File Systems: Fundamentals

Don Porter

Portions courtesy Emmett Witchel

Files

- What is a file?
  - A named collection of related information recorded on secondary storage (e.g., disks)
- File attributes
  - Name, type, location, size, protection, creator, creation time, last-modified-time, …
- File operations
  - Create, Open, Read, Write, Seek, Delete, …
- How does the OS allow users to use files?
  - "Open" a file before use
  - OS maintains an open file table per process, a file descriptor is an index into this file.
  - Allow sharing by maintaining a system-wide open file table

Fundamental Ontology of File Systems

- Metadata
  - The index node (inode) is the fundamental data structure
  - The superblock also has important file system metadata, like block size
- Data
  - The contents that users actually care about
- Files
  - Contain data and have metadata like creation time, length, etc.
- Directories
  - Map file names to inode numbers

Basic Data Structures

- Disk
  - An array of blocks, where a block is a fixed size data array
- File
  - Sequence of blocks (fixed length data array)
- Directory
  - Creates the namespace of files
    - Heirarchical – traditional file names and GUI folders
    - Flat – like the all songs list on an ipod
- Design issues: Representing files, finding file data, finding free blocks

Blocks and Sectors

- Recall: Disks write data in units of **sectors**
  - Historically 512 Bytes; Today mostly 4KiB
  - A sector write is all-or-nothing
- File systems allocate space to files in units of **blocks**
  - A block is 1+ **consecutive** sectors

Selecting a Block Size

- Convenient to have blocks match or be a multiple of page size (why?)
  - Cache space in memory can be managed with same page allocator as used for processes; mmap of a block to a virtual page is 1:1
- Large blocks can be more efficient for large read/writes (why?)
  - Fewer seeks per byte read/written (if all of the data useful)
- Large blocks can *amplify* small writes (why?)
  - One byte update may cause entire block to be rewritten

Functionality and Implementation

- File system functionality:
  - Allocate physical sectors for logical file blocks
    - Must balance locality with expandability.
    - Must manage free space.
  - Index file data, such as a hierarchical name space
- File system implementation:
  - File header (descriptor, inode): owner id, size, last modified time, and location of all data blocks.
    - OS should be able to find metadata block number N without a disk access (e.g., by using math or cached data structure).
  - Data blocks.
    - Directory data blocks (human readable names)
    - File data blocks (data).
  - Superblocks, group descriptors, other metadata…

File System Properties

- Most files are small.
  - Need efficient support for small files.
  - Block size can’t be too big.
- Some files are very large.
  - Must allow large files (64-bit file offsets).
  - Large file access also should be reasonably efficient.

If my file system only has lots of big video files what block size do I want?

1. Large
2. Small

Three Problems for Today

- Indexing data blocks in a file:
  - What is the LBA of is block 17 of The_Dark_Knight.mp4?
- Allocating free disk sectors:
  - I add a block to rw-trie.c, where should it go on disk?
- Indexing file names:
  - I want to open /home/porter/foo.txt, does it exist, and where on disk is the metadata?

Problem 0: Indexing Files&Data

The information that we need:

For each file, a file header points to data blocks

Block 0 --> Disk block 19

Block 1 --> Disk block 4,528

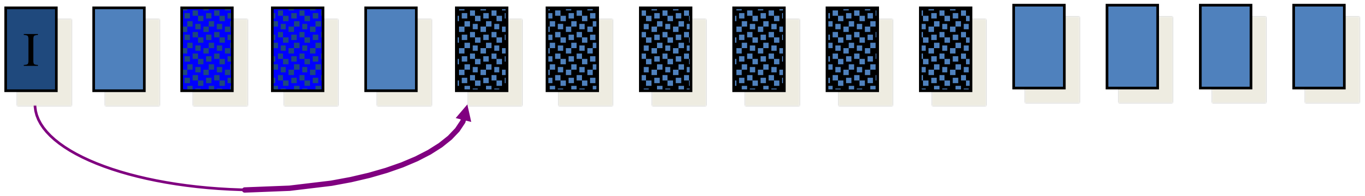
…

Key performance issues:

1. We need to support sequential and random access.
2. What is the right data structure in which to maintain file location information?
3. How do we lay out the files on the physical disk?

We will look at some data indexing strategies

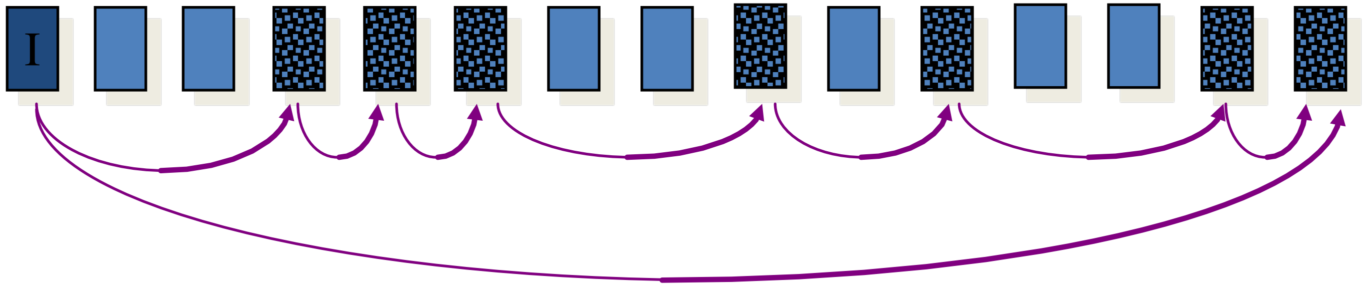
Strategy 0: Contiguous Allocation



- File header specifies starting block & length
- Placement/Allocation policies
  - – First-fit, best-fit, ...

- Pluses
  - Best file read performance
  - Efficient sequential & random access
- Minuses
  - Fragmentation!
  - Problems with file growth
    - Pre-allocation?
    - On-demand allocation?

Strategy 1: Linked Allocation



- Files stored as a linked list of blocks
- File header contains a pointer to the first and last file blocks

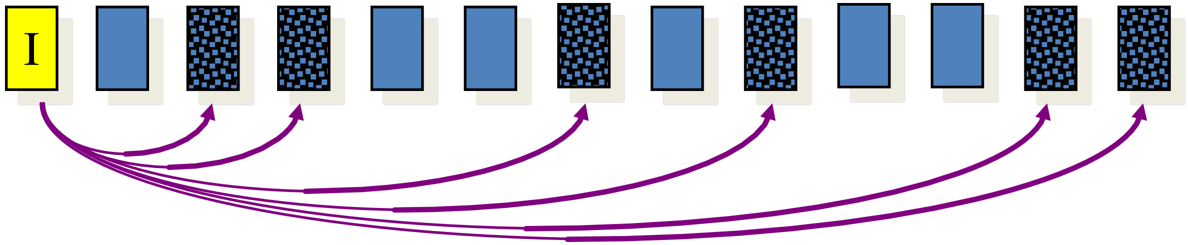
- Pluses
  - Easy to create, grow & shrink files
  - No external fragmentation
    - Can ”stitch” fragments together!

- Minuses
  - Impossible to do true random access
  - Reliability
    - Break one link in the chain and…

Strategy 2: File Allocation Table (FAT)

- Create a table with an entry for each block
  - Overlay the table with a linked list
  - Each entry serves as a link in the list
  - Each table entry in a file has a pointer to the next entry in that file (with a special "eof" marker)
  - A "0" in the table entry ➔ free block
- Comparison with linked allocation
  - If FAT is cached ➔ better sequential and random access performance
    - How much memory is needed to cache entire FAT?
      - 400GB disk, 4KB/block ➔ 100M entries in FAT ➔ 400MB
    - Solution approaches
      - Allocate larger clusters of storage space
      - Allocate different parts of the file near each other ➔ better locality for FAT

Strategy 3: Direct Allocation

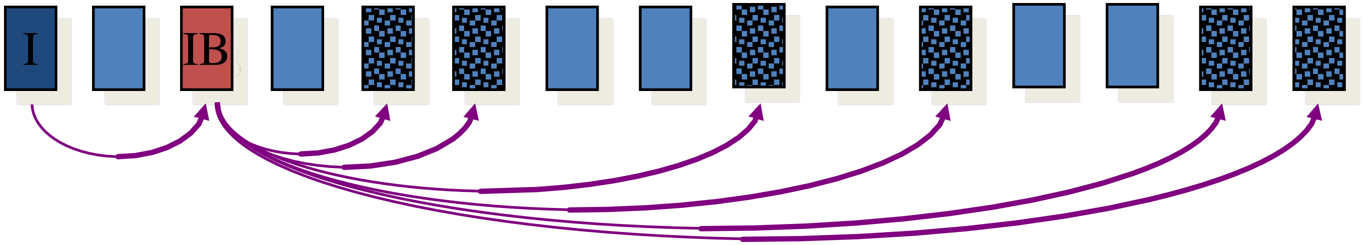


- File header points to each data block

- Pluses
  - Easy to create, grow & shrink files
  - Little fragmentation
  - Supports direct access

- Minuses
  - Inode is big or variable size
  - How to handle large files?

Strategy 4: Indirect Allocation

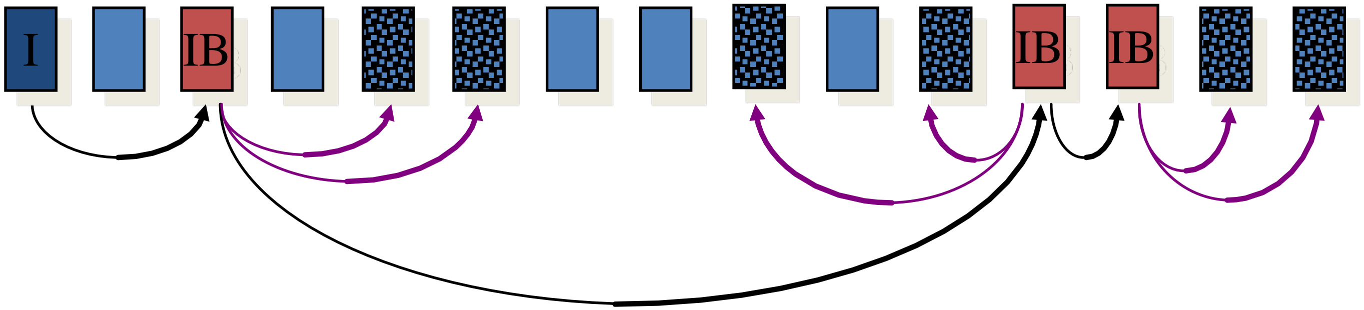


- Create a non-data block for each file called the *indirect block*
  - A list of pointers to file blocks
- File header contains a pointer to the indirect block

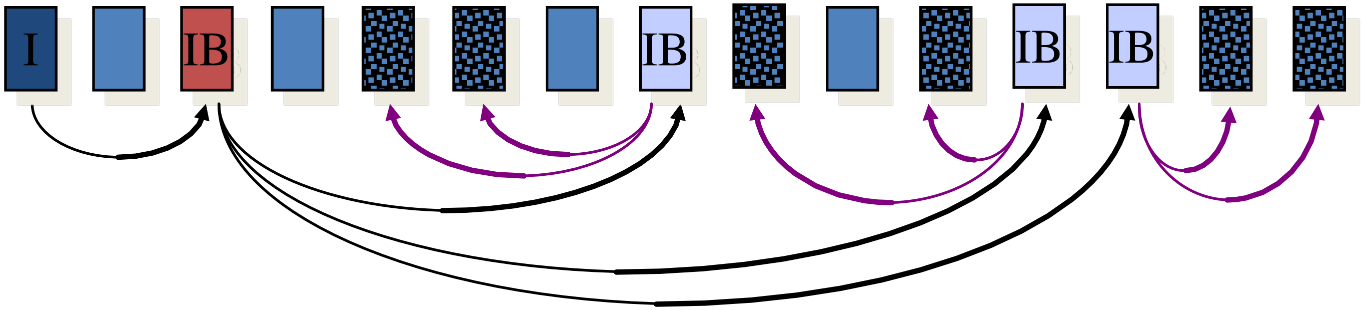
- Pluses
  - Easy to create, grow & shrink files
  - Little fragmentation
  - Supports direct access
- Minuses
  - Overhead of storing index when files are small
  - How to handle large files?

Indexed Allocation for Large Files

- Linked indirect blocks (IB+IB+…)



- Multilevel indirect blocks (IB*IB*…)

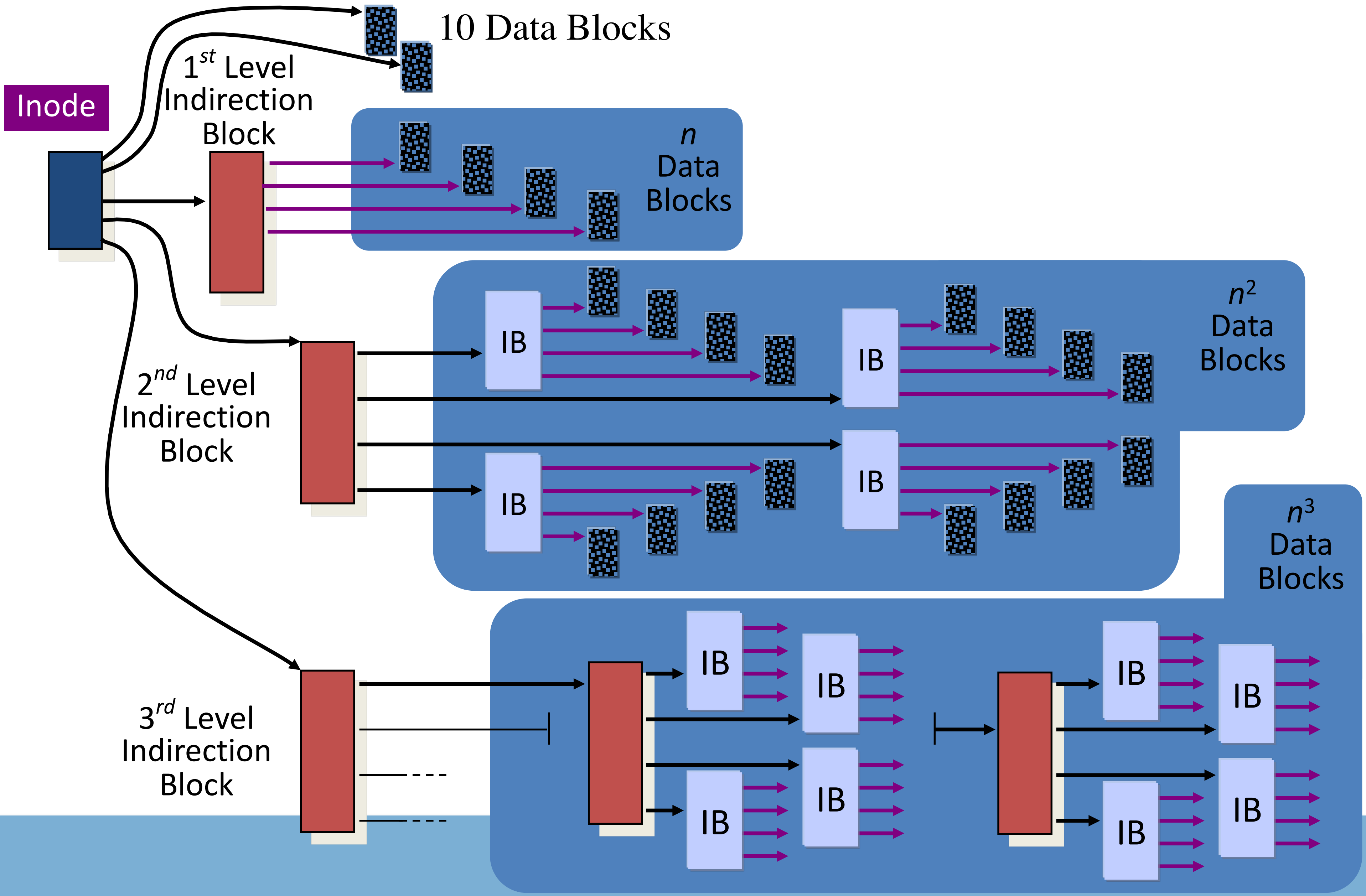


- Why bother with indirect blocks?
  - A. Allows greater file size.
  - B. Faster to create files.
  - C. Simpler to grow files.
  - D. Simpler to prepend and append to files.

Direct/Indirect Hybrid Strategy in Unix

- File header contains 13 pointers
  - 10 pointes to data blocks; 11th pointer → indirect block; 12th pointer → doubly-indirect block; and 13th pointer → triply-indirect block
- Implications
  - Upper limit on file size (~2 TB)
  - Blocks are allocated dynamically (allocate indirect blocks only for large files)
- Features
  - Pros
    - Simple
    - Files can easily expand (add indirect blocks proportional to file size)
    - Small files are cheap (fit in direct allocation)
  - Cons
    - Large files require a lot of seek to access indirect blocks

Visualization



- How big is an inode?
  - A. 1 byte
  - B. 16 bytes
  - C. 128 bytes
  - D. 1 KB
  - E. 16 KB

Three Problems for Today

- Indexing data blocks in a file:
  - What is the LBA of is block 17 of The_Dark_Knight.mp4?
- Allocating free disk sectors:
  - I add a block to rw-trie.c, where should it go on disk?
- Indexing file names:
  - I want to open /home/porter/foo.txt, does it exist, and where on disk is the metadata?

How to store a free list on disk?

- Recall: Disks can be big (currently in TB)
  - Allocations can be small (often 4KB)
- Any thoughts?

Strategy 0: Bit vector

- Represent the list of free blocks as a *bit vector*:

  11111111111111001110101011101111...

  – If bit *i* = 0 then block *i* is *free*, if *i* = 1 then it is *allocated*

Simple to use and vector is compact:
1TB disk with 4KB blocks is 2^28 bits or 32 MB

If free sectors are uniformly distributed across the disk then the expected number of bits that must be scanned before finding a “0” is

$$n/r$$

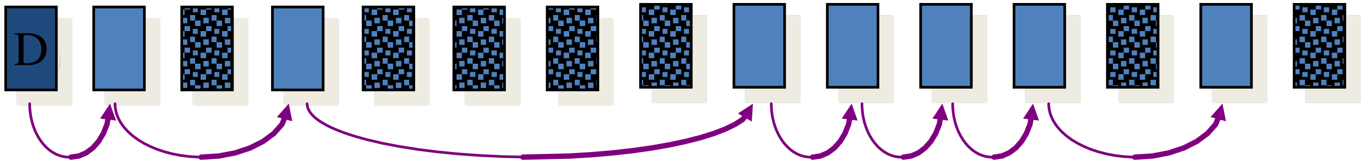
where

n = total number of blocks on the disk,
r = number of free blocks

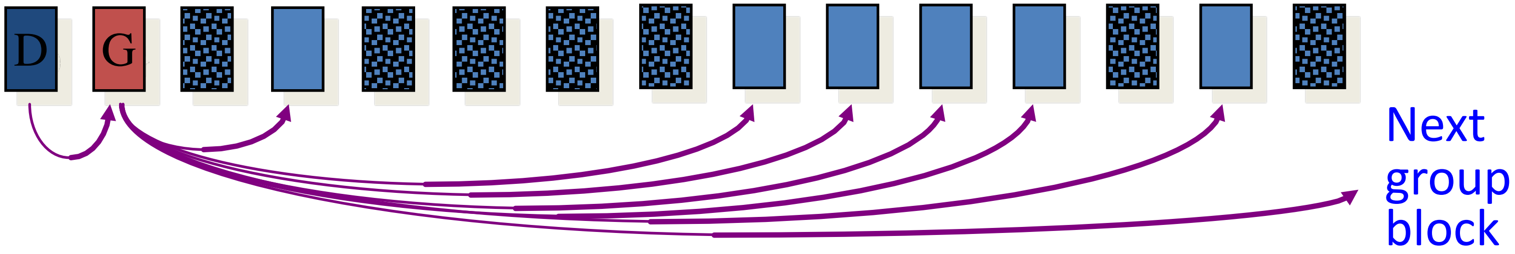
If a disk is 90% full, then the average number of bits to be scanned is 10, independent of the size of the disk

Other choices

- In-situ linked lists



- Grouped lists



▨ Allocated block   ▭ Empty block

Block allocation redux

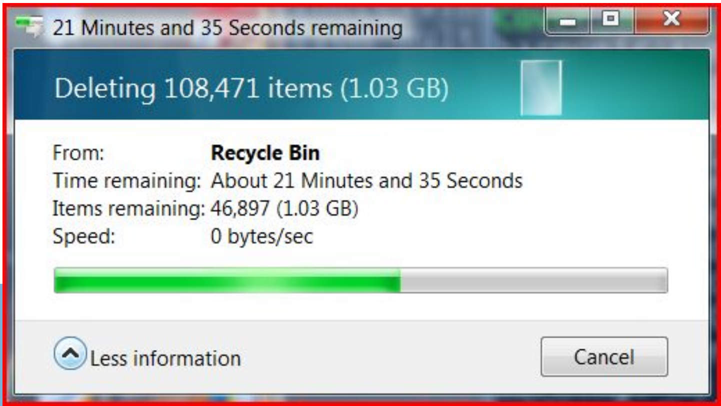
- Bitmap strategy pretty widely used
- Space efficient, but fine-grained
  - Tolerates faults reasonably well
  - (i.e., one corrupted sector loses free info for one sector’s worth of bitmap, not whole list)

Allocating Inodes

- Need a data block
  - – Suppose we have a list of free blocks
- Need an inode
  - – Consult a list of free inodes
- Why do inodes have their own free list?
  - – A. Because they are fixed size
  - – B. Because they exist at fixed locations
  - – C. Because there are a fixed number of them

Deleting a file is a lot of work

- Data blocks back to free list
  - Coalescing free space
- Indirect blocks back to free list
  - Expensive for large files, an ext3 problem
- Inodes cleared (makes data blocks "dead")
- Inode free list written
- Directory updated
- The order of updates matters!
  - Can put block on free list only after no inode points to it



21 Minutes and 35 Seconds remaining

Deleting 108,471 items (1.03 GB)

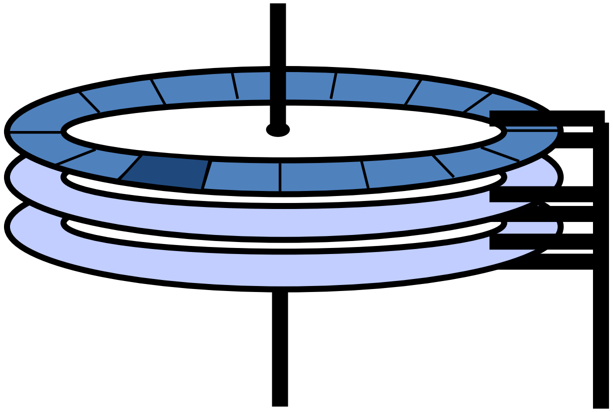
From: Recycle Bin
Time remaining: About 21 Minutes and 35 Seconds
Items remaining: 46,897 (1.03 GB)
Speed: 0 bytes/sec

Three Problems for Today

- Indexing data blocks in a file:
  - What is the LBA of is block 17 of The_Dark_Knight.mp4?
- Allocating free disk sectors:
  - I add a block to rw-trie.c, where should it go on disk?
- Indexing file names:
  - I want to open /home/porter/foo.txt, does it exist, and where on disk is the metadata?

Naming Files and Directories

- Files are organized in directories
  - Directories are themselves files
  - Contain <name, pointer to file header> table
- Only OS can modify a directory
  - Ensure integrity of the mapping
  - Application programs can read directory (e.g., ls)
- Directory operations:
  - List contents of a directory
  - Search (find a file)
    - Linear search
    - Binary search
    - Hash table
  - Create a file
  - Delete a file



- Every directory has an inode
  - A. True
  - B. False

- Given only the inode number (inumber) the OS can find the inode on disk
  - A. True
  - B. False

Directory Hierarchy and Traversal

- Directories are often organized in a hierarchy

- Directory traversal:
  - How do you find blocks of a file? Let's start at the bottom
    - Find file header (inode) – it contains pointers to file blocks
    - To find file header (inode), we need its I-number
    - To find I-number, read the directory that contains the file
    - But wait, the directory itself is a file
    - Recursion !!
  - Example: Read file /A/B/C
    - C is a file
    - B/ is a directory that contains the I-number for file C
    - A/ is a directory that contains the I-number for file B
    - How do you find I-number for A?
      - "/" is a directory that contains the I-number for file A
      - What is the I-number for "/"? In Unix, it is 2

Directory Traversal, Cont'd

- How many disk accesses are needed to access file /A/B/C?
  1. Read I-node for "/" (root) from a fixed location
  2. Read the first data block for root
  3. Read the I-node for A
  4. Read the first data block of A
  5. Read the I-node for B
  6. Read the first data block of B
  7. Read I-node for C
  8. Read the first data block of C

◆ Optimization:
  - Maintain the notion of a current working directory (CWD)
  - Users can now specify relative file names
  - OS can cache the data blocks of CWD

Naming and Directories

- Once you have the file header, you can access all blocks within a file
  - How to find the file header? Inode number + layout.
- Where are file headers stored on disk?
  - In early Unix:
    - Special reserved array of sectors
    - Files are referred to with an index into the array (I-node number)
    - Limitations: (1) Header is not near data; (2) fixed size of array → fixed number of files on disk (determined at the time of formatting the disk)
  - Berkeley fast file system (FFS):
    - Distribute file header array across cylinders.
  - Ext2 (linux):
    - Put inodes in block group header.
- How do we find the I-node number for a file?
  - Solution: directories and name lookup

- A corrupt directory can make a file system useless
  - A. True
  - B. False

Summary

- Understand how file systems map blocks to files
- Understand how free blocks are tracked
- Understand hierarchical directory structure
  - And what an inode is