Lecture 10:Disks & File Systems

CSE 120: Principles of Operating Systems

UC San Diego: Summer Session I, 2009
Frank Uyeda

Announcements

- Homework 2 is due now.
- Project 3 milestone Wednesday night.
- Project 2 bonus points
  - Fix your bugs from Project 2
  - Resubmit by the Project 3 deadline
  - Earn ½ credit back for all the things you fixed.

Announcements

- Lab Hours:
  - Frank: tomorrow 4p - ?, CSE basement

- Final Exam: 3p-6p on Saturday, August 1

- If you are lost, please come to Office Hours! You can make an appointment.

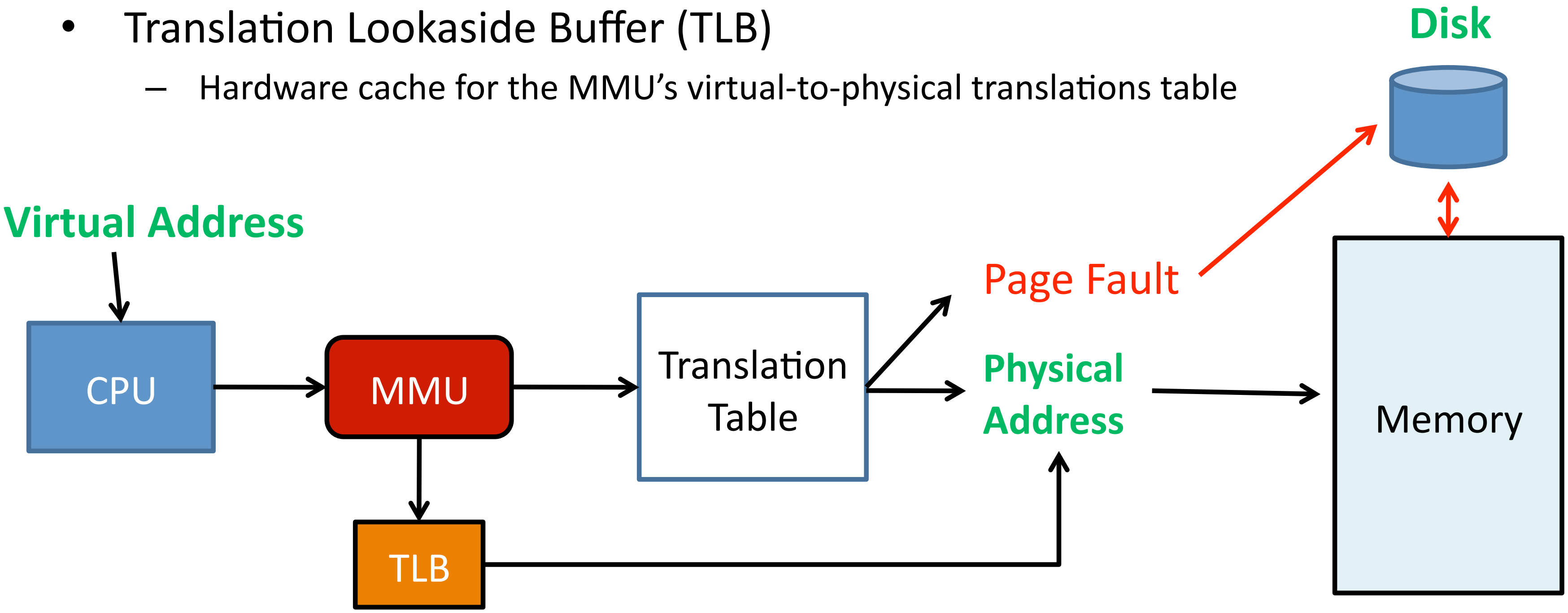
Review Question

- Which of the following scenarios is/are possible?
  - A) A PTE is valid in the TLB and valid in the page table
  - B) A PTE is valid in the TLB and invalid in the page table
  - C) A PTE is invalid in the TLB and valid in the page table
  - D) A PTE is invalid in the TLB and invalid in the page table
  - E) A PTE is not in the TLB and valid in the page table
  - F) A PTE is not in the TLB and invalid in the page table

Review: Demand Paging

- Memory Management Unit (MMU)
  - Hardware unit that translates a virtual address to a physical address
- Translation Table
  - Stored in main memory
- Translation Lookaside Buffer (TLB)
  - Hardware cache for the MMU's virtual-to-physical translations table

Page Faults handled silently by OS



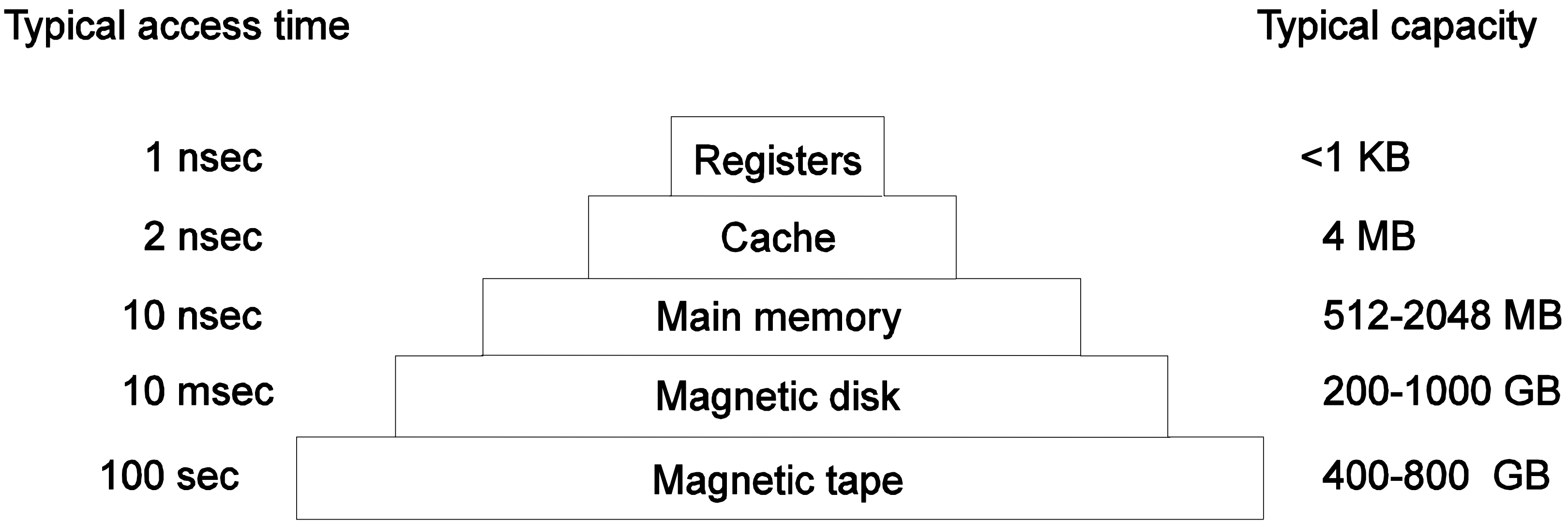
Review

- Page Sharing
  - Copy on write
- Page Replacement
  - Global vs. Local replacement
  - Algorithms:
    - Belady’s Algorithm
    - FIFO
    - LRU
    - Clock (LRU approximation)
- Working Sets
  - Page Fault Frequency
  - Thrashing

Disks and File Systems

- First we'll discuss properties of physical disks
  - Structure
  - Performance
  - Scheduling
- Disk properties motivate how we build file systems on them
  - Files
  - Directories
  - Sharing
  - Protection
  - File System Layouts
  - File Buffer Cache
  - Read Ahead

Data Storage



Typical access time		Typical capacity
1 nsec	Registers	<1 KB
2 nsec	Cache	4 MB
10 nsec	Main memory	512-2048 MB
10 msec	Magnetic disk	200-1000 GB
100 sec	Magnetic tape	400-800 GB

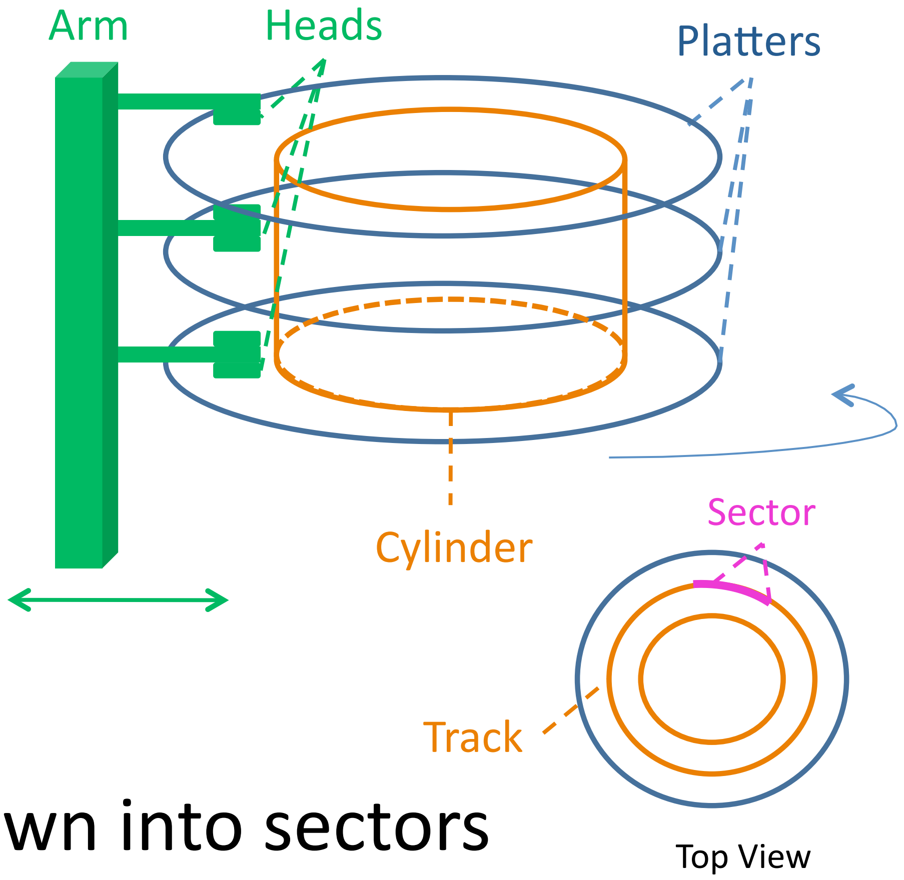
1 msec = 1,000,000 nsec

Memory (DDR2): 2 GB: ~$30
Disk: 1.5 TB = ~$130
(source: tigerdirect.com)

Note: image from Tanenbaum MOS 3/e

Physical Disk Structure

- Disk components
  - Platters (2 surfaces)
  - Tracks
  - Sectors
  - Cylinders
  - Arm
  - Heads (1 per side)



- Logically, disk broken down into sectors
  - Addressed by cylinder, head, sector

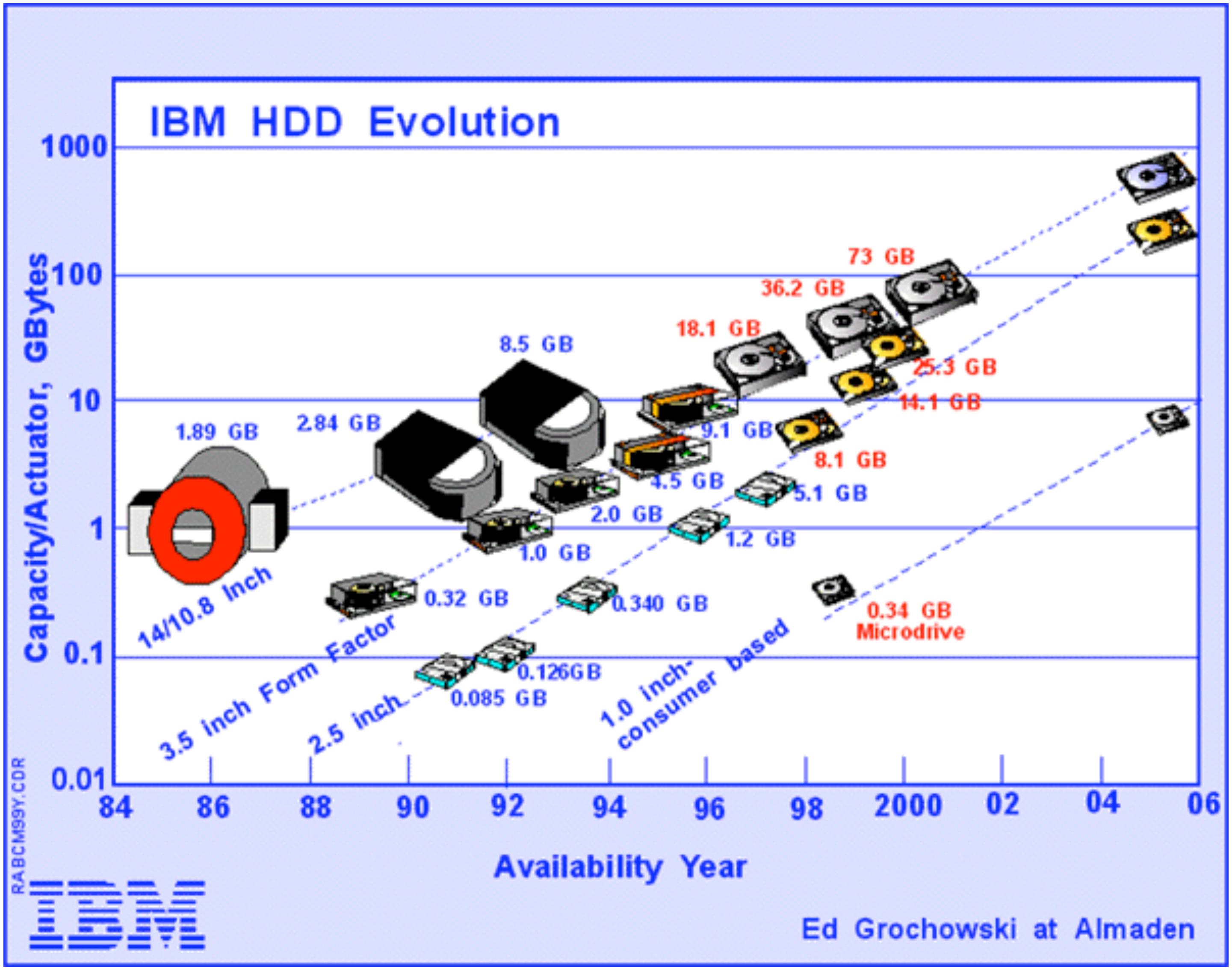
Disks and the OS

- Disks are messy and slow physical devices:
  - Disks just write to sectors, no notion of files or other logical partitions
  - Errors, bad blocks, missed seeks, etc.
  - Access times are *many* orders of magnitude slower than memory

- The OS hides much of this mess from higher level software
  - Hide low-level device control (initiate a disk read, etc.)
  - Present higher-level abstractions (files, databases, etc.)

Disk Interaction

- Specifying disk requests requires a lot of info:
  - Cylinder#, platter surface#, track#, sector#, transfer size…
- Older disks required the OS to specify all of this
  - The OS needed to know *all* disk parameters
- Modern disks are more complicated
  - Not all sectors are the same size, sectors are remapped,…
- Current disks provide a higher-level interface (SCSI)
  - The disk exports its data as a logical array of blocks [0…N]
    - Disk maps logical blocks to cylinder/surface/track/sector
  - Only need to specify the logical block # to read/write
  - But now the disk parameters are hidden from the OS



Source: pcguide.com

Disk Parameters (2009)

Seagate Barracuda 7200.11	
Capacity	1.5 TB
Platters, Surfaces	4, 8
Cache	32 MB
Transfer rate	62 MB/s (inner) – 120 MB/s (outer)
Sector size	512 B
Spindle speed	7200 RPM
Random read seek time	~ 8.5 msec
Random write seek time	~ 9.5 msec
MTBF	750,000 hours

Disk interface speeds	
SCSI	5 MB/sec to 320 MB/sec
ATA	33 MB/sec to 100 MB/sec
Serial ATA (SATA)	150 MB/sec to 300 MB/sec
USB 2.0	60 MB/sec
Firewire	50 MB/sec

Disk Performance

- Disk request performance depends upon…..
  - I/O request overhead: issuing the command to the disk
    - Process file access traps into kernel, which needs to issue hw request
  - Seek: moving the disk arm to the correct cylinder
    - Depends on how fast the disk arm can move (increasing very slowly)
  - Rotation: waiting for the sector to rotate under the head
    - Depends upon rotation rate of disk (increasing, but slowly)
  - Transfer: transferring data from surface into disk controller electronics, sending it back to the host
    - Depends on density (increasing quickly)
    - Faster for tracks near the outer edge of the disk – why?
- The OS tries to minimize the cost of all of these steps
  - Particularly seeks and rotation (why?)

Disk Scheduling

- Because seeks are so expensive (milliseconds!), it helps to schedule disk requests that are queued waiting for the disk
  - FCFS/FIFO (do nothing)
    - Reasonable when load is low
    - Long waiting times for long request queues
  - SSTF (shortest seek time first)
    - Minimize arm movement (seek time), maximize request rate
    - Favors middle tracks
  - SCAN (elevator)
    - Service requests in one direction until done, then reverse
    - Discriminates against the highest and lowest tracks
  - C-SCAN
    - Like SCAN, but only go in one direction (typewriter)
    - Reduce variance in seek times

Disk Scheduling (2)

- In general, unless there are request queues, disk scheduling does not have much impact
  - Important for servers, less so for PCs

- Modern disks often do the disk scheduling themselves
  - Disks know their layout better than OS, can optimize better
  - Ignores, undoes any scheduling done by OS

File Systems

- How do file systems fit in?
- Implement an abstraction (files) for secondary storage
- Organize files logically (directories)
- Permit sharing of data between processes, people, and machines
- Protect data from unwanted access (security)

Files

- A file is data with some properties
  - Contents, size, owner, last read/write time, protection, etc.
- A file can also have a type
  - Understood by the file system
    - Block, character, device, portal, link, etc.
  - Understood by other parts of the OS or runtime libraries
    - Executable, dll, source, object, text, etc.
- A file's type can be encoded in its name or contents
  - Windows encodes type in name
    - .com, .exe, .bat, .dll, .jpg, etc.....
  - Unix encodes type in contents
    - Magic numbers, initial characters (e.g., #! for shell scripts)

Basic File Operations

Unix	Windows NT
creat(name)	CreateFile(name, CREATE)
open(name, how)	CreateFile(name, OPEN)
read(fd, buf, len)	ReadFile(handle, …)
write(fd, buf, len)	WriteFile(handle,…)
sync(fd)	FlushFileBuffers(handle,…)
seek(fd, pos)	SetFilePointer(handle,…)
close(fd)	CloseHandle(handle,…)
unlink(name)	DeleteFile(name)

Directories

- Directories serve two purposes
  - For users, they provide a structured way to organize files
  - For the file system, they provide a convenient naming interface that allows the implementation to separate logical file organization from physical file placement on the disk
    - Why might this help?
- Most file systems support multi-level directories
  - Naming hierarchies (/, /usr, /usr/local/, …)
- Most file systems support the notion of a current directory
  - Relative names specified with respect to current directory
  - Absolute names start from the root of directory tree

Directory Internals

- A directory is a list of entries
  - <name, location>
  - Name is just the name of the file or directory
  - Location depends upon how file is represented on disk
- List is usually unordered (effectively random)
  - Entries usually sorted by program that reads directory
- Directories typically stored in files
  - Only need to manage one kind of secondary storage unit

Path Name Translation

- Let’s say you want to open “/one/two/three”
- What does the file system do?
  - Open directory “/” (well known, can always find)
  - Search for the entry, “one”, get location of “one” (in dir entry)
  - Open directory “one”, search for “two”, get location of “two”
  - Open directory “two”, search for “three”, get location of “Three”
  - Open file “three”

- Systems spend a lot of time walking directory paths
  - This is why open is separate from read/write
  - OS will cache prefix lookups for performance
    - /a/b, /a/bb, /a/bbb, etc., all share “/a” prefix

Storing Files

- Disk is partitioned into Blocks or Sectors
  - Modern disks have 512-byte sectors
  - File systems usually work in block sizes of 4 KB

- Files can span multiple blocks
  - File sizes may span multiple blocks, or may be small

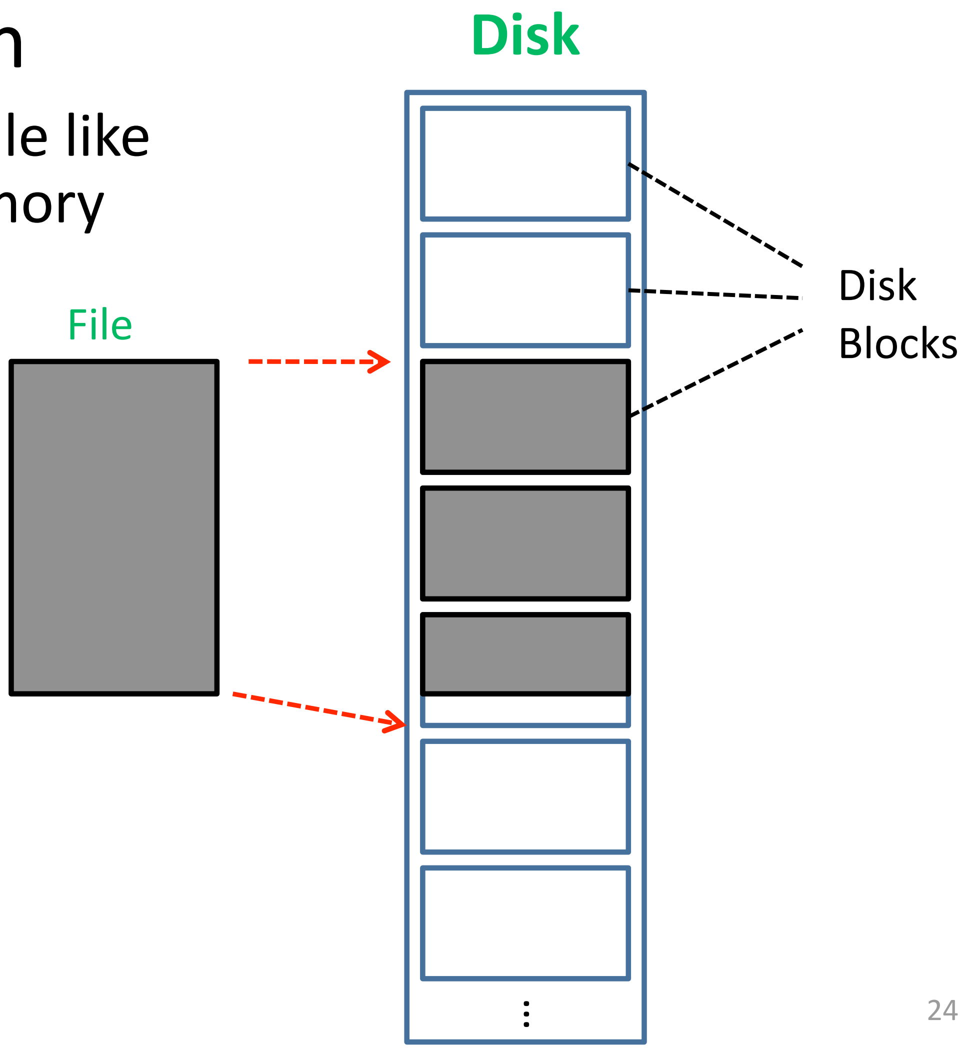
- Things to consider
  - File access: is it random, sequential?
  - File size: how often does it grow/shrink?

- Sound familiar?

Disk Layout Strategies

Contiguous allocation

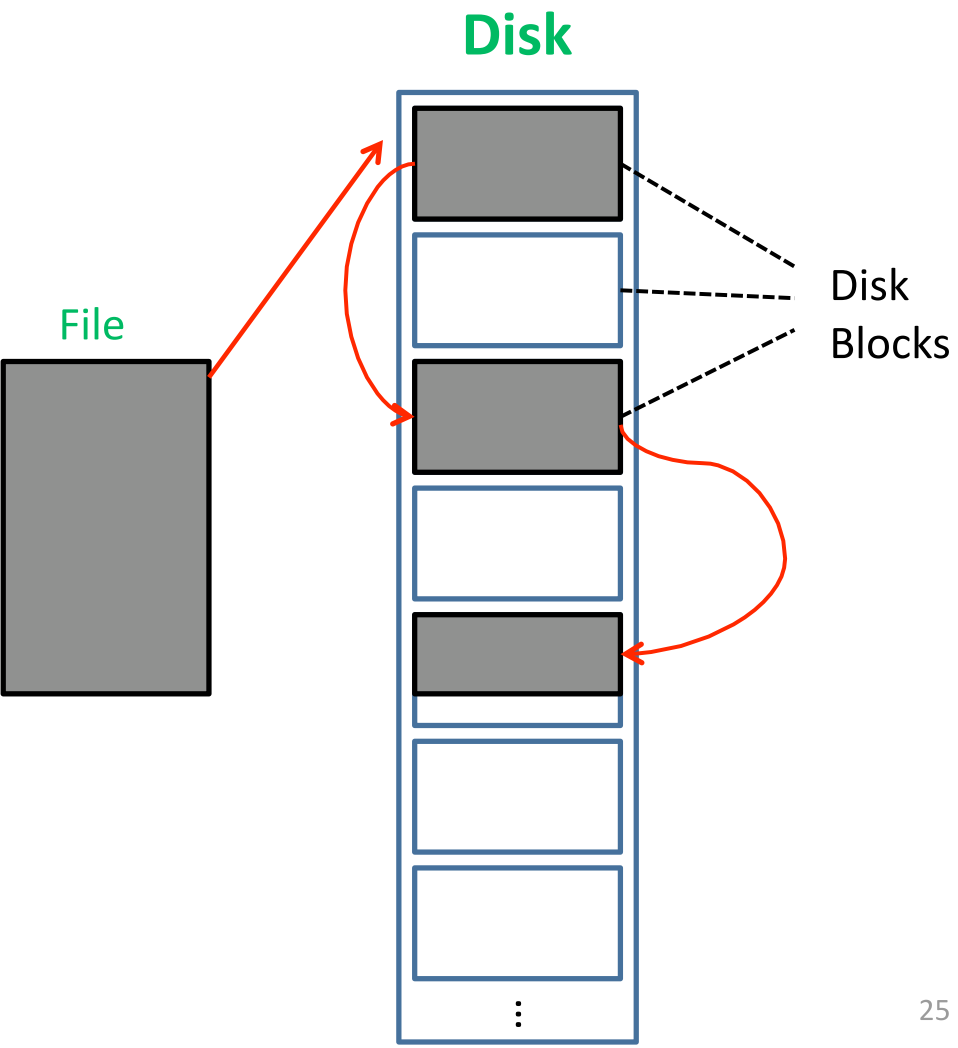
- Idea: Allocate space for file like done for contiguous memory organization
- Pros: Fast file access
- Cons: Fragmentation, needs compaction
  - What happens when you need to grow?



Disk Layout Strategies

Linked Allocation

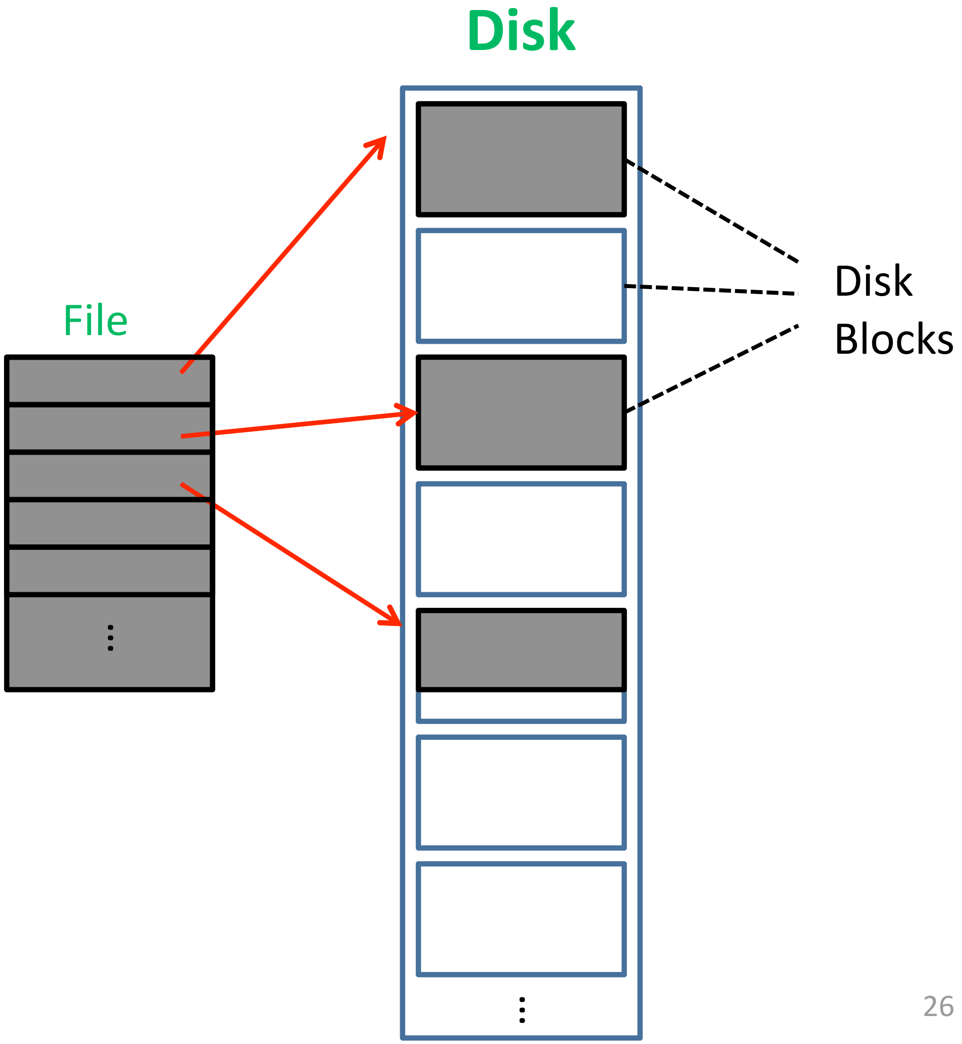
- Idea: Linked list of blocks, each pointing to next
- Pros: Easy to grow; fast sequential access
- Cons: Slow non-sequential access; what happens if you have one bad block?



Disk Layout Strategies

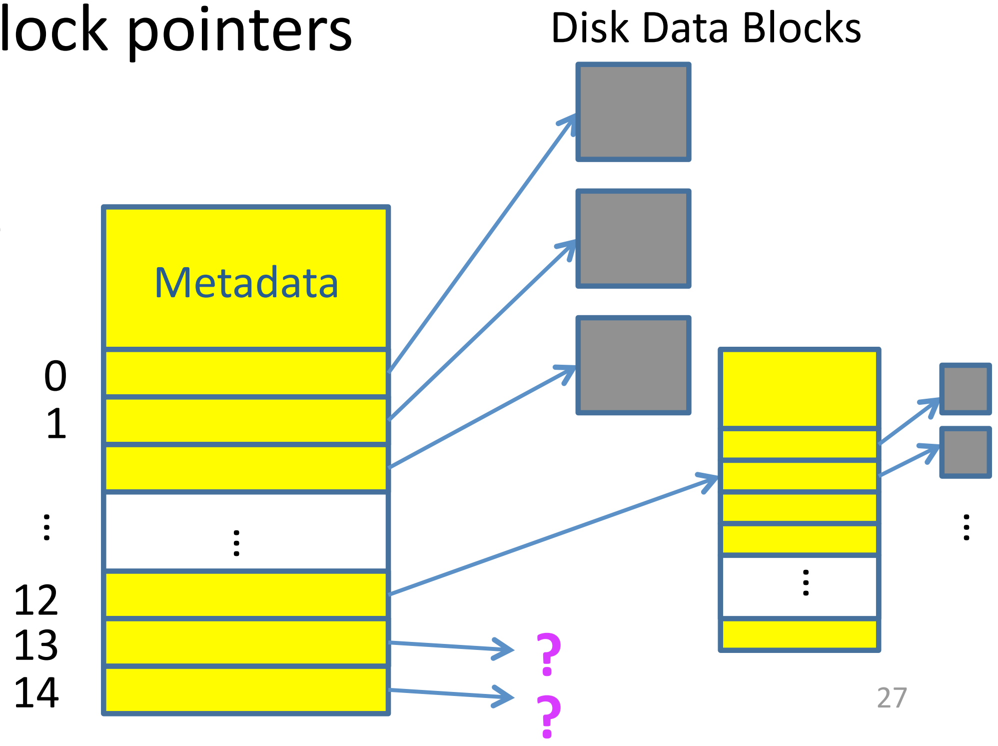
Indexed Allocation

- Idea: Store ordered list of block pointers
- Pros: Good for random access, not bad for sequential
- Cons: Size limit, not as fast for sequential access



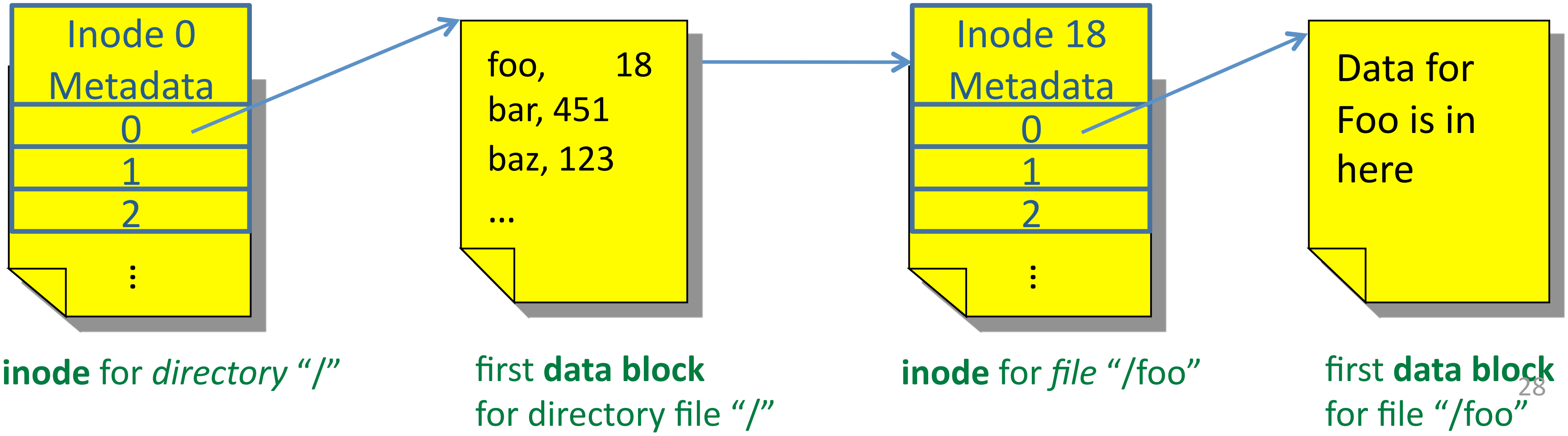
Unix Inodes

- Unix uses an indexed allocation structure
  - An inode (index node) stores both metadata and the pointers to disk blocks
    - Metadata is information *about* the file (protection, timestamp, length, ref count, etc….)
- Each inode contains 15 block pointers
  - First 12 are *direct* blocks (e.g., 4 KB disk blocks)
  - Then single, double, triple *indirect* blocks



Resolving File Location/Data

- Inodes describe where on disk the blocks for a file are placed
  - Unix inodes are *not* directories
  - Directores are represented internally as *files*
    - *What does this mean for how inodes are stored?*
- Directory entries map file names to inodes
  - Want to access "/foo"



Resolving File Location/Data

- Inodes describe where on disk the blocks for a file are placed
  - Unix inodes are *not* directories
  - Directores are represented internally as *files*
    - *What does this mean for how inodes are stored?*
- Directory entries map file names to inodes
  - To open "/foo", use Master Block to find "/" on disk
  - Open "/", look for entry "foo"
  - This entry contains the disk block number for inode for "foo"
  - Read the inode "foo" into memory
  - The inode says where the first data block is on disk
  - Read first data block into memory to access data in file "foo"

That was a lot of work to read one file!

Improving Performance

- We understand how file systems are structured
  - Inodes, data blocks, files, directories, etc…..
- Now we’ll focus on how they perform
  - Where do we place data?
  - Are there any tricks we can play to mask latencies?
- Three case studies:
  - Berkeley Fast File System (FFS)
  - Log-Structured File System (LFS)
  - Redundant Array of Inexpensive Disks (RAID)

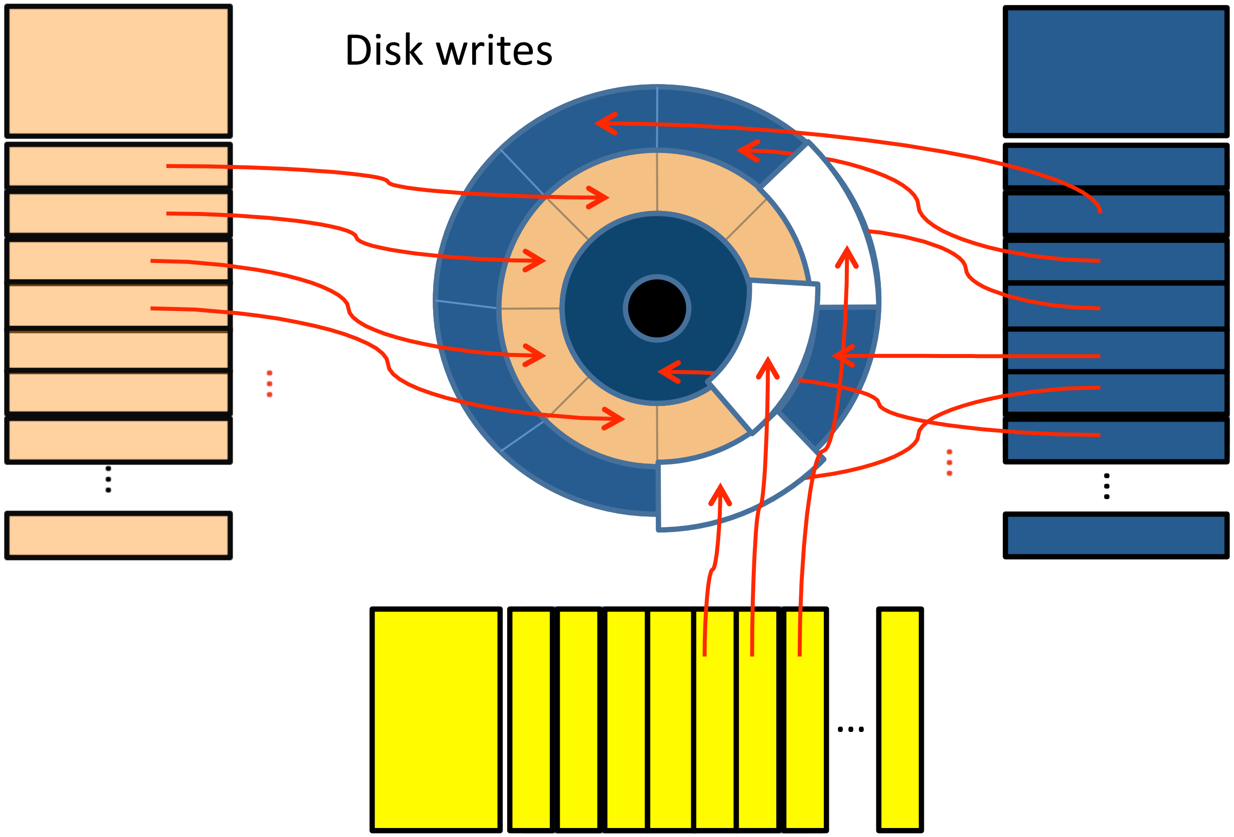
Berkeley Fast File System (FFS)

- 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 file system from which all other Unix file systems have been compared
- Good example of being device-aware for performance

Data and Inode Placement Problem

- 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

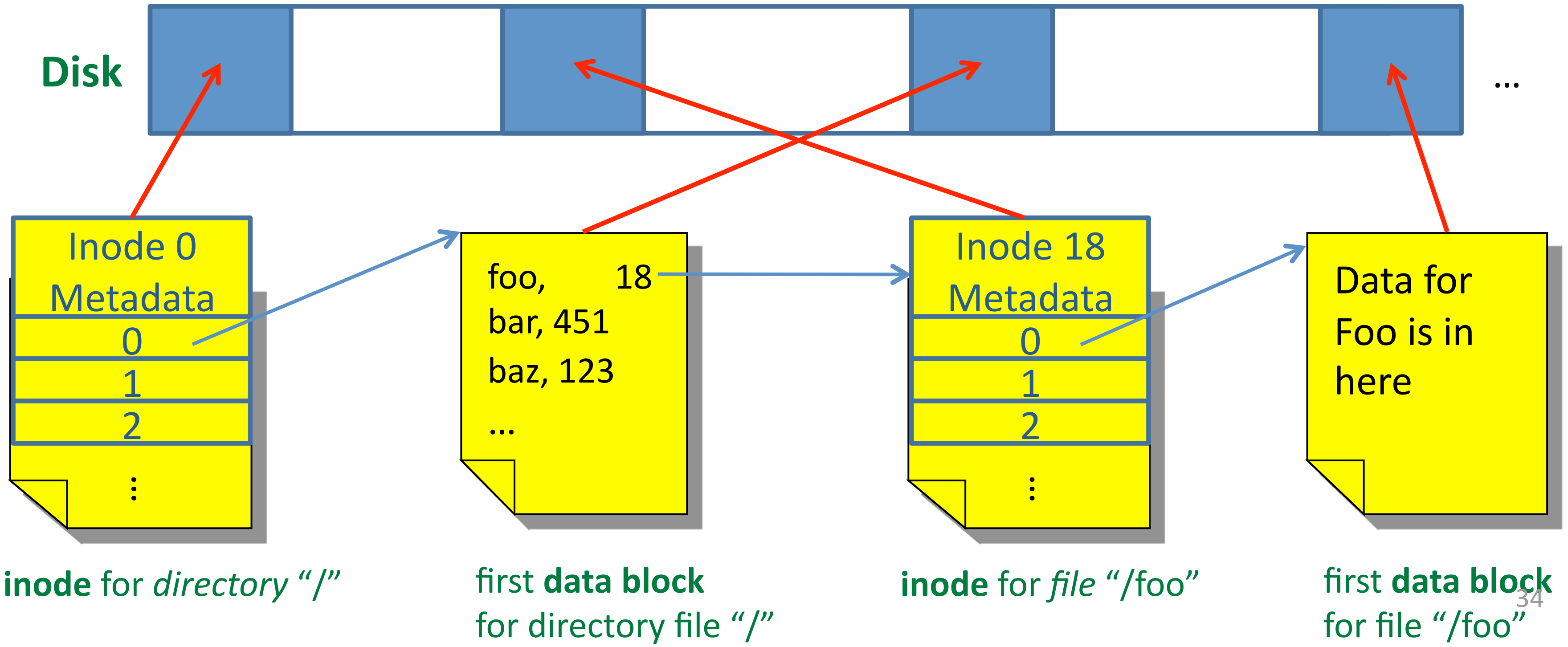
Data and Inode Placement Problem



Over time, block placement gets scattered:
("swiss cheese" effect)

Data and Inode Placement Problem

- 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
    - Remember accessing "/foo" example?



inode for *directory "/"*

first **data block** for directory file "/"

inode for *file "/foo"*

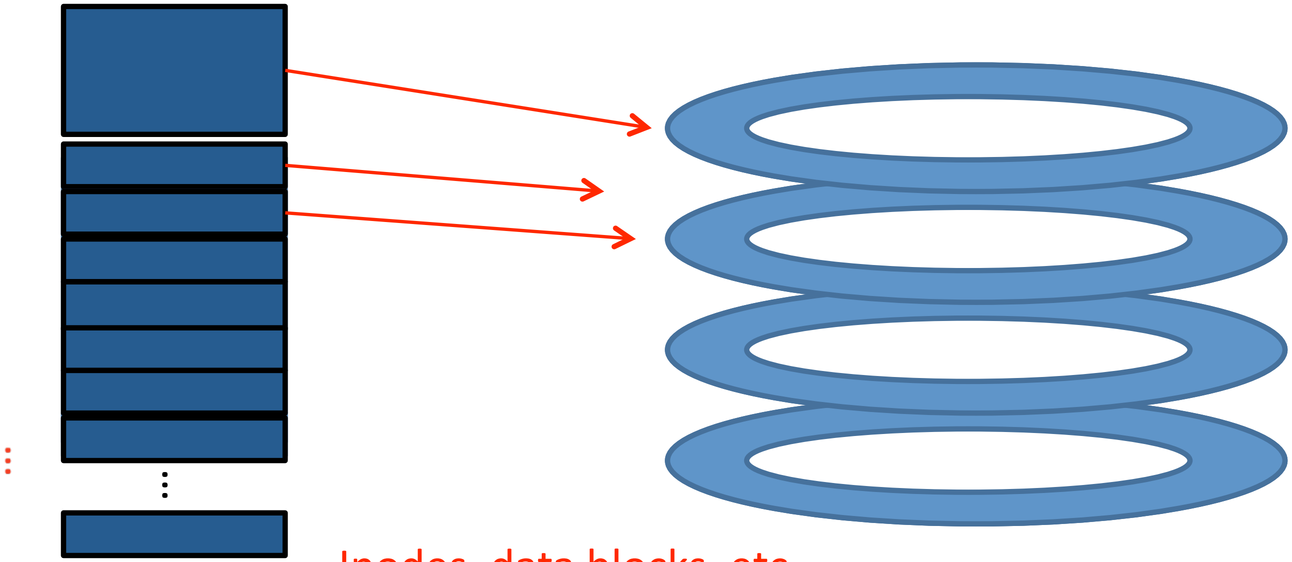
first **data block** for file "/foo"

Data and Inode Placement Problem

- 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

Cylinder Groups

- BSD FFS addressed both of these problems using the notion of a cylinder group
  - Disk partitioned into groups of cylinders
  - Data blocks in same file allocated in same cylinder
  - Files in same directory allocated in same cylinder
  - Inodes for files allocated in same cylinder as file data blocks



Inodes, data blocks, etc…

Reduces number of seeks!

Cylinder Groups

- BSD FFS addressed both of these problems using the notion of a cylinder group
  - Disk partitioned into groups of cylinders
  - Data blocks in same file allocated in same cylinder
  - Files in same directory allocated in same cylinder
  - Inodes for files allocated in same cylinder as file data blocks
- Free space requirement
  - To be able to allocate according to cylinder groups, the disk must have free space scattered across cylinders
  - 10% of the disk is reserved just for this purpose
    - Only used by root – why it is possible for “df” to report > 100%

Problems with Small Blocks

- Small blocks (1K) caused two problems:
  - Low bandwidth utilization
  - Small max file size (function of block size)

Maximum File Size: 1 KB Blocks

- Recall Unix inodes have:
  - 12 direct blocks
  - 1 single indirect block, 1 double indirect block, 1 triple indirect block
- How large can a file be with 1KB blocks?
- Single indirect block:
  - Assuming 32-bit addresses, we have 4 bytes per block pointer, so 1 KB/4 = 256 blocks
  - So … 256 * 1 KB = 256 KB
- Double-indirect block:
  - 256 * 256 * 1 KB = 64 MB
- Triple Indirect block:
  - 256 * 256 * 256 * 1 KB = 16 GB
- Total: ~16 GB

Problems with Small Blocks

- Small blocks (1K) caused two problems:
  - Low bandwidth utilization
  - Small max file size (function of block size)

- Fix using larger blocks (4K)
  - Very large files, only need two levels of indirection for supporting files of size 2^32

Maximum File Size: 4 KB Blocks

- Recall Unix inodes have:
  - 12 direct blocks
  - 1 single indirect block, 1 double indirect block, 1 triple indirect block
- How large can a file be with 4KB blocks?
- Single indirect block:
  - Assuming 32-bit addresses, we have 4 bytes per block pointer, so 4 KB/4 = 1024 B blocks
  - So … 1024 * 1 KB = 1 MB
- Double-indirect block:
  - 1024 * 1024 * 1 KB = 1 GB
- Triple Indirect block:
  - 1024 * 1024 * 1024 * 1 KB = 1 TB
- Total: ~1 TB

Problems with Small Blocks

- Small blocks (1K) caused two problems:
  - Low bandwidth utilization
  - Small max file size (function of block size)

- Fix using larger blocks (4K)
  - Very large files, only need two levels of indirection for supporting files of size 2^32
  - *Why not just use all indirect blocks?*
    - Over 65% of files are smaller than 4 KB (Tanenbaum , OSR 2006)
      - What's the problem with that?

Problems with Small Blocks

- Small blocks (1K) caused two problems:
  - Low bandwidth utilization
  - Small max file size (function of block size)

- Fix using larger blocks (4K)
  - Very large files, only need two levels of indirection for supporting files of size 2^32
  - Problem: internal fragmentation
  - Fix: Introduce "fragments" (1K pieces of a block can be used for other, small files)

Other Problems

- Problem: Media failures
  - If you lose the superblock, you lose everything
    - Or at least recovery is expensive
  - Solution: Replicate master block (superblock)

- Problem: reduced seeks, but even one is expensive
  - What if we can avoid going to disk at all?

- Next: other File System tricks

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 (usually LRU used)

Caching Writes

- Applications assume writes make it to 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
  - Non-volatile RAM (NVRAM)
    - As with write-behind, but maintain queue in NVRAM
    - Expensive

Read Ahead (Prefetching)

- Many file systems implement “read ahead”
  - FS predicts that the process will request next block
  - FS goes ahead and requests it from the disk…
  - ..while the process is computing on previous block!
  - When the process requests block, it will be in cache
  - Complements the disk cache, which also is doing read ahead
- For sequentially accessed files can make big difference
  - Unless blocks for the file are scattered across the disk
  - File systems try to prevent that, though (during allocating)
- Unfortunately, this doesn’t do anything for writes
  - What if we could make write-behind sequential as well?

Log-structured File System

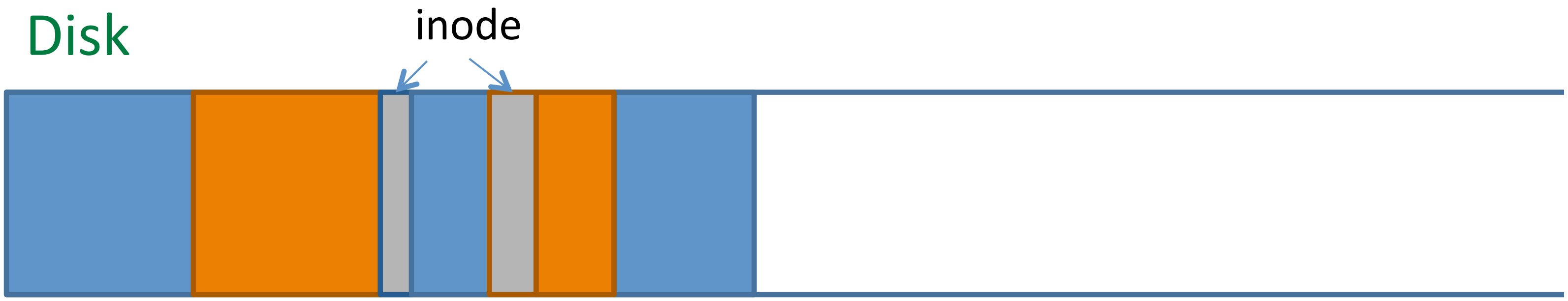
- The Log-structured File System (LFS) was designed in response to two trends in workload and technology:
- 1) Disk bandwidth scaling significantly (40% a year)
  - Latency is not
- 2) Large main memories in machines
  - Large buffer caches
  - Absorb large fraction of read requests
  - Can use for writes as well
  - Coalesce small writes into large writes
- LFS takes advantage of both of these to increase FS performance
  - Rosenblum and Ousterhout (Berkeley, ‘91)

LFS: Approach

Optimize for disk writes

- Batch writes in disk cache
  - Utilize increase in disk throughput
- Treat the disk as one big log for writes
  - No need to worry about special seeks or placement
- All data in file system appended to log
  - Data blocks, metadata, inodes, etc.

LFS: Example



File 1

🟦

Write a file
Modify file
Write to file

File 2

🟧

Write to file
Modify file
…

LFS Challenges

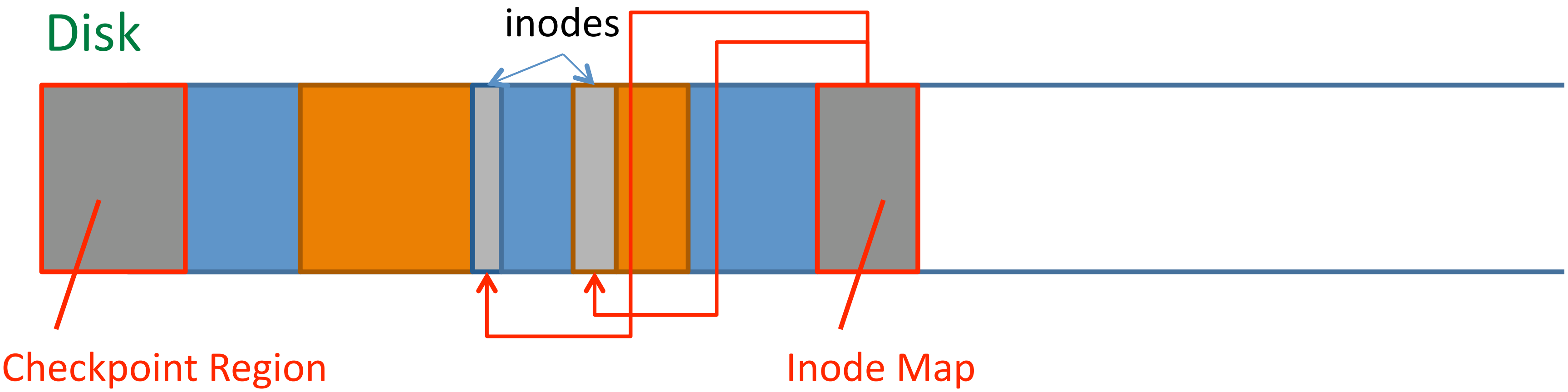
- How do you locate data?
  - FFS places files in a particular location
  - LFS appends data to the end of the log

- How do you free data?
  - At some point, you can’t “append” anymore
  - How do you track and recover stale blocks in the log?

LFS: Locating Data

- FFS uses inodes to locate data blocks
  - Inodes pre-allocated in each cylinder group
  - Directories contain locations of inodes
- LFS appends inodes and data (basically everything) to end of the log
  - Makes them hard to find
- Approach
  - Use another level of indirection: Inode maps
  - Inode maps map file #s to inode location
  - Location of inode map blocks kept in checkpoint region
  - Checkpoint region has a fixed location
  - Cache inode maps in memory for performance

LFS: Example (inode maps)



File 1

🟦

Write a file
Modify file
Write to file

File 2

🟧

Write to file
Modify file
…

Aren't reads still slow?

Rely on buffer cache to store inode maps.

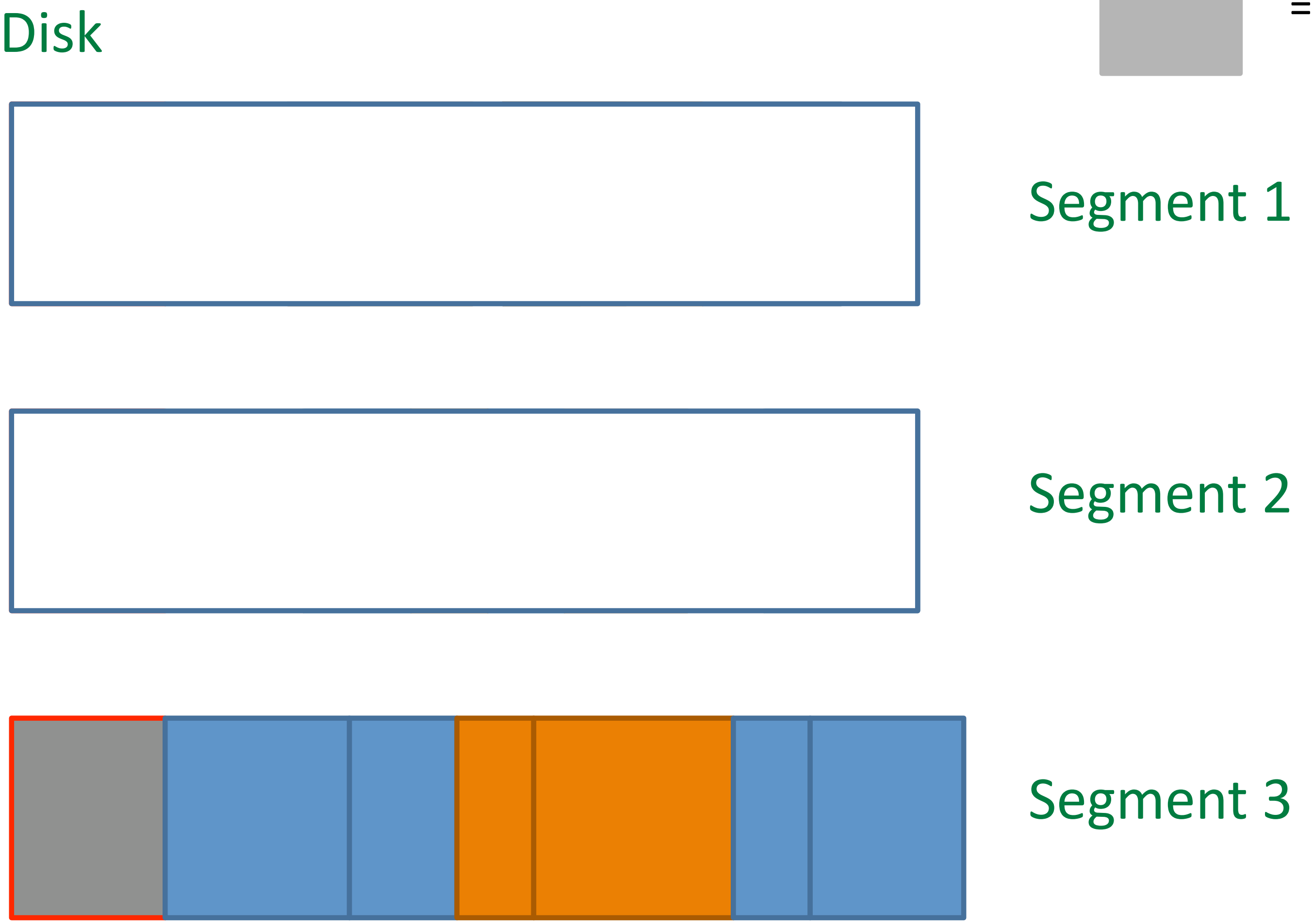
Large buffer cache means don't need to worry about reads!

LFS: Free Space Management

- LFS append-only quickly runs out of disk space
  - Need to recover deleted blocks
- Approach:
  - Fragment log into segments
  - Thread segments on disk
    - Segments can be anywhere
  - Reclaim space by cleaning segments
    - Read segment
    - Copy live data to end of log
    - Now have free segment you can reuse

LFS Example (cleaning)

▨ = dead region



LFS: Free Space Management

- LFS append-only quickly runs out of disk space
  - Need to recover deleted blocks
- Approach:
  - Fragment log into segments
  - Thread segments on disk
    - Segments can be anywhere
  - Reclaim space by cleaning segments
    - Read segment
    - Copy live data to end of log
    - Now have free segment you can reuse
- Cleaning is a big problem
  - Costly overhead

LFS: Now

- Revolutionary (at the time) design concept that spurred a lot of debate and research in the area in the 90s
- Present-day file systems use soft updates or journaling, which seem to be due in large part to the concepts from LFS

Summary

- We’ve explained how file systems can be structured
  - Many techniques are similar to those in memory management
  - Unix-style: Inodes, data blocks, files, directories, etc…..
- Performance of file systems highly dependent on disk technology
  - Seeks take a long time
  - Placement of data matters (swiss-cheese problem and seek avoidance)
- Berkeley Fast File System (FFS)
  - Cylinder groups (which files are likely to be accessed together)
  - Larger block sizes to increase throughput
- Log-Structured File System (LFS)
  - Optimize for writes (batch writes)
  - Rely on cache for reads (data placement practically ignored)
- Assorted other tricks
  - Pre-fetching (avoid extra fetches and put in buffer cache)
  - Delayed writes (like LFS; used in modern journaling file systems)

Next Time

- Read Chapter 11.9, 12.7, 15
- Check Web site for course announcements
  - http://www.cs.ucsd.edu/classes/su09/cse120

RAID

- Problem:
  - Disk drives fail frequently
  - Disks are SLOW (seek times & transfer rates)
- Idea: Use many disks in parallel to increase storage bandwidth, improve reliability
  - Files are striped across disks
  - Each stripe portion is read/written in parallel
  - Bandwidth increases with more disks
- Redundant Array of Inexpensive Disks (RAID)
  - A storage system, not a file system
  - Patterson, Katz, and Gibson (Berkeley, ’88)

RAID Levels

- In marketing literature, you will see RAID systems advertised as supporting different "RAID Levels"
- Here are some common levels:
  - RAID 0: Striping
    - Good for random access (no reliability)
  - RAID 1: Mirroring
    - Two disks, write data to both (expensive, 1X storage overhead)
  - RAID 5: Floating Parity
    - Parity blocks for different stripes written to different disks
    - No single parity disk, hence no bottleneck at that disk
  - Raid "10": Striping plus mirroring
    - Higher bandwidth, but still have large overhead
    - See this on UltraDMA PC RAID disk cards

RAID Challenges

- Small files (small writes less than a full stripe)
  - Need to read entire stripe, update with small write, then write entire segment out to disks
- Reliability
  - More disks increases the chance of media failure (MTBF)
- Turn reliability problem into a feature
  - Use one disk to store parity data
    - XOR of all data blocks in stripe
  - Can recover any data block from all others + parity block
  - Hence "redundant" in name
  - Introduces overhead, but assuming disks are "inexpensive"