File Systems

Inf-2201, University of Tromsø
Loic Guegan (loic.guegan@uit.no)

Based on slides from Lars Ailo Bongo(UiT), Kai Li (Princeton University)

Overview

- Part I
  - File system abstractions and operations
  - Protection
  - File system structure
    - Disk allocation and i-nodes
    - Directory and link implementations
    - Physical layout for performance
- Part II
  - Performance and reliability
    - File buffer cache
    - Disk failure and file recovery tools
    - Consistent updates
    - Transactions and logging

Why Files?

- Can’t we just use main memory?
- Can’t we use a mechanism like swapping to disk?

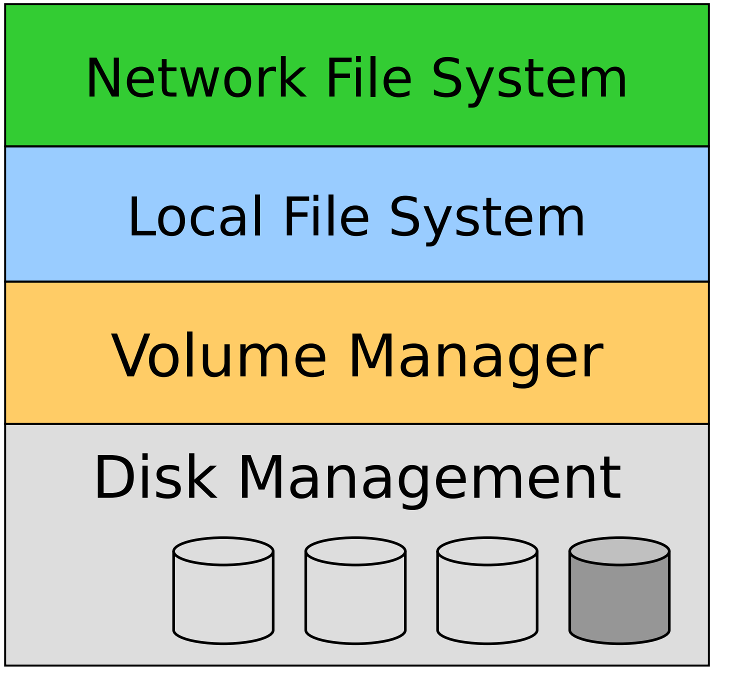
- Need to store large amount of information
- Need the information to survive process termination
- Need the information to be shareable by processes

Recall Some High-level Abstractions

- Processes are an abstraction for processors (CPU)
- Virtual memory is an abstraction for memory
- File systems are an abstraction for disks (disk blocks)

File System Layers and Abstractions

- Network file system maps a network file system protocol to local file systems
  - NFS, CIFS, DAFS, GFS, HDFS, Dropbox, etc
- Local file system implements a file system on blocks in volumes
  - Local disks or network of disks
- Volume manager maps logical volume to physical disks
  - Provide logical unit
  - RAID and reconstruction
- Disk management manages physical disks
  - Sometimes part of volume manager
  - Drivers, scheduling, etc



Volume Manager

- Group multiple disk partitions into a logical disk volume
  - No need to deal with physical disk, sector numbers
  - To read a block: read(vol#, block#, buf, n);
- Volume can include RAID, tolerating disk failures
  - No need to know about parity disk in RAID-5, for example
  - No need to know about reconstruction
- Volume can provide error detections at disk block level
  - Some products use a checksum block for 8 blocks of data
- Volume can grow or shrink without affecting existing data
- Volume can have remote volumes for disaster recovery
- Remote mirrors can be split or merged for backups

Files vs. Block Storage

File abstraction

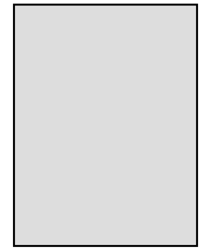
- Byte oriented
- Named files
- Users protected from each other
- Robust to machine failures

Disk abstraction

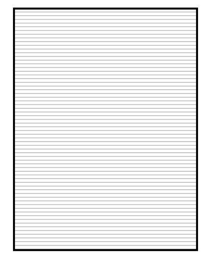
- Block oriented
- Block numbers
- No protection among users of the system
- Data might be corrupted if machine crashes

File Structure Possibilities

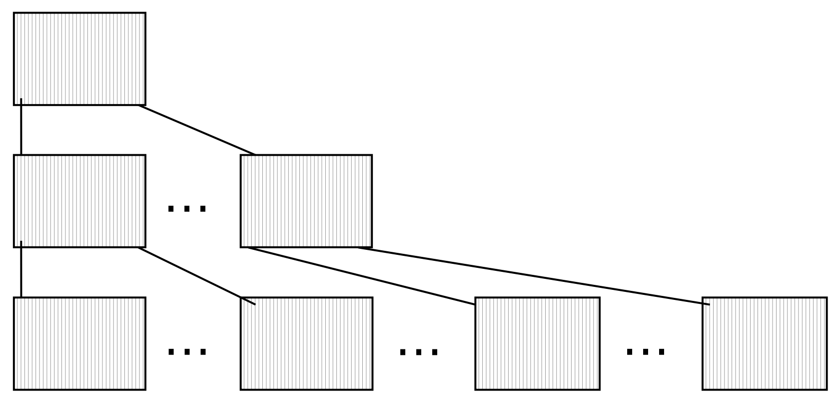
- Byte sequence
  - Read or write a number of bytes
  - Unstructured or linear
  - Unix, Windows



- Record sequence
  - Fixed or variable length
  - Read or write a number of records
  - Not used: punch card days



- Tree
  - Records with keys
  - Read, insert, delete a record (typically using B-tree, sorted on key)
  - Used in mainframes for commercial data processing



File Types, examples

- ASCII
- Binary data
  - Record
  - Tree
  - An Unix executable file
    - header: magic number, sizes, entry point, flags
    - text
    - data
    - relocation bits
    - symbol table
- Devices
- Everything else in the system

Most common file operations

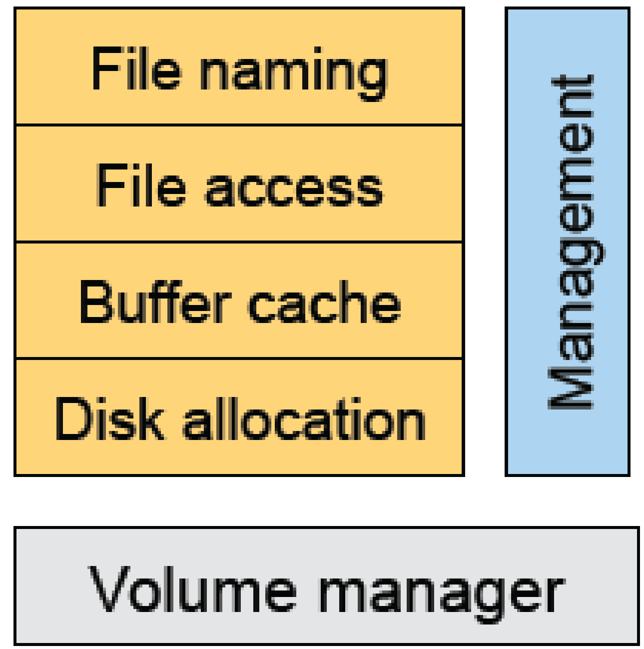
- Operations for “sequence of bytes” files
  - Create: create a mapping from a name to bytes
  - Delete: delete the mapping
  - Open: authentication, bring key attributes, disk info into RAM
  - Close: free up table space, force last block write
  - Seek: jump to a particular location in a file
  - Read: read some bytes from a file
  - Write: write some bytes to a file
  - Get attributes, Set attributes
- Implementation goal
  - Operations should have as few disk accesses as possible and have minimal space overhead

Access Patterns

- Sequential (the common pattern)
  - File data processed sequentially
  - Examples
    - Editor writes out a new file
    - Compiler reads a file
- Random access
  - Address a block in file directly without passing through predecessors
  - Examples:
    - Data set for demand paging
    - Databases
- Keyed access
  - Search for a record with particular values
  - Usually not provided by today’s file systems
  - Examples
    - Database search and indexing

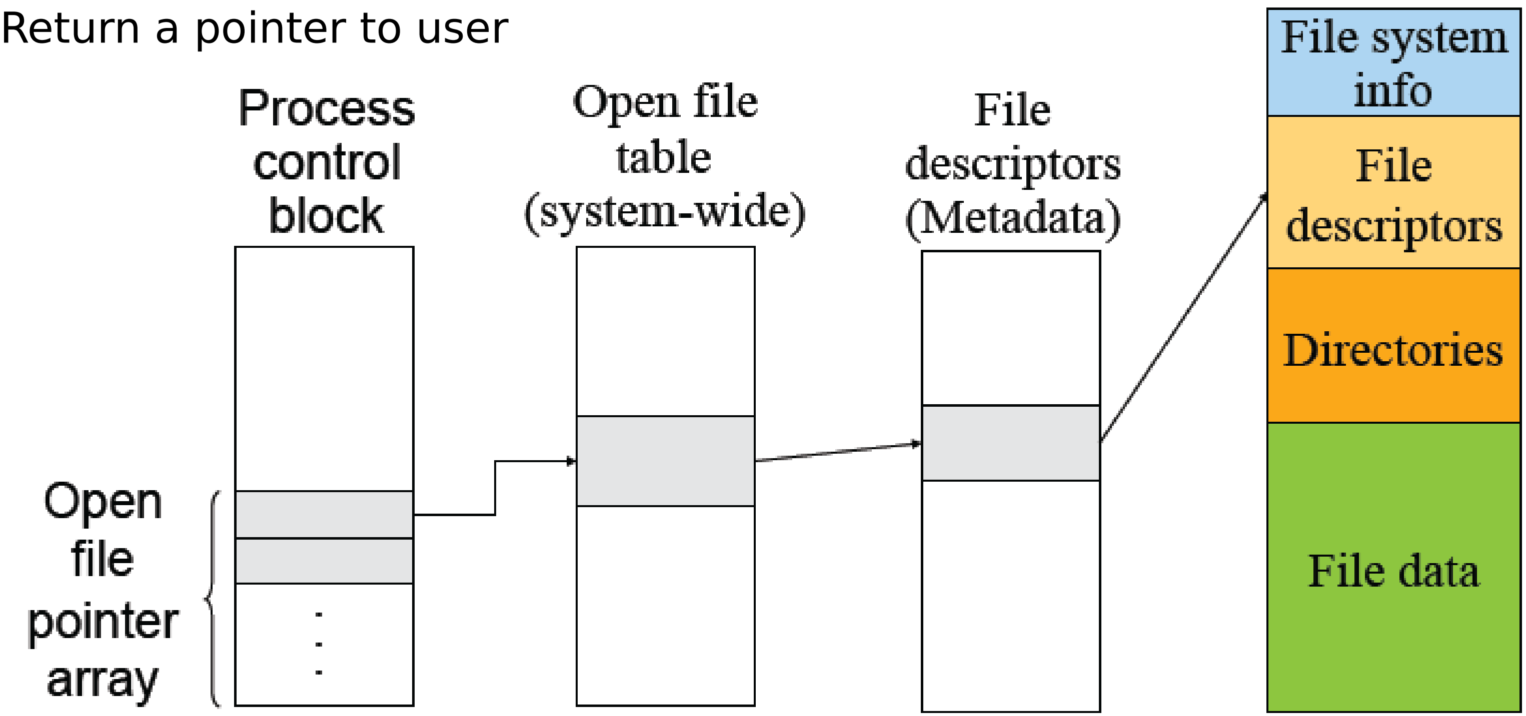
File System Components

- Naming
  - File and directory naming
  - Local and remote operations
- File access
  - Implement read/write and other functionalities
- Buffer cache
  - Reduce client/server disk I/Os
- Disk allocation
  - File data layout
  - Mapping files to disk blocks
- Management
  - Tools for system administrators to manage file systems



Steps to Open a file

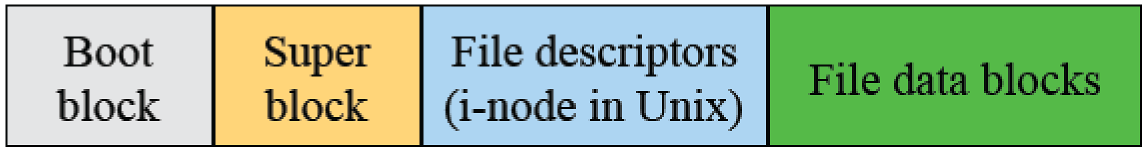
- File name lookup and authenticate
- Copy the file descriptors into the in-memory data structure, if it is not in yet
- Create an entry in the open file table (system wide) if there isn't one
- Create an entry in PCB
- Link up the data structures
- Return a pointer to user



File Read and Write

- Read 10 bytes from a file starting at byte 2?
  - seek byte 2
  - fetch the block
  - read 10 bytes
- Write 10 bytes to a file starting at byte 2?
  - seek byte 2
  - fetch the block
  - write 10 bytes in memory
  - write out the block

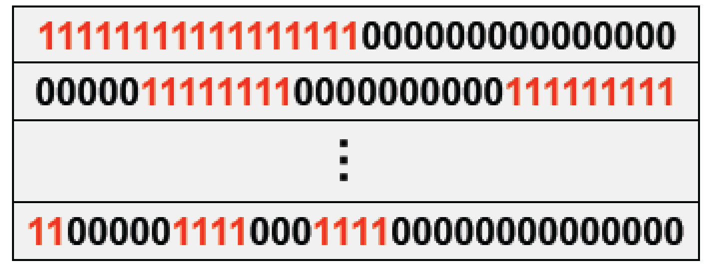
Disk Layout

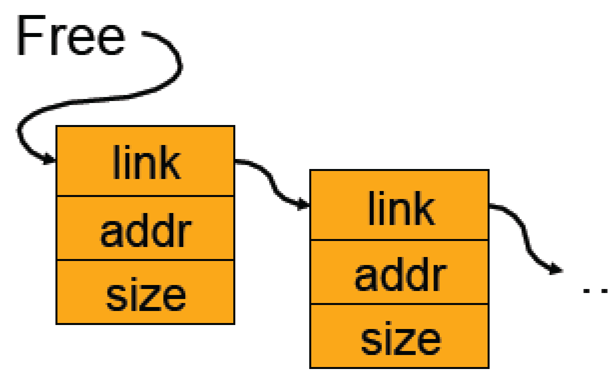


- Boot block
  - Code to bootstrap the operating system
- Super-block defines a file system
  - Size of the file system
  - Size of the file descriptor area
  - Free list pointer, or pointer to bitmap
  - Location of the file descriptor of the root directory
  - Other meta-data such as permission and various times
  - Kernel keeps in main memory, replicated on disk
- File descriptors
- File data blocks
  - Data for the files, the largest portion on disk

Data Structures for Disk Allocation

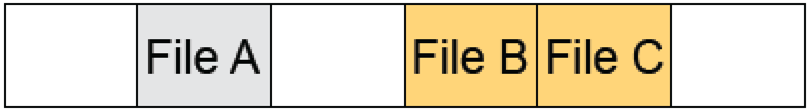
- The goal is to manage the allocation of a volume
- A file header for each file
  - Disk blocks associated with each file
- A data structure to represent free space on disk
  - Bit map that uses 1 bit per block (sector)
  - Linked list that chains free blocks together
  - …





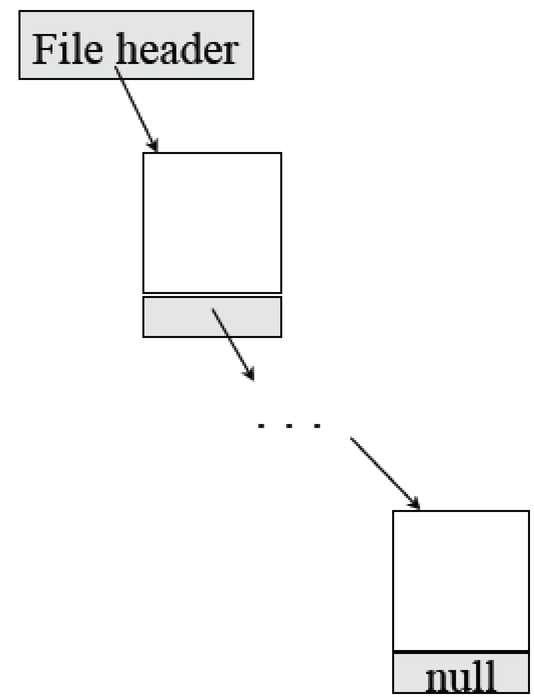
Contiguous Allocation

- Request in advance for the size of the file
- Search bit map or linked list to locate a space
- File header
  - First block in file
  - Number of blocks
- Pros
  - Fast sequential access
  - Easy random access
- Cons
  - External fragmentation (what if file C needs 3 blocks)
  - Hard to grow files: may have to move (large) files on disk
  - May need compaction



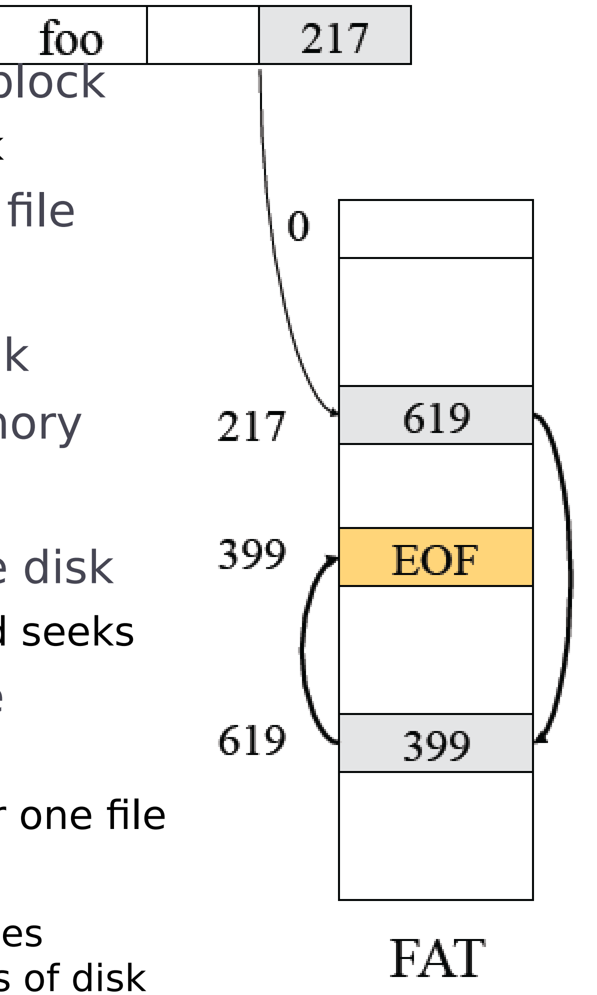
Linked Files

- File header points to 1st block on disk
- A block points to the next
- Pros
  - Can grow files dynamically
  - Free list is similar to a file
  - No external fragmentation or need to move files
- Cons
  - Random access: horrible
  - Even sequential access needs one seek per block
  - Unreliable: losing a block means losing the rest



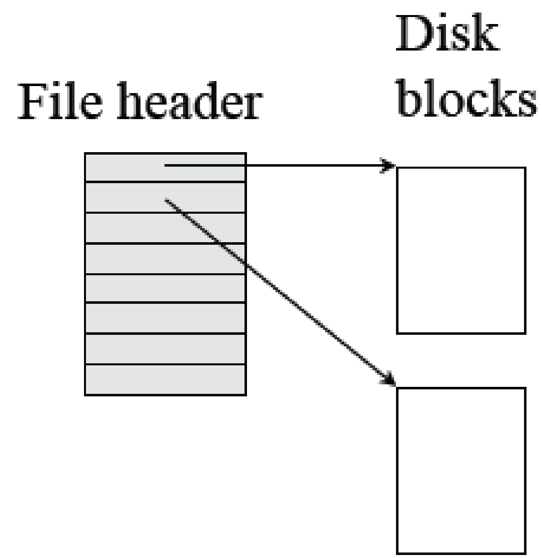
File Allocation Table (FAT)

- Approach
  - Table of “next pointers”, indexed by block
    - Instead of pointer stored with block
  - Directory entry points to 1st block of file
- Pros
  - No need to traverse list to find a block
  - Cache FAT table and traverse in memory
- Cons
  - FAT table takes lots of space for large disk
    - Hard to fit in memory; so may need seeks
  - Pointers for all files on whole disk are interspersed in FAT table
    - Need full table in memory, even for one file
    - Solution: indexed files
      - Keep block lists for different files together, and in different parts of disk



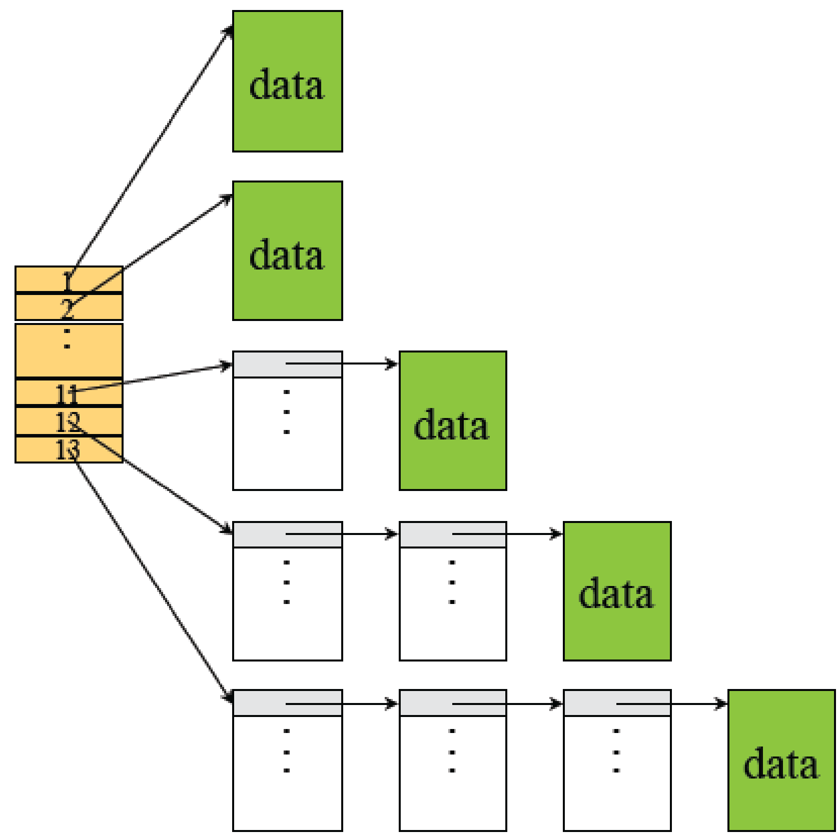
Single-Level Indexed Files

- A file header holds an array of pointers to point to disk blocks
- Pros
  - Can grow up to a limit
  - Random access is fast
- Cons
  - Clumsy to grow beyond the limit
  - Still lots of seeks



Multi-Level Indexed Files (Unix)

- 13 Pointers in a header
  - 1…10: direct pointers
  - 11: 1-level indirect
  - 12: 2-level indirect
  - 13: 3-level indirect
- Pros & Cons
  - In favor of small files
  - Can grow
  - Limit is 16G and lots of seek

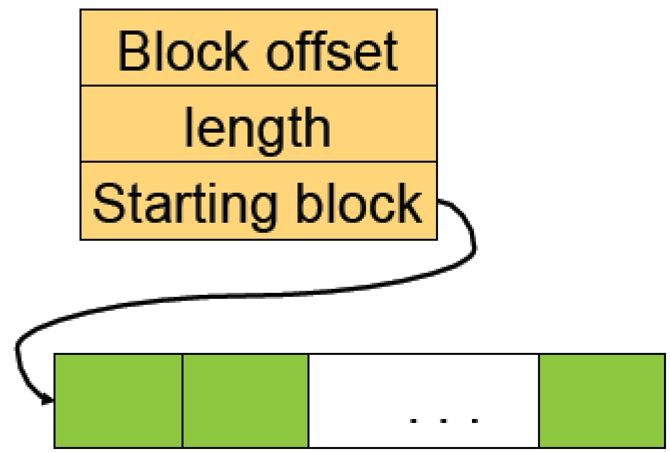


What’s in Original Unix i-node?

- Mode: file type, protection bits, setuid, setgid bits
- Link count: number of directory entries pointing to this
- Uid: uid of the file owner
- Gid: gid of the file owner
- File size
- Times (access, modify, change)
- No filename (why?)
- 10 pointers to data blocks
- Single indirect pointer
- Double indirect pointer
- Triple indirect pointer

Extents

- Instead of using a fixed size block, use a number of blocks
  - XFS uses 8Kbyte block
  - Max extent size is 2M blocks
- Index nodes need to have
  - Block offset
  - Length
  - Starting block



Directory Organization Examples

- Flat
  - All files are in one directory
- Hierarchical (Unix)
  - /home/foo/bar
  - Directory is stored in a file containing (name, i-node) pairs
  - The name can be either a file or a directory

Mapping File Names to i-nodes

- Create/delete
  - Create/delete a directory
- Open/close
  - Open/close a directory for read and write
- Link/unlink
  - Link/unlink a file
- Rename
  - Rename the directory

Linear List

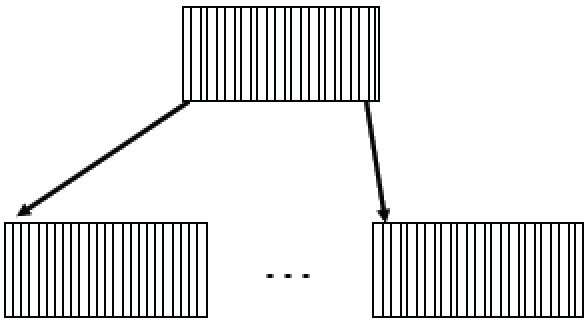
- Method
  - <FileName, i-node> pairs are linearly stored in a file
  - Create a file
    - Append <FileName, i-node>
- Delete a file
  - Search for FileName
  - Remove its pair from the directory
  - Compact by moving the rest
- Pros
  - Space efficient
- Cons
  - Linear search
  - Need to deal with fragmentation

- /home/userY/foo/bar/… veryLongFileName

<foo,1234> <bar,1235> … <veryLongFileName, 4567>

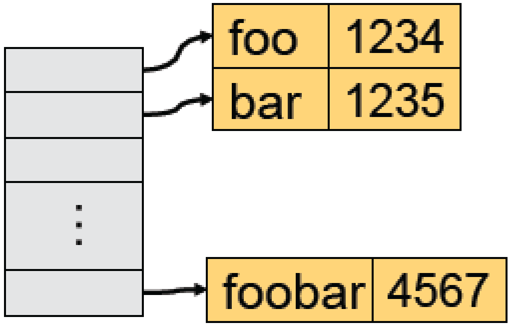
Tree Data Structure

- Method
  - Store <fileName, i-node> a tree data structure such as B-tree
  - Create/delete/search in the tree data structure
- Pros
  - Good for a large number of files
- Cons
  - Inefficient for a small number of files
  - More space
  - Complex



Hashing

- Method
  - Use a hash table to map FileName to i-node
  - Space for name and metadata is variable sized
  - Create/delete will trigger space allocation and free
- Pros
  - Fast searching and relatively simple
- Cons
  - Not as efficient as trees for very large directory (wasting space for the hash table)



Disk I/Os to Read/Write A File

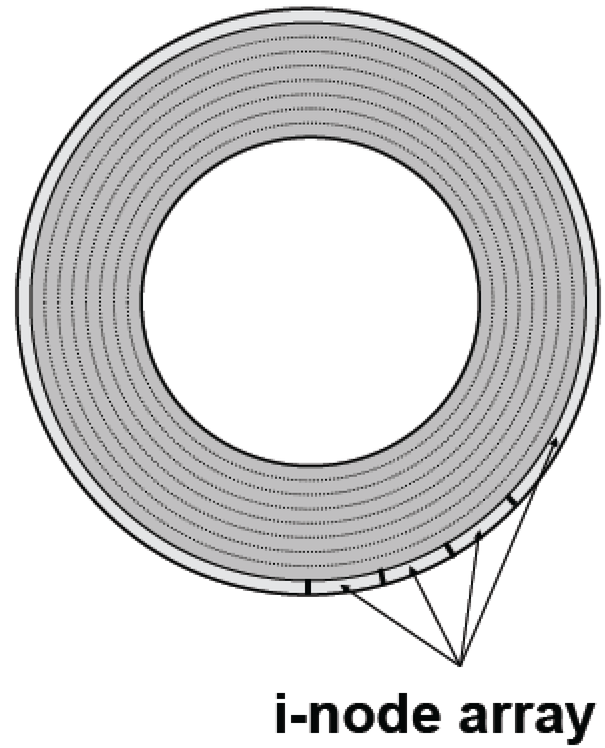
- Disk I/Os to access a byte of /home/foo/bar
  - Read the i-node and first data block of "/"
  - Read the i-node and first data block of "home"
  - Read the i-node and first data block of "foo"
  - Read the i-node and first data block of "bar"
- Disk I/Os to write a file
  - Read the i-node of the directory and the directory file.
  - Read or create the i-node of the file
  - Read or create the file itself
  - Write back the directory and the file
- Too many I/Os to traverse the directory
  - Solution is to use ***Current Working Directory***

Links

- Symbolic (soft) links
  - A symbolic link is just the name of the file
  - Original owner still owns the file, deleted on rm by owner
  - Use a new i-node for the symbolic link
    ln –s source target
- Hard links
  - A link to a file with the same i-node
    ln source target
  - Delete may or may not remove the target depending on whether it is the last one (link reference count)

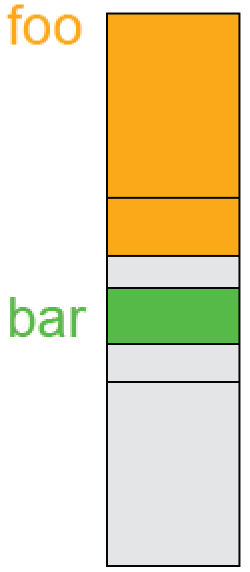
Original Unix File System

- Simple disk layout
  - Block size is sector size (512 bytes)
  - i-nodes are on outermost cylinders
  - Data blocks are on inner cylinders
  - Use linked list for free blocks
- Issues
  - Index is large
  - Fixed max number of files
  - i-nodes far from data blocks
  - i-nodes for directory not close together
  - Consecutive blocks can be anywhere
  - Poor bandwidth (20Kbytes/sec even for sequential access!)



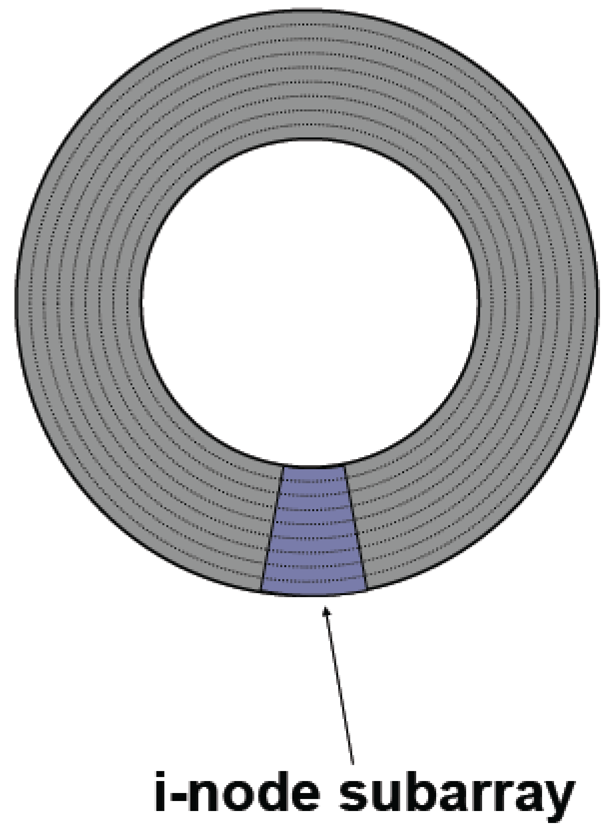
BSD FFS (Fast File System)

- Use a larger block size: 4KB or 8KB
  - Allow large blocks to be chopped into fragments
- Use bitmap instead of a free list
  - Try to allocate contiguously
  - 10% reserved disk space



FFS Disk Layout

- i-nodes are grouped together
  - A portion of the i-node array on each cylinder
- Do you ever read i-nodes without reading any file blocks?
- Overcome rotational delays
  - Skip sector positioning to avoid the context switch delay
  - Read ahead: read next block right after the first



What Has FFS Achieved?

- Performance improvements
  - 20-40% of disk bandwidth for large files (10-20x original)
  - Better small file performance
- We can still do a lot better
  - Extent based instead of block based
    - Use a pointer and size for all contiguous blocks (XFS, Veritas file system, etc)
  - Synchronous metadata writes hurt small file performance
    - Asynchronous writes with certain ordering (“soft updates”)
    - Logging (talk about this later)
    - Play with semantics (/tmp file systems)

Side note : Protection Policy vs. Mechanism

- A protection system is the mechanism to enforce a security policy
  - Roughly the same set of choices, no matter what policy
- A security policy determines what is acceptable or not
  - Example security policies:
    - Each user can only allocate 40GB of disk
    - No one but root can write to the password file
    - You cannot read my mail

Protection Mechanisms

- Authentication
  - Make sure system knows whom it is talking to
    - Unix: password
    - US banks: account # + last transactions
    - Bars: driver's license
- Authorization
  - Determine if "X" is allowed to do "Y"
  - Need a simple database
- Access enforcement
  - Enforce authorization decision
  - Must make sure there are no loopholes
  - Hard to assert

Protection Domain

- A set of (objects, rights) pairs
  - Domain may correspond to single user, or more general
  - Process runs in a domain at a given instant in time
- Once identity known, what is Bob allowed to do?
  - More generally: must be able to determine what each “principal” is allowed to do with what
- Can be represented as a “protection matrix” with one row per domain, one column per resource
- What are the pros and cons of this approach?

	File A	Printer B	File C
Domain 1	R	W	RW
Domain 2	RW	W	…
Domain 3	R	…	RW

Access Control Lists (ACLs)

- By column: For each object, indicate which users are allowed to perform which operations
  - In most general form, each object has a list of <user,privileged> pairs
- Access control lists are simple, and are used in almost all file systems
  - Owner, group, world
- Implementation
  - Stores ACLs in each file
  - Use login authentication to identify
  - Kernel implements ACLs

Capabilities

- By rows: For each user, indicate which files may be accessed and in what ways
  - Store a lists of <object, privilege> pairs for each user.
    - Called a *Capability List*
- Capabilities frequently do both naming and protection
  - Can only “see” an object if you have a capability for it.
  - Default is no access
- Implementation
  - Capability lists
    - Architecture support
    - Stored in the kernel
    - Stored in the user space but in encrypted format
  - Checking is easy: no enumeration

Access Enforcement

- Use a trusted party to
  - Enforce access controls
  - Protect authorization information
- Kernel is the trusted party
  - This part of the system can do anything it wants
  - If it has a bug, the entire system can be destroyed
  - Want it to be as small & simple as possible
- Security is only as strong as the weakest link in the protection system

Summary - Part 1

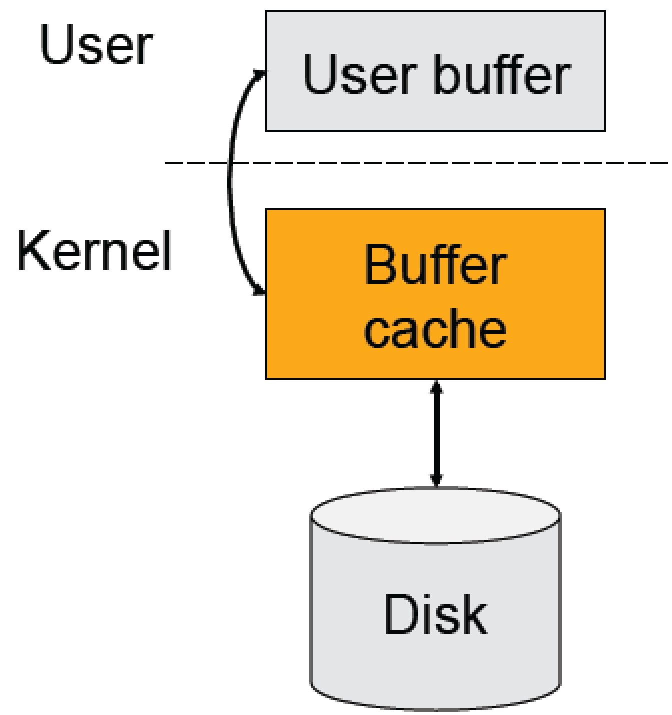
- Protection
  - We basically live with access control list
  - More protection is needed in the future
- File system structure
  - Boot block, super block, file metadata, file data
- File metadata
  - Consider efficiency, space and fragmentation
- Directories
  - Consider the number of files
- Links
  - Soft vs. hard
- Physical layout
  - Where to put metadata and data

Overview

- Part I
  - File system abstractions and operations
  - Protection
  - File system structure
    - Disk allocation and i-nodes
    - Directory and link implementations
    - Physical layout for performance
- Part II
  - Performance and reliability
    - File buffer cache
    - Disk failure and file recovery tools
    - Consistent updates
    - Transactions and logging

File Buffer Cache for Performance

- Cache files in main memory
  - Check the buffer cache first
  - Hit will read from or write to the buffer cache
  - Miss will read from the disk to the buffer cache
- Usual questions
  - What to cache?
  - How to size?
  - What to prefetch?
  - How and what to replace?
  - Which write policies?



What to Cache?

- Things to consider
  - I-nodes and indirect blocks of directories
  - Directory files
  - I-nodes and indirect blocks of files
  - Files
- What is a good strategy?
  - Cache i-nodes and indirect blocks if they are in use?
  - Cache only the i-nodes and indirect blocks of the current directory?
  - Cache an entire file vs. referenced blocks of files?

How to Size?

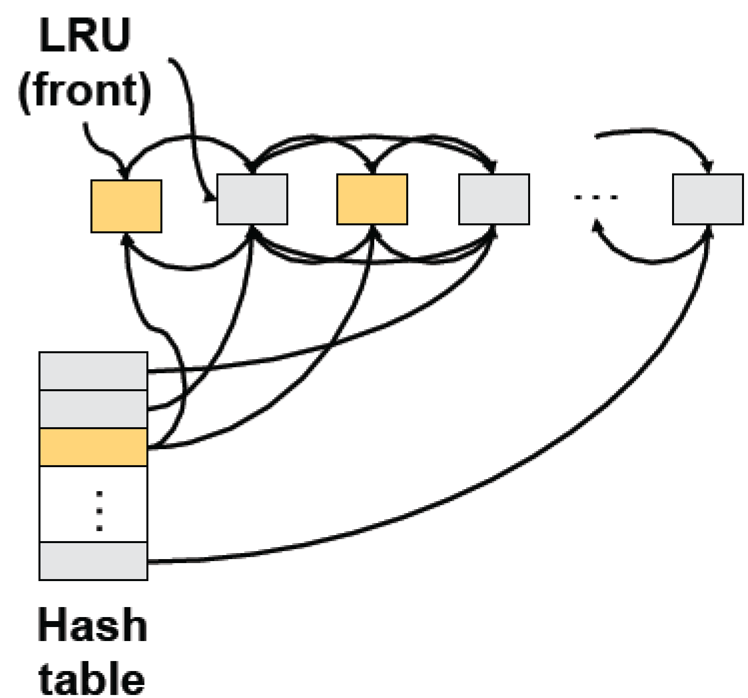
- An important issue is how to partition memory between the buffer cache and VM cache
- Early systems use fixed-size buffer cache
  - It does not adapt to workloads
- Later systems use variable size cache
  - But, large files are common, how do we make adjustment?
- Basically, we solve the problem using the *working set idea*

What to Prefetch?

- Optimal
  - The blocks are fetched in just enough time to use them
  - But, too hard to do
- The good news is that files also have locality
  - Temporal locality
  - Spatial locality
- Common strategies
  - Prefetch next k blocks together (typically > 64KB)
  - Some discard unreferenced blocks
  - Cluster blocks of the same directory and i-nodes if possible (to the same cylinder group and neighborhood) to make prefetching efficient

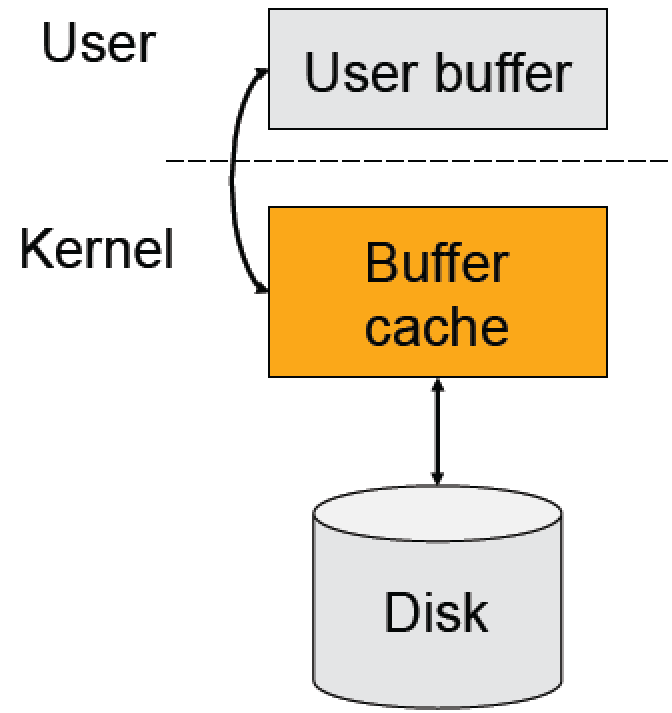
How and What to Replace?

- Page replacement theory
  - Use past to predict future
  - LRU is good
- Buffer cache with LRU replacement mechanism
  - If b is in buffer cache, move it to front and return b
  - Otherwise, replace the tail block, get b from disk, insert b to the front
  - Use double linked list with a hash table



Which Write Policies?

- Write through
  - Whenever modify cached block, write block to disk
  - Cache is always consistent
  - Simple, but cause more I/Os
- Write back
  - When modifying a block, mark it as dirty & write to disk later
  - Fast writes, absorbs writes, and enables batching
  - So, what's the problem?

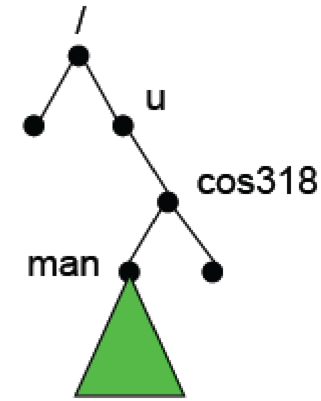


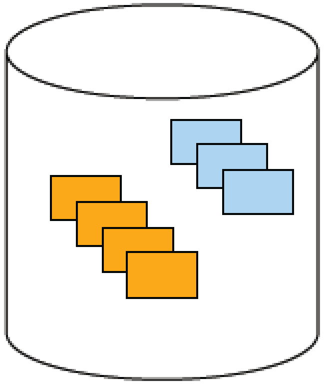
Write Back Complications

- Fundamental tension
  - On crash, all modified data in cache is lost.
  - The longer you postpone write backs, the faster you are but the worst the damage is on a crash
- When to write back
  - When a block is evicted
  - When a file is closed
  - On an explicit flush
  - When a time interval elapses (30 seconds in Unix)
- Issues
  - These write back options have no guarantees
  - A solution is consistent updates (later)

File Recovery Tools

- Physical backup (dump) and recovery
  - Dump disk blocks by blocks to a backup system
  - Backup only changed blocks since the last backup as an incremental
  - Recovery tool built accordingly
- Logical backup (dump) and recovery
  - Traverse the logical structure from the root
  - Selectively dump what you want to backup
  - Verify logical structures as you backup
  - Recovery tool selectively move files back
- Consistency check (e.g. fsck)
  - Start from the root i-node
  - Traverse the whole tree and mark reachable files
  - Verify the logical structure
  - Figure out what blocks are free



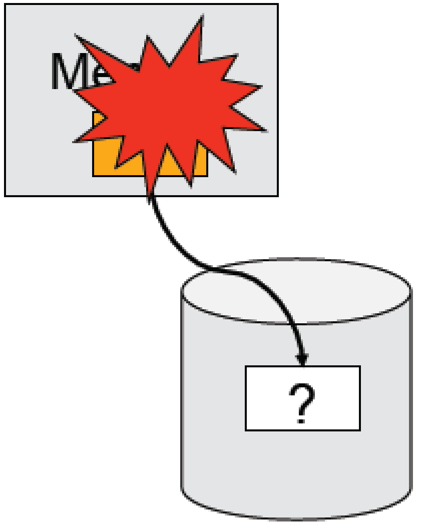


What fsck does

- Get default list of file systems to check from */etc/fstab*
- Inconsistencies checked:
  - Blocks claimed by more than one i-node or the free map
  - Blocks claimed by an i-node outside range of the filesystem
  - Incorrect link counts
  - Size checks (directory size etc)
  - Bad i-node format
  - Blocks not accounted anywhere
  - Directory checks:
    - File pointing to unallocated i-node; I-node number out of range; . or .. Not first two entries of a directory or have wrong i-node number
  - Super Block checks
    - More blocks for i-nodes than are in the filesystem; Bad free block map format; Total free block and/or free i-node count incorrect
  - Put orphaned files and directories in lost+found directory

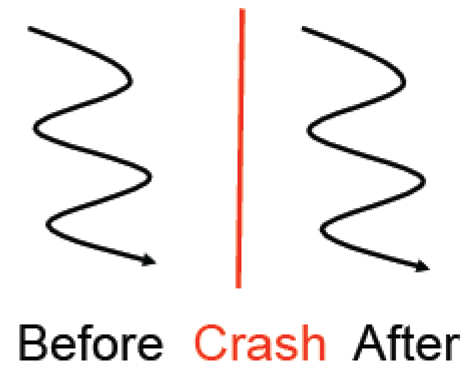
Persistency and Crashes

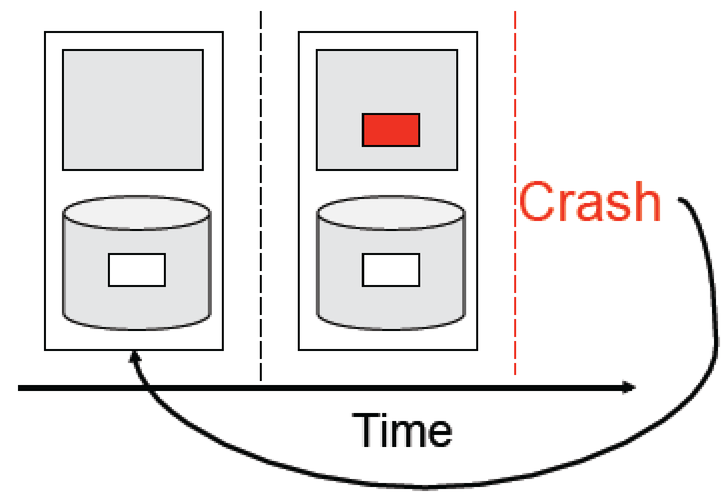
- File system promise: Persistency
  - File system will hold a file until its owner explicitly deletes it
- Why is this hard?
  - A crash will destroy memory content
  - Cache more ⇒ better performance
  - Cache more ⇒ lose more on a crash
  - A file operation often requires modifying multiple blocks, but the system can only atomically modify one at a time
  - Systems can crash anytime



What is a Crash?

- Crash is like a context switch
  - Think about a file system as a thread before the context switch and another after the context switch
  - Two threads read or write same shared state?
- Crash is like time travel
  - Current volatile state lost; suddenly go back to old state
  - Example: move a file
    - Place it in a directory
    - Delete it from old
    - Crash happens and both directories have problems





Consistent Updates: Problem

- Modify /u/cos318/foo
  - Traverse to /u/cos318/

Crash → **Consistent**

  - Allocate data block

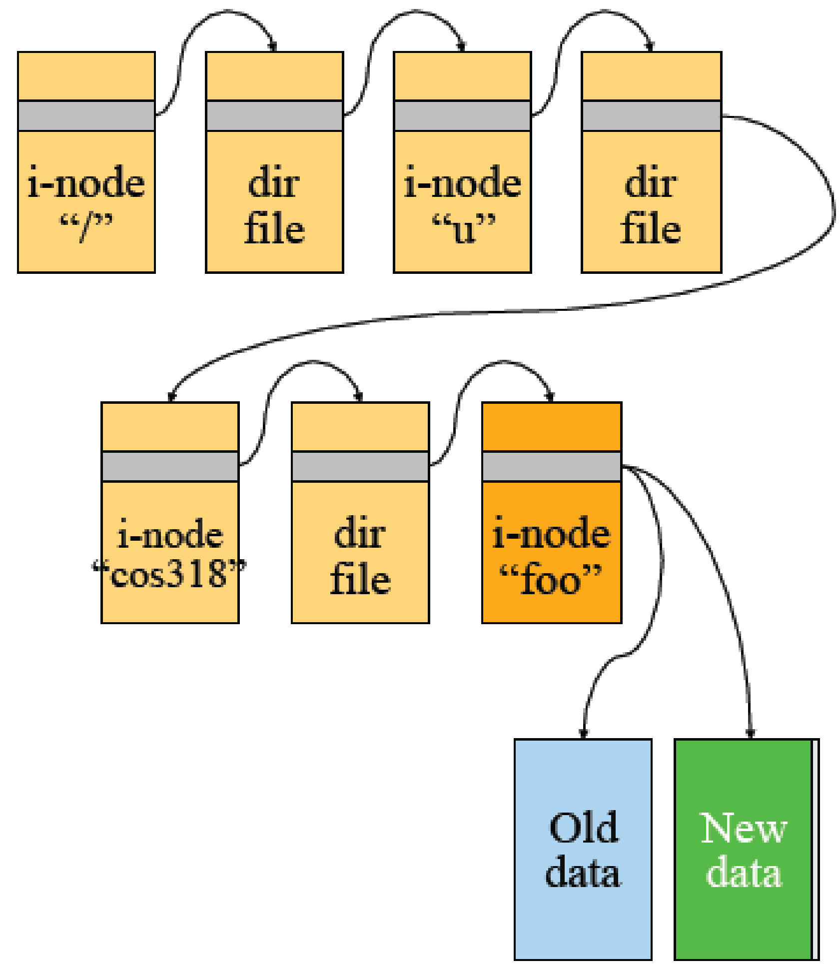
Crash → **Consistent**

  - Write pointer into i-node

Crash → **Inconsistent**

  - Write new data to foo

Crash → **Consistent**



Writing metadata first can cause inconsistency

Consistent Updates: Data Before Metadata

- Modify /u/cos318/foo
  - Traverse to /u/cos318/

Crash → Consistent

  - Allocate data block

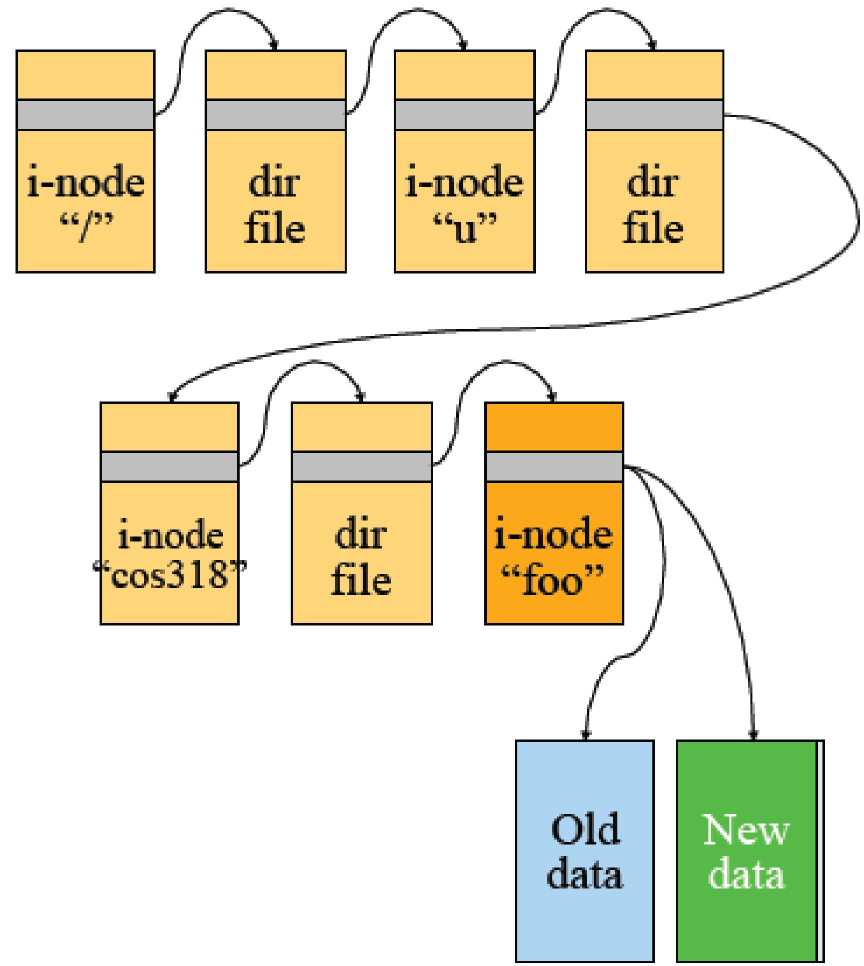
Crash → Consistent

  - Write new data to foo

Crash → Consistent

  - Write pointer into i-node

Crash → Consistent



Consistent Updates: Bottom-Up Order

- The general approach is to use a “bottom up” order
  - File data blocks, file i-node, directory file, directory i-node, …
- What about file buffer cache?
  - Write back all data blocks
  - Update file i-node and write it to disk
  - Update directory file and write it to disk
  - Update directory i-node and write it to disk (if necessary)
  - Continue until no directory update exists
- Does this solve the write back problem?
  - Updates are consistent but leave garbage blocks around
  - May need to run fsck to clean up once a while
  - Ideal approach: consistent update without leaving garbage

Operations as transactions in FileSys

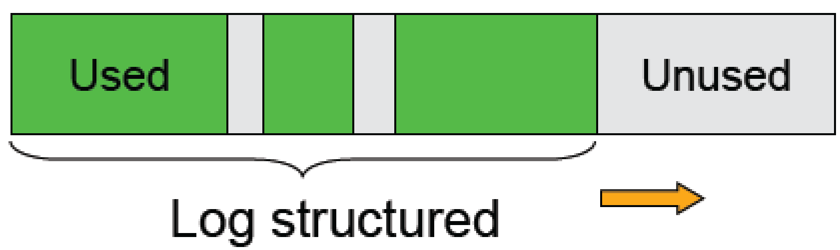
- Make a file operation a transaction
  - Create a file
  - Move a file
  - Write a chunk of data
  - …
- Make arbitrary number of file operations a transaction
  - Just keep logging but make sure that things are idempotent: making a very long transaction
  - Recovery by replaying the log and correct the file system
  - This is called logging file system or journaling file system
  - Almost all new file systems are journaling (Windows NTFS, Veritas file system, file systems on Linux)

Log Management

- How big is the log? Same size as the file system?
- Observation
  - Log what’s needed for crash recovery
- Management method
  - Checkpoint operation: flush the buffer cache to disk
  - After a checkpoint, we can truncate log and start again
  - Log needs to be big enough to hold changes in memory
- Some logging file systems log only metadata (file descriptors and directories) and not file data to keep log size down

Log-structured File System (LFS)

- Structure the entire file system as a log with segments
- A segment has i-nodes, indirect blocks, and data blocks
- All writes are sequential (no seeks)
- There will be holes when deleting files



Summary – Part 2

- File buffer cache
  - True LRU is possible
  - Simple write back is vulnerable to crashes
- Disk block failures and file system recovery tools
  - Individual recovery tools
  - Top down traversal tools
- Logging file systems