Lecture 19: Network File Systems

Instructor: Mitch Neilsen

Office: N219D

Quote of the Day

" Life is like a ten speed bicycle.

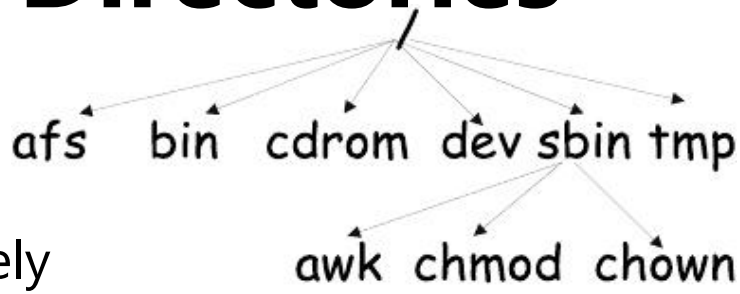
Most of us have gears we never use. "

-- Charles M. Schulz

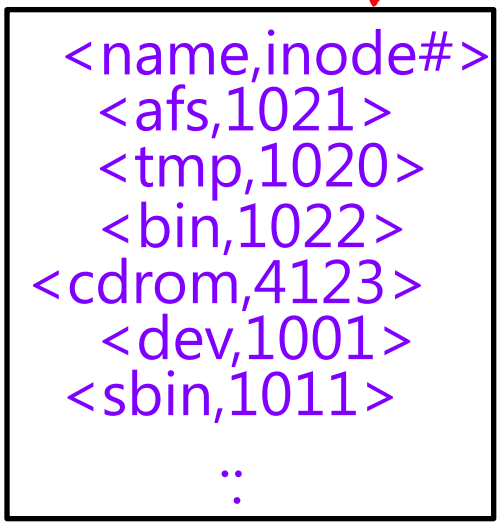
Outline – Chapters 11/12

- File System Structure
- File System Implementation
- **Directory Implementation**
- Allocation Methods
- Free-Space Management
- **Efficiency and Performance**
- **Recovery**
- **NFS**

Hierarchical Unix Directories



- **Used since CTSS (1960s)**
  - Unix picked up and used really nicely
- **Directories stored on disk just like regular files**
  - Inode contains special flag bit set
  - User 's can read just like any other file
  - Only special programs can write (why?)
  - Inodes at fixed disk location
  - File pointed to by the index may be another directory
  - Makes FS into hierarchical tree (what needed to make a DAG?)

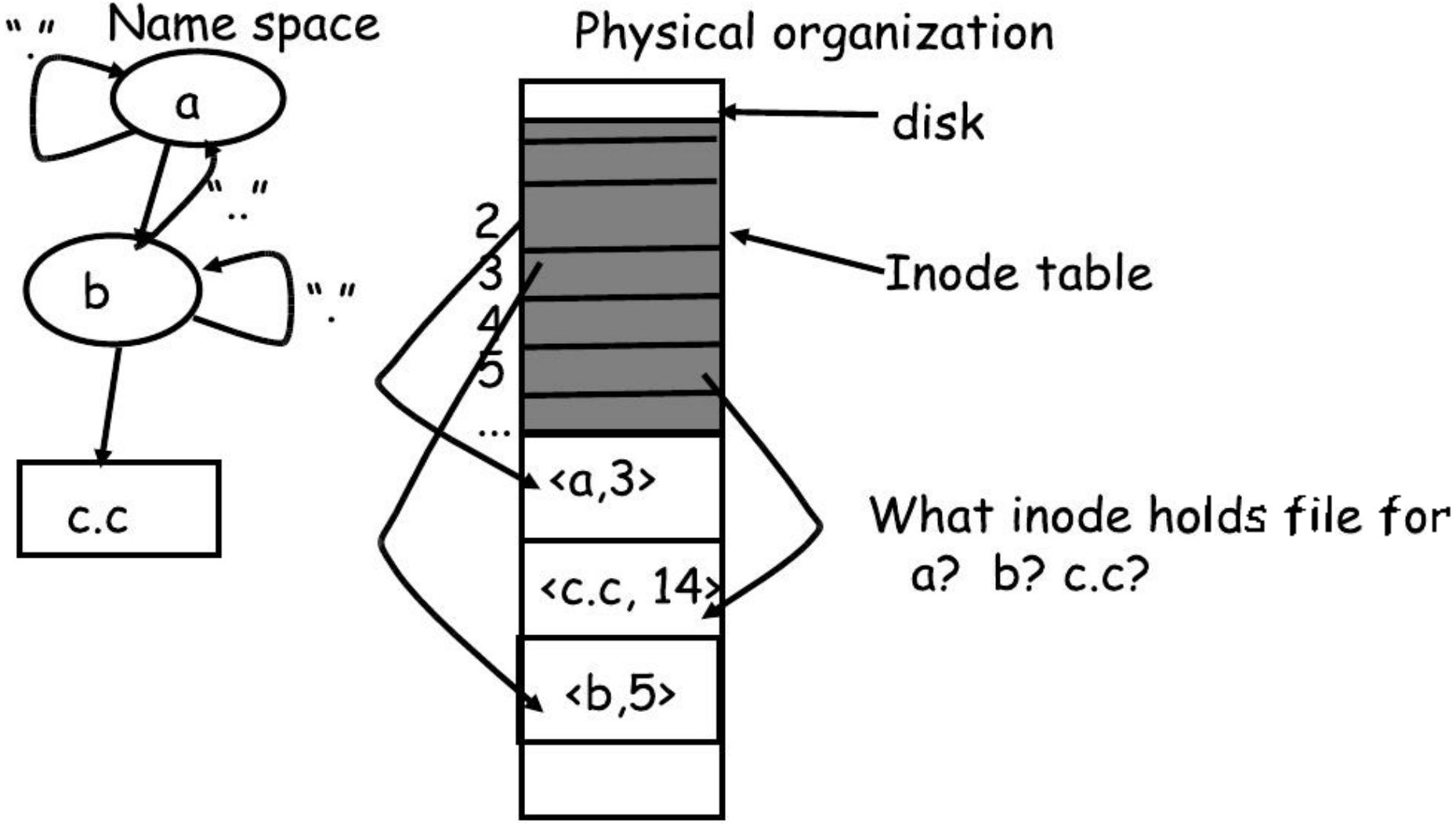


- **Simple, plus speeding up file ops speeds up dir ops!**

Naming magic

- **Bootstrapping: Where do you start looking?**
  - Root directory always inode #2 (0 and 1 historically reserved)
- **Special names:**
  - Root directory: "/"
  - Current directory: "."
  - Parent directory: ".."
- **Special names not implemented in FS:**
  - User 's home directory: "~"
  - Globbing: "foo.*" expands to all files starting "foo."
- **Using the given names, only need two operations to navigate the entire name space:**
  - cd *name*: move into (change context to) directory *name*
  - ls : enumerate all names in current directory (context)

Unix example: /a/b/c.c

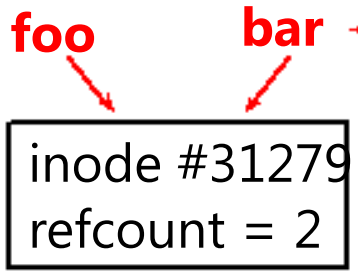


Default context: working directory

- **Cumbersome to constantly specify full path names**
  - In Unix, each process associated with a "current working directory"
  - File names that do not begin with "/" are assumed to be **relative** to the working directory, otherwise translation happens as before
- **Shells track a default list of active contexts**
  - A "search path" for programs you run - $PATH
  - Given a search path *A* : *B* : *C*, a shell will check in A, then check in B, then check in C
  - Can escape using explicit paths: "./foo"
- **Example of locality**

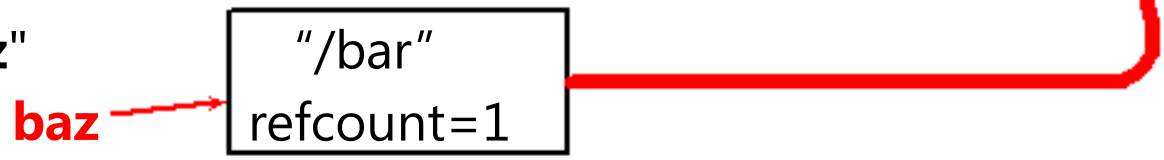
Hard and soft links (synonyms)

- **More than one dir entry can refer to a given file**
  - Unix stores count of pointers ("hard links") to inode
  - To make: "**ln foo bar**" creates a synonym (bar) for *file* foo



- **Soft links = synonyms for *names***
  - Point to a file (or dir) *name*, but object can be deleted from underneath it (or never even exist).
  - Unix implements like directories: inode has special "sym link" bit set and contains pointed to name

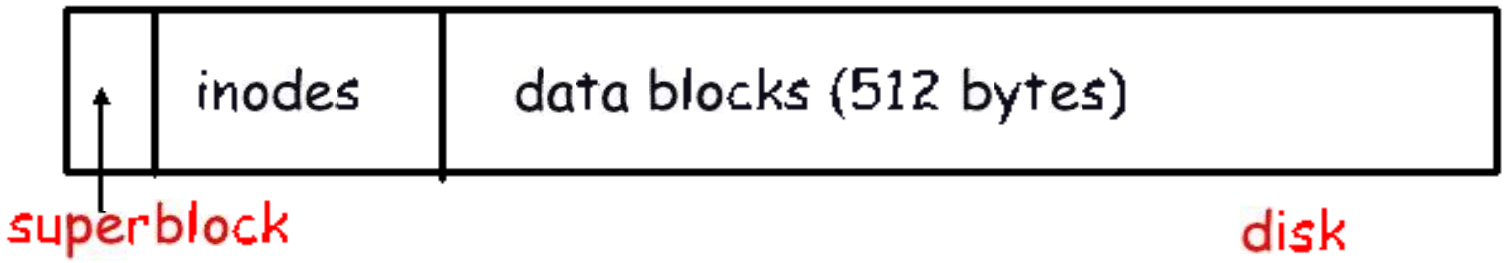
"**ln -s bar baz**"



  - When the file system encounters a symbolic link it automatically translates it (if possible).

Case study: speeding up FS

- **Original Unix FS: Simple and elegant:**



- **Components:**
  - Data blocks
  - Inodes (directories represented as files)
  - Hard links
  - Superblock. (specifies number of blks in FS, counts of max # of files, pointer to head of free list)
- **Problem: slow**
  - Only gets 20KB/sec (2% of disk maximum) even for sequential disk transfers!

A plethora of performance costs

- **Blocks too small (512 bytes)**
  - File index too large
  - Too many layers of mapping indirection
  - Transfer rate low (get one block at time)
- **Bad clustering of related objects:**
  - Consecutive file blocks not close together
  - Inodes far from data blocks
  - Inodes for directory not close together
  - Poor enumeration performance: e.g., “ls”, “grep foo *.c”
- **Next: how FFS fixes these problems (to a degree)**

FFS = Berkeley Fast File System = Unix File System = UFS

Problem: Internal fragmentation

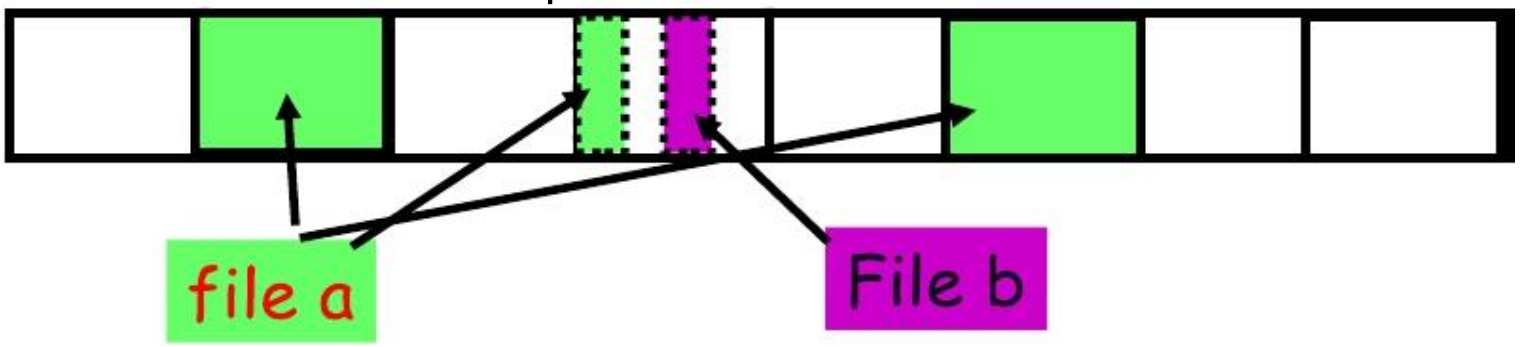
- **Block size was to small in original Unix FS**
- **Why not just make bigger?**

Block size	space wasted	file bandwidth
512	6.9%	2.6%
1024	11.8%	3.3%
2048	22.4%	6.4%
4096	45.6%	12.0%
1MB	99.0%	97.2%

- **Bigger block increases bandwidth, but how to deal with wastage ("internal fragmentation")?**
  - Use idea from malloc: split unused portion.

Solution: fragments

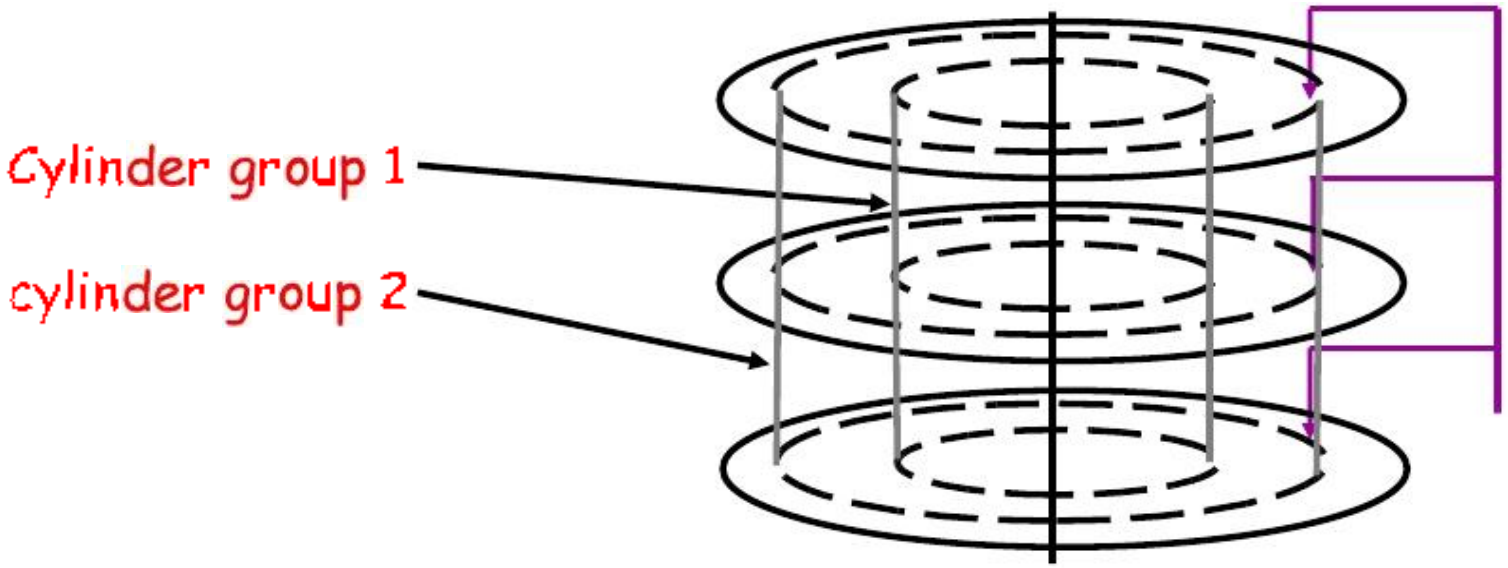
- **BSD FFS = UFS:**
  - Has large block size (4096 or 8192)
  - Allow large blocks to be chopped into small ones ("fragments")
  - Used for little files and pieces at the ends of files



- **Best way to eliminate internal fragmentation?**
  - Variable sized splits of course
  - Why does FFS use fixed-sized fragments (1024, 2048)?

Clustering related objects in FFS

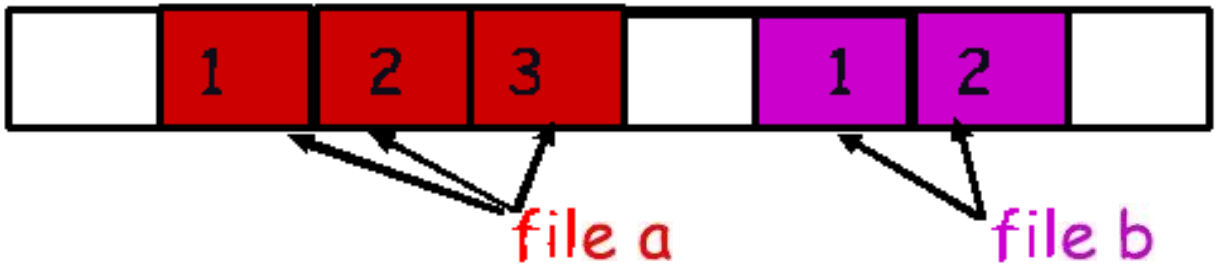
- **Group 1 or more consecutive cylinders into a "*cylinder group*"**



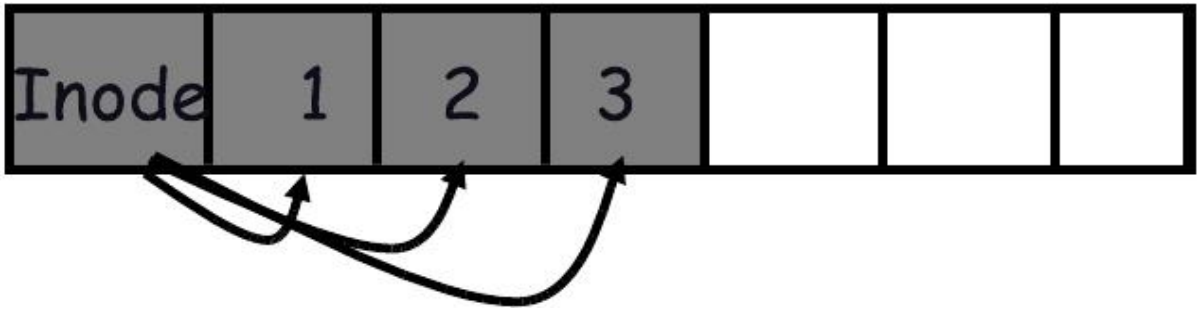
  - Key: can access any block in a cylinder without performing a seek. Next fastest place is adjacent cylinder.
  - Tries to put everything related in same cylinder group
  - Tries to put everything not related in different group (?!)

Clustering in FFS

- **Tries to put sequential blocks in adjacent sectors**
  - (Access one block, probably access next)



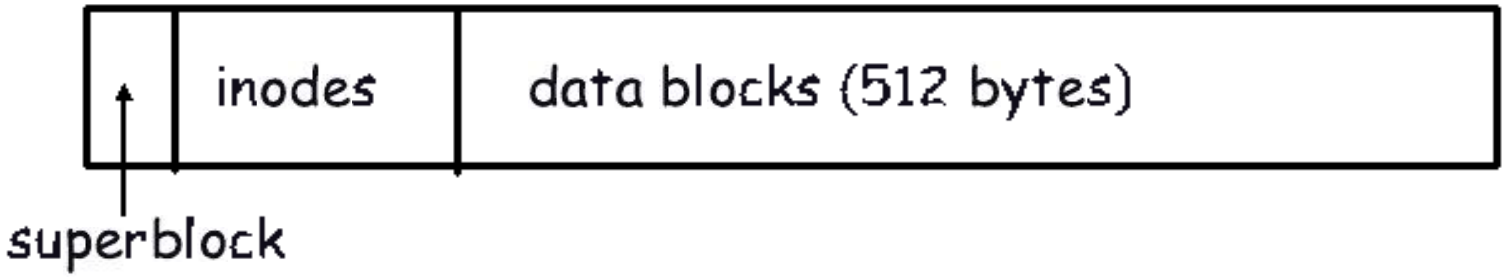
- **Tries to keep inode in same cylinder as file data:**
  - (If you look at inode, most likely will look at data too)



- **Tries to keep all inodes in a dir in same cylinder group**
  - Access one name, frequently access many, e.g., "ls -l"

What does a cylinder group look like?

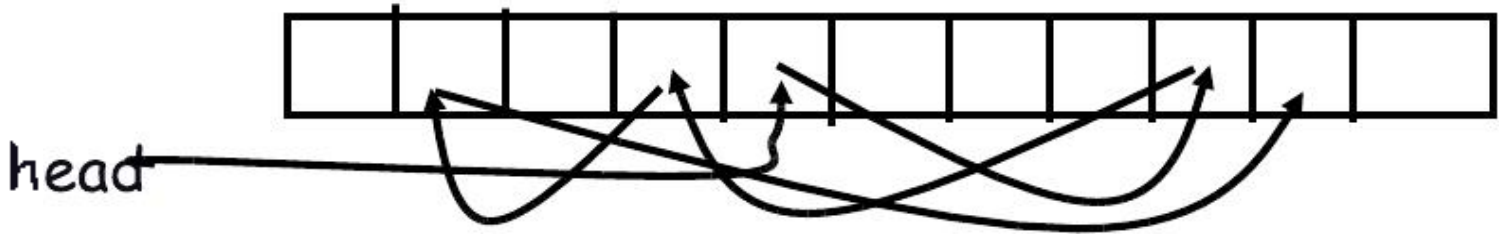
- **Basically a mini-Unix file system:**



- **How to ensure there's space for related stuff?**
  - Place different directories in different cylinder groups
  - Keep a "free space reserve" so can allocate near existing things
  - When file grows too big (1MB) send its remainder to different cylinder group.

Finding space for related objects

- **Old Unix (& dos): Linked list of free blocks**
  - Just take a block off of the head. Easy.



  - Bad: free list gets jumbled over time. Finding adjacent blocks hard and slow
- **FFS: switch to bit-map of free blocks**
  - 1010101111111000001111111000101100
  - Easier to find contiguous blocks.
  - Small, so usually keep entire thing in memory
  - Key: keep a reserve of free blocks. Makes finding a close block easier

Using a bitmap

- **Usually keep entire bitmap in memory:**
  - 4G disk / 4K byte blocks. How big is map?
- **Allocate block close to block x?**
  - Check for blocks near bmap[x/32]
  - If disk almost empty, will likely find one near
  - As disk becomes full, search becomes more expensive and less effective.
- **Trade space for time (search time, file access time)**
- **Keep a reserve (e.g., 10%) of disk always free, ideally scattered across disk**
  - Don't tell users (df → 110% full)
  - With 10% free, can almost always find one of them free

So what did we gain?

- **Performance improvements:**
  - Able to get 20-40% of disk bandwidth for large files
  - 10-20x original Unix file system!
  - Better small file performance (why?)
- **Is this the best we can do? No.**
- **Block based rather than extent based**
  - Name contiguous blocks with single pointer and length
  - (Linux ext2fs)
- **Writes of metadata done synchronously**
  - Really hurts small file performance
  - Make asynchronous with write-ordering ("soft updates") or logging (the episode file system, ~LFS)
  - Play with semantics (/tmp file systems)

Other hacks

- **Obvious:**
  - Big file cache.
- **Fact: no rotation delay if get whole track.**
  - How to use?
- **Fact: transfer cost negligible.**
  - Recall: Can get 50x the data for only ~3% more overhead
  - 1 sector: 10ms + 8ms + 10μs (= 512 B/(50 MB/s)) ≈ 18ms
  - 50 sectors: 10ms + 8ms + .5ms = 18.5ms
  - How to use?
- **Fact: if transfer huge, seek + rotation negligible**
  - Hoard data, write out MB at a time.

Network File Systems

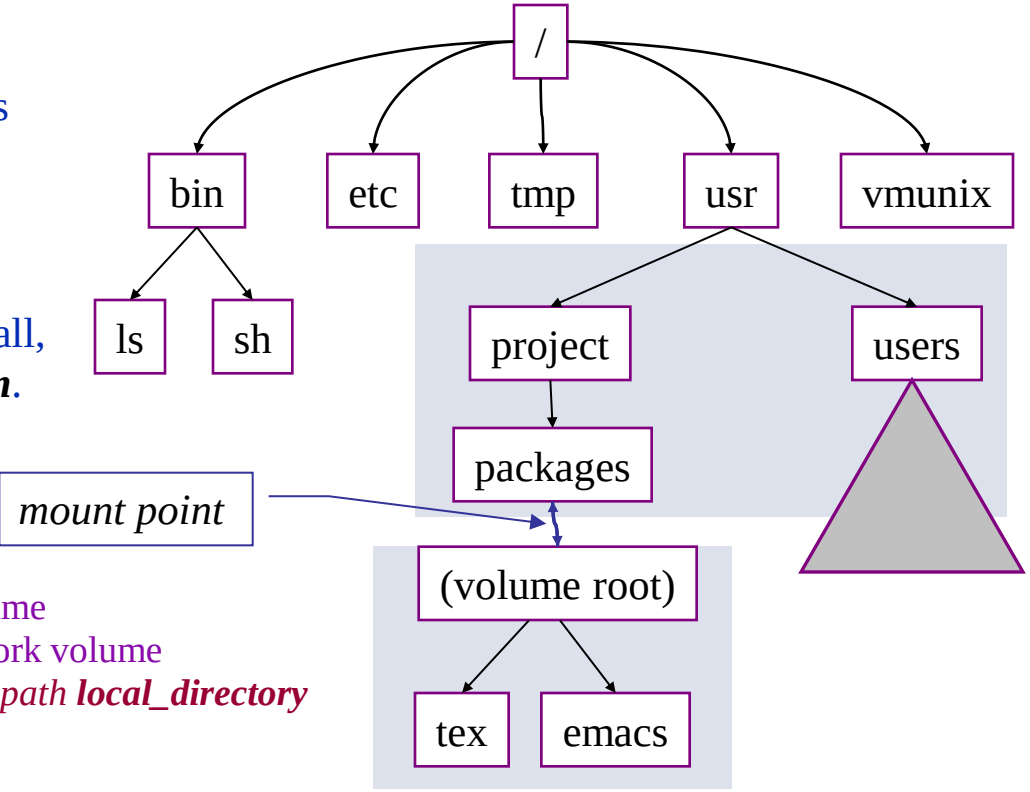
- **What's a network file system?**
  - Looks like a **local** file system to applications
  - But data potentially stored on another machine
  - Reads and writes must go over the network
  - Also called **distributed file system**
- **Advantages of network file systems**
  - Easy to share if files available on multiple machines
  - Often easier to administer servers than clients
  - Access way more data than what fits on your local disk
  - Network + remote buffer cache faster than local disk
- **Disadvantages**
  - Network + remote disk slower than local disk
  - Network or server may fail even when client OK
  - Complexity, security issues

Typical Unix File Tree

Each volume is a set of directories and files; a host's ***file tree*** is the set of directories and files visible to processes on a given host.

File trees are built by ***grafting*** volumes from different volumes or from network servers.

In Unix, the graft operation is the privileged ***mount*** system call, and each volume is a ***file system***.



mount (local_directory, volume)

local_directory: directory pathname

volume: device specifier or network volume

volume root contents become visible at path ***local_directory***

e.g., /usr/project/packages/tex

File Systems

Each file volume (***file system***) has a ***type***, determined by its disk layout or the network protocol used to access it.

> ***ufs (ffs), lfs, nfs, rfs, cdfs,*** etc.
>
> File systems are administered independently.

Modern systems also include “logical” pseudo-file systems in the naming tree, accessible through the file syscalls.

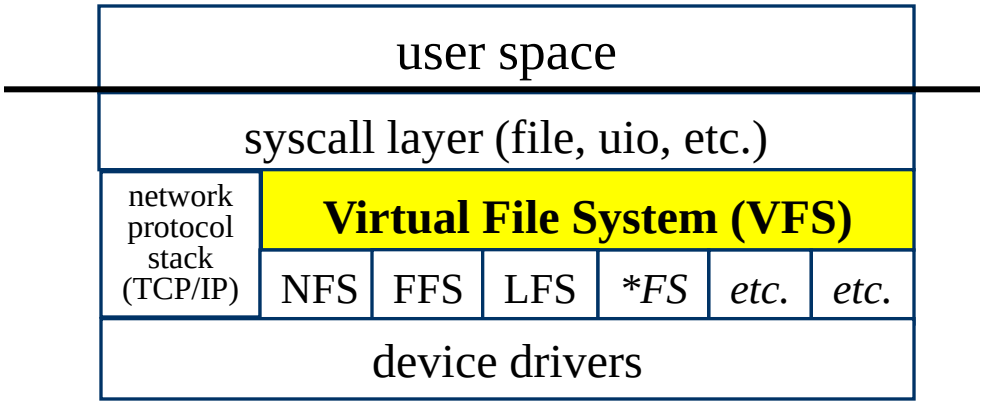
> *procfs*: the ***/proc*** file system allows access to process internals.
>
> *mfs*: the ***memory file system*** is a memory-based scratch store.

Processes access file systems through common system calls.

VFS: the File System Switch

Sun Microsystems introduced the ***virtual file system*** interface in 1985 to accommodate diverse file system types cleanly.

VFS allows diverse *specific file systems* to coexist in a file tree, isolating all FS-dependencies in pluggable file system modules.



Other abstract interfaces in the kernel: device drivers, file objects, executable files, memory objects.

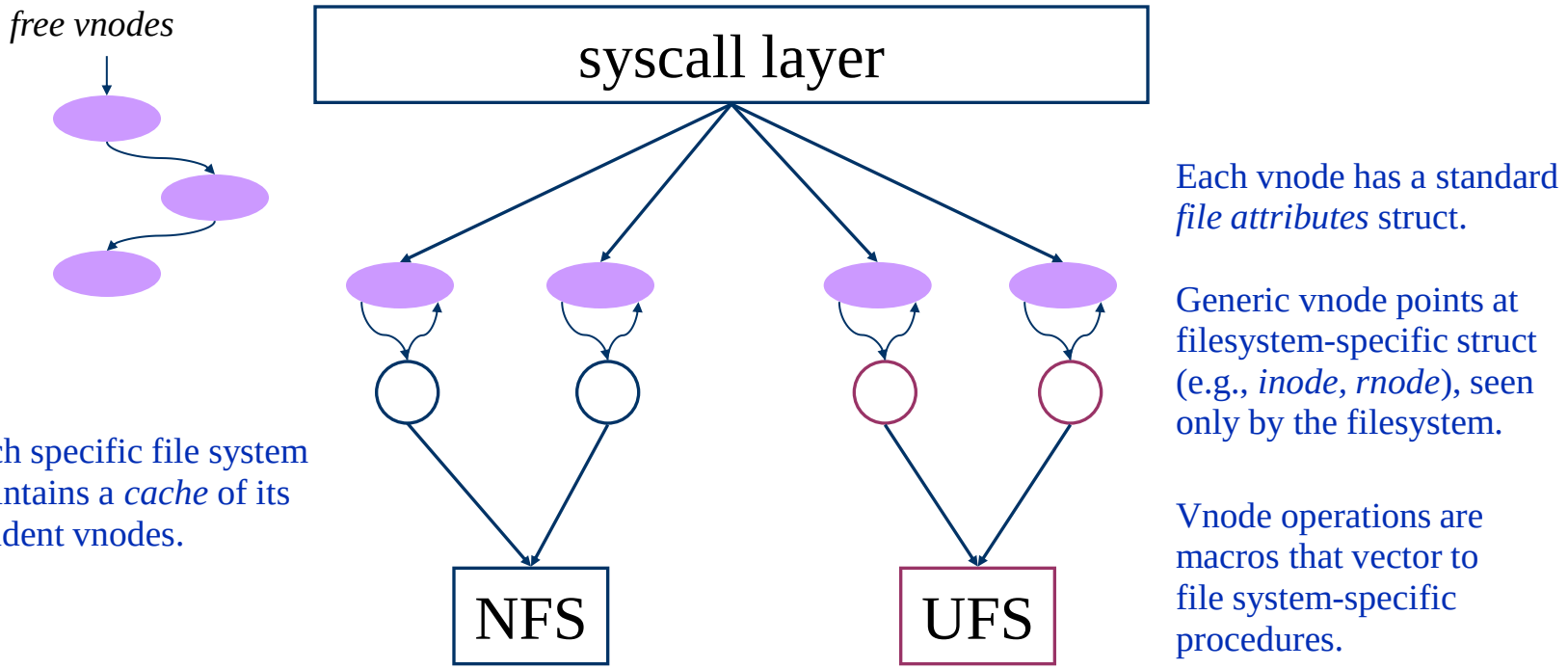
VFS was an internal kernel restructuring with no effect on the syscall interface.

Incorporates object-oriented concepts: a generic procedural interface with multiple implementations.

Based on abstract objects with dynamic method binding by type...in C.

Vnodes

In the VFS framework, every file or directory in active use is represented by a ***vnode*** object in kernel memory.



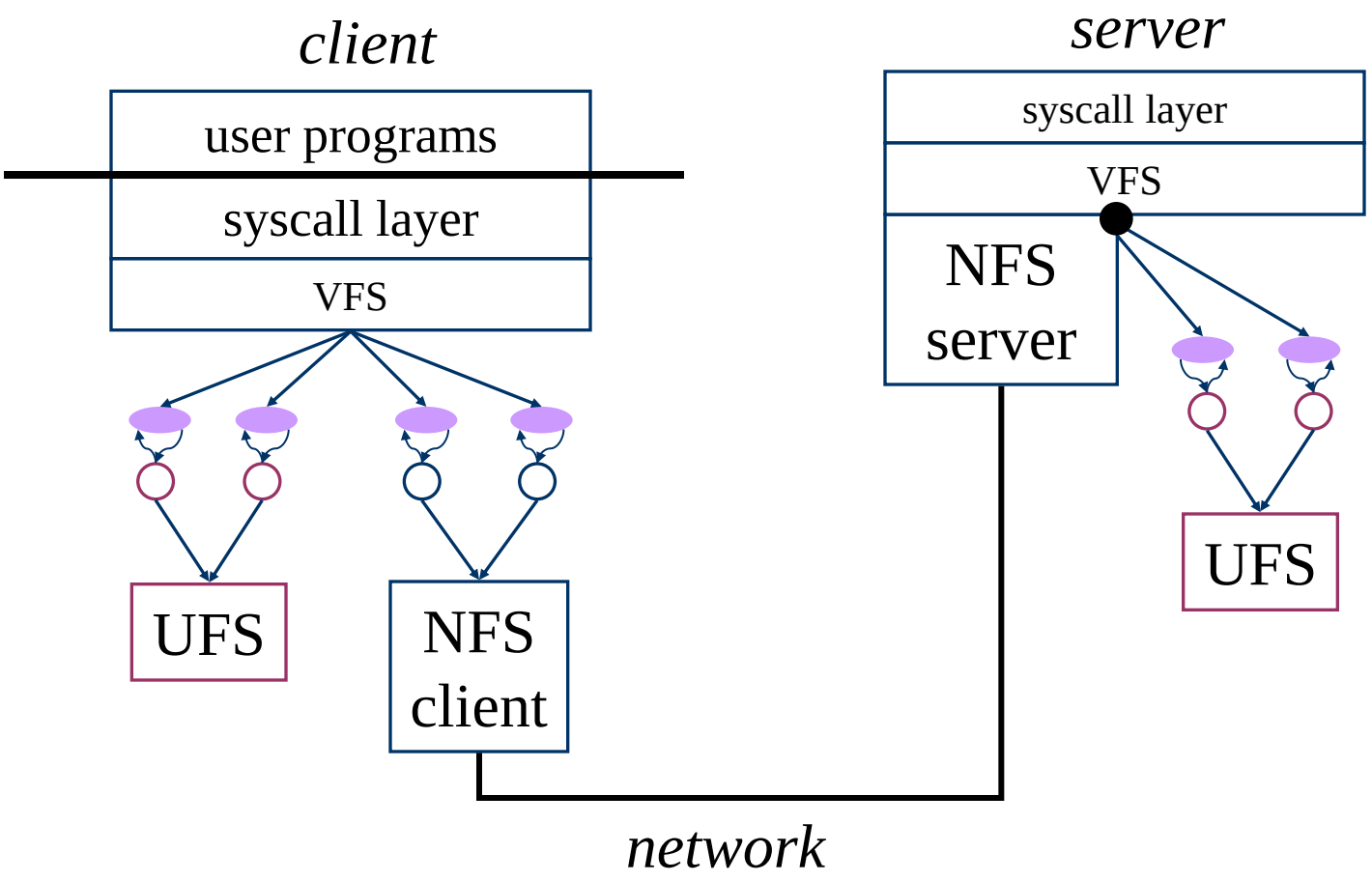
Each specific file system maintains a *cache* of its resident vnodes.

Each vnode has a standard *file attributes* struct.

Generic vnode points at filesystem-specific struct (e.g., *inode*, *rnode*), seen only by the filesystem.

Vnode operations are macros that vector to file system-specific procedures.

Network File System (NFS)



Naming Structures

- **Location transparency – file name does not reveal the file's physical storage location**
  - File name still denotes a specific, although hidden, set of physical disk blocks
  - Convenient way to share data
  - Can expose correspondence between component units and machines
- **Location independence – file name need not change when the file's physical storage location changes**
  - Better file abstraction
  - Promotes sharing the storage space itself
  - Separates the naming hierarchy form the storage-devices hierarchy

Naming Schemes

- **File names include server and local path (URLs)**
  - E.g., http://host.cis.ksu.edu/home/dm – unique name
  - Variation: Include cryptographically secure name for server
- **Attach remote directories to local directories (NFS)**
  - Gives appearance of a coherent directory tree
  - Only previously mounted remote directories accessible
- **Total integration of the component file systems (AFS)**
  - A single global name structure spans all the files in the system
  - If a server is unavailable, some arbitrary set of directories on different machines also becomes unavailable
- **Name by the data you want (Chord CFS, IVY)**
  - Very non-standard administrative model (impractical?)
  - Some big advantages like scalability & fault tolerance

NFS Implementation

- **Virtualized the file system with *vnodes***
  - Basically poor man's C++ (like protosw struct)
- **Vnode structure represents an open (or openable) file**
- **Bunch of generic "vnode operations":**
  - lookup, create, open, close, getattr, setattr, read, write, fsync, remove, link, rename, mkdir, rmdir, symlink, readdir, readlink, . . .
  - Called through function pointers, so most system calls don't care what type of file system a file resides on
- **NFS implements vnode operations through *Remote Procedure Calls* (RPC)**
  - Client request to server over network, awaits response
  - Each system call may require a series of RPCs
  - **System mostly determined by NFS RPC *Protocol***

Stateless Operation

- **NFS version 2 protocol specified in [RFC 1094]**
- **Designed for "stateless operation"**
  - Motivated by need to recover from server crashes
- **Requests are self-contained**
- **Requests are idempotent**
  - Unreliable UDP transport
  - Client retransmits requests until it gets a reply
  - Writes must be stable before server returns
- **Can this really work?**

Stateless Operation

- **NFS version 2 protocol specified in [RFC 1094]**
- **Designed for "stateless operation"**
  - Motivated by need to recover from server crashes
- **Requests are self-contained**

  mostly
- **Requests are ∧ idempotent**
  - Unreliable UDP transport
  - Client retransmits requests until it gets a reply
  - Writes must be stable before server returns
- **Can this really work?**
  - Of course, FS not stateless – it stores files
  - E.g., *mkdir* can't be idempotent – second time dir exists
  - But many operations, e.g., *read*, *write* are idempotent

Semantics

- **Attach remote file system on local directory**
  - mount server:/server/path to /client/path
  - Hard mount – if server unavailable, keep trying forever
  - Soft mount – if server unavailable, time out and return error
- **Component-by-component file name lookup**
- **Authenticate client, assume same users as server**
- **Open files should be usable even if unlinked**
  - Kludge: client just renames the file
- **Permissions usually checked when files opened**
  - So if user owns file but no write perms, allow write anyway
- **Cache consistency**
  - With multiple clients, some departure from local FS semantics

NFS version 3

- **Same general architecture as NFS 2**
- **Specified in RFC 1813 (subset of Open Group spec)**
  - Based on XDR protocol specification language [RFC 1832]
  - XDR defines C structures that can be sent over network; includes typed unions (to know which union field active)
  - Protocol defined as a set of Remote Procedure Calls (RPCs)
- **New access RPC**
  - Supports clients and servers with different uids/gids
- **Better support for caching**
  - Unstable writes while data still cached at client
  - More information for cache consistency
- **Better support for exclusive file creation**

Write discussion

- **When is it okay to lose data after a crash?**
  - *Local file system?*
    If no calls to *fsync*, OK to lose 30 seconds of work after crash
  - *Network file system?*
    What if server crashes but not client?
    Application not killed, so shouldn't lose previous writes
- **NFSv2 addresses problem by having server write data to disk before replying to a write RPC**
  - Caused performance problems
- **Could NFS2 clients just perform write-behind?**
  - Implementation issues – used blocking kernel threads on write
  - Semantics – how to guarantee consistency after server crash
  - Solution: small # of pending write RPCs, but write through on close; if server crashes, client keeps re-writing until acked

Stateful file service (E.g., CIFS)

- **Mechanism:**
  - Client opens a file
  - Server returns client-specific identifier like a file descriptor
  - Identifier used for subsequent accesses until the session ends
  - Server keeps active identifiers in memory; must reclaim
- **Possible advantages**
  - Easier for server to detect sequential access and read ahead
  - Easier to implement callbacks if know all clients w. open file
  - Easier to implement local FS semantics (e.g., unlink file open on different server)
- **Disadvantages**
  - Harder to recover from server crash (lost open file state)

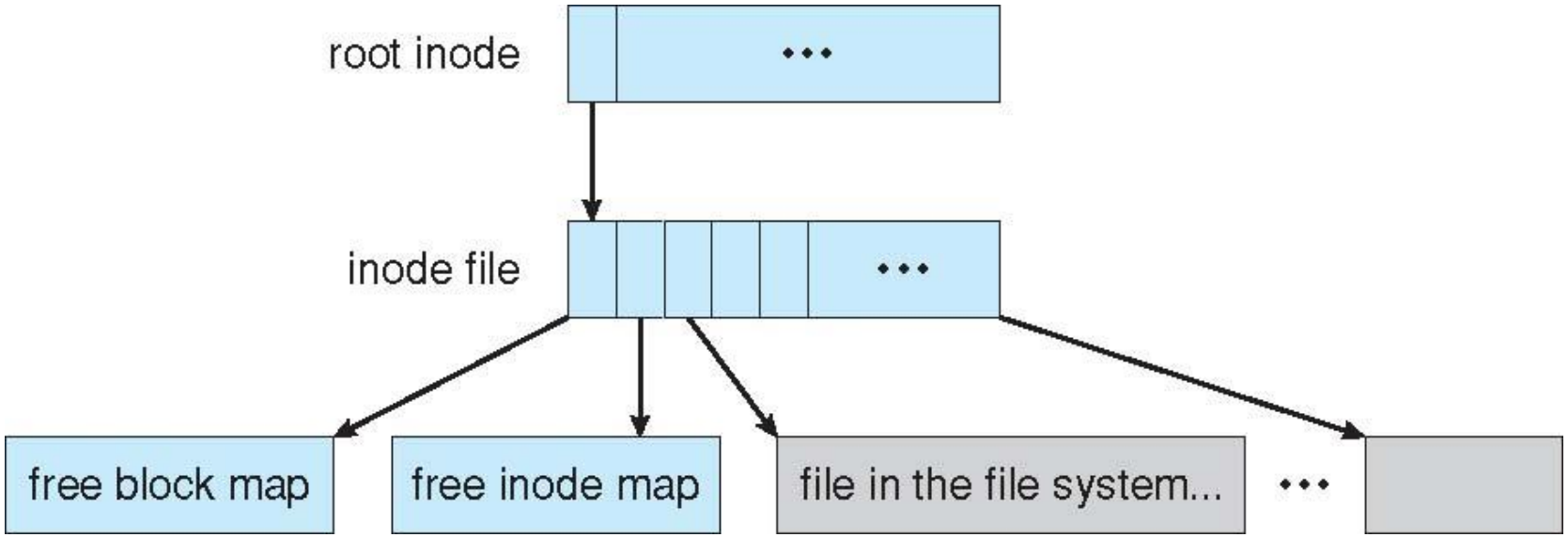
File replication

- **Replicate same file on failure-independent machines**
  - Improves availability and can shorten read time
- **Naming scheme maps file name → good replica**
  - Existence of replicas should be invisible to higher levels
  - Replicas must be distinguished from one another by different lower-level names
- **Updates**
  - Replicas of a file denote the same logical entity
  - Updates must be reflected on all replicas of a file
- **Demand replication – reading a non-local replica causes it to be cached locally, thereby generating a new non-primary replica**

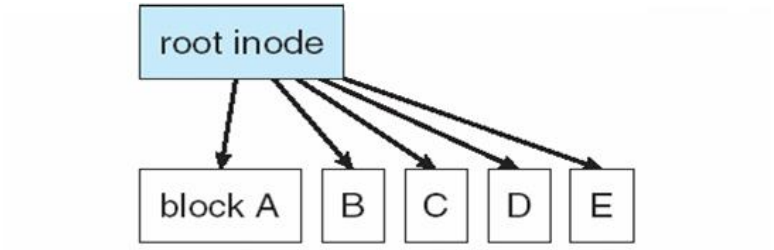
Example: WAFL File System

- Used on Network Appliance “Filers” – distributed file system appliances
- “Write-anywhere file layout”
- Serves up NFS, CIFS, http, ftp
- Random I/O optimized, write optimized
  - NVRAM for write caching
- Similar to Berkeley Fast File System, with extensive modifications

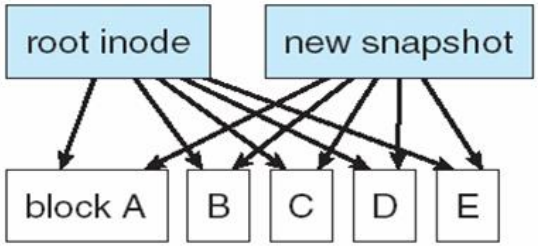
The WAFL File Layout



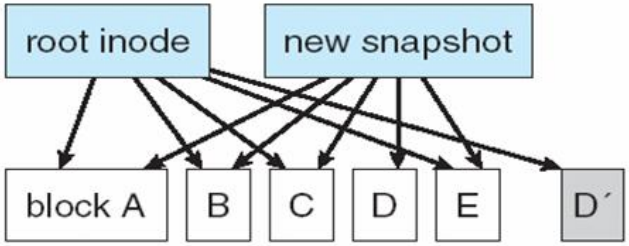
Snapshots in WAFL



(a) Before a snapshot.



(b) After a snapshot, before any blocks change.



(c) After block D has changed to D´.

Summary

- Read Ch. 1-12
- Processes and Threads (Ch. 4)
- Process Scheduling (Ch. 5)
- Synchronization (Ch. 6)
- Deadlock (Ch. 7)
- Memory Management (Ch. 8)
- Virtual Memory (Ch. 9)
- Mass-Storage Structure (Ch. 10)
- File System Interface (Ch. 11)
- File System Implementation (Ch. 12)
- Project #2 – System Calls and User-Level Processes