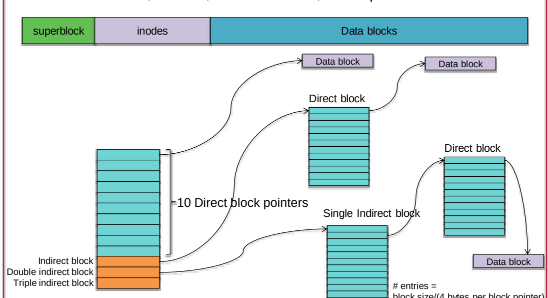
Extent lists

- 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?
  - Fewer 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

14

Unix File System (UFS)

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



15

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)

16

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!

17

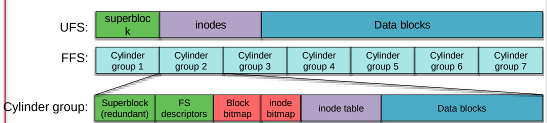
BSD Fast File System (FFS)

- 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

18

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



19

How do you find inodes?

- UFS was easy – to get block # for and inode:
  - 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

20

BSD Fast File System (FFS)

- Optimize for sequential access
- Allocate data blocks that are 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

21

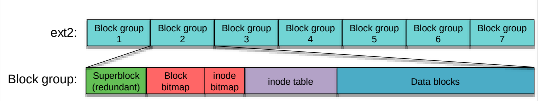
BSD Fast File System (FFS)

- 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

22

Linux ext2

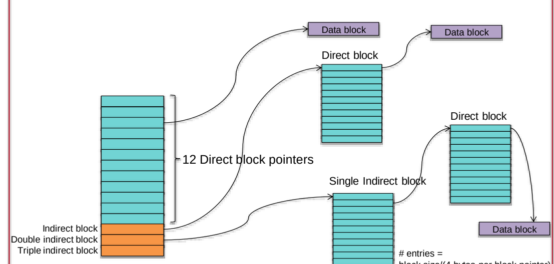
- Similar to BSD FFS
- No fragments
- No cylinder groups (not useful in modern disks) – block groups
- Divides disk into fixed-size block groups
  - Like FFS, somewhat fault tolerant: recover chunks of disk even if some parts are not accessible



23

Linux ext2

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



24

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

25

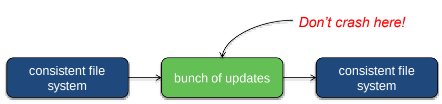
Journaling

26

Consistent Update Problem

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



27

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*

28

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
  - Wait for the *writes* to complete
  - Then write transaction-end
- If the log is replayed and a transaction-end is missing, ignore the log entry

wait for writes to complete

```
jwrite("Transaction-begin")
jwrite("New inode 779")
jwrite("New block bitmap, group 4")
jwrite("New data block 24120")
```

```
jwrite("Transaction-end")
```

29

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

30

Linux ext3

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

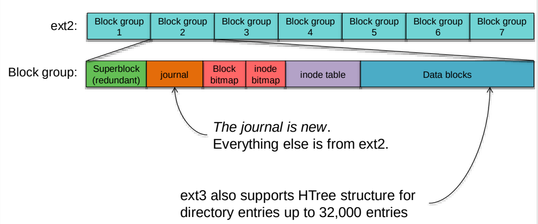
31

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]

32

ext3 layout



33

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: less need for indirect blocks
  - 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

34

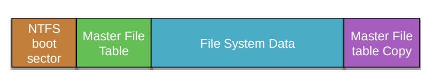
Linux ext4: extensions to ext3

- 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

35

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)

36

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

37

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

38

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

39

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

40

Latest MS file system: ReFS

- ReFS = Resilient File System for Windows Server 2012
- 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

41

The End

3/25/2015 © 2014-2015 Paul Krzyzanowski 42