CS 153
Design of Operating Systems

Winter 2016

Lecture 20: File Systems

Protection

- File systems implement some kind of protection system
  - Who can access a file
  - How they can access it
- More generally…
  - Objects are “what”, subjects are “who”, actions are “how”
- A protection system dictates whether a given action performed by a given subject on a given object should be allowed
  - You can read and/or write your files, but others cannot
  - You can read “/etc/motd”, but you cannot write to it

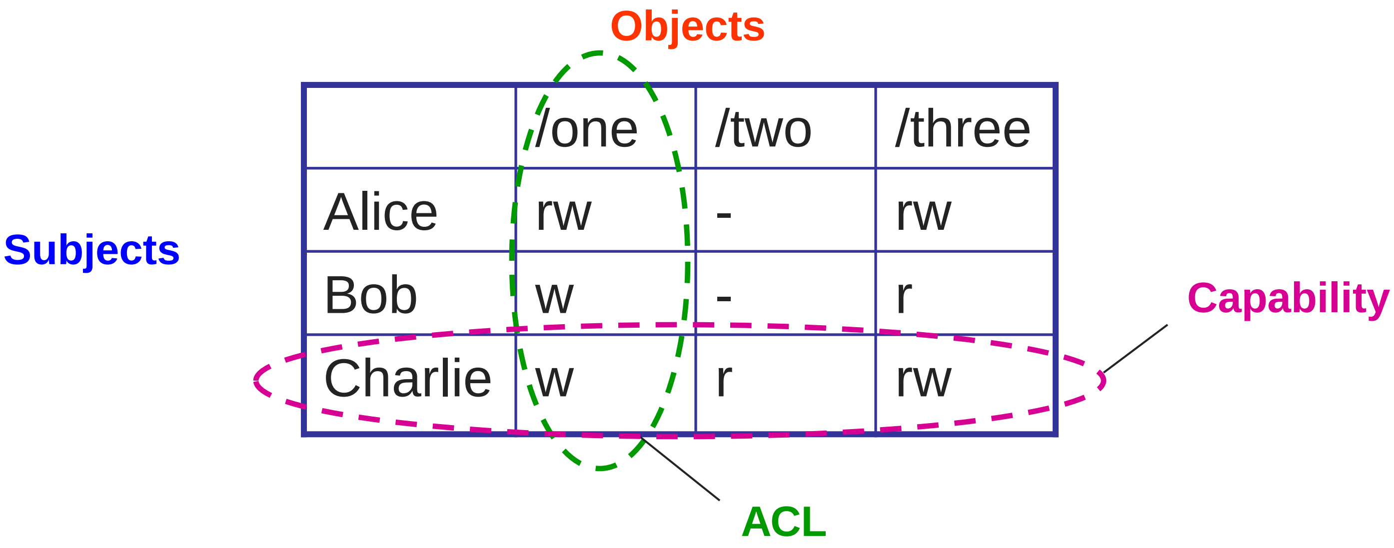
Representing Protection

Access Control Lists (ACL)

- For each object, maintain a list of subjects and their permitted actions

Capabilities

- For each subject, maintain a list of objects and their permitted actions



	/one	/two	/three
Alice	rw	-	rw
Bob	w	-	r
Charlie	w	r	rw

ACLs and Capabilities

- The approaches differ only in how table is represented
  - What approach does Unix use?
- Capabilities are easier to transfer
  - They are like keys, can handoff, does not depend on subject
- In practice, ACLs are easier to manage
  - Object-centric, easy to grant, revoke
  - To revoke capabilities, have to keep track of all subjects that have the capability – a challenging problem
- ACLs have a problem when objects are heavily shared
  - The ACLs become very large
  - Use groups (e.g., Unix)

File System Layout

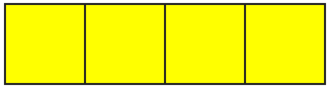
How do file systems use the disk to store files?

- File systems define a block size (e.g., 4KB)
  - Disk space is allocated in granularity of blocks
- A “Master Block” determines location of root directory
  - Always at a well-known disk location
  - Often replicated across disk for reliability
- A free map determines which blocks are free, allocated
  - Usually a bitmap, one bit per block on the disk
  - Also stored on disk, cached in memory for performance
- Remaining disk blocks used to store files (and dirs)
  - There are many ways to do this

Disk Layout Strategies

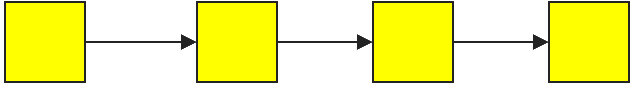
- Files span multiple disk blocks
- How do you find all of the blocks for a file?

1. Contiguous allocation



    - » Like memory
    - » Fast, simplifies directory access
    - » Inflexible, causes fragmentation, needs compaction

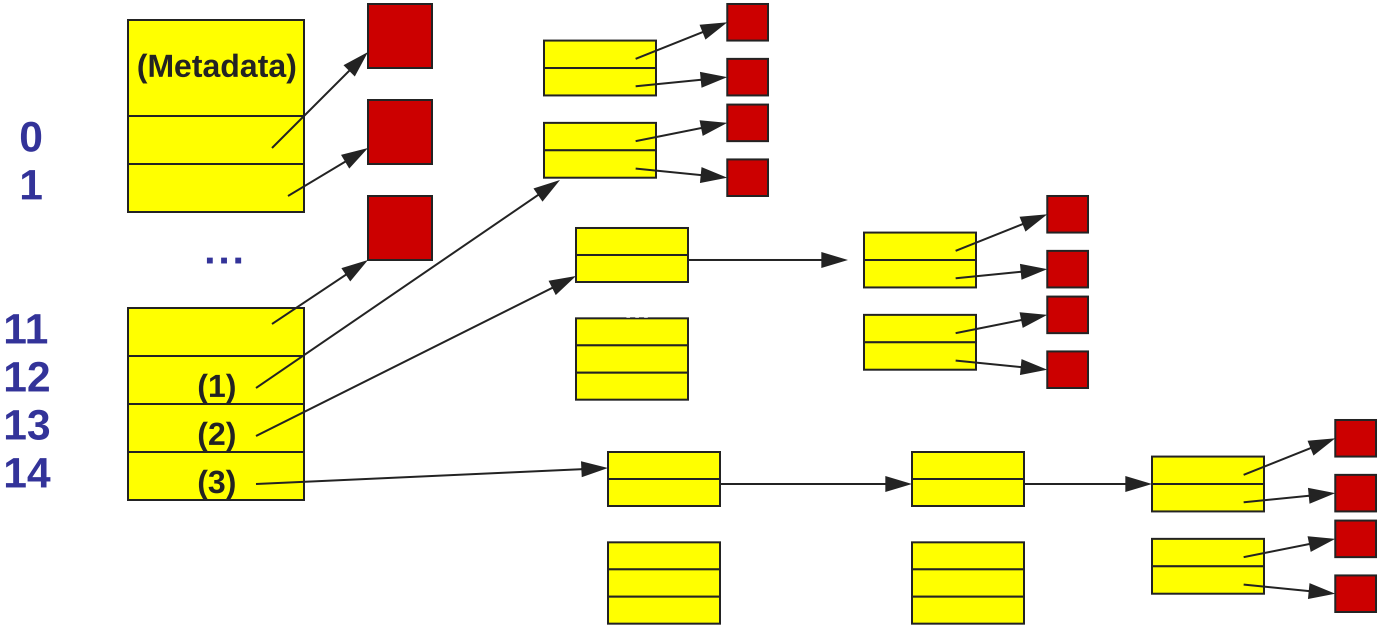
2. Linked structure



    - » Each block points to the next, directory points to the first
    - » Bad for random access patterns
3. Indexed structure (indirection, hierarchy)
    - » An “index block” contains pointers to many other blocks
    - » Handles random better, still good for sequential
    - » May need multiple index blocks (linked together)

Unix Inodes

- Unix inodes implement an indexed structure for files
    - Also store metadata info (protection, timestamps, length, ref count…)
- Each inode contains 15 block pointers
    - First 12 are direct blocks (e.g., 4 KB blocks)
    - Then single, double, and triple indirect



Unix Inodes and Path Search

- Unix Inodes are not directories
- Inodes describe where on disk the blocks for a file are placed
  - Directories are files, so inodes also describe where the blocks for directories are placed on the disk
- Directory entries map file names to inodes
  - To open “/one”, use Master Block to find inode for “/” on disk
  - Open “/”, look for entry for “one”
  - This entry gives the disk block number for the inode for “one”
  - Read the inode for “one” into memory
  - The inode says where first data block is on disk
  - Read that block into memory to access the data in the file
- This is why we have *open* in addition to *read* and *write*

File Buffer Cache

- Applications exhibit significant locality for reading and writing files
- Idea: Cache file blocks in memory to capture locality
  - This is called the file buffer cache
  - Cache is system wide, used and shared by all processes
  - Reading from the cache makes a disk perform like memory
  - Even a 4 MB cache can be very effective
- Issues
  - The file buffer cache competes with VM (tradeoff here)
  - Like VM, it has limited size
  - Need replacement algorithms again (LRU usually used)

Caching Writes

- On a write, some applications assume that data makes it through the buffer cache and onto the disk
    - As a result, writes are often slow even with caching
- Several ways to compensate for this
    - "write-behind"
        - Maintain a queue of uncommitted blocks
        - Periodically flush the queue to disk
        - Unreliable
    - Battery backed-up RAM (NVRAM)
        - As with write-behind, but maintain queue in NVRAM
        - Expensive
    - Log-structured file system
        - Always write next block after last block written
        - Complicated

Read Ahead

- Many file systems implement “read ahead”
  - FS predicts that the process will request next block
  - FS goes ahead and requests it from the disk
  - This can happen while the process is computing on previous block
    - Overlap I/O with execution
  - When the process requests block, it will be in cache
  - Compliments the disk cache, which also is doing read ahead
- For sequentially accessed files can be a big win
  - Unless blocks for the file are scattered across the disk
  - File systems try to prevent that, though (during allocation)

Improving Performance

- Disk reads and writes take order of milliseconds
  - Very slow compared to CPU and memory speeds

- How to speed things up?
  - File buffer cache
  - Cache writes
  - Read ahead

FFS, LFS, RAID

- Now we're going to look at some example file and storage systems
  - BSD Unix Fast File System (FFS)
  - Log-structured File System (LFS)
  - Redundant Array of Inexpensive Disks (RAID)

Fast File System

- The original Unix file system had a simple, straightforward implementation
  - Easy to implement and understand
  - But very poor utilization of disk bandwidth (lots of seeking)
- BSD Unix folks did a redesign (mid 80s) that they called the Fast File System (FFS)
  - Improved disk utilization, decreased response time
  - McKusick, Joy, Leffler, and Fabry
- Now the FS from which all other Unix FS's are compared
- Good example of being device-aware for performance

Data and Inode Placement

Original Unix FS had two placement problems:

1. Data blocks allocated randomly in aging file systems
   - Blocks for the same file allocated sequentially when FS is new
   - As FS “ages” and fills, need to allocate into blocks freed up when other files are deleted
   - Problem: Deleted files essentially randomly placed
   - So, blocks for new files become scattered across the disk
2. Inodes allocated far from blocks
   - All inodes at beginning of disk, far from data
   - Traversing file name paths, manipulating files, directories requires going back and forth from inodes to data blocks

Both of these problems generate many long seeks

Summary

- Protection
  - ACLs vs. capabilities
- File System Layouts
  - Unix inodes
- File Buffer Cache
  - Strategies for handling writes
- Read Ahead
- UNIX file system
  - Indexed access to files using inodes
- FFS
  - Improve performance by localizing files to cylinder groups

Next time...

- File system optimizations