CSCI 350
Ch. 13 – File & Directory Implementations

Mark Redekopp
Michael Shindler & Ramesh Govindan

Introduction

- File systems primarily map filenames to the disk blocks that contain the file data
- File system can also impact
  - Performance (Seek times for poorly placed blocks)
  - Flexibility (Various access patterns)
    - Sequential, random, many reads/few writes, frequent writes, etc.
  - Consistency/persistence
  - Reliability

Illusions Provided by File System

Physical Storage Device	OS Abstraction
Physical block/sector #'s	File names + directory hierarchy
Read/write sectors	Read/write bytes
No protection/access rights for sectors	File protection / access rights
Possibly inconsistent structure or corrupted data	Reliable and robust recovery

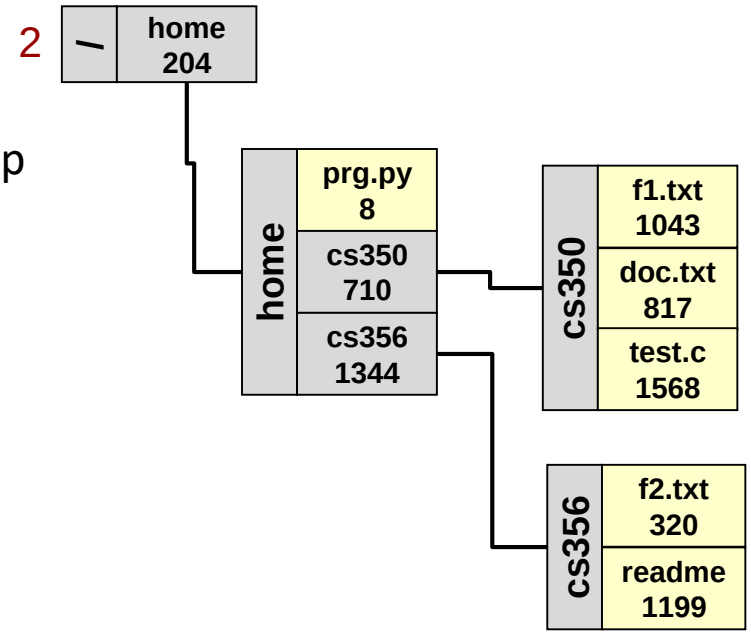
Analogous to VM

- Maintain directories and filenames which map to physical disk blocks
- Keep track of free resources (disk blocks vs. physical memory frames)
  - Usually some kind of bitmap to track free disk blocks
- Locality heuristics *(Look for these in coming slides)*
  - Keep related files physically close on disk (i.e. files in a directory)
  - Keep blocks of a file close (Defragmenting)
  - Use a log structure (sequential writes)

DIRECTORIES

Directory Representation

- Map filenames to **file numbers**
  - **File number**: Unique ID that can be used to lookup physical disk location of a file
- Directory info can be stored in a normal file (i.e. directories are files that contain mappings of filenames to file numbers)
  - Maintain metadata indicating a file is a directory
  - Usually you are not allowed to write these files but the OS provides special system calls to make/remove directories
    - Only the OS writes the directory files. **When would the OS write to a directory file?**
    - Each process maintains the "current working directory"
  - Root directory has a **predefined** (**"well-known"**) file number (e.g. 1)

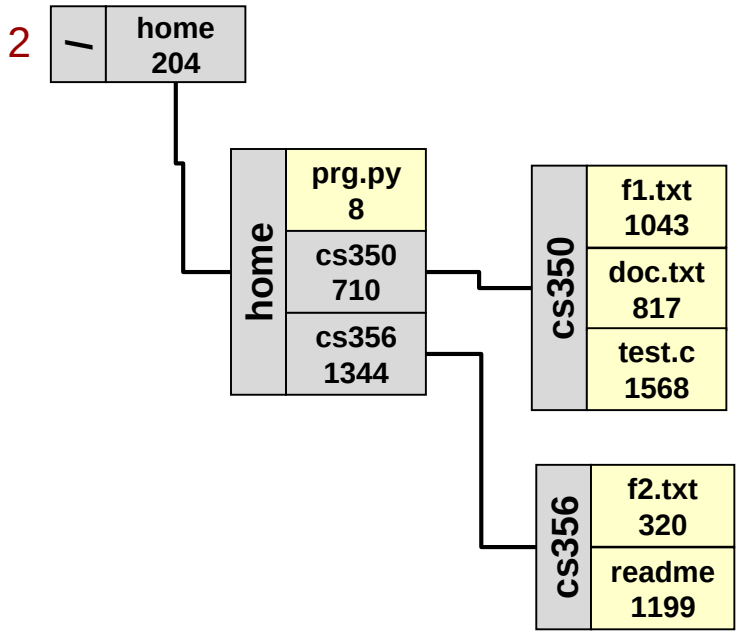


PINTOS Directory-related system calls:

- **bool chdir(const char* dir);**
- **bool mkdir(const char* dir);**
- **bool readdir(int fd, char* name)**
  - **Returns next filename entry in the directory file indicated by fd**
- **bool isdir(int fd);**
  - **Returns true if the file indicated by fd is a directory**

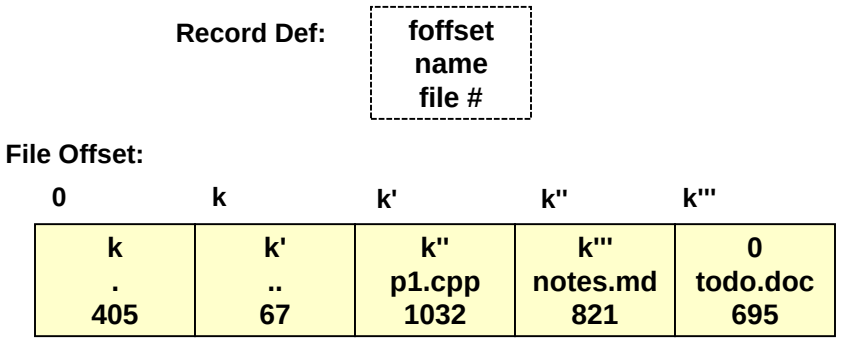
Directory Read Issues

- Problems
  - **A**: Opening a file can require many reads to follow the path (e.g. /home/cs350/f1.txt)
  - **B**: Finding a file in a directory file
    - Directory can have 1000's of files
    - Linear search may be very slow
- Solutions
  - **A**: Caching of recent directory files (often locality in subsequent directory accesses)
  - **B**: Use more efficient data structures to store the filename to file number information



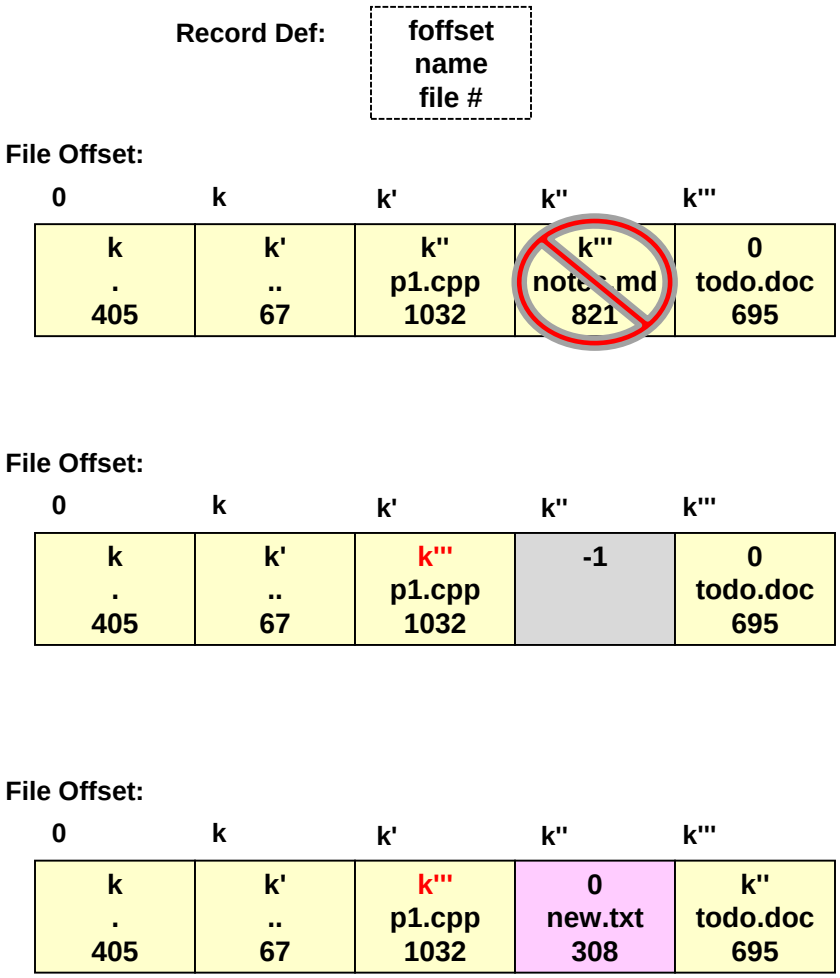
Linear Directory File Layout

- Simplest Approach
  - Linear List
  - Do we really need to store the next file offset?



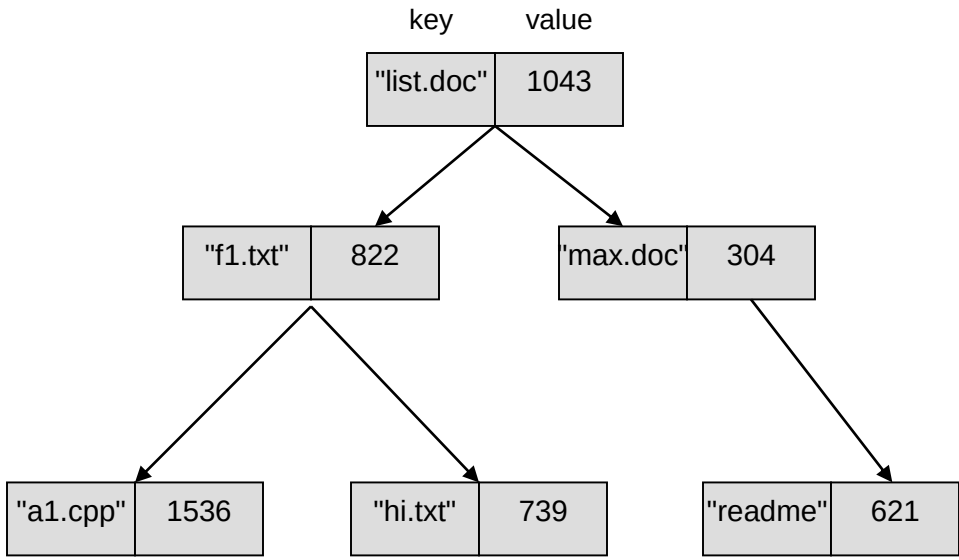
Linear Directory File Layout

- Simplest Approach
  - Linear List
  - Do we really need to store the next file offset?
  - Yes, we may delete files
  - Then, we may create new ones
- Requires linear, O(n), search to find an entry in a directory



Tree Directory File Layout

- Use a more efficient directory file structure
- Could use a balanced binary search tree
  - The "pointers" (arrows) in the diagram would be file offsets to where the child entry starts
  - Jumping to a new offset is likely a different disk block
  - Recall the penalty for non-sequential reads from disk
  - For larger directories walking the tree would be expensive
- Often a B+ Tree is used

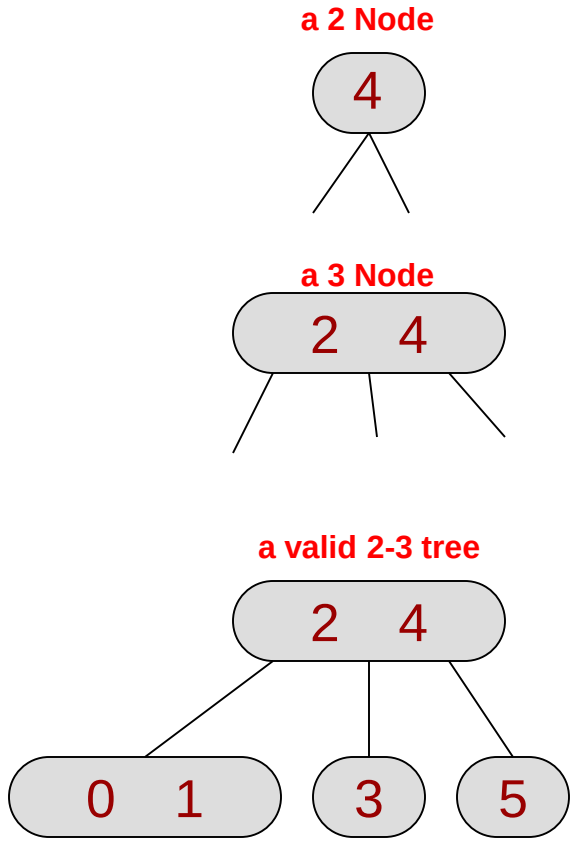


"Interesting" technical look: http://lwn.net/2001/0222/a/dp-ext2.php3

REVIEW OF B-TREES FROM CS104

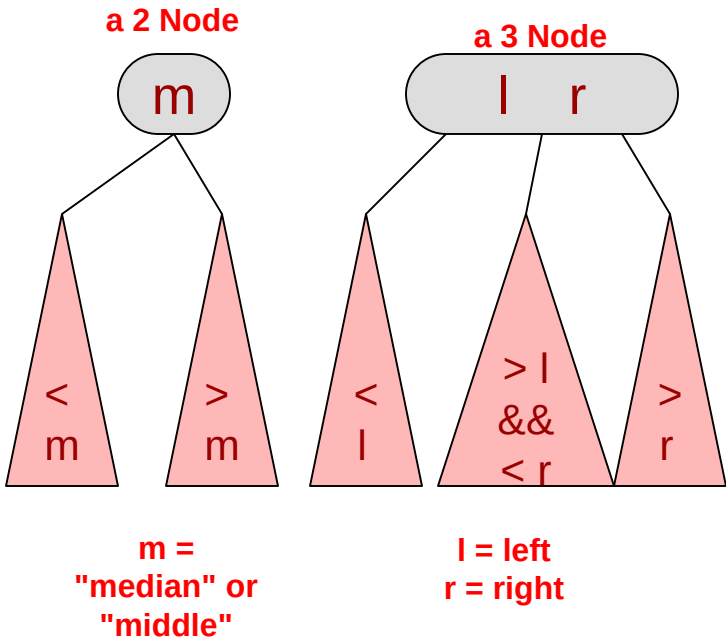
Definition

- B-trees have *d* to *2d* keys and *(d+1)* to *(2d+1)* child pointers
- 2-3 Tree is a B-tree (d=1) where
  - Non-leaf nodes have 1 value & 2 children or 2 values and 3 children
  - All leaves are at the same level
- Following the line of reasoning…
  - All leaves at the same level with internal nodes having at least 2 children implies a **full** tree
    - FULL
  - A full tree with n nodes implies…
    - Height that is bounded by $\log_2(n)$



2-3 Search Trees

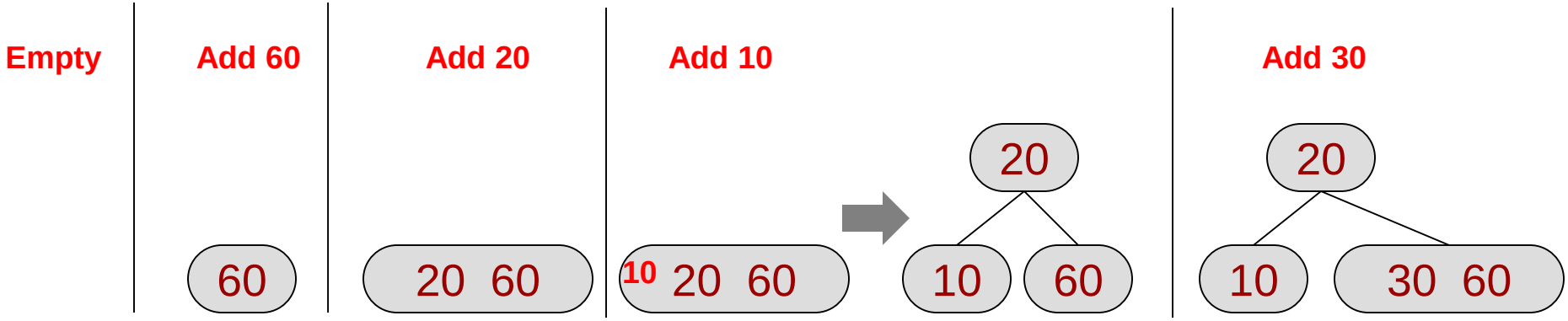
- Similar properties as a BST
- 2-3 Search Tree
  - If a 2 Node with value, *m*
    - Left subtree nodes are < node value
    - Right subtree nodes are > node value
  - If a 3 Node with value, *l* and *r*
    - Left subtree nodes are < *l*
    - Middle subtree > *l* and < *r*
    - Right subtree nodes are > *r*
- 2-3 Trees are almost always used as search trees, so from now on if we say 2-3 tree we mean 2-3 search tree



2-3 Insertion Algorithm

- Key: Since all leaves must be at the same level ("leaves always have their feet on the ground"), insertion causes the tree to "grow upward"
- To insert a value,
  - 1. walk the tree to a leaf using your search approach
  - 2a. If the leaf is a 2-node (i.e.1 value), add the new value to that node
  - 2b. Else break the 3-node into two 2-nodes with the smallest value as the left, biggest as the right, and median value promoted to the parent with smallest and biggest node added as children of the parent
  - Repeat step 2(a or b) for the parent
- Insert 60, 20, 10, 30, 25, 50, 80

Key: Any time a node accumulates 3 values, split it into single valued nodes (i.e. 2-nodes) and promote the median

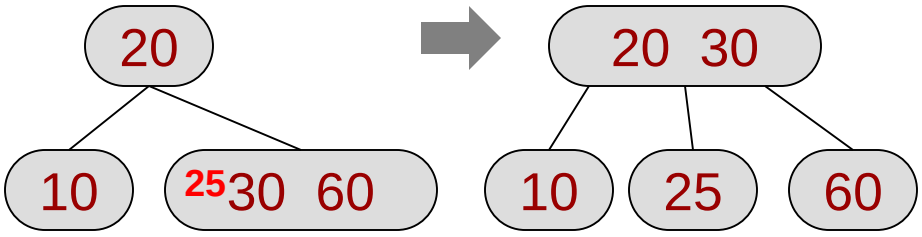


2-3 Insertion Algorithm

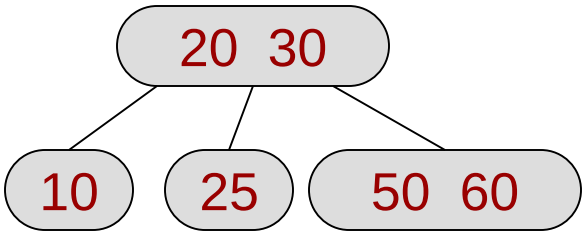
- Key: Since all leaves must be at the same level ("leaves always have their feet on the ground"), insertion causes the tree to "grow upward"
- To insert a value,
  - 1. walk the tree to a leaf using your search approach
  - 2a. If the leaf is a 2-node (i.e.1 value), add the new value to that node
  - 2b. Else break the 3-node into two 2-nodes with the smallest value as the left, biggest as the right, and median value promoted to the parent with smallest and biggest node added as children of the parent
  - Repeat step 2(a or b) for the parent
- Insert 60, 20, 10, 30, 25, 50, 80

Key: Any time a node accumulates 3 values, split it into single valued nodes (i.e. 2-nodes) and promote the median

Add 25



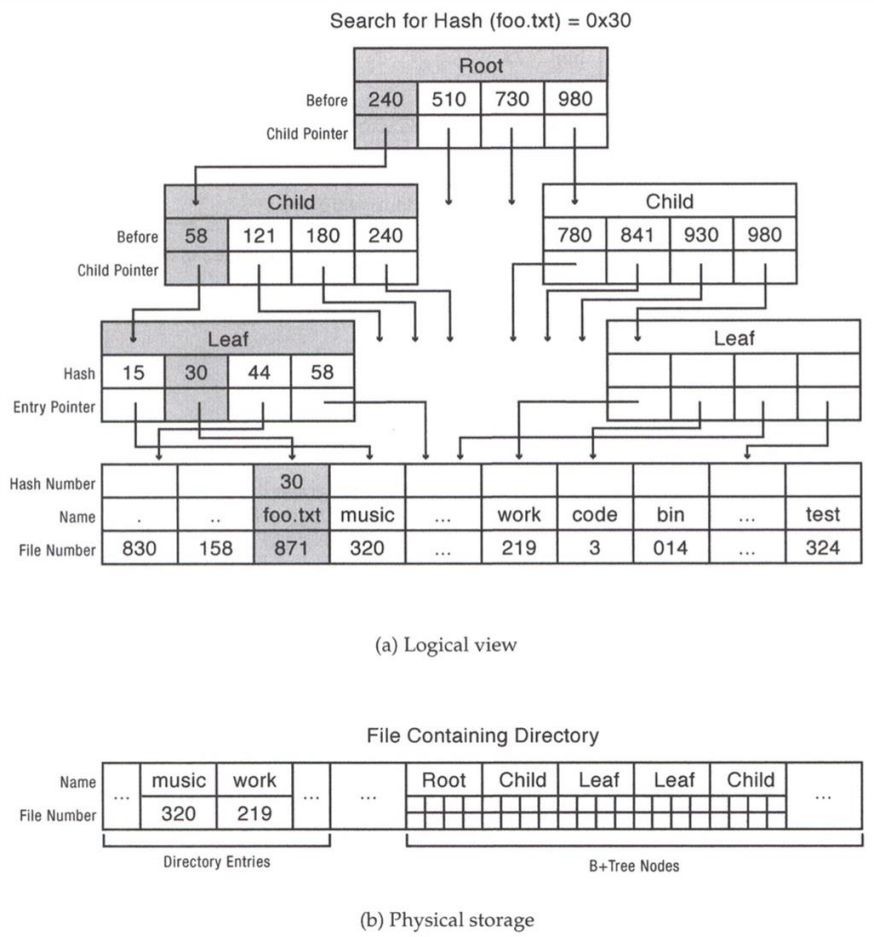
Add 50



BACK TO DIRECTORIES

Tree Directory File Layout

- Use a more efficient directory file structure
- Often a B+ Tree is used
  - Each node holds an array whose size would likely be matched to the disk block size
  - Filename (string) is hashed to an integer
  - Integer is used as a key to the B+ Tree
  - All keys live in the leaf nodes (keys are repeated in root/child nodes for indexing)
  - Leaf nodes of B+ tree store the file offset of where in the directory file that particular file's info/entry is located



"Interesting" technical look: http://lwn.net/2001/0222/a/dp-ext2.php3

Allowing for growth

FILE IMPLEMENTATION

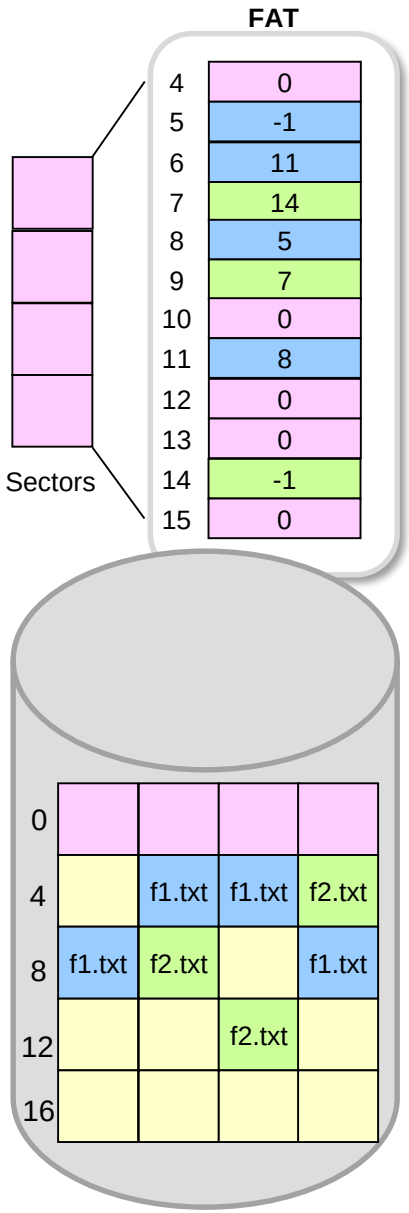
Overview

	FAT	FFS	NTFS	ZFS
Index structure	Linked List	Fixed, asymmetric tree	Dynamic tree	Dynamic, COW tree
Index structure granularity	Block	Block	Extent	Block
Free space management	FAT array	Bitmap	Bitmap in file	Space map (log-structured)
Locality heuristics	Defragmentation	Block groups (reserve space)	Best-fit / defragmentation	Write anywhere (block groups)

MICROSOFT FAT (FAT-32) FILE SYSTEM

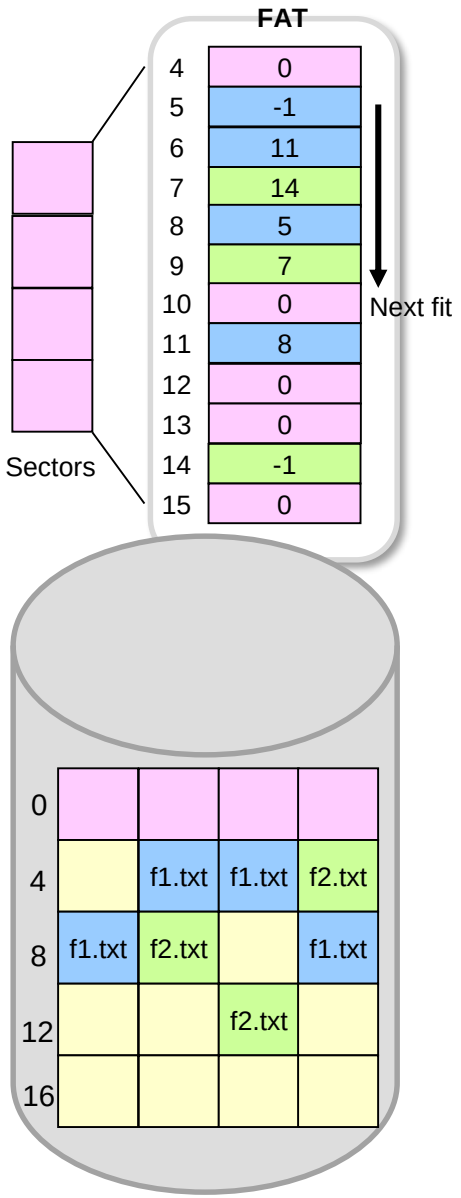
FAT-32

- An array of entries (1 per available block on the volume)
  - Stored in some well-known area/sectors on the disk
- Array entries specify **both** *file structure* and *free-map*
  - If FAT[i] is NULL (0), then block i on the disk is available
  - If FAT[i] is non-NULL and for all j, FAT[j] != i, then block i is the starting point of a file and FAT[i] is the next block in the file
  - A special value (usually all 1's = -1 = 0x?fffffff) will be used to indicate the of the chain (last block of a file)



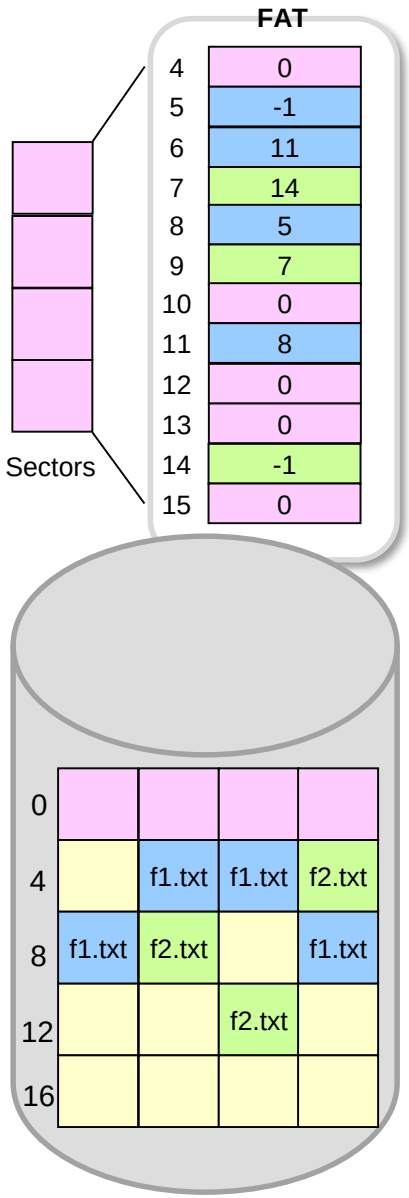
FAT-32

- How do you grow a file?
  - Use simple approaches like next fit (next free block starting from the last block of the file)
    - If we add to f1.txt block 10 would be selected
  - Recall sequential access is fastest for disks. What performance issues are likely to arise? How can they be mitigated?
- How do you know where a file starts?
  - Recall that is what directories are for though the previous slide provides the method



FAT-32

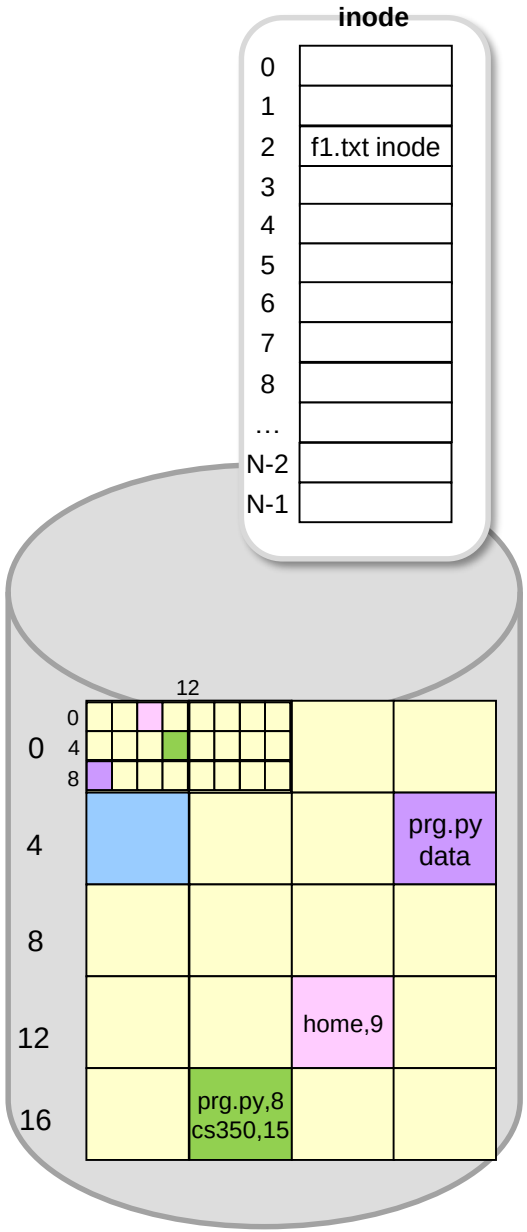
- Other FAT-32 issues
  - Limited metadata and access control
  - No support for hard links
  - File size (stored in metadata) is limited to 32-bits limiting file size to ____?
  - Each FAT-32 entry uses 28-bits for the next block pointer, limiting the FAT to ___ entries
  - If each disk block corresponds to 4KB then the max volume size is ____?
  - Note: Block size can be chosen. Is bigger better?
- Still used in many simple devices
  - Flash-based USB drive, camera storage devices, etc.
  - FAT approach is mimicked in some file formats (.doc)
    - 1 document is made of many objects and the objects are tracked using a FAT like system



UNIX FFS (FAST FILE SYSTEM)

inodes

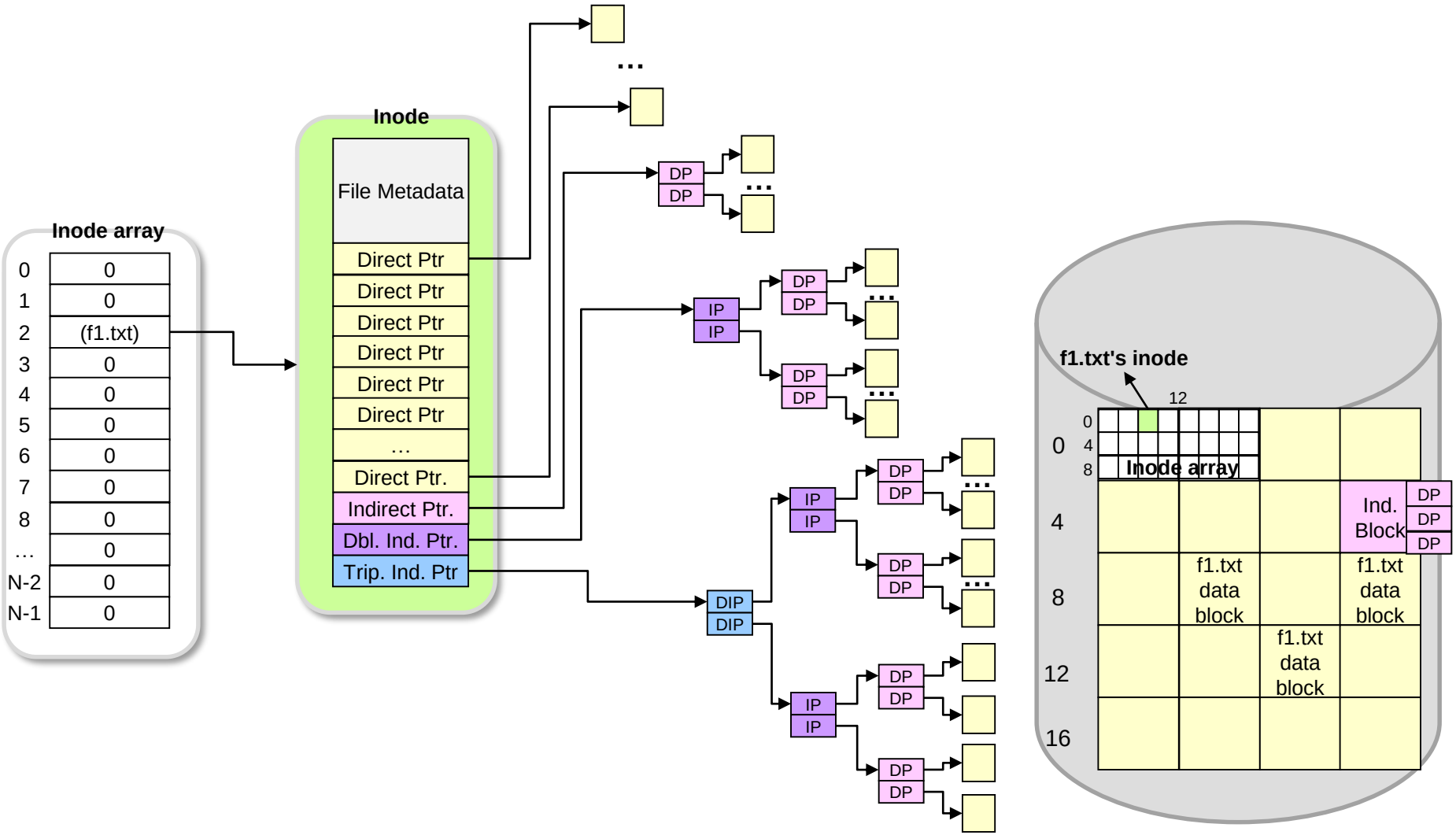
- Inode = (index node?)
- 1 inode = 1 file
  - inode contains file metadata and location of the files data blocks
  - Number of inodes often set when drive is formatted
- Inodes may be stored in an array at some well-known location on the disk
- Directories map filenames to file numbers which is simply the inode number (index in the inode array)
  - If "f1.txt" has file number 2 then f1.txt's inode is at index 2 in the inode array



inode's File to Block Mapping

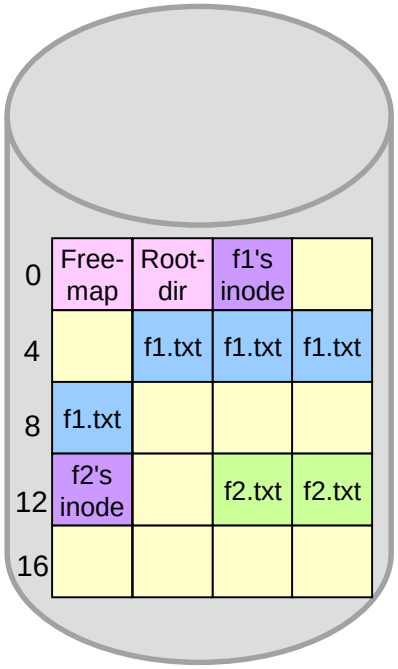
- Rather than using a linked list like FAT-32, FFS uses a **multilevel index**
  - Really a *fixed-depth, asymmetric tree*
  - The leaves of the tree are the data blocks of the file
  - The root is the inode
  - Internal nodes provide indirection to support ever-larger file sizes
    - Internal nodes are usually whole sectors or blocks that store many pointers

Inode Multi-level Tree Index



Pintos Base Filesys Implementation

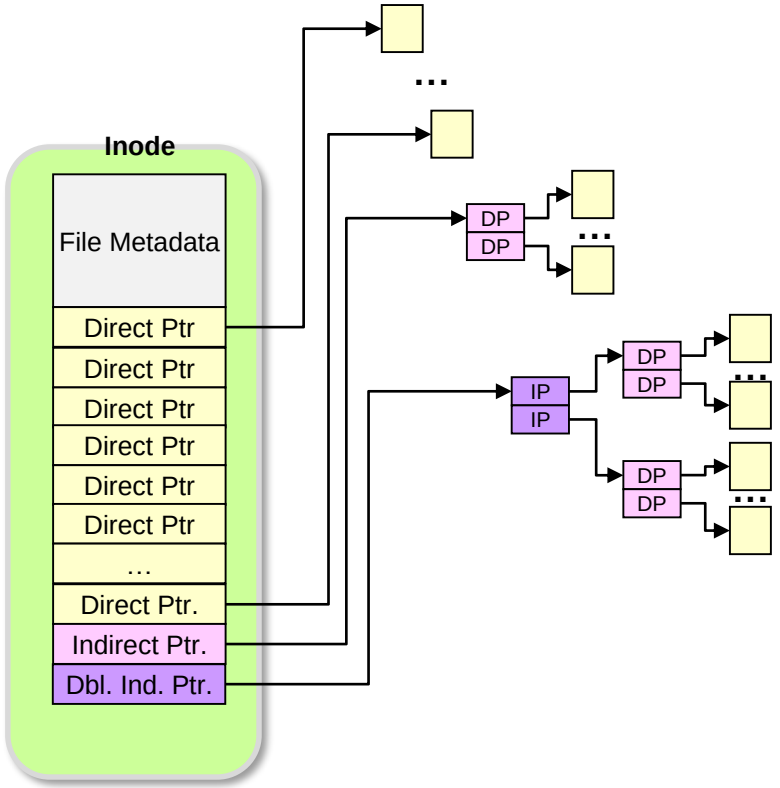
- Uses an "extent" based approach where a file's size is fixed at creation time and must occupy a **contiguous** range of disk blocks
- Inode is used but simply stores the file's size and has one "direct" pointer to the starting block of the file
- Inode occupies an entire sector, thus that sector's number becomes the "inode"/file number
  - In the illustration, the root-dir would have entries {f1.txt,2} and {f2.txt,12}
  - f1.txt's inode at sector 2 would indicate the file's size and "point to" sector 5



Pintos inodes occupy an entire sector

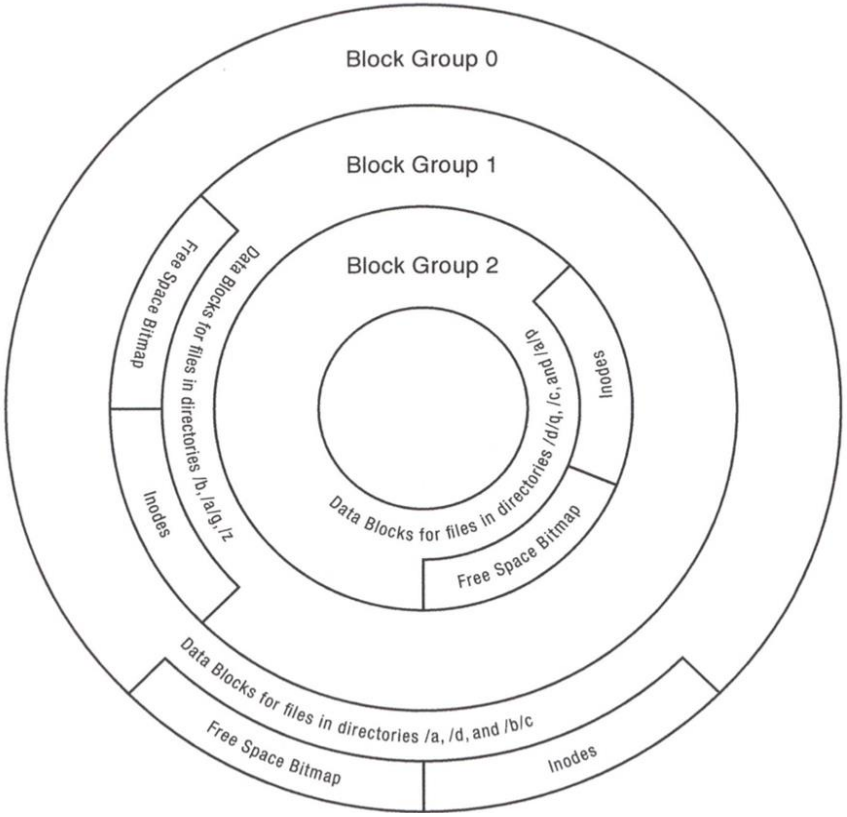
Pintos Base Implementation

- You will add code to enable extensible files using an asymmetric tree that provides at least doubly-indirect blocks
- You may continue to use an entire sector per inode thus allowing the sector number to be the file number and easily access the inode



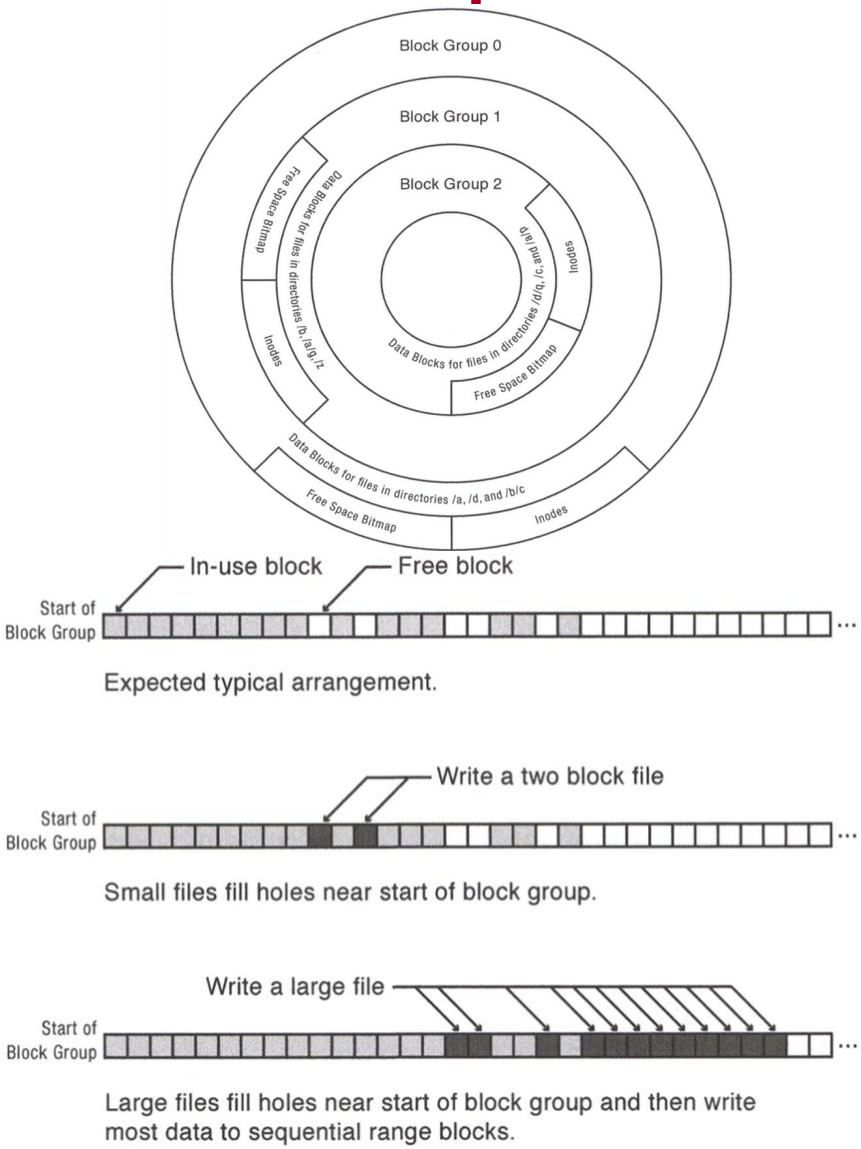
Locality Heuristics

- Disk is broken into groups of *contiguous tracks* called a **block group**
- Block group has a partition of the inode array and bitmap
- Attempt to place files of a directory in the block group where the directory's inode is located
- Subdirectories can be located in different block groups to avoid filling it



More About Block Groups

- Within a block group, **first-fit** algorithm is used
  - Attempts to fill holes thus leaving greater contiguous free space at the end of the block group
- To increase ability to support sequential allocation, some amount of the disk is reserved (i.e. disk is "full" even if block groups have an average of 10% free space remaining)
  - Want to ensure each block group has free space so large files might be split across a block group



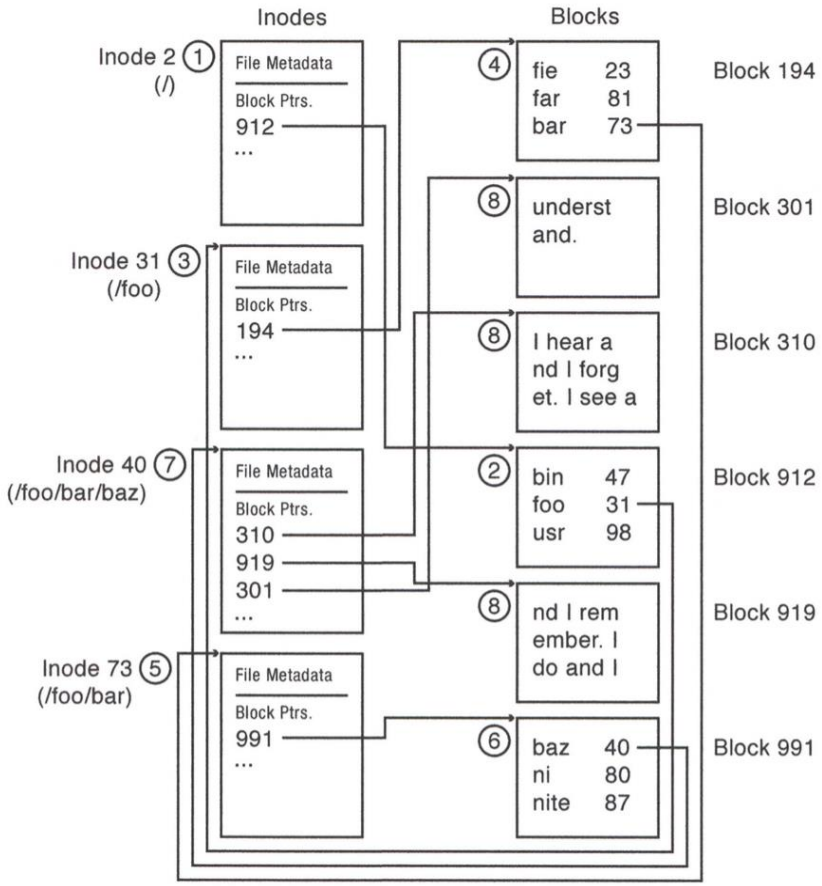
Good reference for FFS:
http://pages.cs.wisc.edu/~remzi/Classes/537/Fall2008/Notes/file-ffs.txt

Opening and Reading a File

- List the sequence for finding and opening /tmp/f1.txt
  - Use the well-known inode index for root directory, /
  - Use the inode to go to the block where the file for "/" is stored
  - Read through the data (possibly spanning multiple blocks) to find the mappint of 'tmp' to its file (inode) number
  - Go back to the inode array and possibly read a new sector/block to get the inode
  - Use the inode for tmp to go to the block where the file for "tmp" is stored
  - Read through the data to find the file (inode) number associated with "f1.txt"
  - Go back to the inode array to read the inode for "f1.txt"
  - Use the inode for "f1.txt" to start reading through its direct blocks
  - If "f1.txt" is large enough to require use of an indirect block, read the indirect block to obtain the subsequent direct pointers and then continue to read the blocks indicated by those direct pointers
  - And so on for double or triply indirect blocks

Opening and Reading a File

- Reading a file may require many "random" accesses to walk the directory and file index structures

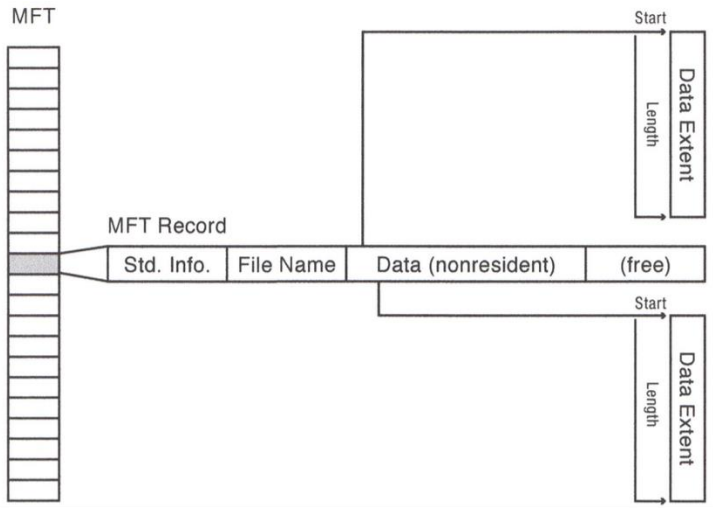


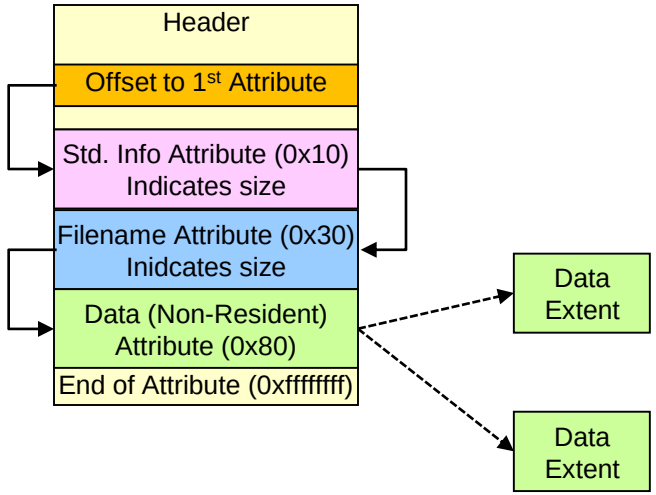
OS:PP 2nd Ed. Fig. 13.25
Read of /foo/bar/baz in the FFS file system

NTFS

MFT Records

- Uses an **extent-based** (variable size, contiguous ranges) approach
- Allocates by unit of clusters (multiple sectors) usually starts at 4KB
- Master File Table (MFT)
  - Entries are 1KB
  - Each entry contains
    - A header (hard link count, in use, dir/file, size of MFT entry)
    - Some number of attributes
  - Attributes hold file metadata (name, std. info) or data
  - Attributes can be resident (in the current extent) or non-resident (pointers to other extents)
- Usually 1/8 of disk set aside for MFT growth

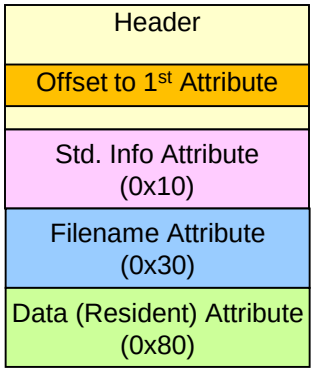




http://www.kes.talktalk.net/ntfs/

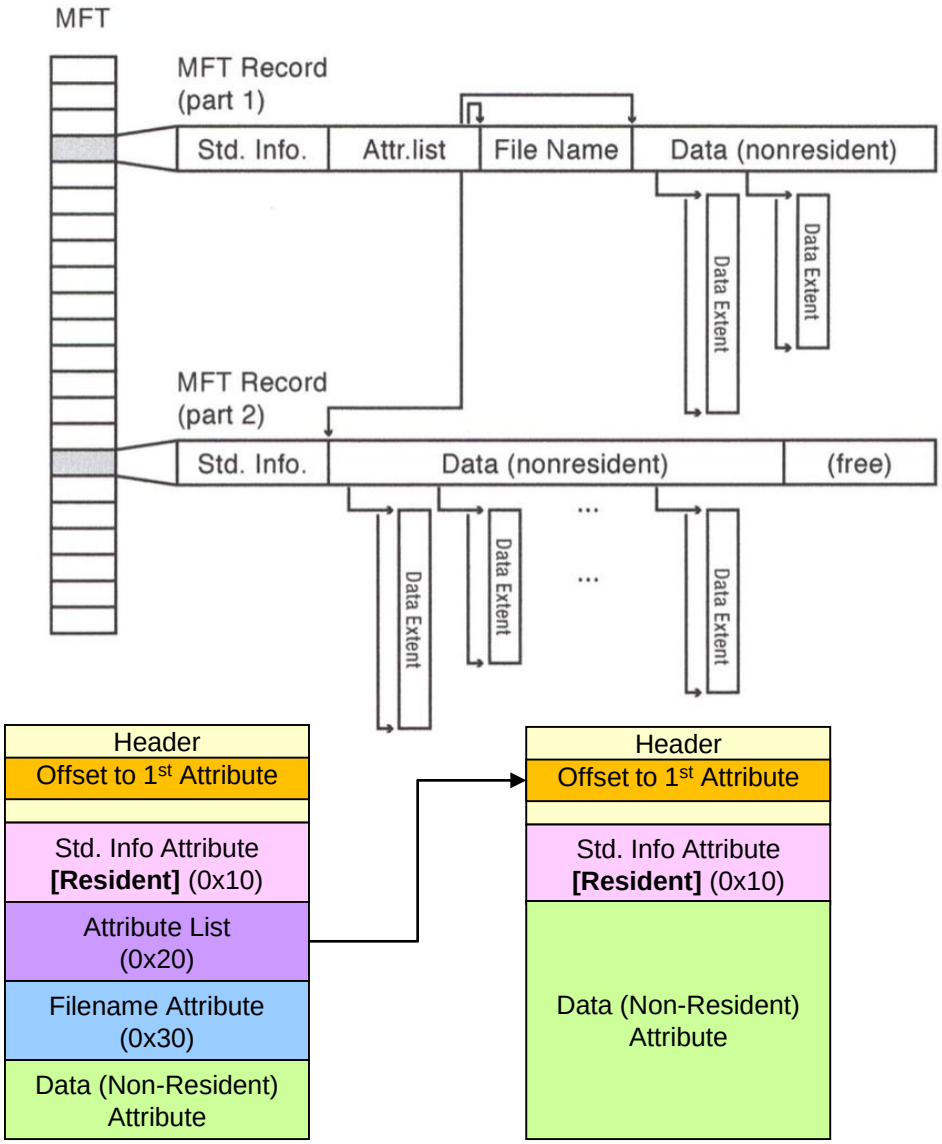
Resident Data Attributes

- A small file's data (smaller than 1KB) can be contained in a "data" attribute entirely in the MFT record



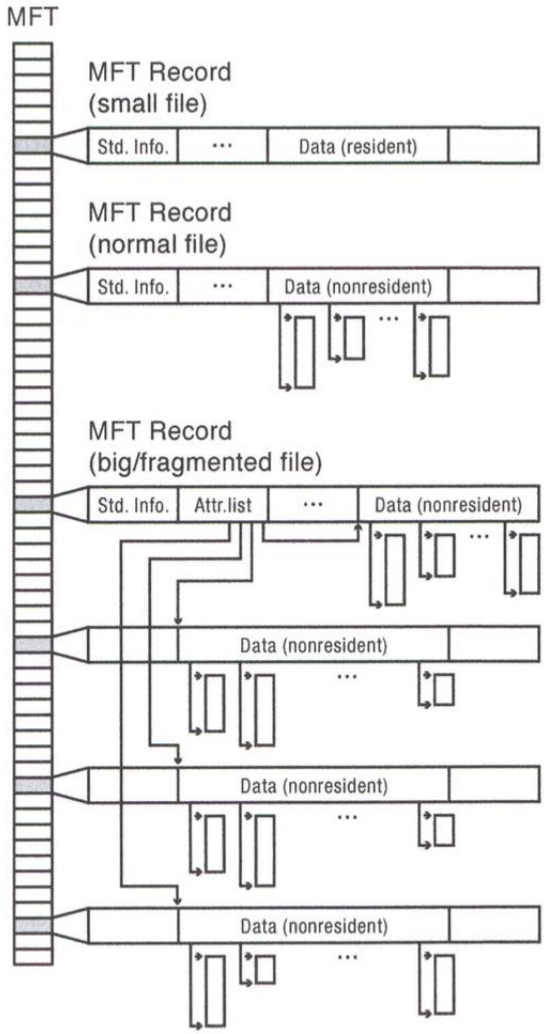
Entry with Large Number of Attributes

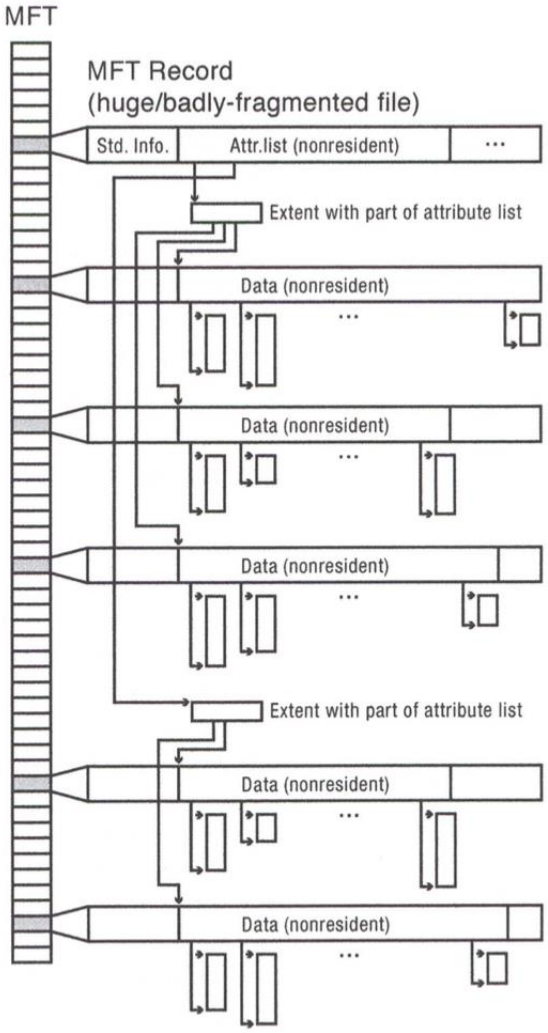
- If there are too many attributes or an attribute is too large to fit in the MFT entry, an extension record can be created using another MFT entry
- This may occur if the file is fragmented over many extents
- First entry acts as the file number/index

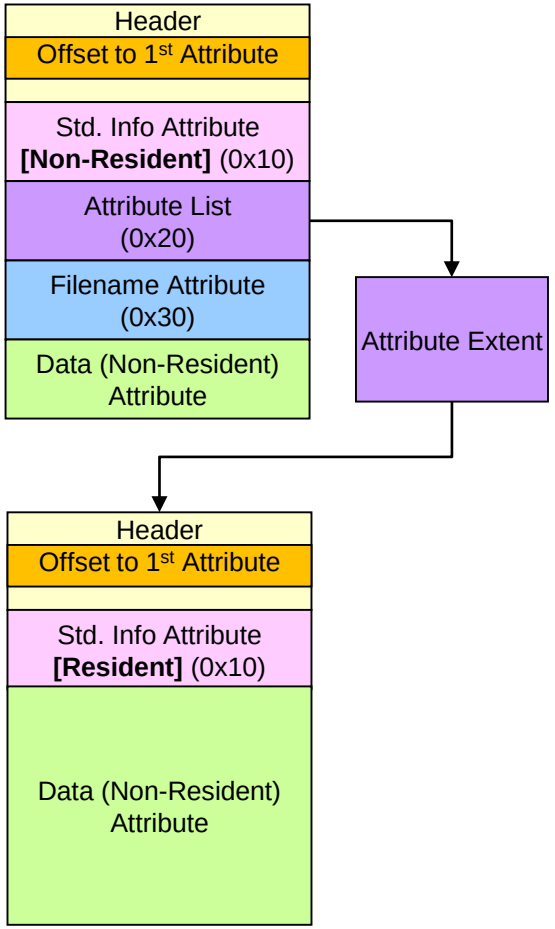


Non-resident Attribute Lists (Very Large Files)

- Even attribute lists can be non-resident







Comparison

FFS (Linux ext2, ext3)

- Block-based
  - Can be susceptible to long seek times
- Fixed-depth (asymmetric) index tree
  - Limits file size
- Data never stored in inode entry
- Block groups using **first-fit**
  - Internal Fragmentation

NTFS (Linux ext4, Apple HFS)

- Extent-based
  - Allows better sequential access
- Variable-depth index structure
  - Arbitrary file size
- Small files in MFT entry itself
- **Best-fit** algorithm (API allows estimated file size to be communicated)
  - External Fragmentation

ZFS, Btrfs

COPY-ON-WRITE FILE SYSTEMS

Motivation

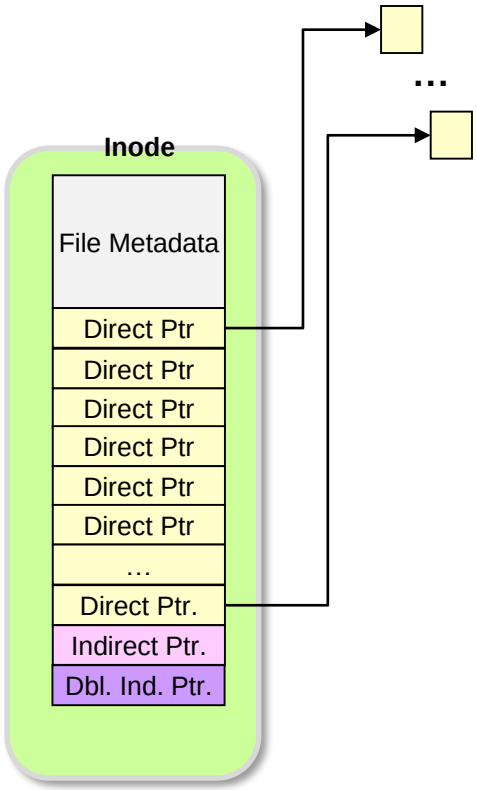
- Small writes are expensive…
  - Random access (data block, indirect node, i-node)
  - Especially to RAID (if one block on a disk changes, need to update parity block)
- …but sequential writes are faster
- Block caches (data from files maintained in RAM by the OS) filter reads
- Prevalence of flash
  - Need to even wearing
- Greater disk space
  - Allows versioning

Basic Idea

- Don't update blocks in place, simply write them all sequentially to a new location
  - Data block, indirect block, i-node, directory, free-space bitmap, etc
- Make everything mobile
  - Main issue: inodes need to be mobile rather than at fixed locations
    - Solution: Store inode array itself in a file

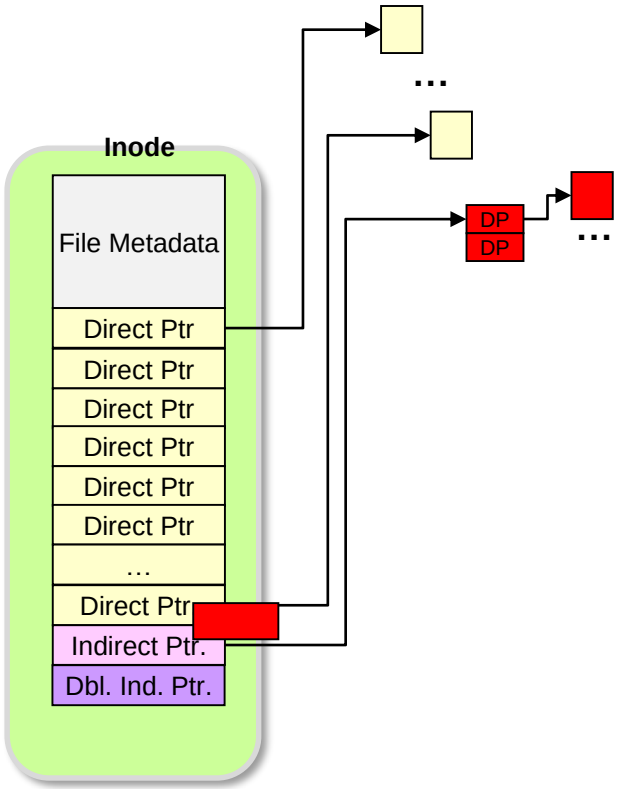
Copy-On-Write Idea

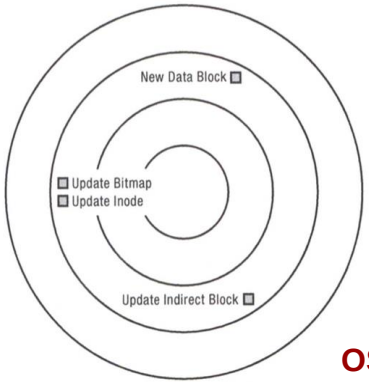
- Suppose we need to add a block to a file whose old size fit in direct blocks but now needs to start using indirect pointers



Copy-On-Write Idea

- Suppose we need to add a block to a file whose old size fit in direct blocks but now needs to start using indirect pointers
- We would allocate and update
  - The actual data block
  - The indirect block
  - The inode indirect ptr.
  - The freespace bitmap
- The writes would like be non-sequential (spread) over the disk

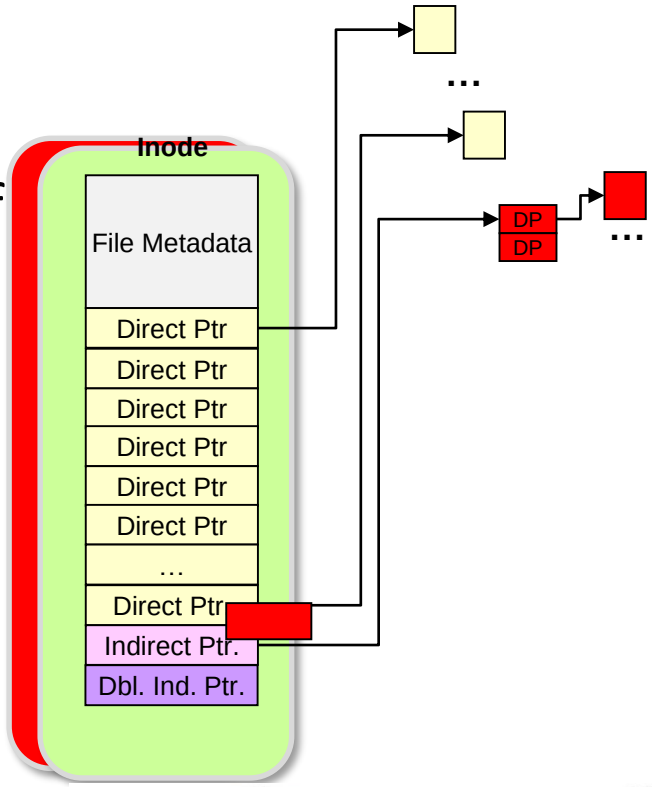


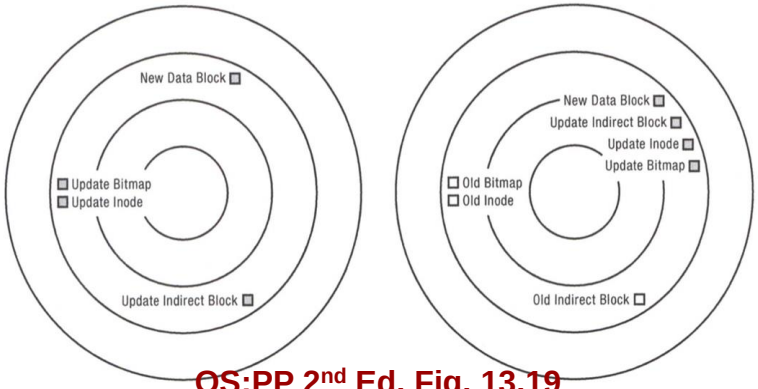


OS:PP 2nd Ed. Fig. 13.19

Copy-On-Write Idea

- Instead, COW file systems would **sequentially** write new versions of the following blocks
  - Data block
  - Indirect block
  - I-node
  - Freespace bitmap

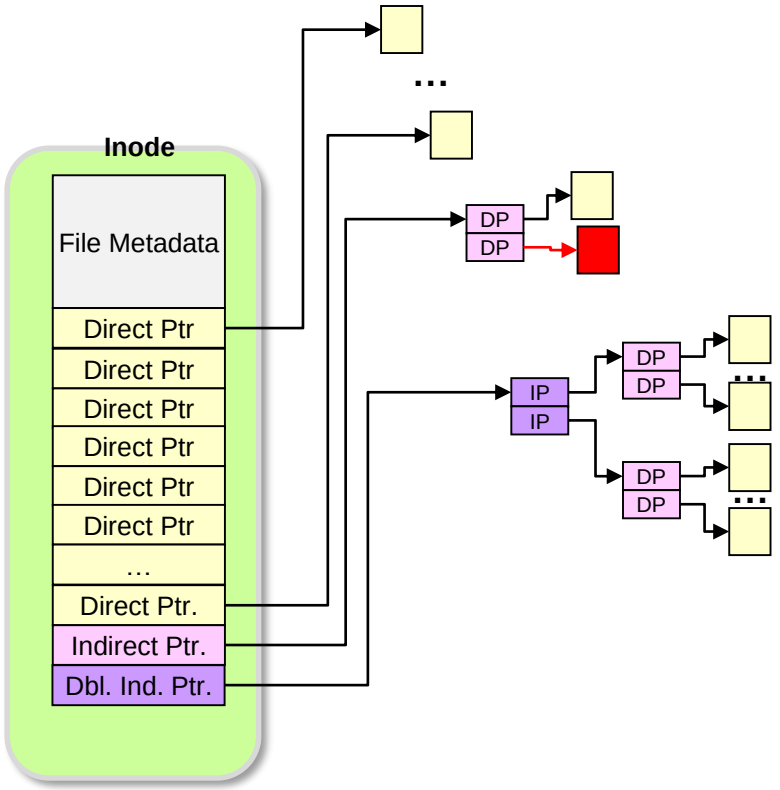




OS:PP 2nd Ed. Fig. 13.19

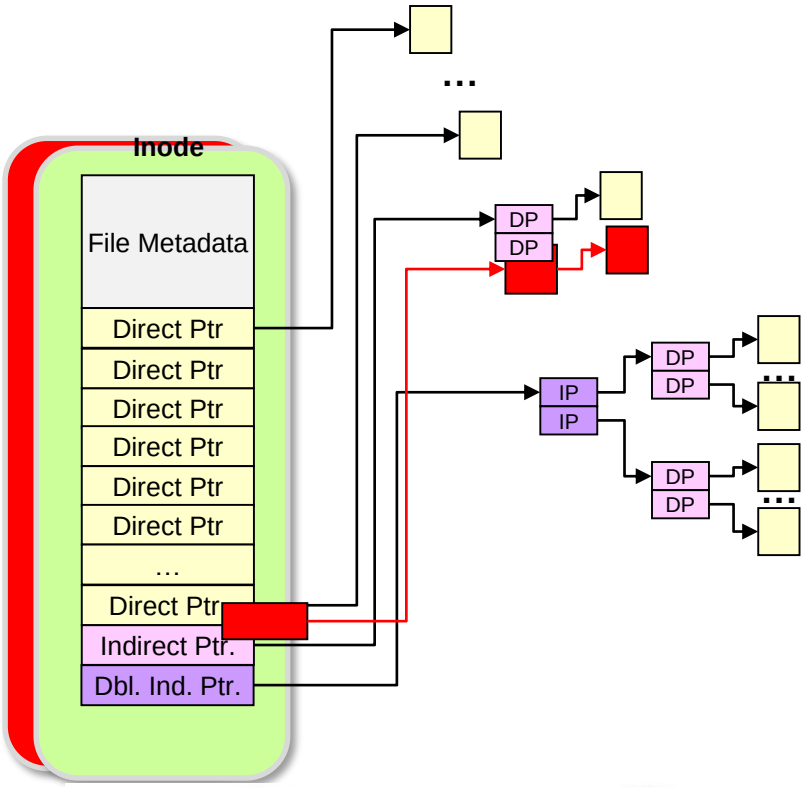
Updates Lead to Re-Writes

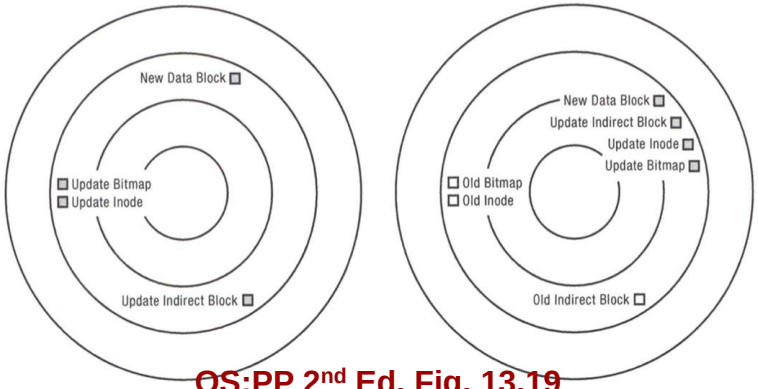
- If we already had an indirect block of pointers and wanted to add a new data block, would the inode need to change?



Updates Lead to Re-Writes

- No, but in COW we don't update in place instead making a new indirect block
  - And since the indirect block location is different the inode we need to be updated
  - Since the inode would need to be updated we'd simply write a new version of it sequentially with the other updated blocks
  - And since the inode got updated, our directory entry would have to change, so we'd rewrite the directory file
  - And since the directory file changed…

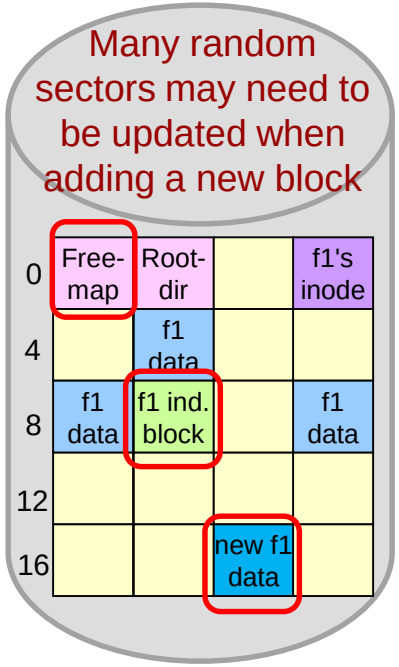


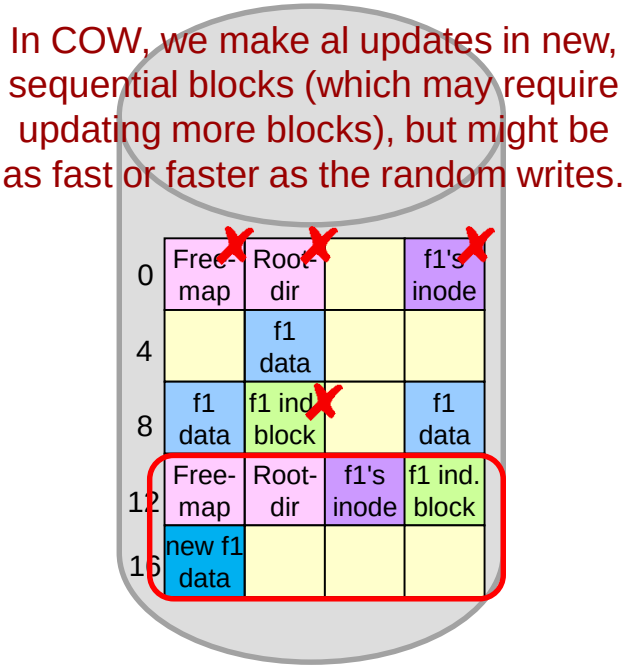


OS:PP 2nd Ed. Fig. 13.19

Chain of Updates

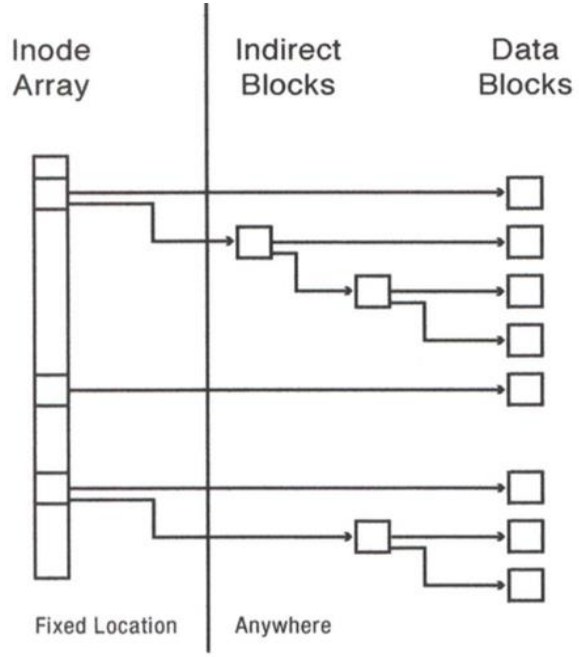
- In COW, we would re-write all the updated blocks sequentially
  - Since blocks are moving, may need to update inode and directory entries
  - If the file is deep in the directory path, all parent directories would likely need to be updated
- We would move all the way to the root node of the file system



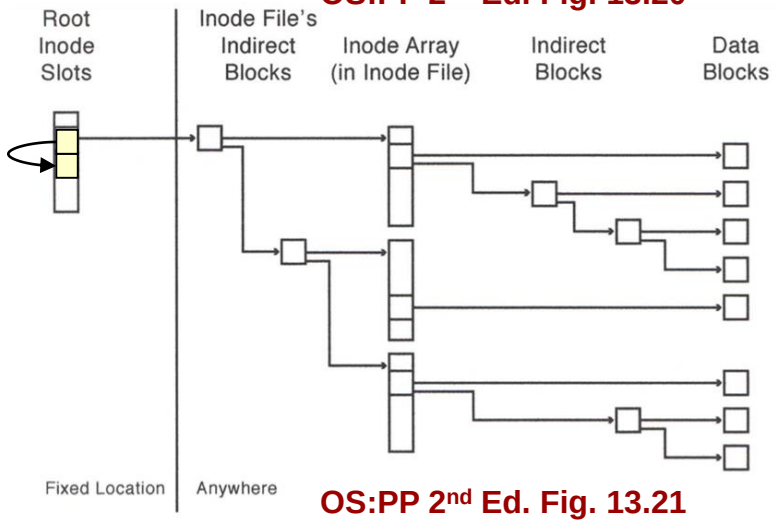


Implementation

- In FFS the inode array would need to be in fixed (well-known) locations so that they could be found when needed
- In COW, inodes change and thus need to be mobile
  - We could place them in a file (files are extensible)
  - But how do we know where that file is?
  - We could have a small "circular" buffer of "root inode" slots with only one (the latest) in use at a time
  - Each update moves the root inode on



OS:PP 2nd Ed. Fig. 13.20

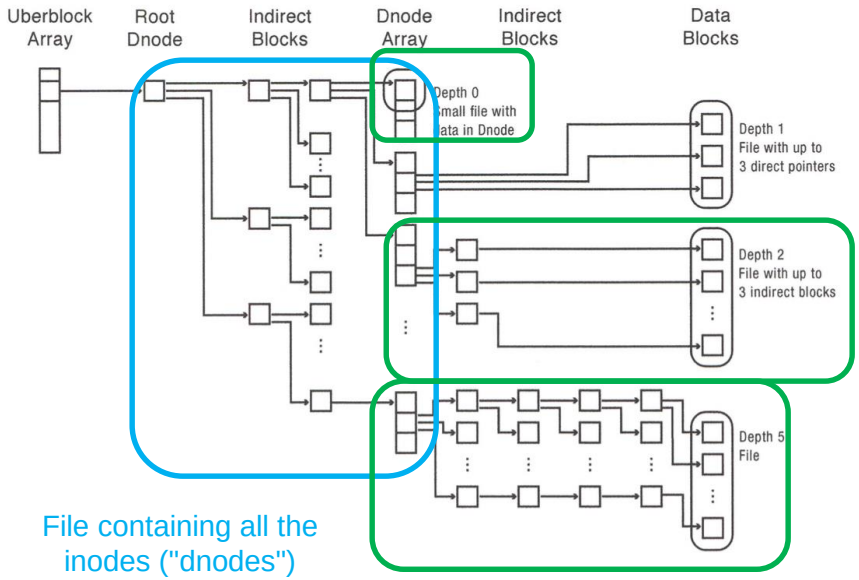


OS:PP 2nd Ed. Fig. 13.21

ZFS Implementation

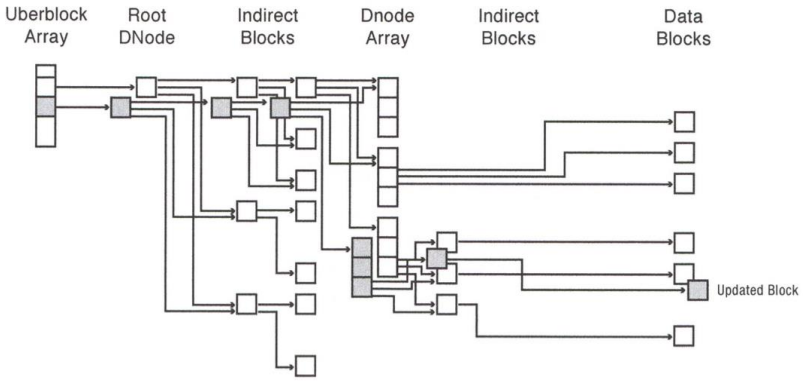
- Sun's ZFS implements COW concepts
- Uberblock array is like the root inode array slots (rotates entries on updates) and stores a pointer to the current "root Dnode" which is essentially the inode for the file containing all the other inodes
- Dnodes are like i-nodes
  - Variable depth tree of block pointers
  - Initial Dnode has room for 3 block pointers
  - Support small files with data in the Dnode itself (i.e. tree depth=0)
    - Like MFT entry of NTFS

OS:PP 2nd Ed. Fig. 13.23



ZFS Example and Block Sizes

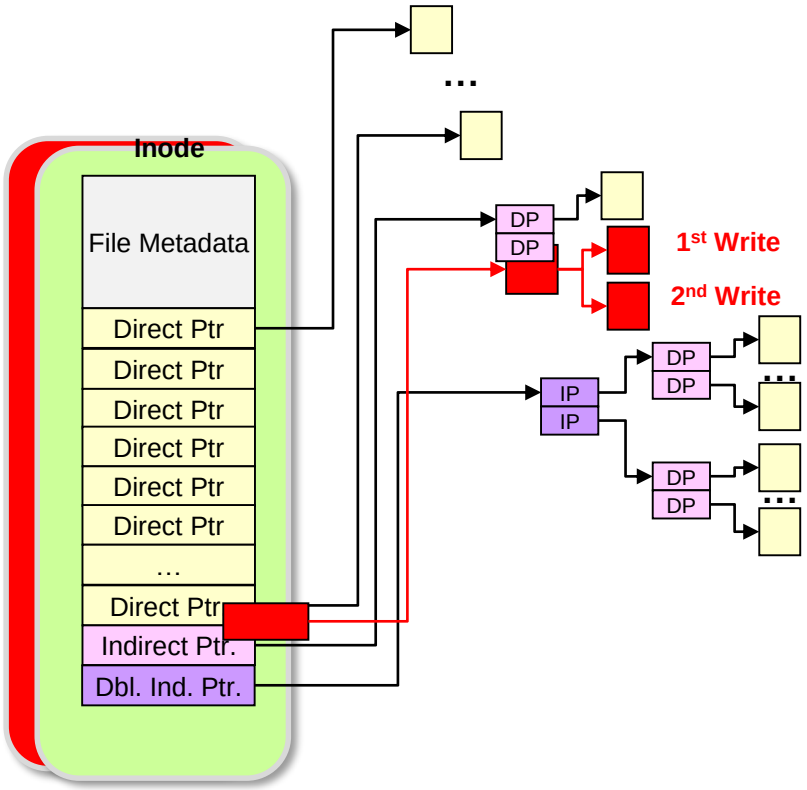
- Figure to the right shows an update of a file that uses 1 level of indirection
- Files can specify block size ranging from 512 bytes to 128KB
  - But block pointers are large 128-byte structures (not just 4-bytes like in Pintos) as they hold checksums and other info, snapshots, etc.
  - Large blocks…
    - Larger file sizes w/ same size index tree
    - Potentially less free-space tracking overhead
- But it seems like a lot of work…



OS:PP 2nd Ed. Fig. 13.22

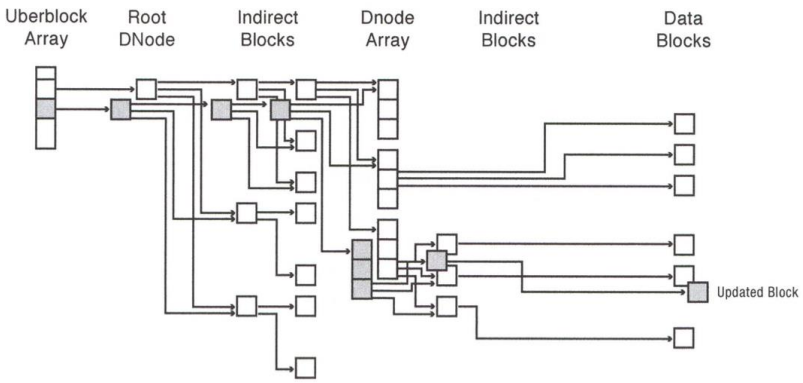
Performance Enhancements

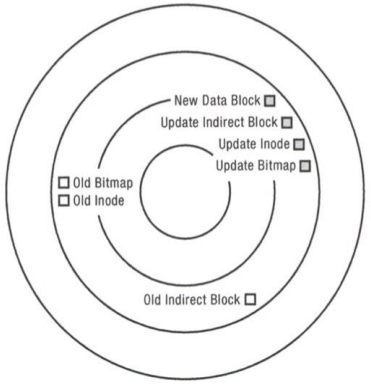
- Recall: Sequential writes are fast
- Writes can be buffered in memory allowing writes to the same file which cause multiple updates of the indirect pointers and dnodes to be coalesced
  - In the figure if we did two writes, EACH write may require re-writing the indirect block, inode, etc.
  - But if we buffer these updates in memory and coalesce the writes we would only need to write the indirect block and inode once (amortizing the cost over the 2 data blocks written)
- After a short interval the writes are performed on disk



COW Benefits

- Versioning!
  - Notice old version of file and directory structure are still present
- Consistency
  - Transactional approach (all data maintained until atomic switch of uberblock)
  - Power failure or crash still presents a consistent (old or new) view of the FS
- Even wearing
  - Many updates to one file will now result in many rewrites to different locations

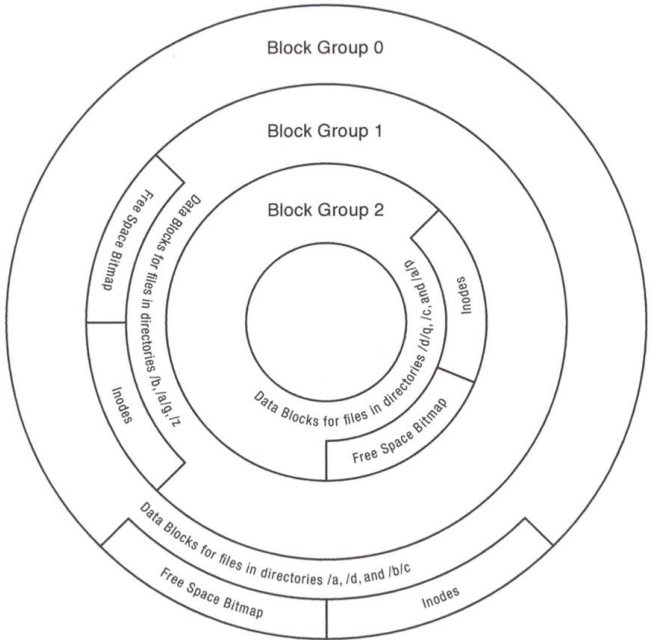


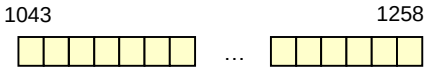


OS:PP 2nd Ed. Fig. 13.22

ZFS Locality (Free-Space) Management

- Bitmaps for large drives can still consume vast amounts of space
  - 32 TB drive with 4KB pages = 1GB of bitmap space
- ZFS uses a similar notion of FFS **block groups**
- Maintains free-space:
  - Per block group
    - Partitions the free-space data structure (bitmap or extent tree) to make lookups faster
  - As extents (contiguous ranges)
    - Large, sequential free extents can be stored compactly rather than 1 bit per block
    - Stores extents in an AVL tree indexed on size
  - Using log-based updates
    - Frees are simply logged (in memory) and then the free-space tree is updated only when a new allocation need be performed





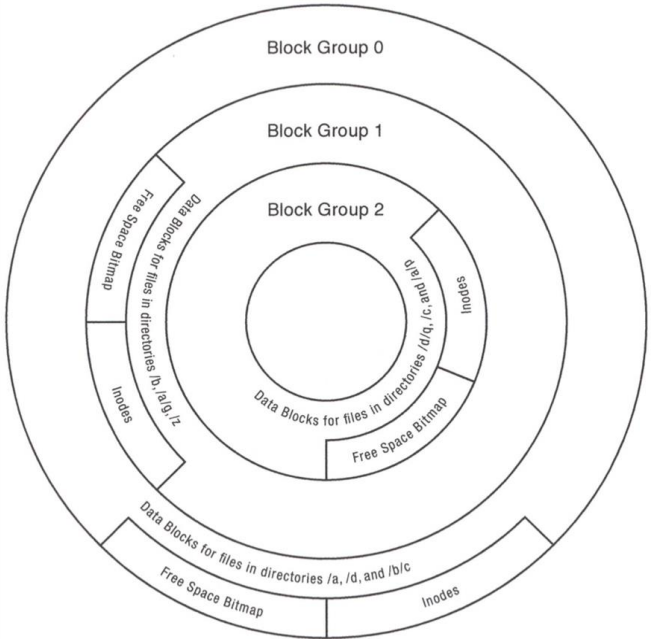
Bitmap for a large contiguous set of free blocks

Start: 1043
Size: 216

Extent Representation

Allocation Strategies

- When we do write to disk we must choose which block group and then allocate blocks within that group
- ZFS may span multiple disks
  - Round robin between disks with some bias towards those with more free space
- Choose a block group
  - Prefer spatial locality and continue in the block group where you last wrote
  - Once a block group reaches a certain limit of free space, move on to the next (biasing selection based on free-space, location [nearby / outer-tracks of disk], etc.)
  - Use **first fit** *until* the block group is close to full then use **best-fit**



COW Summary

- Versioning and Error-detection (checksumming)
- Much better write performance
- Comparable read performance
- Comparable file sizes (support for large volumes)
- Flash optimized