File System Naming and Reliability

Naming

- File needs to have a handle that allows for references
- OS uses simple numbers as names, which aren't usable by people
- Need a better way to name files which is user friendly and allows for easy organization

Hierarchical Name Spaces

- Graphical organization
- Typically organized using directories
    - A file containing references to other files
- Nested directories can form a tree
    - A file name is a path through the tree
    - Can form a directed graph if files are allowed to have multiple names

Directories are Files

- Directories are a special type of file that maps file names to associated files
- Directory contains multiple directory entries
    - Each entry describes open file and its name
- User apps are allowed to read directories
    - Can get info about files and allowed to see which file exist
    - Usually write permissions are granted by the OS

Traversing the Tree

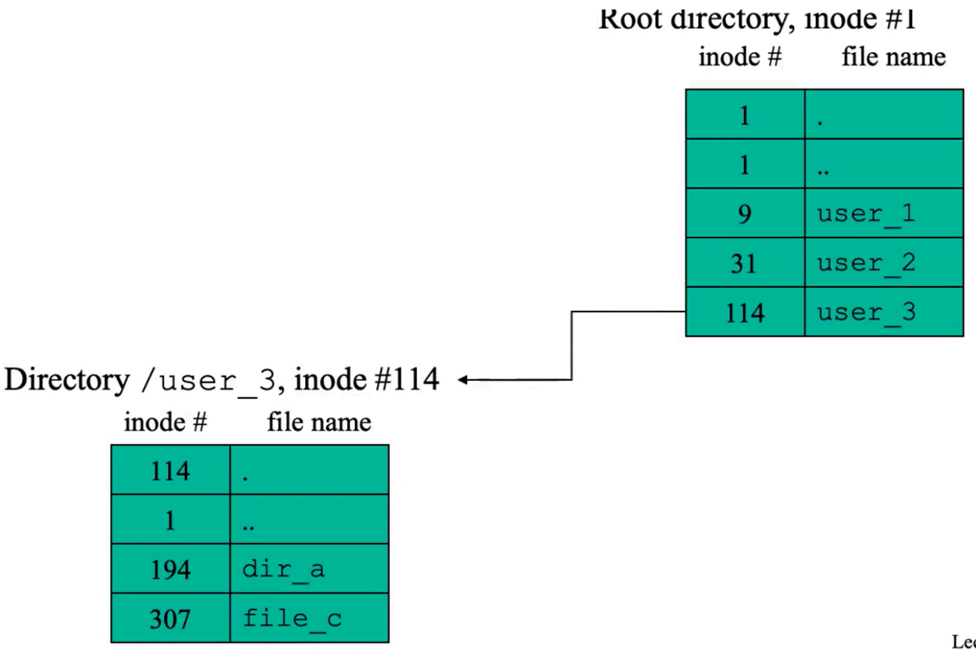
- Entries in directories point to child directories
    - Describes a lower level in the hierarchy
- To name a file at that level, name the parent directory and the child directory
- Moving up the hierarchy is often useful
    - Directories have special entries to point to the parent ".."

File Names vs Path Names

- True Names
    - One possible name for the file
    - In DOS, file is described by a directory entry

Hard Links

- File names separated by slashes
    - /user/fdsf
- Directory entries only point to inodes and are a mapping between file names and inodes
- Hard Links / Directory entry
    - File name and inode number, known as "hardlink"
    - Multiple directory entries point to the same inode



Multiple File Names in Unix

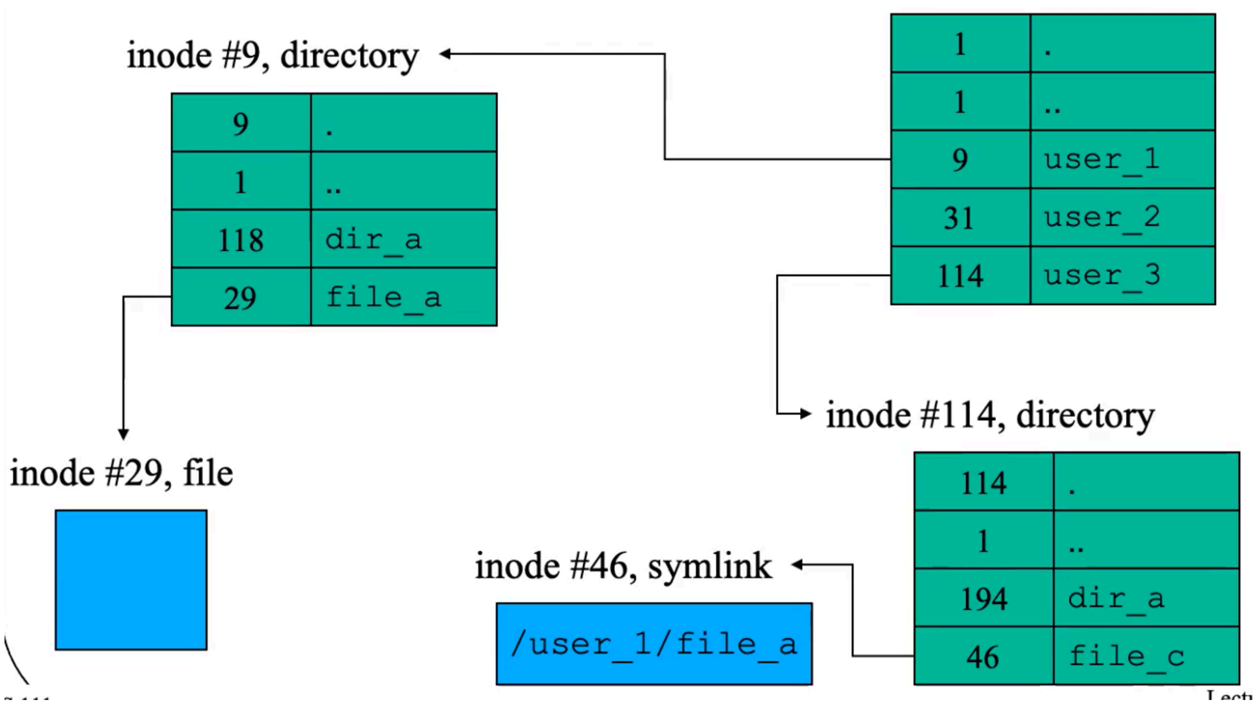
- How links relate to files
  - They're only names
- All other metadata is stored in the file inode
- All links provide the same access to the file
  - Anyone with read access can create a new link
- All links are equal
  - Nothing special about the first link

Links and Deallocation

- If files exist under many names, can't remove the data until all names have been removed
- Hard link count is stored in the inode

Symbolic Links

- Different way of giving files multiple names
- Symbolic links implemented as a special type of file
  - Indirect reference to some other file
  - Contents is a path to another file
- File system recognizes symbolic links
- Symbolic link isn't a reference to the inode
  - Symbolic links don't prevent deletion
  - Deleting the symbolic name doesn't change the link count



File Systems Reliability

- Multiple operations
    - Writing one or more metadata blocks
    - Inode, free list, and directory blocks
- All must be committed to disk for the write to succeed
- Each block write is a separate hardware operation

Deferred Writes

- Process allocates a new block to file A
    - Get a new block from the free list
    - Write out the updated inode for file A
    - Defer free list write back
- System crashes and reboot
    - New process wants a new block for file B
    - But file A is already using the block to store data

Core Reliability Problem

- File system writes typically involve multiple operations
    - Not just writing a data block to disk/flash
    - Writing one or more metadata blocks
- All must be committed to disk for the write to succeed
    - Failure on one operation could be bad
- Each block write is a separate hardware operation

Deferred Writes

- Process allocates a new block to file A
  - We get a new block from the free list
  - We write out the updated inode for file A
  - Defer free list write back (delay the write operation to batch it with more data)

- System crashes and after it reboots
  - New process wants a new block for file B
  - We get a block from the stale free list

- Two different files now contain the same block
  - File A is written, causing B to get corrupted and vice versa

App expectations When Writing

- App makes sys call to perform writes
- Upon syscall return, we expect the write to be safe
- We can block the writing app until its safe

Buffered Writes

- Don't wait for the write to actually be persisted
- Keep track of it in RAM
- Tell the app its OK
- At some later point actually write to persistent memory
- Advantages
  - Less app blocking
  - Deeper and optimizable write queries
- Disadvantages
  - Crashes could cause problems with memory allocation

Ordered Writes

- Carefully ordered writes can reduce potential damage

- Write out data before writing pointers to it
  - We want to make sure the data exists before creating the pointer to it
  - Unreferenced objects can be garbage collected
  - Pointers to incorrect info can be serious

- Write out deallocations before allocations
  - Disassociate resources from old file immediately
  - Free list can be corrected by garbage collection
  - Improperly shared data is more serious than missing data (better to lose data than to corrupt the file system)

Practicality of Ordered Writes

- Greatly reduced I/O performance
  - Eliminates accumulation of near by operations
  - Eliminated consolidation of updates to same block
- May not be possible due to device constraints
- Doesn't solve the problem
  - Ordered writes don't eliminate incomplete writes

Audit and Repair

- Design file system structures for audit and repair
- Audit file system for correctness and use redundant info to enable automatic repair
- Used to be standard practice, but isn't practical since its so slow

Journaling

- Part of our storage is devoted to a journal, which keeps track of everything that I intend to write

- Circular buffer journaling device
  - Always sequential
  - Writes can be batched
  - Journal is relatively small

- Efficiently schedule actual file system updates
  - Journal keeps track of the ordering that updates need to be made in

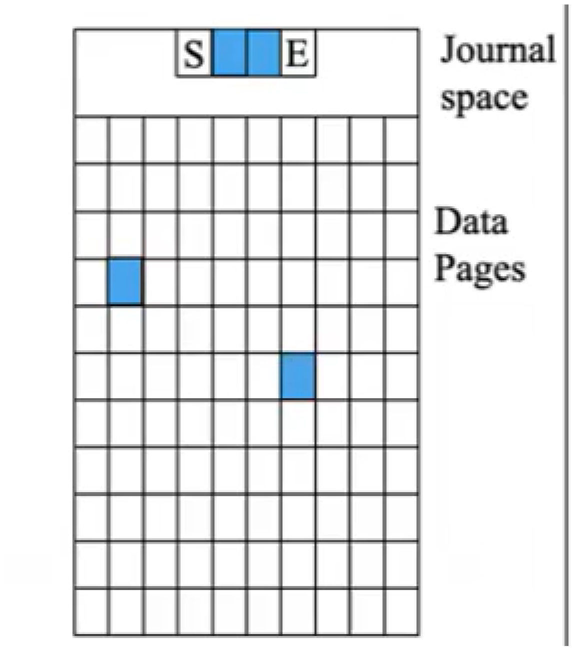
- Journal completions when <u>real writes happen</u>
  - Once journal updates are made, the contents that have been written don't need to be stored anymore

- Upon system crash, go to the journal to see what needs to be done
  - Even if the journal wasn't cleared during the crash or there was an intermediate step crash, it will still work

Journal example

- Write 2 pages

1. Put start record into the journal
2. Put the 2 pages into the journal
3. Put the end record into the journal
4. Overwrite the data, then delete entries from the journal



Batching Journal Entries

- Operation is safe after journal entry persisted
- Small writes are still inefficient
- Accumulate batch until full or max wait time

Journal Recovery

- It's a small circular buffer that can be recycled after old ops have completed
  - Time stamps distinguish new entries from old
- After the system restarts
  - Review entire journal
  - Note which ops were known to have completed
  - Perform all writes not known to have completed
- Truncate journal and resume normal operation

Journalling Functionality

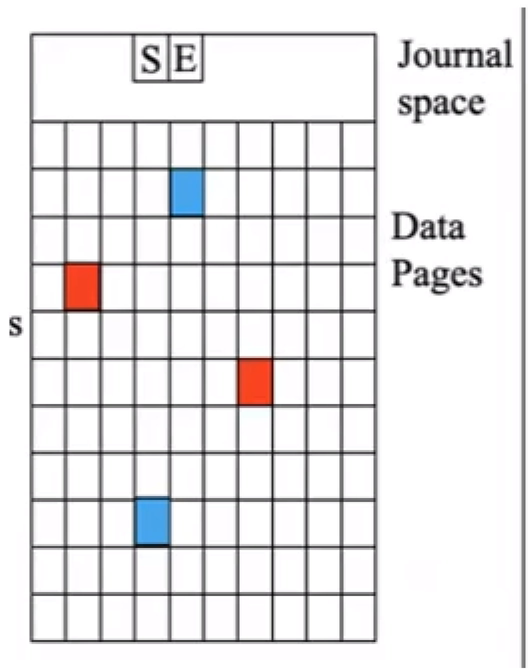
- Journal writes are much faster than data writes
- In normal operation, journal is write only
  - File system never reads/processes the journal
- Scanning the journal on restart is fast
  - Very small
  - Can read sequentially with large and efficient reads
  - All recovery processing is done in memory
- Journal pages may contain information for multiple files
  - Performed by different processes and users

Metadata Only Journaling

- Journal meta-data
  - Only contains the metadata
  - Small and random
  - Integrity critical

- Journal data
  - Large and sequential
  - Less order sensitive

- Safe metadata journaling
  1. Allocate new space for the data and write it there
     a. Data could take a lot of space, so we should write it to disk instead of storing in journal
  2. Journal the metadata updates
     a. Consistency in the metadata is extremely important
     b. Metadata points to the data that we stored on disk



Log Structured File Systems

- Journal is the file system (called a log)
  - All inodes and data updates written to the log
  - All updates in a log are updates using redirect on write (don't overwrite the old data, write it elsewhere then change the metadata pointer to it)
    - This means that old versions remain, but the inode pointer points to new data
  - In memory index caches inode locations

- Dominant architecture
    - Flash file systems
    - Key/value store

- Issues
    - Recovery time (to reconstruct index/cache)
    - Log defragmentation and garbage collection

Navigating a Logging File System

- Inodes point at data segments (pages) in the log
    - Sequential writes may be contiguous in log
    - Random updates can be spread across the log
- Updated inodes are added to the end of the log
- Index points to latest version of each inode
    - Index is periodically appended to the log
- Recovery
    - Find and recover the latest index
    - Replay all log updates since then

Redirect on Write

- Once written, blocks and inode are immutable
    - Add new info to the log and update the index
- Old inodes and data remain in the log
    - If we have an old index, we can access them
- Garbage collections is used to recycle old log entries