CS162
Operating Systems and Systems Programming
Lecture 19

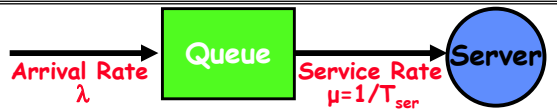
File Systems continued
Distributed Systems

November 7, 2005

Prof. John Kubiatowicz

http://inst.eecs.berkeley.edu/~cs162

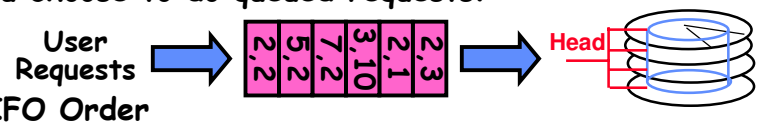
Review: A Little Queuing Theory



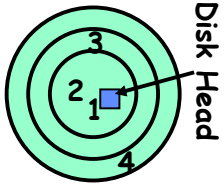
- Parameters that describe our system:
  - λ: mean number of arriving customers/second
  - T_{ser}: mean time to service a customer ("m1")
  - C: squared coefficient of variance = $\sigma^2/m1^2$
  - μ: service rate = $1/T_{ser}$
  - u: server utilization ($0 \le u \le 1$): $u = \lambda/\mu = \lambda \times T_{ser}$
- Parameters we wish to compute:
  - T_q: Time spent in queue
  - L_q: Length of queue = $\lambda \times T_q$ (by Little's law)
- Basic Approach:
  - Customers before us must finish; mean time = $L_q \times T_{ser}$
  - *If something at server*, takes m1(z) to complete on avg
    - » m1(z): mean residual wait time at server= $T_{ser} \times \frac{1}{2}(1+C)$
    - » Chance something at server = u ⇒ mean time is $u \times m1(z)$
- Computation of wait time in queue (T_q):
  - $T_q = L_q \times T_{ser} + u \times m1(z) = T_{ser} \times \frac{1}{2}(1+C) \times u/(1-u))$

Review: Disk Scheduling

- Disk can do only one request at a time; What order do you choose to do queued requests?



- FIFO Order
  - Fair among requesters, but order of arrival may be to random spots on the disk ⇒ Very long seeks
- SSTF: Shortest seek time first
  - Pick the request that's closest on the disk
  - Although called SSTF, today must include rotational delay in calculation, since rotation can be as long as seek
  - Con: SSTF good at reducing seeks, but may lead to starvation



- SCAN: Implements an Elevator Algorithm: take the closest request in the direction of travel
  - No starvation, but retains flavor of SSTF
- C-SCAN: Circular-Scan: only goes in one direction
  - Skips any requests on the way back
  - Fairer than SCAN, not biased towards pages in middle

Review: Access and Usage Patterns

- Sequential Access: bytes read in order ("give me the next X bytes, then give me next, etc")
  - Almost all file access are of this flavor
- Random Access: read/write element out of middle of array ("give me bytes i—j")
  - Less frequent, but still important. For example, virtual memory backing file: page of memory stored in file
  - Want this to be fast – don't want to have to read all bytes to get to the middle of the file
- Most files are small (for example, .login, .c files)
  - A few files are big – nachos, core files, etc.; the nachos executable is as big as all of your .class files combined
  - However, most files are small – .class's, .o's, .c's, etc.
- Large files use up most of the disk space and bandwidth to/from disk
  - May seem contradictory, but a few enormous files are equivalent to an immense # of small files

Review: Building File Systems

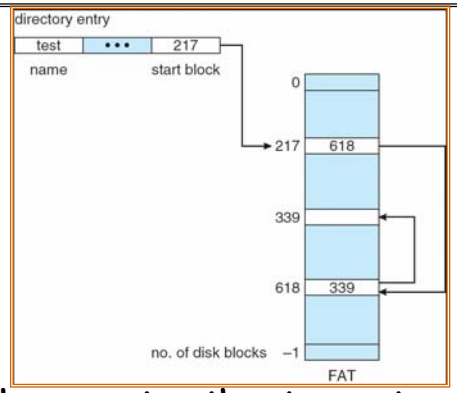
- File System: Layer of OS that transforms block interface of disks (or other block devices) into Files, Directories, etc
- Access disk as linear array of blocks. Two Options:
  - Identify blocks as vectors [cylinder, surface, sector]. Sort in cylinder-major order. Not used much anymore.
  - Logical Block Addressing (LBA). Every block has integer address from zero up to max number of cylinders.
    - » Assumption is that locality within LBA numbers represent locality on disk
  - Use bitmap to represent free space on disk
- Need way to structure files: File Header
  - Track which blocks belong at which offsets within the logical file structure
  - Optimize placement of files disk blocks to match access and usage patterns
- File System Design Goals:
  - Maximize sequential performance
  - Easy random access to file
  - Easy management of file (growth, truncation, etc)

Goals for Today

- Finish Discussion of File Systems
  - Structure, Naming, Directories
- File Caching
- Data Durability
- Beginning of Distributed Systems Discussion

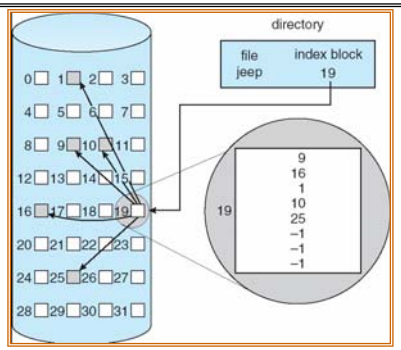
Note: Some slides and/or pictures in the following are adapted from slides ©2005 Silberschatz, Galvin, and Gagne

Linked Allocation: File-Allocation Table (FAT)



- MSDOS links pages together to create a file
  - Links not in pages, but in the File Allocation Table (FAT)
    - » FAT contains an entry for each block on the disk
    - » FAT Entries corresponding to blocks of file linked together
  - Access properies:
    - » Sequential access expensive unless FAT cached in memory
    - » Random access expensive always, but *really* expensive if FAT not cached in memory

Indexed Allocation



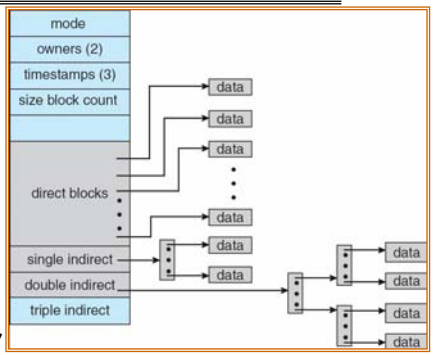
- Indexed Files (Nachos, VMS)
  - System Allocates file header block to hold array of pointers big enough to point to all blocks
    - » User pre-declares max file size;
  - Pros: Can easily grow up to space allocated for index
    Random access is fast
  - Cons: Clumsy to grow file bigger than table size
    Still lots of seeks: blocks may be spread over disk

Multilevel Indexed Files (UNIX BSD 4.1)

- Multilevel Indexed Files: Like multilevel address translation (from UNIX 4.1 BSD)
  - Key idea: efficient for small files, but still allow big files
  - File header contains 13 pointers
    - » Fixed size table, pointers not all equivalent
    - » This header is called an "inode" in UNIX
  - File Header format:
    - » First 10 pointers are to data blocks
    - » Block 11 points to "indirect block" containing 256 blocks
    - » Block 12 points to "doubly indirect block" containing 256 indirect blocks for total of 64K blocks
    - » Block 13 points to a triply indirect block (16M blocks)
- Discussion
  - Basic technique places an upper limit on file size that is approximately 16Gbytes
    - » Designers thought this was bigger than anything anyone would need. Much bigger than a disk at the time…
    - » Fallacy: today, EOS producing 2TB of data per day
  - Pointers get filled in dynamically: need to allocate indirect block only when file grows > 10 blocks.
    - » On small files, no indirection needed

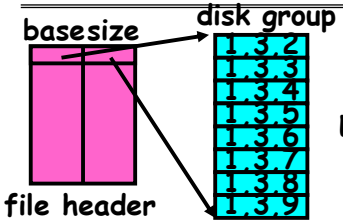
Example of Multilevel Indexed Files

- Sample file in multilevel indexed format:
  - How many accesses for block #23? (assume file header accessed on open)?
    - » Two: One for indirect block, one for data
  - How about block #5?
    - » One: One for data
  - Block #340?
    - » Three: double indirect block, indirect block, and data



- UNIX 4.1 Pros and cons
  - Pros: Simple (more or less)
    Files can easily expand (up to a point)
    Small files particularly cheap and easy
  - Cons: Lots of seeks
    Very large files must read many indirect block (four I/Os per block!)

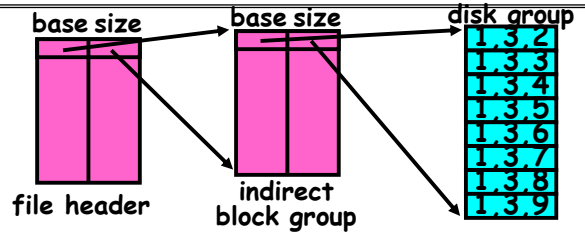
File Allocation for Cray-1 DEMOS



Basic Segmentation Structure:
Each segment contiguous on disk

- DEMOS: File system structure similar to segmentation
  - Idea: reduce disk seeks by
    - » using contiguous allocation in normal case
    - » but allow flexibility to have non-contiguous allocation
  - Cray-1 had 12ns cycle time, so CPU:disk speed ratio about the same as today (a few million instructions per seek)
- Header: table of base & size (10 "block group" pointers)
  - Each block chunk is a contiguous group of disk blocks
  - Sequential reads within a block chunk can proceed at high speed – similar to continuous allocation
- How do you find an available block group?
  - Use freelist bitmap to find block of 0's.

Large File Version of DEMOS



- What if need much bigger files?
  - If need more than 10 groups, set flag in header: BIGFILE
    - » Each table entry now points to an indirect block group
  - Suppose 1000 blocks in a block group ⇒ 80GB max file
    - » Assuming 8KB blocks, 8byte entries⇒
      (10 ptrs×1024 groups/ptr×1000 blocks/group)*8K =80GB
- Discussion of DEMOS scheme
  - Pros: Fast sequential access, Free areas merge simply
    Easy to find free block groups (when disk not full)
  - Cons: Disk full ⇒ No long runs of blocks (fragmentation), so high overhead allocation/access
  - Full disk ⇒ worst of 4.1BSD (lots of seeks) with worst of continuous allocation (lots of recompaction needed)

How to keep DEMOS performing well?

- In many systems, disks are always full
  - CS department growth: 300 GB to 1TB in a year
    - » That's 2GB/day! (Now at 3—4 TB!)
  - How to fix? Announce that disk space is getting low, so please delete files?
    - » Don't really work: people try to store their data faster
  - Sidebar: Perhaps we are getting out of this mode with new disks… However, let's assume disks full for now
- Solution:
  - Don't let disks get completely full: reserve portion
    - » Free count = # blocks free in bitmap
    - » Scheme: Don't allocate data if count < reserve
  - How much reserve do you need?
    - » In practice, 10% seems like enough
  - Tradeoff: pay for more disk, get contiguous allocation
    - » Since seeks so expensive for performance, this is a very good tradeoff

Administrivia

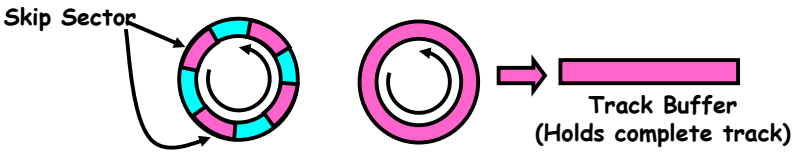
- My office hours
  - New office hour: Thursday 2:30-3:30
  - Will be removing the Monday office hour
- Project 3 design due today
- Project zero-sum game:
  - In the end, we will evaluate how to distribute project points to partners
    - » Normally, we are pretty even about this
    - » However, under extreme circumstances, can give many of points to working members and take them away from non-working members
  - This is a zero-sum game!
- Make sure to do your project evaluations
  - This is supposed to be an individual evaluation, not done together as a group
  - This is part of the information that we use to decide how to distributed points
- Final Exam
  - December 17th, 12:30 – 3:30, 220 Hearst Gym

UNIX BSD 4.2

- Same as BSD 4.2 (same file header and triply indirect blocks), except incorporated ideas from DEMOS:
  - Uses bitmap allocation in place of freelist
  - Attempt to allocate files contiguously
  - 10% reserved disk space
  - Skip-sector positioning (mentioned next slide)
- Problem: When create a file, don't know how big it will become (in UNIX, most writes are by appending)
  - How much contiguous space do you allocate for a file?
  - In Demos, power of 2 growth: once it grows past 1MB, allocate 2MB, etc
  - In BSD 4.2, just find some range of free blocks
    - » Put each new file at the front of different range
    - » To expand a file, you first try successive blocks in bitmap, then choose new range of blocks
  - Also in BSD 4.2: start files from same directory near each other

Attack of the Rotational Delay

- Problem 2: Missing blocks due to rotational delay
  - Issue: Read one block, do processing, and read next block. In meantime, disk has continued turning: missed next block! Need 1 revolution/block!



  - Solution1: Skip sector positioning ("interleaving:)
    - » Place the blocks from one file on every other block of a track: give time for processing to overlap rotation
  - Solution2: Read ahead: read next block right after first, even if application hasn't asked for it yet.
    - » This can be done either by OS (read ahead)
    - » By disk itself (track buffers). Many disk controllers have internal RAM that allows them to read a complete track
- Important Aside: Modern disks+controllers do many complex things "under the covers"
  - Track buffers, elevator algorithms, bad block filtering

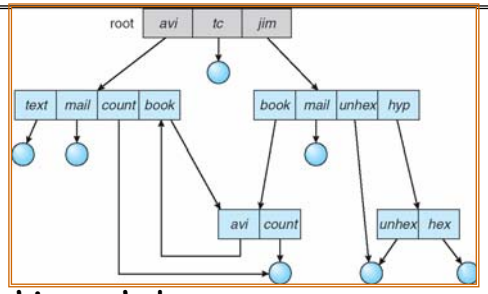
How do we actually access files?

- All information about a file contained in its file header
  - UNIX calls this an "inode"
    - » Inodes are global resources identified by index ("inumber")
  - Once you load the header structure, all the other blocks of the file are locatable
- Question: how does the user ask for a particular file?
  - One option: user specifies an inode by a number (index).
    - » Imagine: open("14553344")
  - Better option: specify by textual name
    - » Have to map name→inumber
  - Another option: Icon
    - » This is how Apple made its money. Graphical user interfaces. Point to a file and click.
- Naming: The process by which a system translates from user-visible names to system resources
  - In the case of files, need to translate from strings (textual names) or icons to inumbers/inodes
  - For global file systems, data may be spread over globe⇒need to translate from strings or icons to some combination of physical server location and inumber

Directories

- Directory: a relation used for naming
  - Just a table of (file name, inumber) pairs
- How are directories constructed?
  - Directories often stored in files
    - » Reuse of existing mechanism
    - » Directory named by inode/inumber like other files
  - Needs to be quickly searchable
    - » Options: Simple list or Hashtable
    - » Can be cached into memory in easier form to search
- How are directories modified?
  - System calls for manipulation: `mkdir, rmdir`
  - Ties to file creation/destruction
    - » On creating a file by name, new inode grabbed and associated with new file in particular directory
- Directory Hierarchy
  - Directories organized into a hierarchical structure
    - » Seems standard, but in early 70s it wasn't
    - » Permits much easier organization of data structures
  - Entries in directory can be either files *or* directories
  - Files named by ordered set (e.g.: /programs/p/list)

Directory Structure



- Not really a hierarchy!
  - Many systems allow directory structure to be organized as an acyclic graph or even a (potentially) cyclic graph
  - Hard Links: different names for the same file
    - » Multiple directory entries point at the same file
  - Soft Links: "shortcut" pointers to other files
    - » Implemented by storing the logical name of actual file
- Name Resolution: The process of converting a logical name into a physical resource (like a file)
  - Traverse succession of directories until reach target file
  - Global file system: May be spread across the network

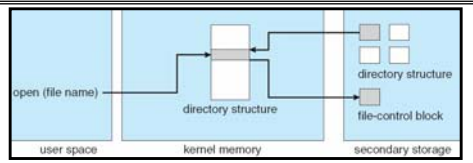
Directory Structure (Con't)

- How many disk accesses to resolve "/avi/book/count"?
  - Read in file header for root (fixed spot on disk)
  - Read in first data bock for root
    - » Table of file name/index pairs. Search linearly – ok since directories typically very small
  - Read in file header for "avi"
  - Read in first data block for "avi"; search for "book"
  - Read in file header for "book"
  - Read in first data block for "book"; search for "count"
  - Read in file header for "count"
- Current working directory: Per-address-space pointer to a directory (inode) used for resolving file names
  - Allows user to specify relative filename instead of absolute path (say CWD="/avi/book" can resolve "count")

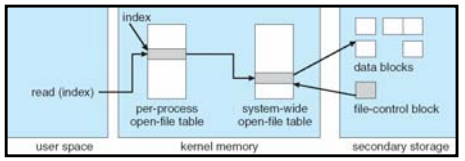
Where are inodes stored?

- In early UNIX and DOS/Windows' FAT file system, headers stored in special array in outermost cylinders
  - Header not stored near the data blocks. To read a small file, seek to get header, seek back to data.
  - Fixed size, set when disk is formatted. At formatting time, a fixed number of inodes were created (They were each given a unique number, called an "*inumber*")
- Later versions of UNIX moved the header information to be closer to the data blocks
  - Often, inode for file stored in same "cylinder group" as parent directory of the file (makes an *ls* of that directory run fast).
  - Pros:
    - » UNIX BSD 4.2 puts a portion of the file header array on each cylinder. For small directories, can fit all data, file headers, etc in same cylinder⇒no seeks!
    - » File headers much smaller than whole block (a few hundred bytes), so multiple headers fetched from disk at same time
    - » Reliability: whatever happens to the disk, you can find many of the files (even if directories disconnected)

In-Memory File System Structures



- Open system call:
  - Resolves file name, finds file control block (inode)
  - Makes entries in per-process and system-wide tables
  - Returns index (called "file handle") in open-file table



- Read/write system calls:
  - Use file handle to locate inode
  - Perform appropriate reads or writes

File System Caching

- Key Idea: Exploit locality by caching data in memory
  - Name translations: Mapping from paths→inodes
  - Disk blocks: Mapping from block address→disk content
- **Buffer Cache:** Memory used to cache kernel resources, including disk blocks and name translations
  - Can contain "dirty" blocks (blocks yet on disk)
- Replacement policy? LRU
  - Can afford overhead of timestamps for each disk block
  - Advantages:
    - » Works very well for name translation
    - » Works well in general as long as memory is big enough to accommodate a host's working set of files.
  - Disadvantages:
    - » Fails when some application scans through file system, thereby flushing the cache with data used only once
    - » Example: `find . -exec grep foo {} \;`
- Other Replacement Policies?
  - Some systems allow applications to request other policies
  - Example, 'Use Once':
    - » File system can discard blocks as soon as they are used

File System Caching (con't)

- Cache Size: How much memory should the OS allocate to the buffer cache vs virtual memory?
  - Too much memory to the file system cache ⇒ won't be able to run many applications at once
  - Too little memory to file system cache ⇒ many applications may run slowly (disk caching not effective)
  - Solution: adjust boundary dynamically so that the disk access rates for paging and file access are balanced
- **Read Ahead Prefetching:** fetch sequential blocks early
  - Key Idea: exploit fact that most common file access is sequential by prefetching subsequent disk blocks ahead of current read request (if they are not already in memory)
  - Elevator algorithm can efficiently interleave groups of prefetches from concurrent applications
  - How much to prefetch?
    - » Too many imposes delays on requests by other applications
    - » Too few causes many seeks (and rotational delays) among concurrent file requests

File System Caching (con't)

- Delayed Writes: Writes to files not immediately sent out to disk
  - Instead, `write()` copies data from user space buffer to kernel buffer (in cache)
    - » Enabled by presence of buffer cache: can leave written file blocks in cache for a while
    - » If some other application tries to read data before written to disk, file system will read from cache
  - Flushed to disk periodically (e.g. in UNIX, every 30 sec)
  - Advantages:
    - » Disk scheduler can efficiently order lots of requests
    - » Disk allocation algorithm can be run with correct size value for a file
    - » Some files need never get written to disk! (e..g temporary scratch files written /tmp often don't exist for 30 sec)
  - Disadvantages
    - » What if system crashes before file has been written out?
    - » Worse yet, what if system crashes before a directory file has been written out? (lose pointer to inode!)

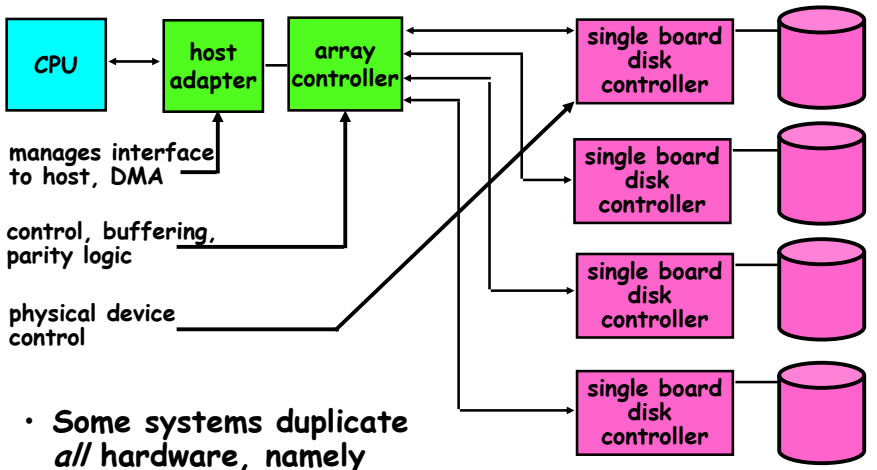
Important "ilities"

- Availability: the probability that the system can accept and process requests
  - Often measured in "nines" of probability. So, a 99.9% probability is considered "3-nines of availability"
  - Key idea here is independence of failures
- Durability: the ability of a system to recover data despite faults
  - This idea is fault tolerance applied to data
  - Doesn't necessarily imply availability: information on pyramids was very durable, but could not be accessed until discovery of Rosetta Stone
- Reliability: the ability of a system or component to perform its required functions under stated conditions for a specified period of time (IEEE definition)
  - Usually stronger than simply availability: means that the system is not only "up", but also working correctly
  - Includes availability, security, fault tolerance/durability
  - Must make sure data survives system crashes, disk crashes, other problems

How to make file system durable?

- Disk blocks contain Reed-Solomon error correcting codes (ECC) to deal with small defects in disk drive
  - Can allow recovery of data from small media defects
- Make sure writes survive in short term
  - Either abandon delayed writes or
  - use special, battery-backed RAM (called non-volatile RAM or NVRAM) for dirty blocks in buffer cache.
- Make sure that data survives in long term
  - Need to replicate! More than one copy of data!
  - Important element: independence of failure
    - » Could put copies on one disk, but if disk head fails…
    - » Could put copies on different disks, but if server fails…
    - » Could put copies on different servers, but if building is struck by lightning….
    - » Could put copies on servers in different continents…
- RAID: Redundant Arrays of Inexpensive Disks
  - Data stored on multiple disks (redundancy)
  - Either in software or hardware
    - » In hardware case, done by disk controller and file system may not even know that there is more than one disk in use

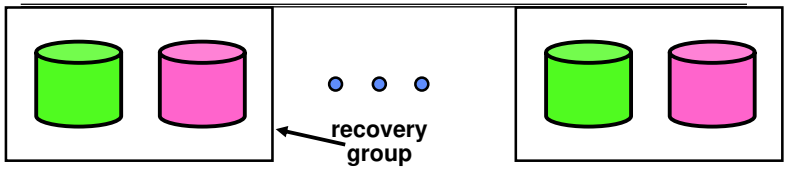
Hardware RAID: Subsystem Organization



- Some systems duplicate *all* hardware, namely controllers, busses, etc.

often piggy-backed in small format devices

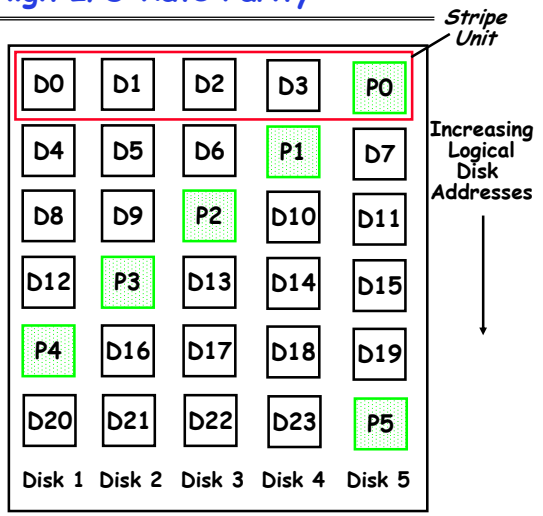
RAID 1: Disk Mirroring/Shadowing



- Each disk is fully duplicated onto its "shadow"
  - For high I/O rate, high availability environments
  - Most expensive solution: 100% capacity overhead
- Bandwidth sacrificed on write:
  - Logical write = two physical writes
  - Highest bandwidth when disk heads and rotation fully synchronized (hard to do exactly)
- Reads may be optimized
  - Can have two independent reads to same data
- Recovery:
  - Disk failure ⇒ replace disk and copy data to new disk
  - Hot Spare: idle disk already attached to system to be used for immediate replacement

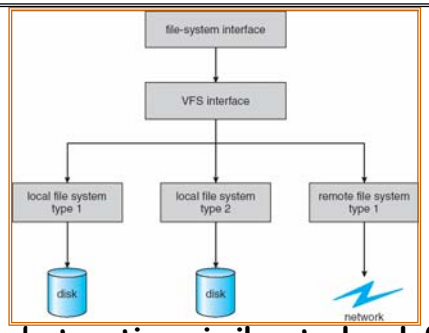
RAID 5+: High I/O Rate Parity

- Data stripped across multiple disks
  - Successive blocks stored on successive (non-parity) disks
  - Increased bandwidth over single disk
- Parity block (in green) constructed by XORing data bocks in stripe
  - P0=D0⊕D1⊕D2⊕D3
  - Can destroy any one disk and still reconstruct data
  - Suppose D3 fails, then can reconstruct: D3=D0⊕D1⊕D2⊕P0



- Later in term: talk about spreading information widely across internet for durability.

Remote File Systems: Virtual File System (VFS)

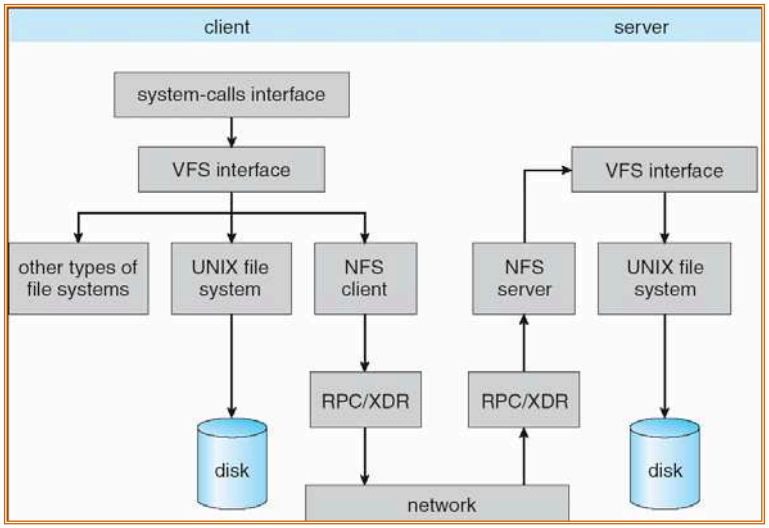


- VFS: Virtual abstraction similar to local file system
  - Instead of "inodes" has "vnodes"
  - Compatible with a variety of local and remote file systems
    - » provides object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
  - The API is to the VFS interface, rather than any specific type of file system

Network File System (NFS)

- Three Layers for NFS system
  - UNIX file-system interface: open, read, write, close calls + file descriptors
  - VFS layer: distinguishes local from remote files
    - » Calls the NFS protocol procedures for remote requests
  - NFS service layer: bottom layer of the architecture
    - » Implements the NFS protocol
- NFS Protocol: remote procedure calls (RPC) for file operations on server
  - Reading/searching a directory
  - manipulating links and directories
  - accessing file attributes/reading and writing files
- NFS servers are stateless; each request provides all arguments require for execution
- Modified data must be committed to the server's disk before results are returned to the client
  - lose some of the advantages of caching
  - Can lead to weird results: write file on one client, read on other, get old data

Schematic View of NFS Architecture



Conclusion

- Cray DEMOS: optimization for sequential access
  - Inode holds set of disk ranges, similar to segmentation
- 4.2 BSD Multilevel index files
  - Inode contains pointers to actual blocks, indirect blocks, double indirect blocks, etc
  - Optimizations for sequential access: start new files in open ranges of free blocks
  - Rotational Optimization
- Naming: act of translating from user-visible names to actual system resources
  - Directories used for naming for local file systems
- Important system properties
  - Availability: how often is the resource available?
  - Durability: how well is data preserved against faults?
  - Reliability: how often is resource performing correctly?
- RAID: Redundant Arrays of Inexpensive Disks
  - RAID1: mirroring, RAID5: Parity block
- VFS: Virtual File System layer
  - NFS: An example use of the VFS layer