Disk Drive Operations
Storage Access Methods

Lecture 2

September 19, 2006

Plan for today

- Questions from last time
- On-disk data organization
  - what they're all about
- Storage access methods
  - Block-based access
  - File-based access
  - Object-based access

Disk Drive Firmware Algorithms

Outline

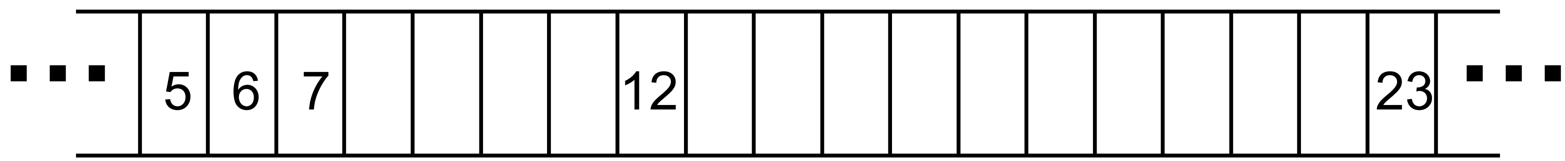
- Mapping LBNs to physical sectors
  - zones
  - defect management
  - track and cylinder skew
- Bus and buffer management
  - optimizing storage subsystem resources
- Advanced buffer space usage
  - prefetching and caching
  - read/write-on-arrival

How functionality is implemented

- Some of it is in ASIC logic
  - error detection and correction
  - signal/servo processing
  - motor/seek control
  - cache hits (often)
- Some of it is in firmware running on control processor
  - request processing
  - request queueing and scheduling
  - LBN-to-PBN mapping
- Key considerations: cost and performance and cost
  - optimize common cases
  - keep things simple and space-conscious

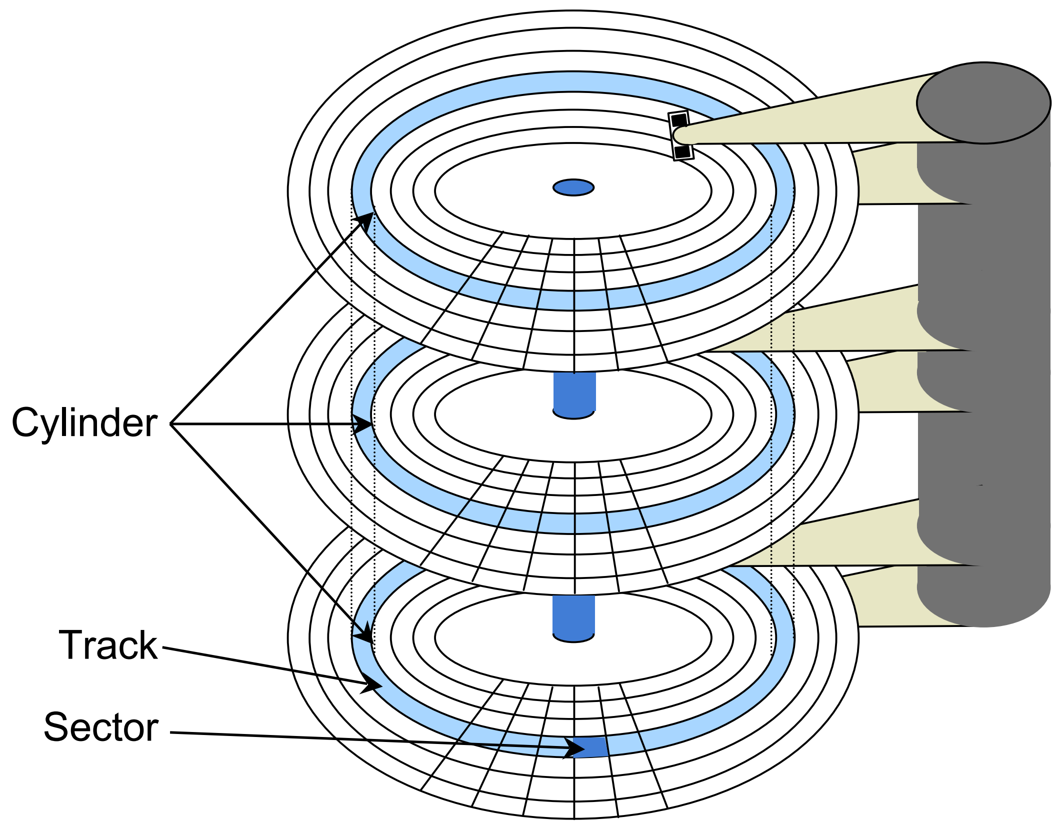
Recall the storage device interface

- Linear address space of equal-sized blocks
  - each identified by logical block number (LBN)

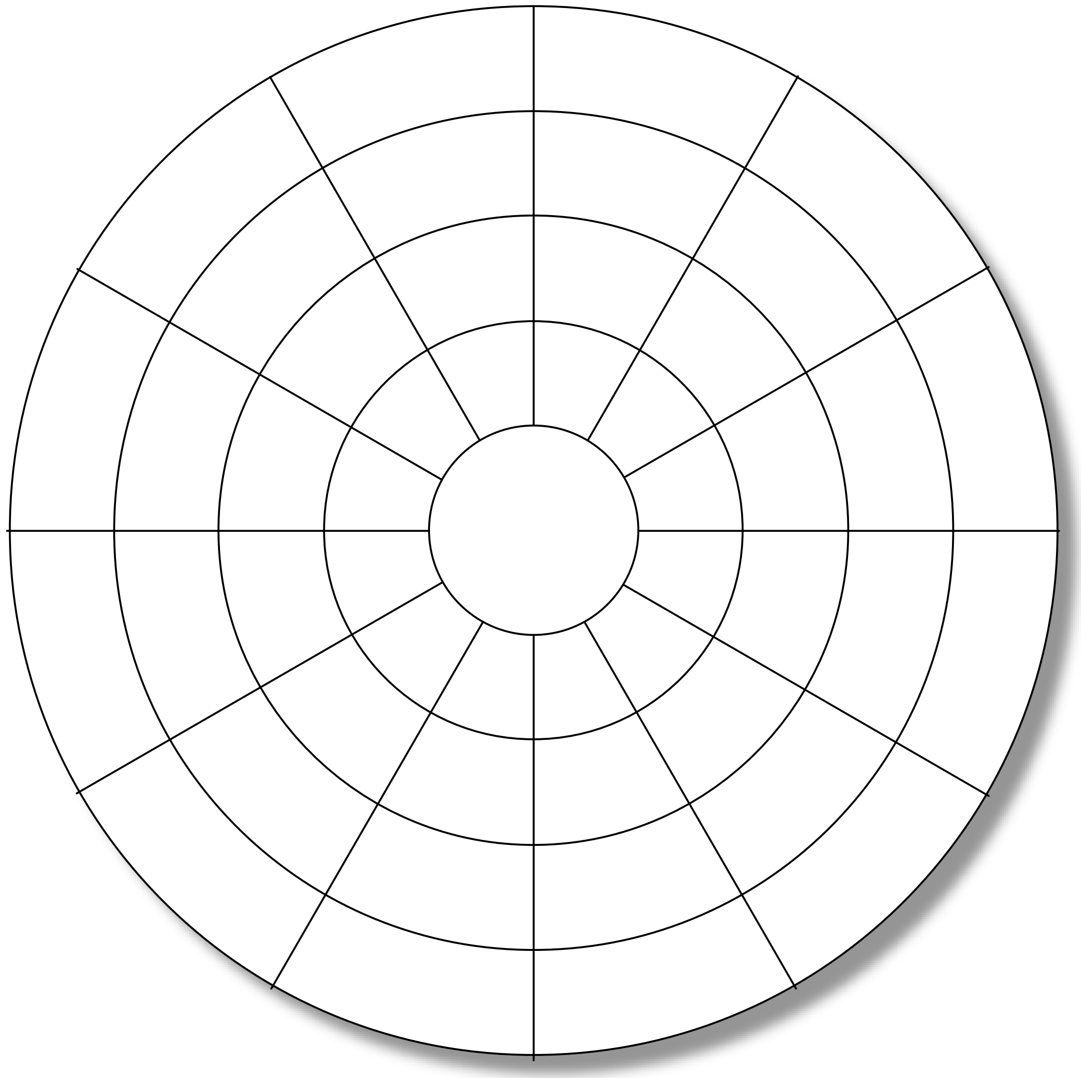


- Common block size: 512 bytes
- Number of blocks: device capacity / block size

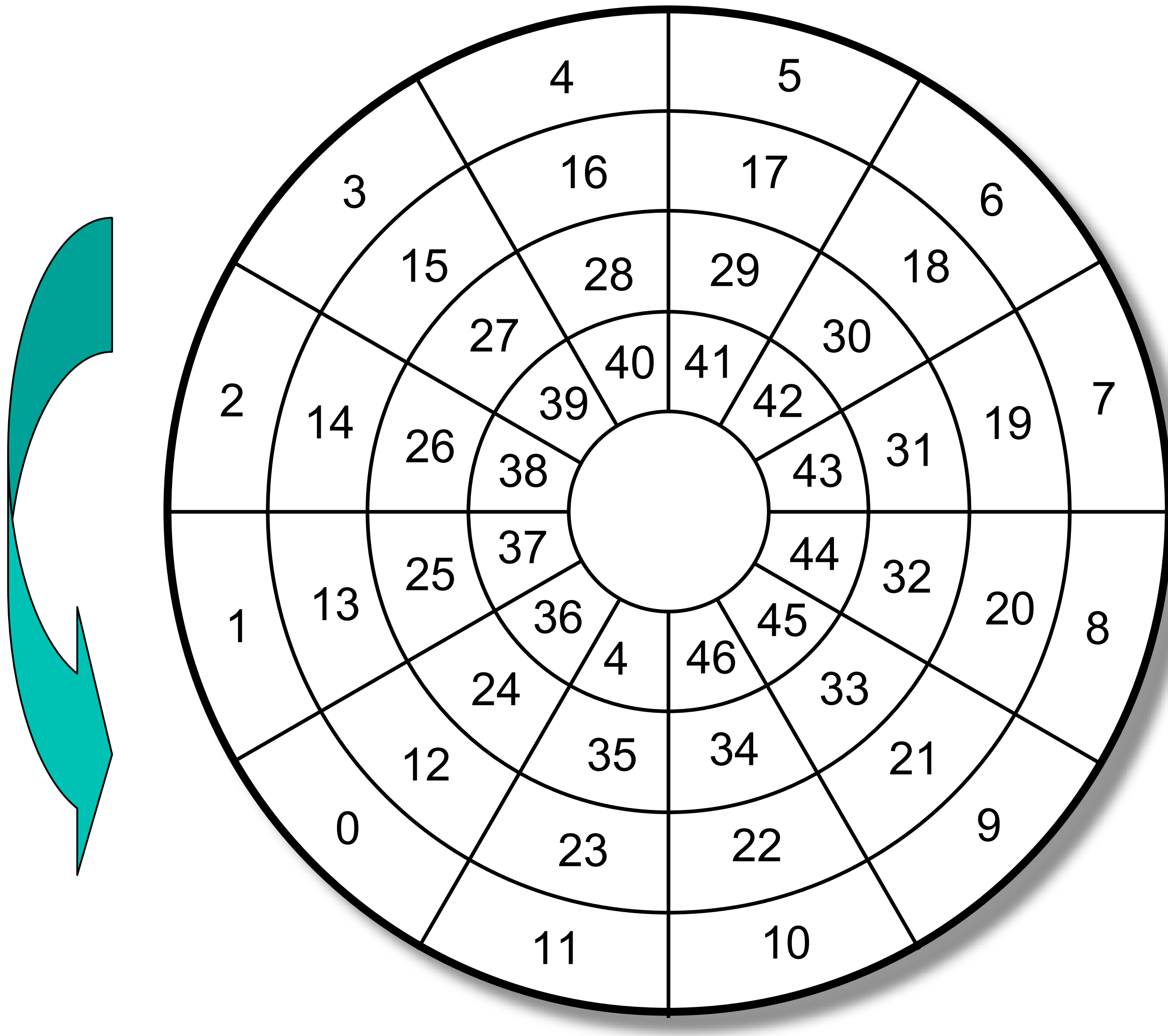
Recall the physical disk storage reality



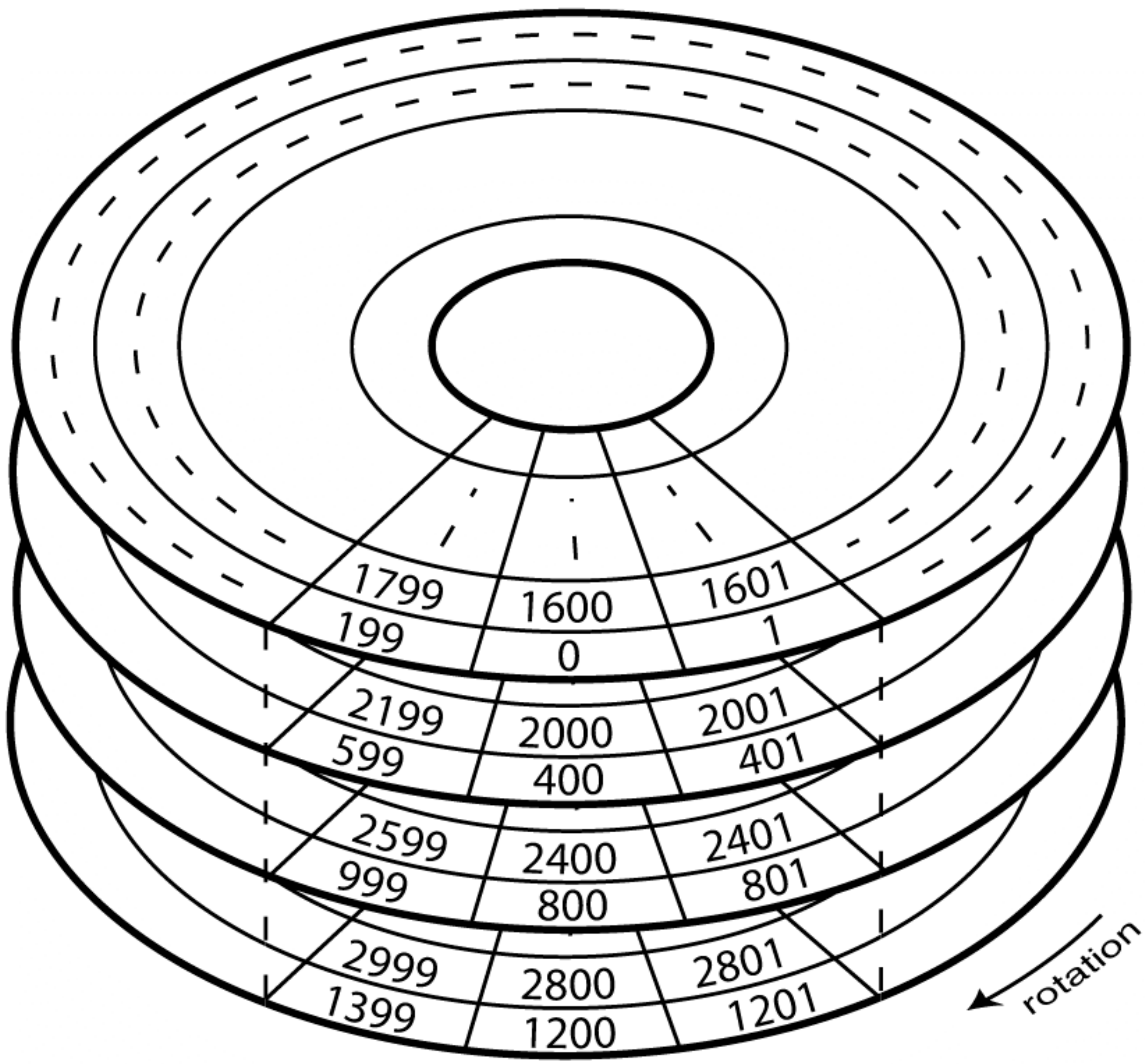
Physical sectors of a single-surface disk



LBN-to-physical for a single-surface disk



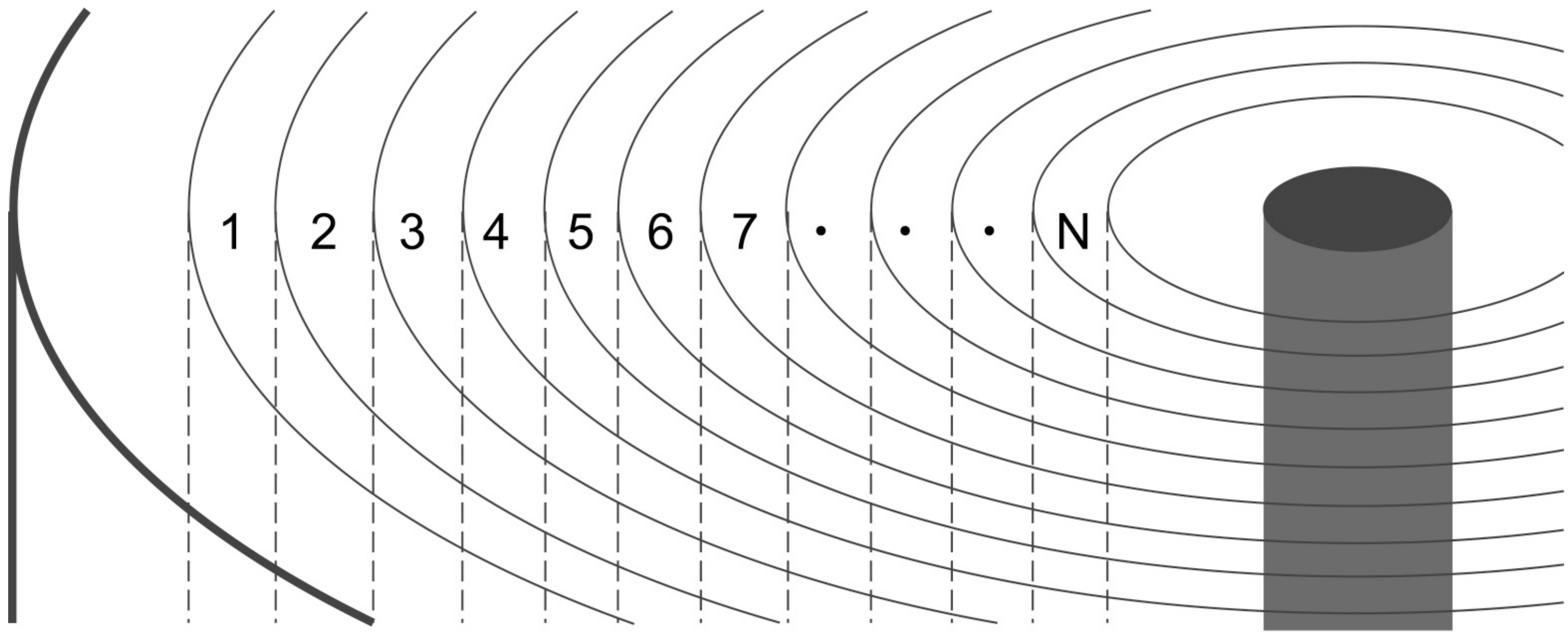
Extending mapping to a multi-surface disk



Some real numbers for modern disks

- # of platters: 1-4
  - 2-8 surfaces for data
- # of tracks per surface: 10s of 1000s
  - same thing as # of cylinders
- # sectors per track: 500-900
  - so, 250-450KB
- # of bytes per sector: usually 512
  - can be chosen by OS for some disks
  - disk manufactures want to make it bigger

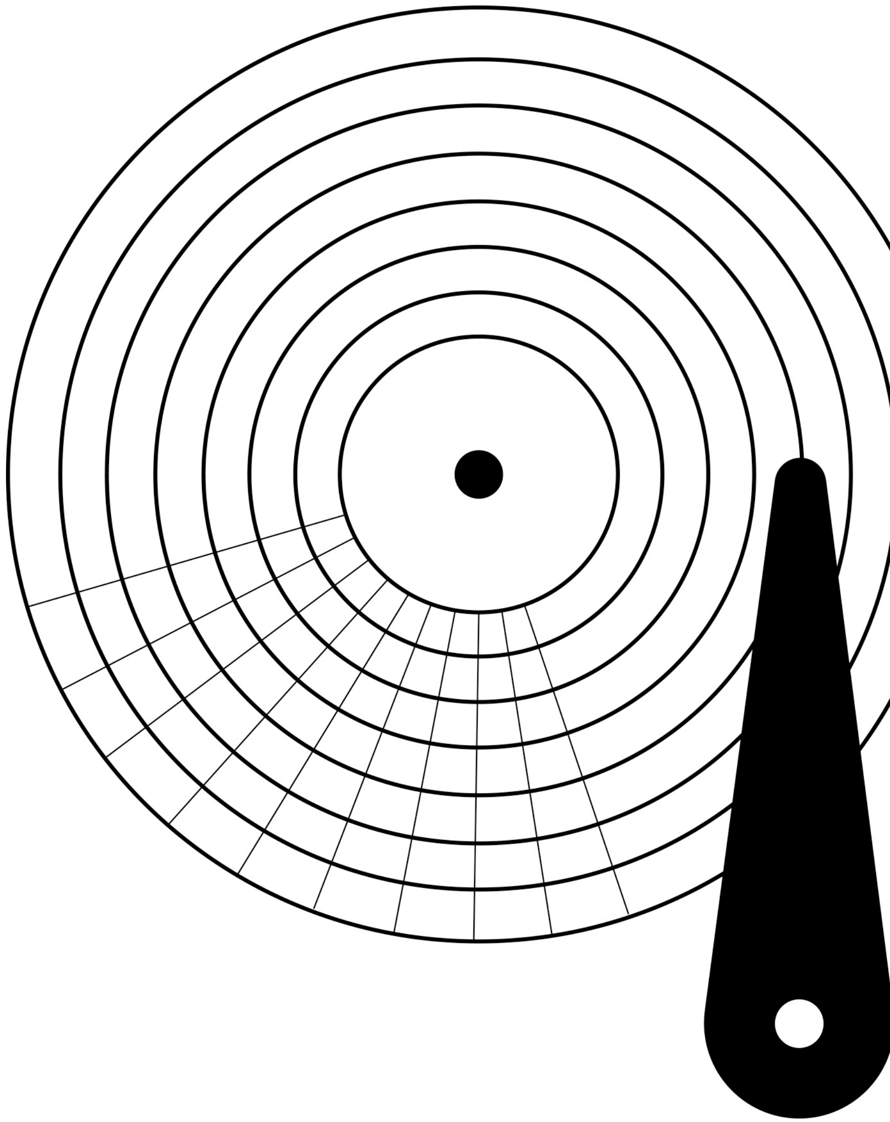
Clarification of Cylinder Numbering



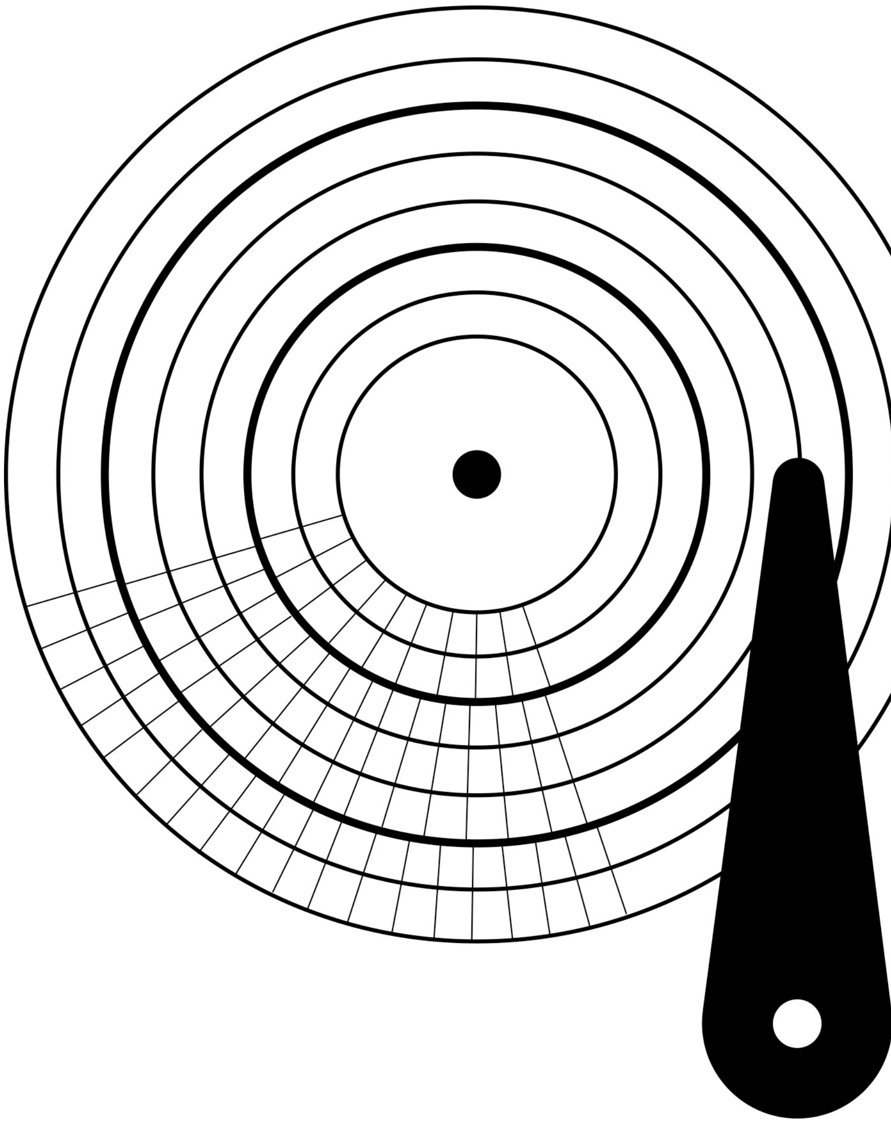
First Complication: Zones

- Outer tracks are longer than inner ones
  - so, they can hold more data
  - benefits: increased capacity and higher bandwidth
- Issues
  - increased bookkeeping for LBN-to-physical mapping
  - more complex signal processing logic
    - because of variable bit rate timing
- Compromise: zones
  - all tracks in each zone hold same number of sectors

Constant number of sectors per track



Multiple "zones"



A real zone breakdown

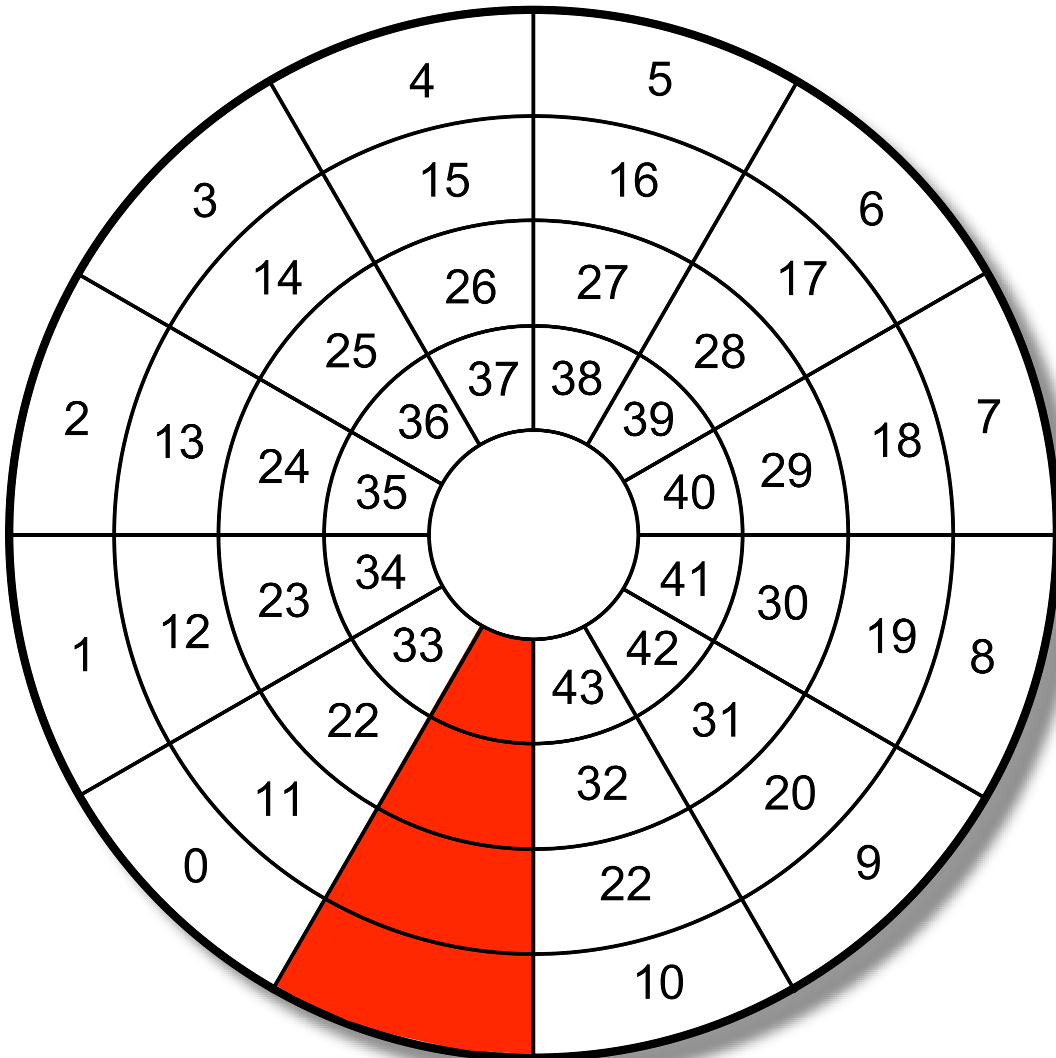
- IBM Ultrastar 18ES (1998)

Zone	Start cylinder	End cylinder	SPT
0	0	377	390
1	378	1263	374
2	1264	2247	364
3	2248	3466	351
4	3466	4504	338
5	4505	5526	325
6	5527	7044	312
7	7045	8761	286
8	8762	9815	273
9	9816	10682	260
10	10683	11473	247

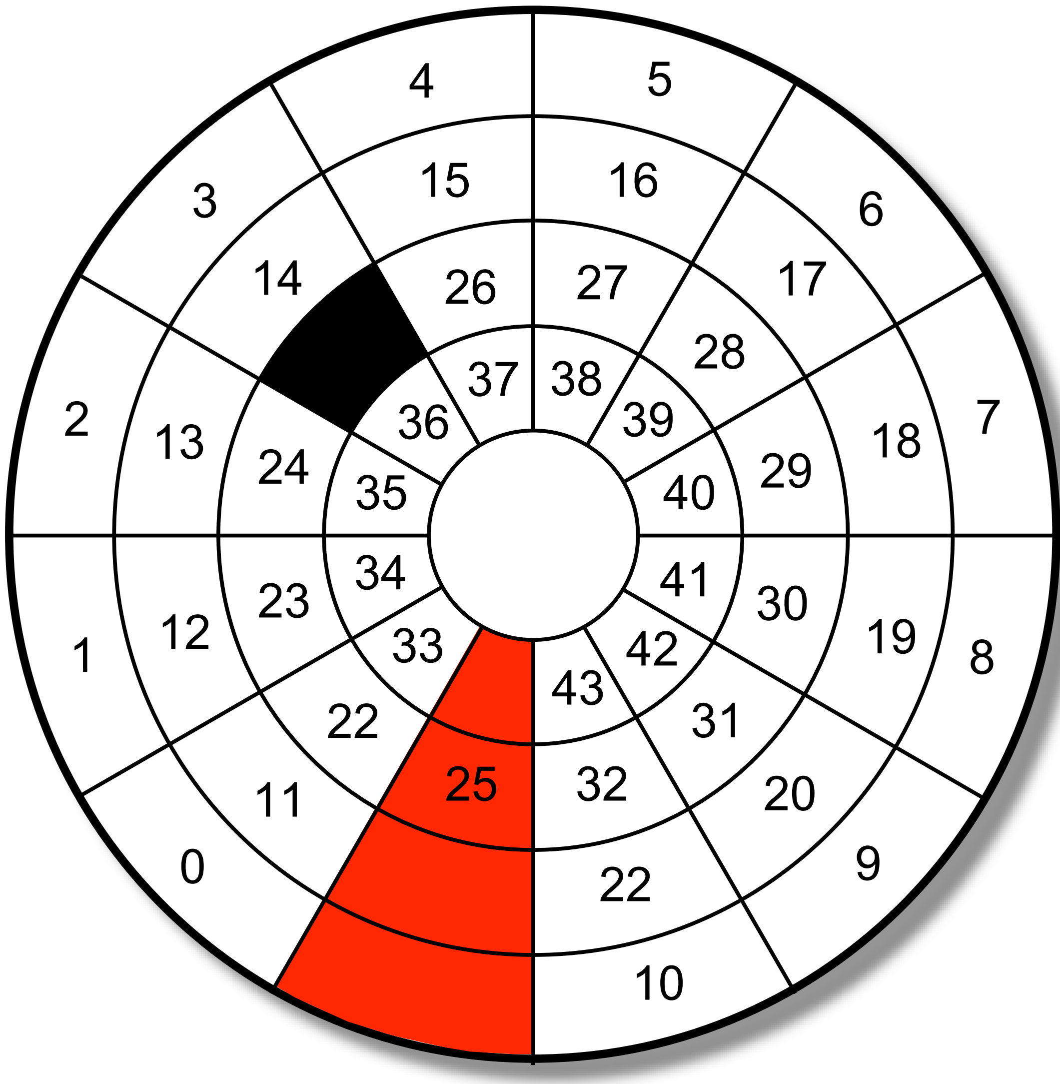
Second Complication: Defects

- Portions of the media can become unusable
  - both before installation and during use
    - former is MUCH more common than latter
- Need to set aside physical space as spares
  - simplicity dictates having no holes in LBN space
  - many different organizations of spare space
    - e.g., sectors per track, cylinder, group of cylinders, zone
- Two schemes for using spare space to handle defects
  - remapping
    - leave everything else alone and just remap the disturbed LBNs
  - slipping
    - change mapping to skip over defective regions

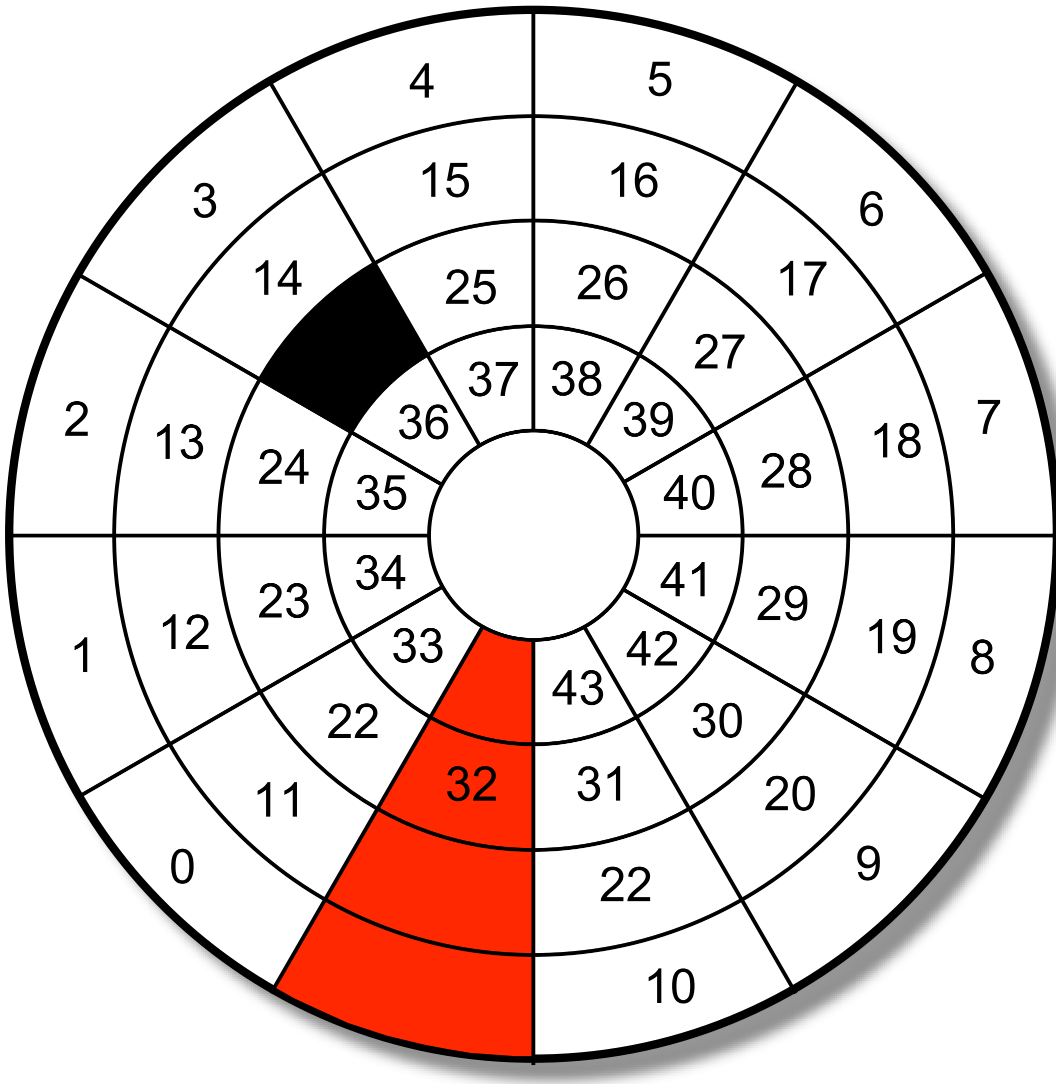
One spare sector per track



Remapping from defective sector to spare



LBN mapping slipped past defective sector



Some Real Defect Management Schemes

- High level facts
  - percentage of space: < 1%
  - always slip if possible
    - much more efficient for streaming data

- One real scheme: Seagate Cheetah 4LP
  - 108 spare sectors every 12 cylinders
    - located on the last track of the 12-cylinder group
    - used only for remapped sectors grown during usage
  - many spare sectors on innermost cylinders
    - used to provide backstop for all slipped sectors

Computing physical location from LBN

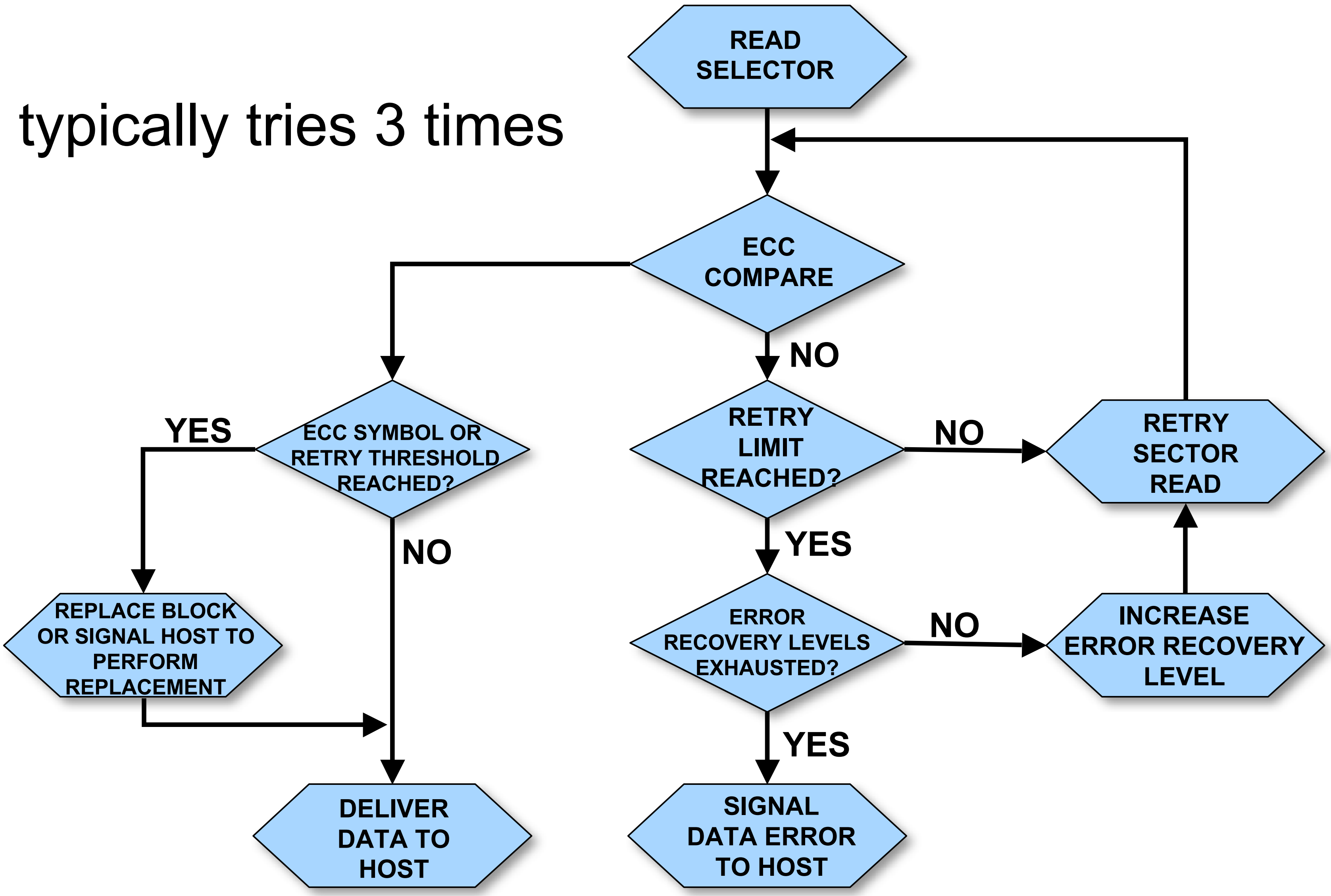
- First, check list of remapped LBNs
  - usually identifies exact physical location of replacement
- If no match, do the steps from before
  - but, also account for slipped sectors that affect desired LBN
- About 10 different management schemes
  - For any given scheme, the computations can be fairly straightforward. However, it is quite complex to discuss them all at once concretely

When defects "grow" during operation

- First, try ECC
  - it can recover from many problems
- Next, try to read the sector again
  - often, failure to read the sector is transient
  - cost is a full rotation added to access time
- Last resort, report failure and remap sector
  - this means that the stored data has been lost
  - until next write to this LBN, reads get error response
    - new data allows the location change to take effect

Error Recovery Algorithm for READs

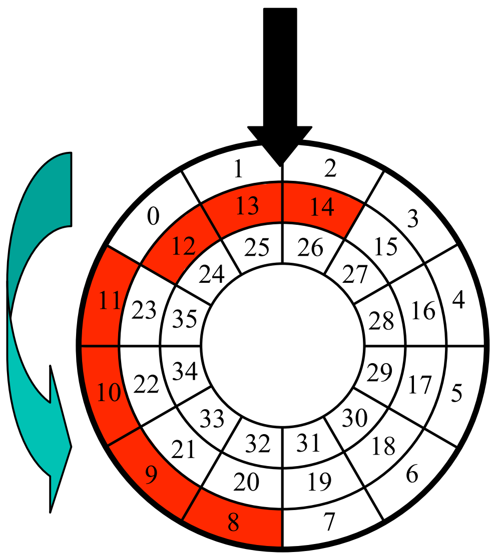
typically tries 3 times



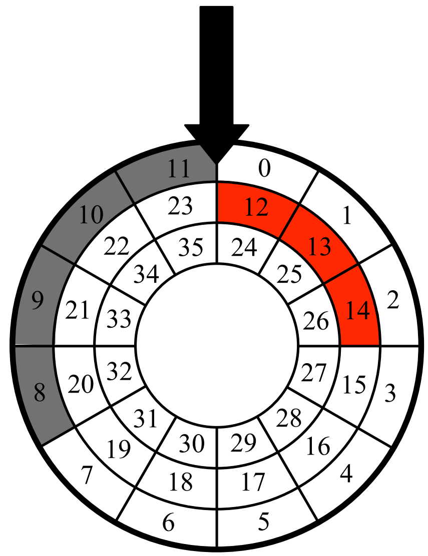
Third Complication: Skew

- Switching from one track to another takes time
  - sequential transfers would suffer full rotation
- Solution: skew
  - offset physical location of first sector to avoid extra rotation
    - selection of skew value made from switch time statistics
- Track skew
  - for when switching to next surface within a cylinder
- Cylinder skew
  - for when switching from last surface of one cylinder to first surface of next cylinder

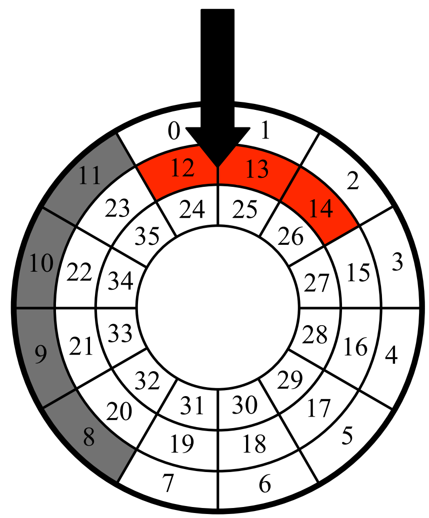
What happens to requests that span tracks?



Request spans 2 tracks

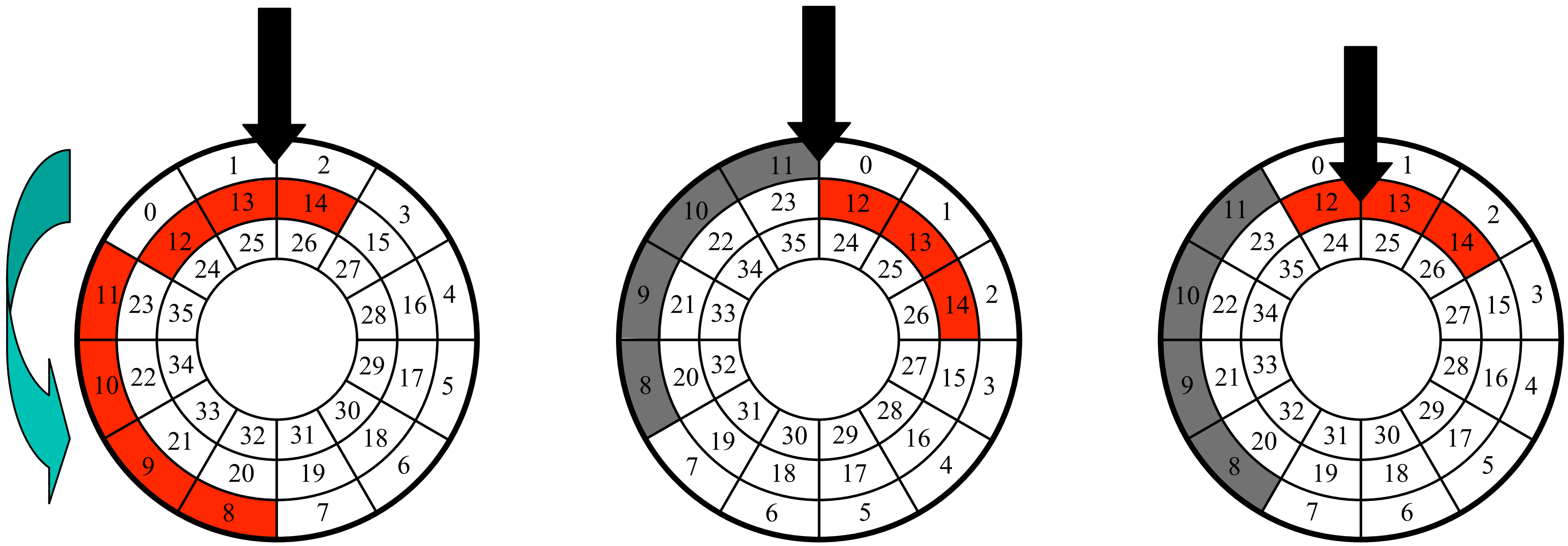


After reading first part



After track switch

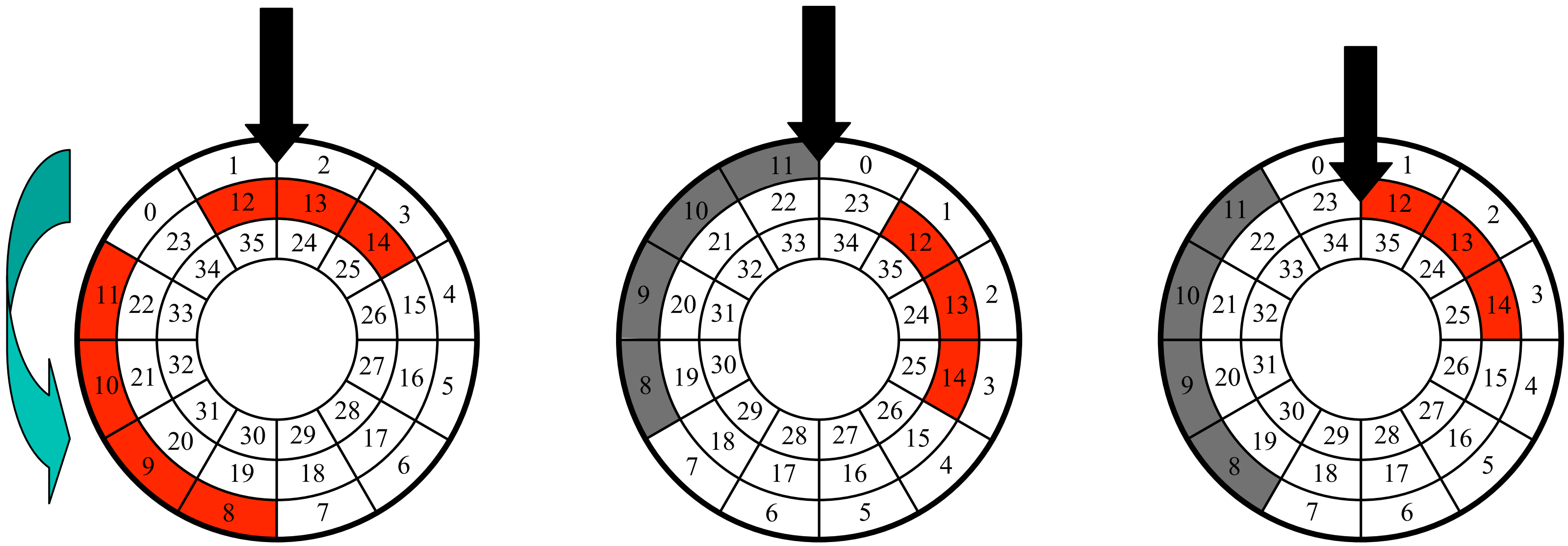
What happens to requests that span tracks?



Request spans 2 tracks | After reading first part | After track switch

Sector 12 rotates past during track switch, so full rotation needed

Same request with track skew of one sector



Request spans 2 tracks | After reading first part | After track switch

Track skew prevents unnecessary rotation

Examples of Track and Cylinder Skews

	Quantum Atlas 10k			IBM Ultrastar 18ES		
Zone \ Skew	SPT	Track	Cylinder	SPT	Track	Cylinder
1	334	64	101	390	58	102
2	324	62	98	374	56	97
3	306	56	93	364	55	95

Computing Physical Location from LBN

- Figure out cylno, surfaceno, and sectno
  - using algorithms indicated previously

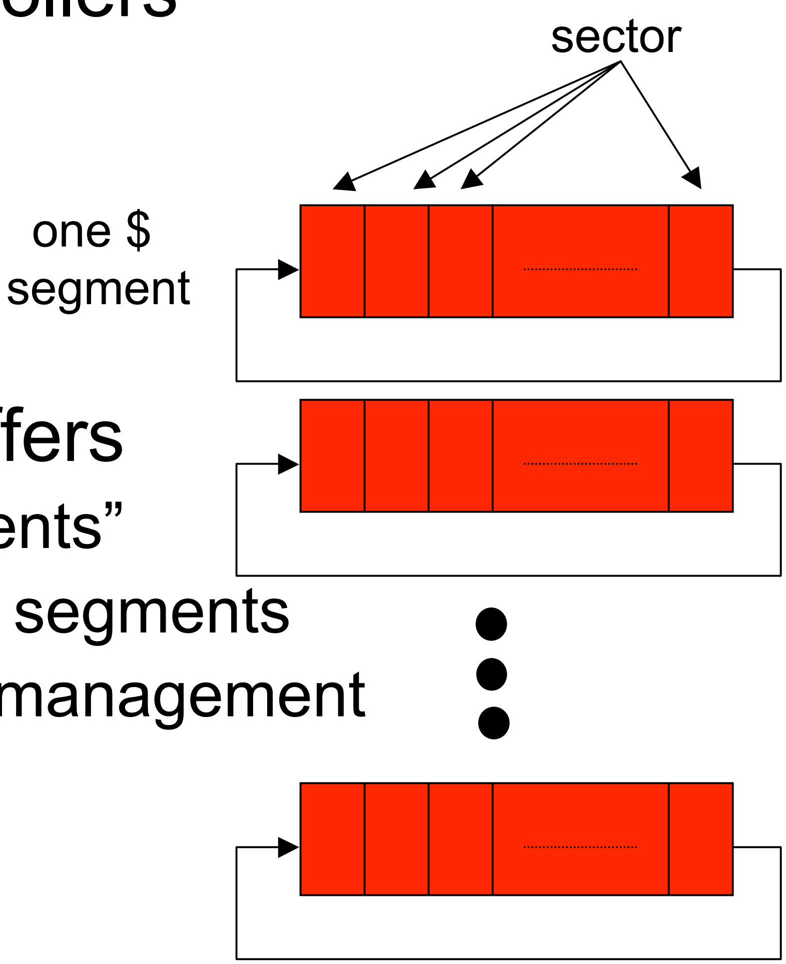
- Compute total skew for first mapped physical sector on this track
  - totalskew = (cylno * cylskew) +
    (surfaceno + (cylno * (surfaces-1)) * trackskew)

- Compute rotational offset on given track
  - offset = (totalskew + sectno) % sectspertrack

Basic On-disk Caching

On-disk RAM

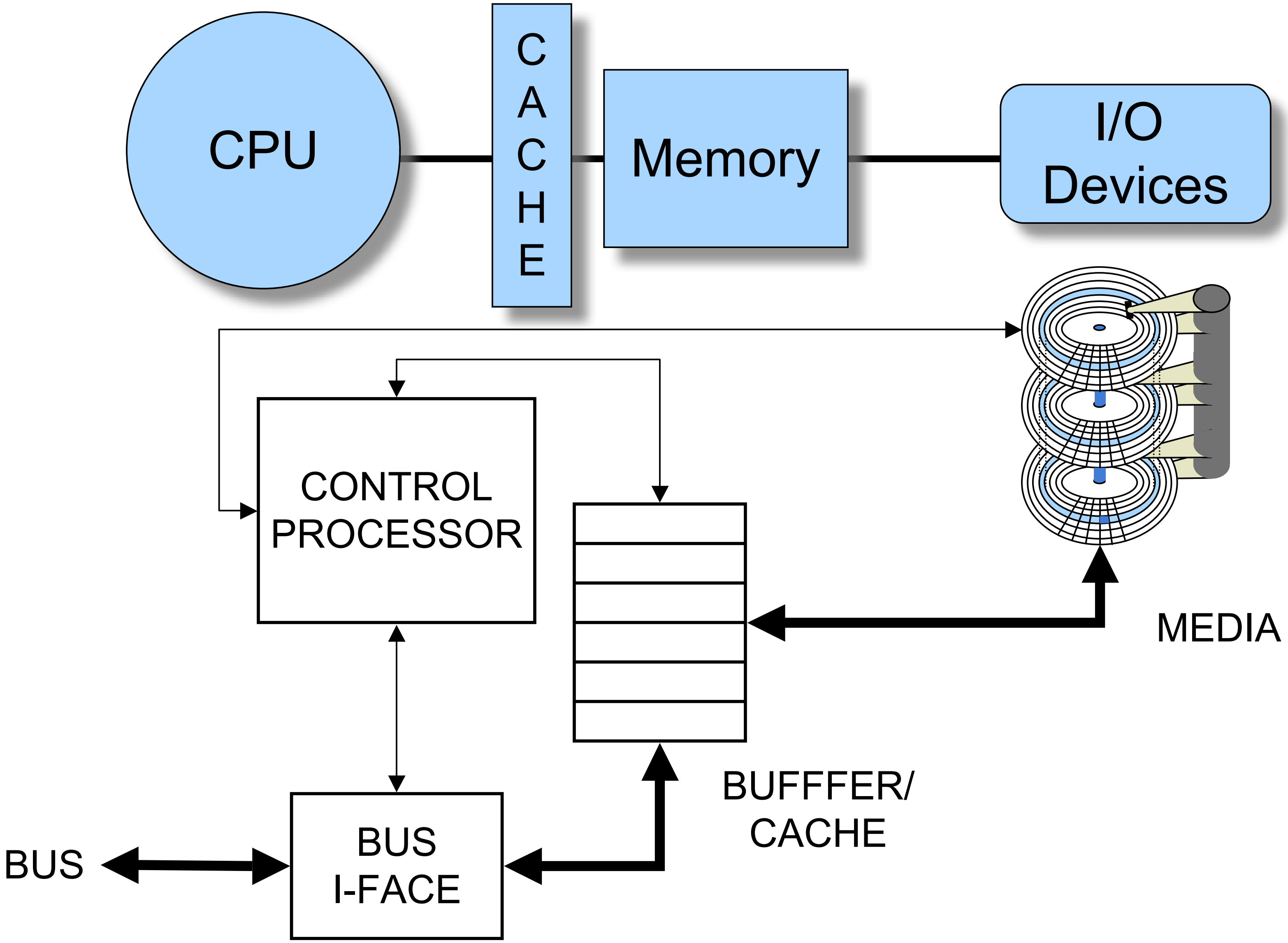
- RAM on disk drive controllers
  - firmware
  - speed matching buffer
  - prefetching buffer
  - cache
- Canonical disk drive buffers
  - several fixed-size "segments"
  - latest thing: variable-size segments
  - down the road: OS style management



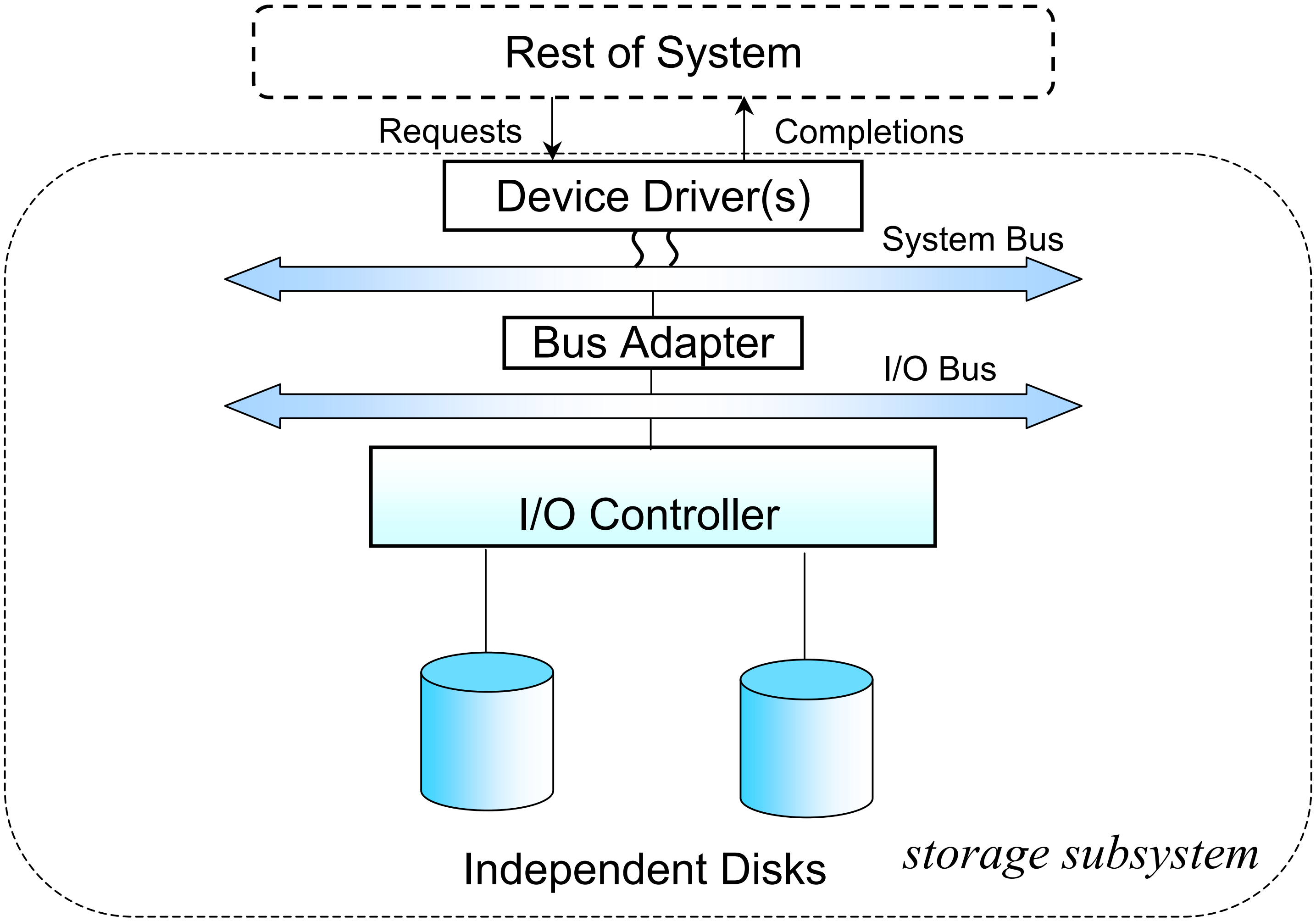
Prefetching and Caching

- Prefetching
  - sequential prefetch essentially free until next request arrives
    - and until track boundary
  - Note: physically sequential sectors are prefetched
    - usefulness depends on access patterns
  - Example algorithms
    - prefetch until buffer is full or next request arrives
    - MIN and MAX values for prefetching
    - if track `n-1` and `n` have been READ, prefetch track `n+1`
- Caching
  - data in buffer segments retained as cache
  - most of the benefit comes from prefetching

Disk Drive – Complete System?



Not really, recall this…



File Systems

Key FS design issues

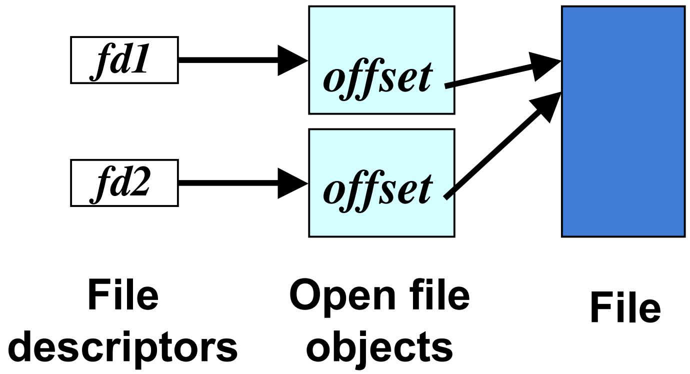
- Application interface and system software
- Data organization and naming
- On-disk data placement
- Cache management
- Metadata integrity and crash recovery
- Access control

Starting at the top: what applications see

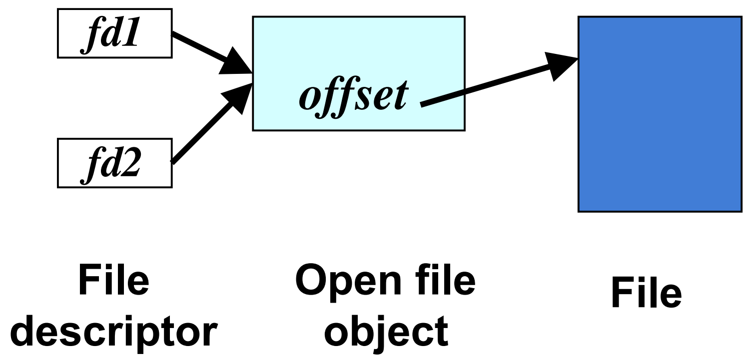
- At the highest level (in most systems)
  - contents of a file: sequence of bytes
  - most basic operations: *open, close, read, write*
- *open* starts a "session" and returns a "handle"
  - in POSIX (e.g., Linux and various UNIXes)
    - – handle is process-specific integer called "file descriptor"
    - – session remembers current offset into file
    - – for local files, session also postpones full file deletion
  - handle is provided with each subsequent operation
- *read* or *write* access bytes at some offset in file
    - – could be explicitly provided or remembered by session
- *close* ends session and destroys the handle

Sidebar: shared and private sessions

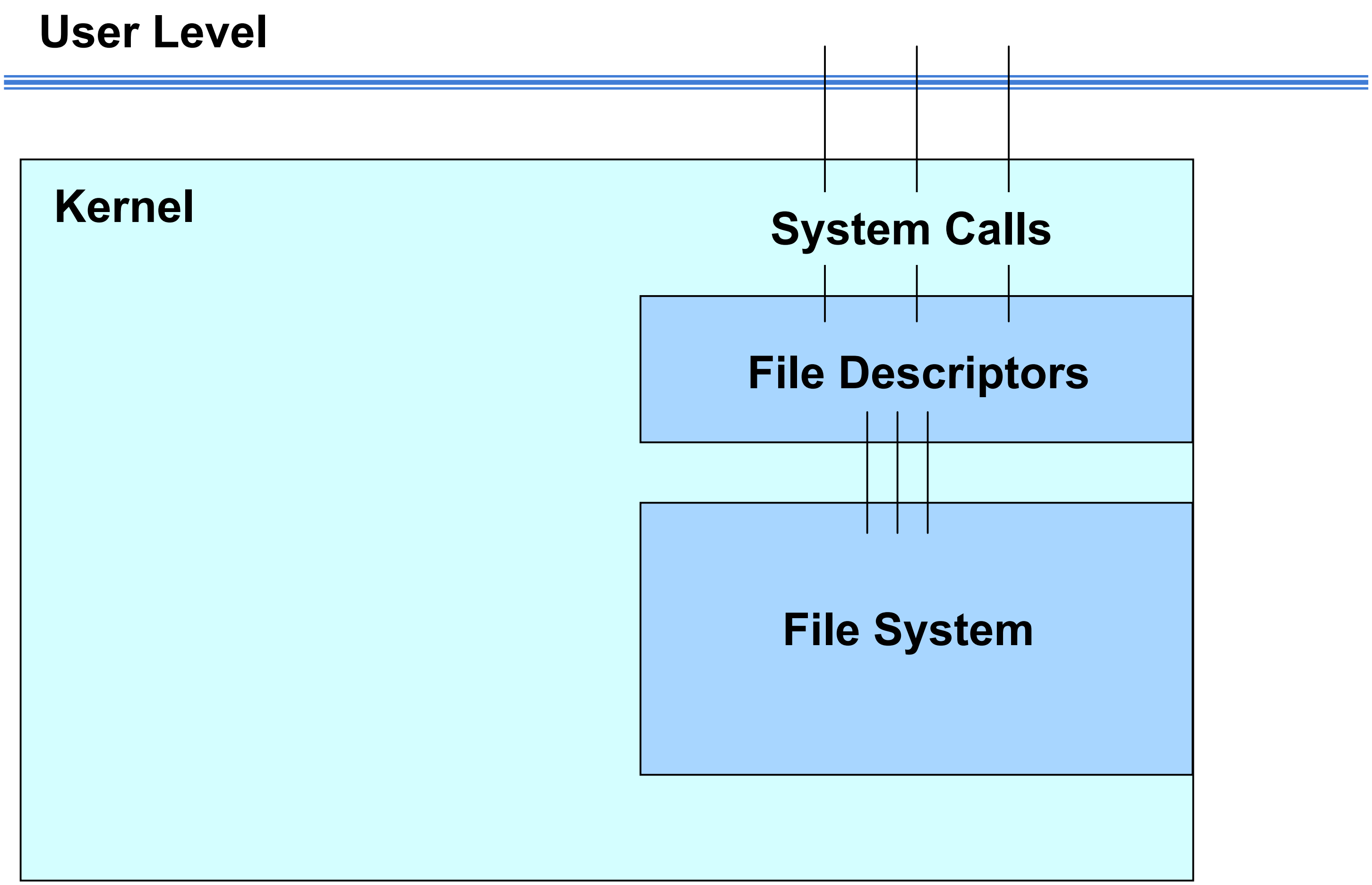
- Two *open*s of the same file yield independent sessions



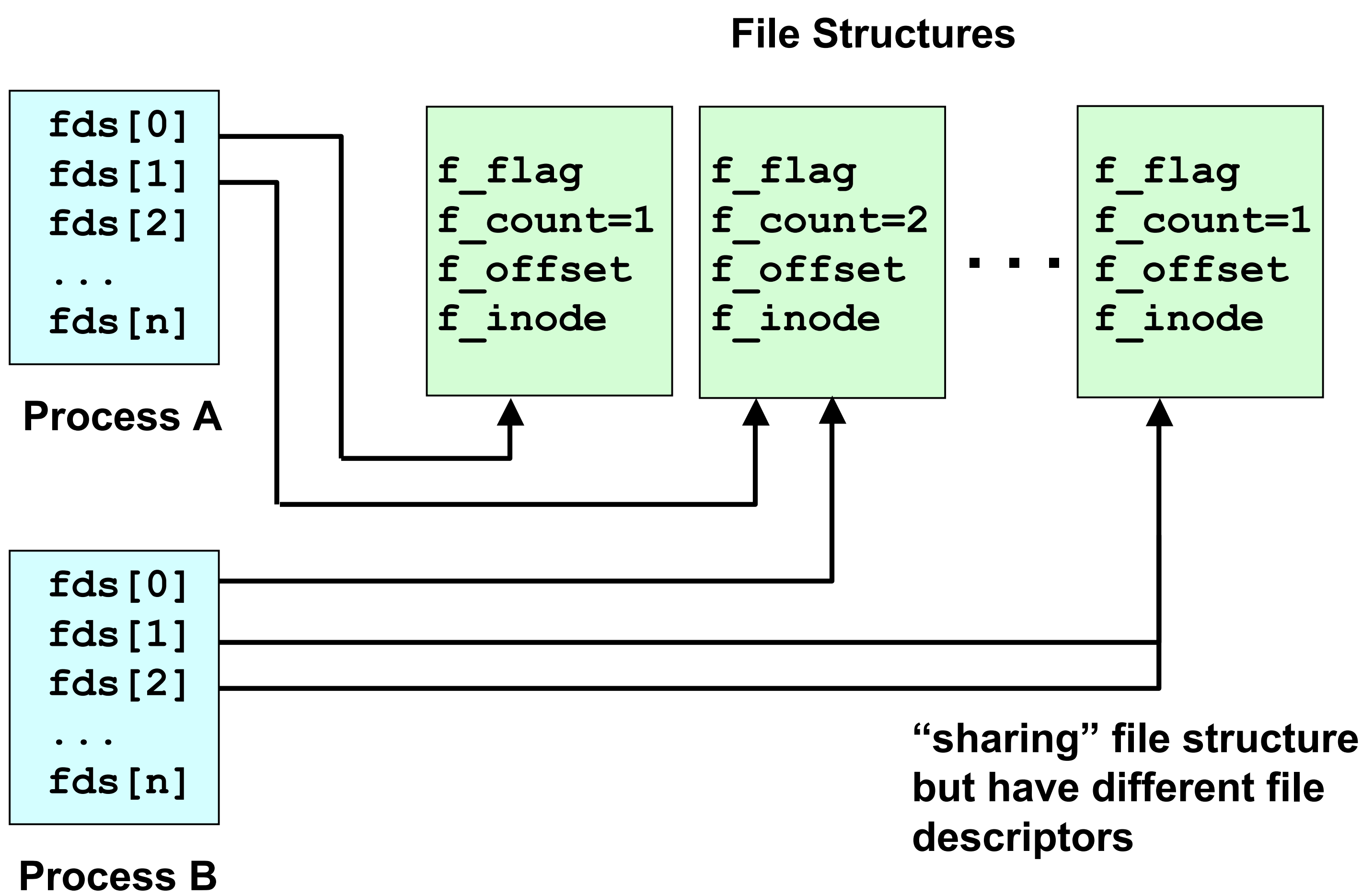
- A session can be shared across handles or processes



Where information resides



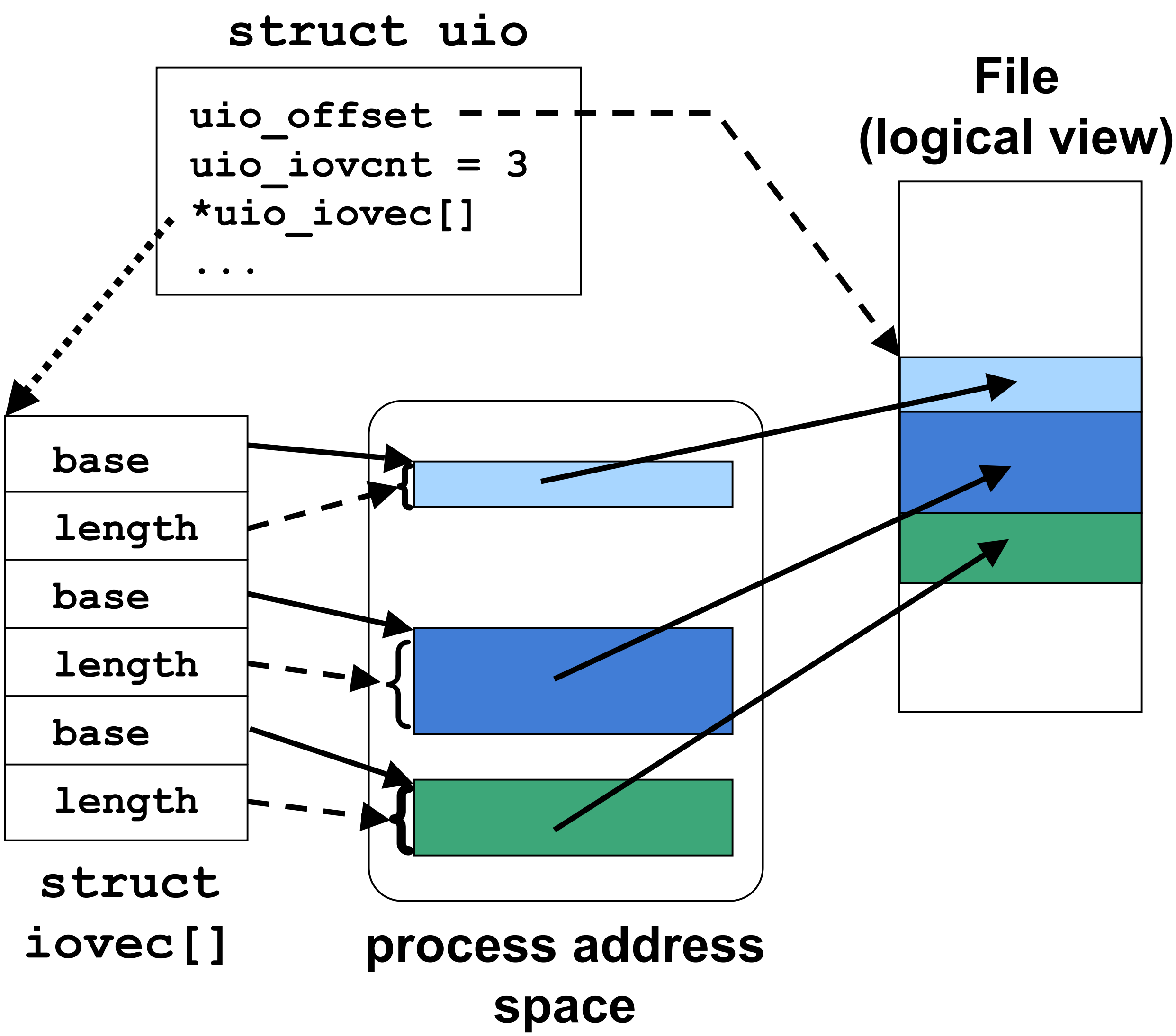
Some associated structures in kernel



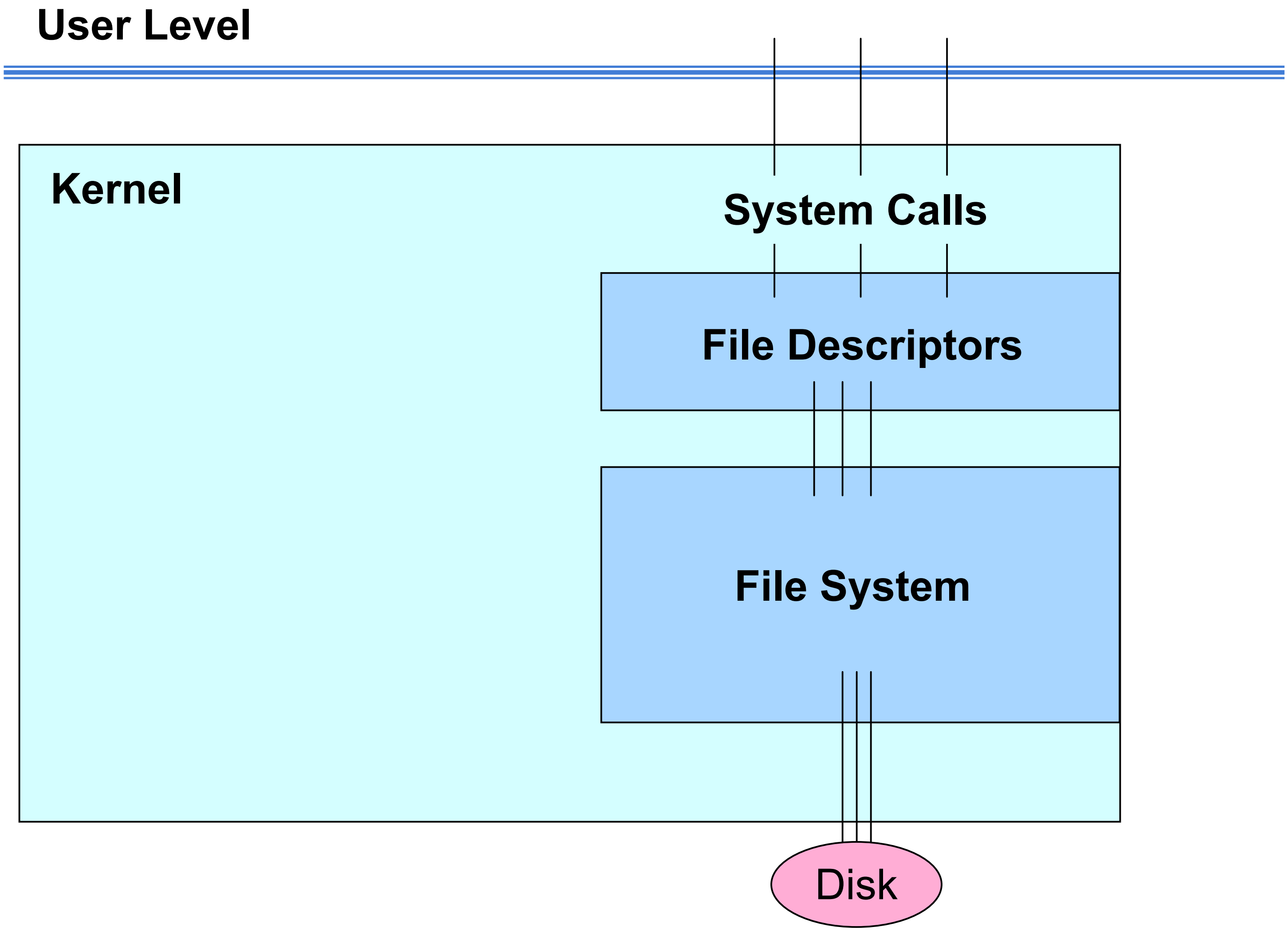
Moving data between kernel and application

- Common approach: copy it
  - as in *read*(fd, buffer, size) or *write*(fd, buffer, size)
  - simplifies things in two ways
    - application knows it can use the memory immediately
      - and that the corresponding data is in that memory
    - kernel has no hidden coordination with application
      - e.g., later changes to buffer do not silently change file
- Sometimes better approach: hand it off
  - as in char *buffer = *read*(fd, size)
    - notice that buffer containing data is returned
      - this allows page swapping (via VM) rather than copying
  - downsides
    - sometimes not much of a performance improvement
    - makes file caching more difficult
    - can be confusing for application writers

The *uio* structure for scatter/gather I/O

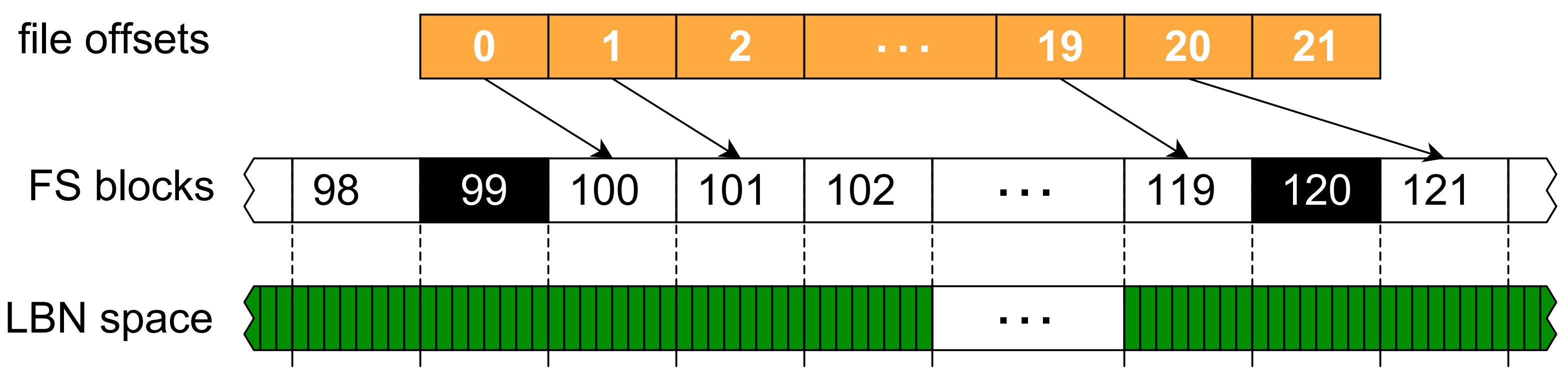


Remember that the FS data lives on disk



From file offsets to LBNs

- File offsets
  - `0` to `num_blocks_in_file`
    - offset to a file given in block number
- File System blocks
  - `0` to `num_blocks_in_filesystem`
  - Single block may span multiple disk LBNs

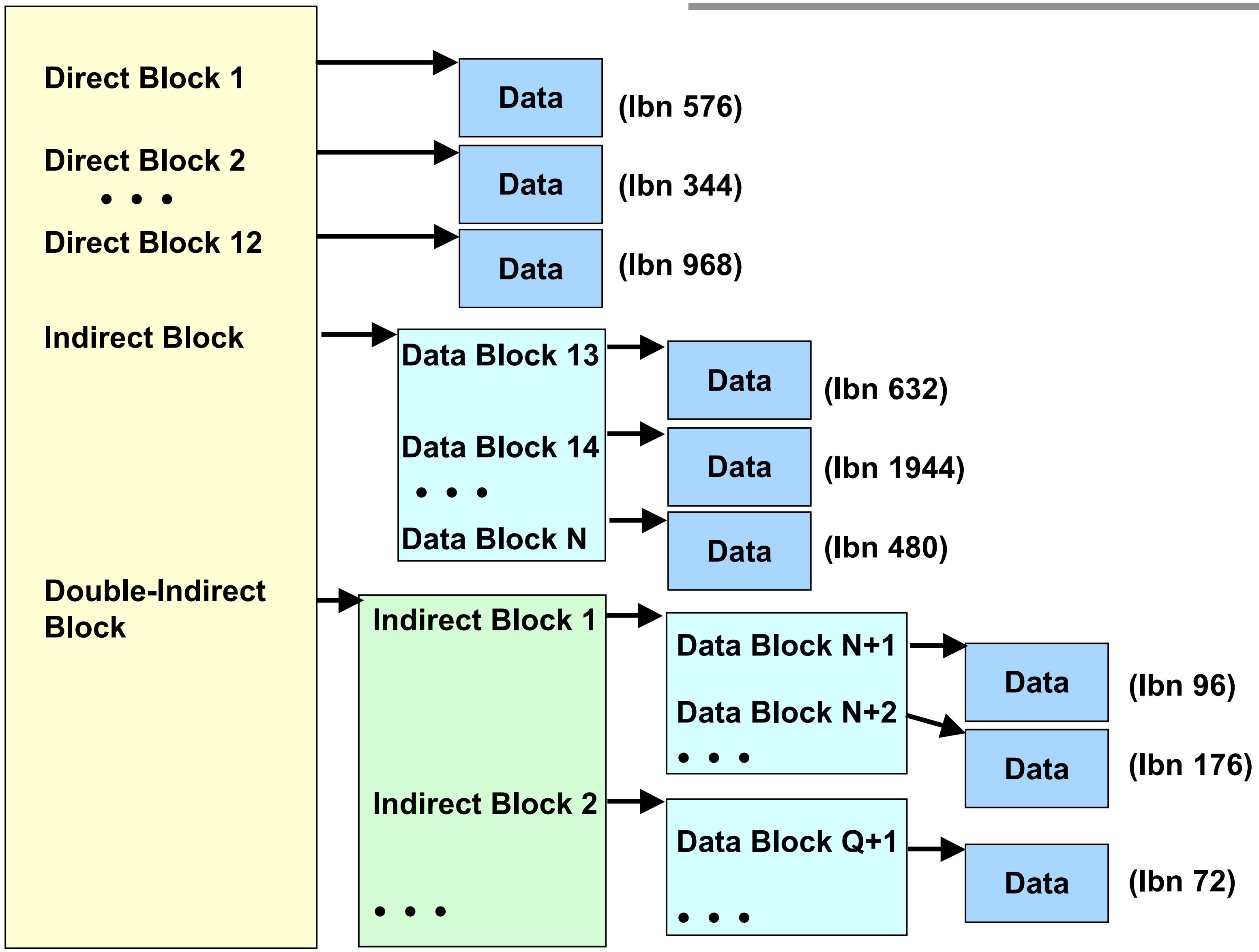


FS block size = 8 LBNs (4KB)

Mapping file offsets to disk LBNs

- Issue in question
  - must know which LBNs hold which file's data
- Trivial mapping: just remember start location
  - then keep entire file in contiguous LBNs
    - what happens when it grows?
  - alternately, include a "next pointer" in each "block"
    - how does one find location of a particular offset?
- Most common approach: block lists
  - an array with one LBN per block in the file
  - Note: file block size can exceed one logical block
    - file system treats groups of logical blocks as a unit

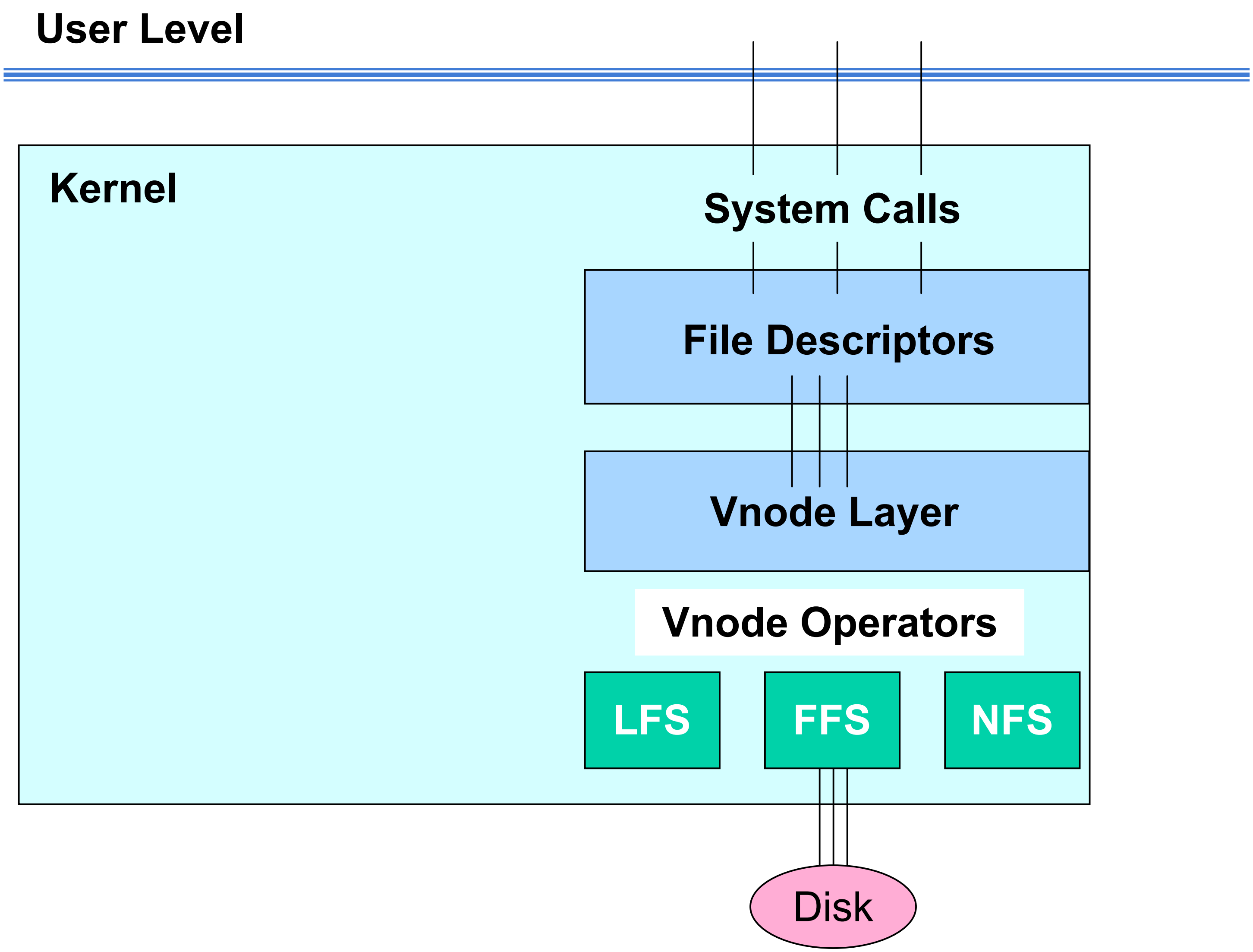
A common approach to recording a block list



Inodes

- FS stores other per-file information as well
  - length of file
  - owner
  - access permissions
  - last modification time
  - …
- Usually kept together with the block list

Supporting multiple file system types

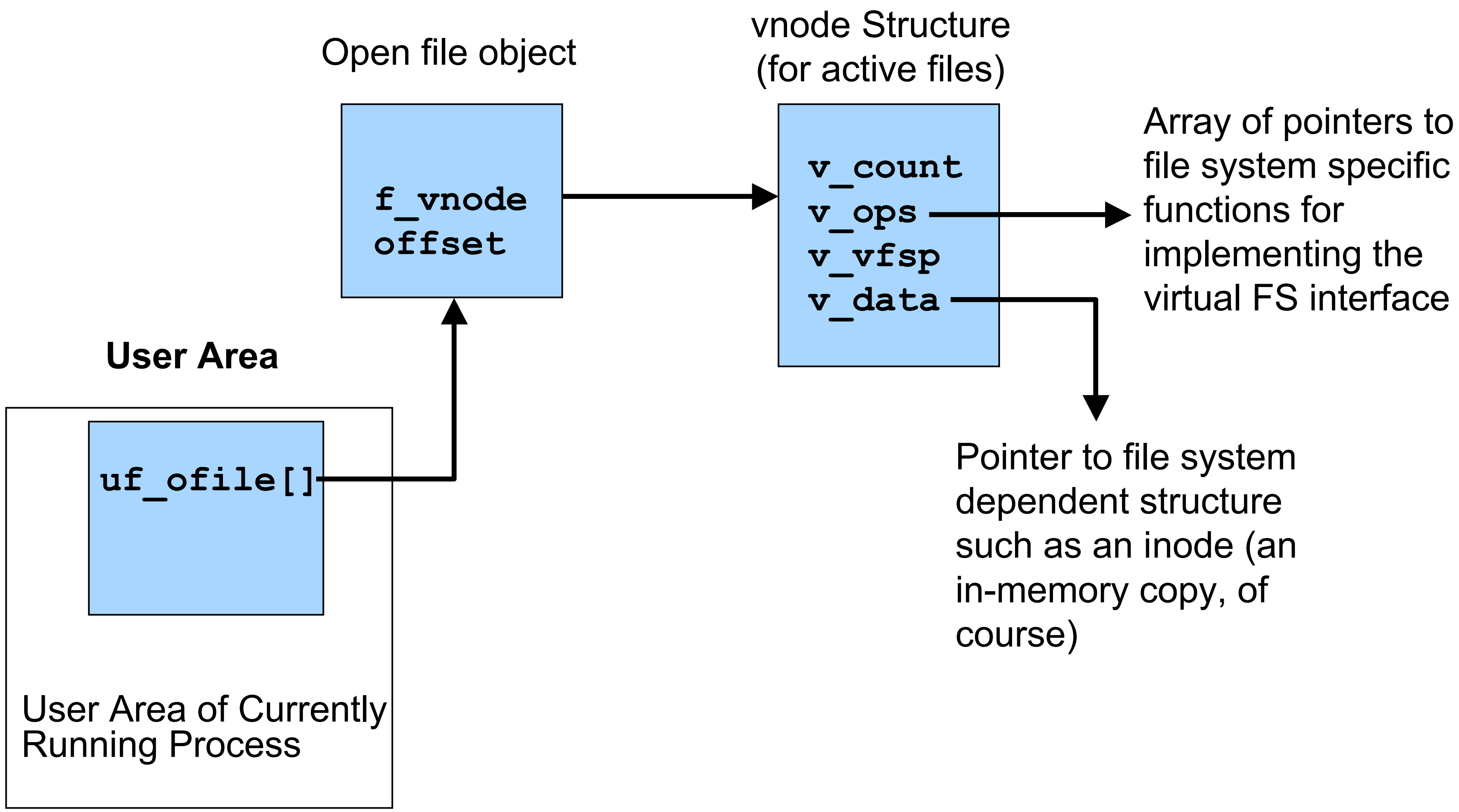


Vnode layer: inside kernel

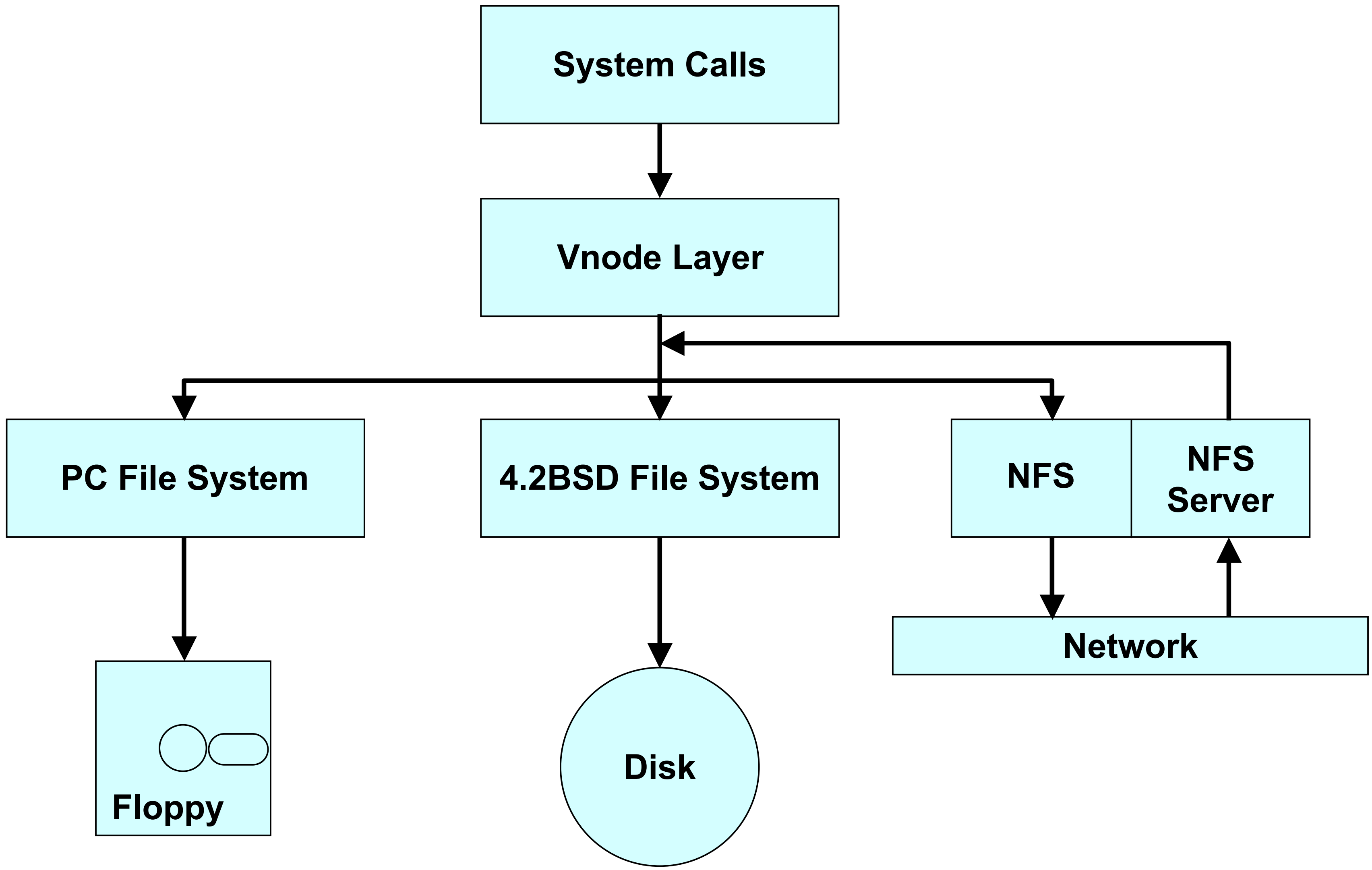
- Want to have multiple file systems at once
  - and possibly of differing types
- Solution: virtual file system layer
  - adding level of indirection always seems to help…

- Everything in kernel interacts with FS via a virtualized layer of functions
  - these function calls are routed to the appropriate FS-specific implementations
    - once the correct FStype has been identified

Open file object points to a vnode



Can also support non-disk FS



Key FS design issues

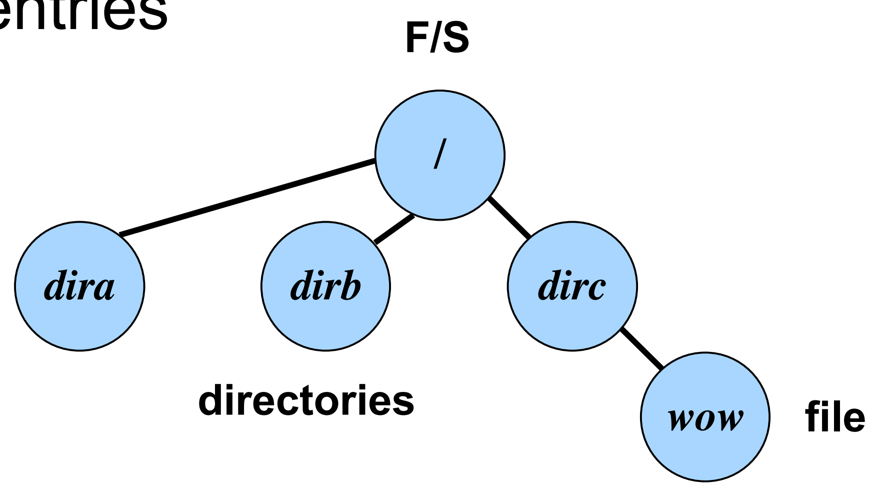
- Application interface and system software
- Data organization and naming
- On-disk data placement
- Cache management
- Metadata integrity and crash recovery
- Access control

What makes this so important?

- One of the biggest problems, looking ahead
  - with TBs of data, how does one organize things
  - how to ensure we can find what we want later?
- Not nearly as easy as it seems
  - try to find some old piece of paper sometime
    - e.g., your exam #2 from Calculus 3
  - think ahead to when you're a lot busier…

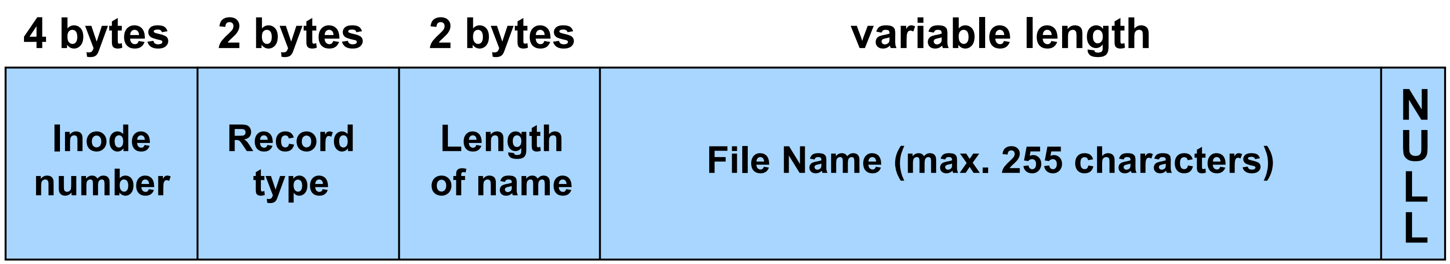
Common approach: directory hierarchy

- Hierarchies are good to deal with complexity
  - … and data organization is a complex problem
- It works well for moderate-sized data sets
  - easy to identify course breakdowns
  - when it gets too big, split it and refine namespace
- Traversing the directory hierarchy
  - the '.' and '..' entries

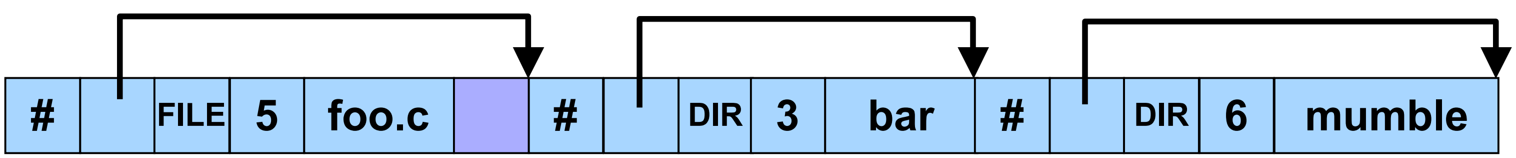


What's in a directory

- Directories to translate file names to inode IDs
  - just special file with entries formatted in some way



  - often sets of entries put in sector-sized chunks

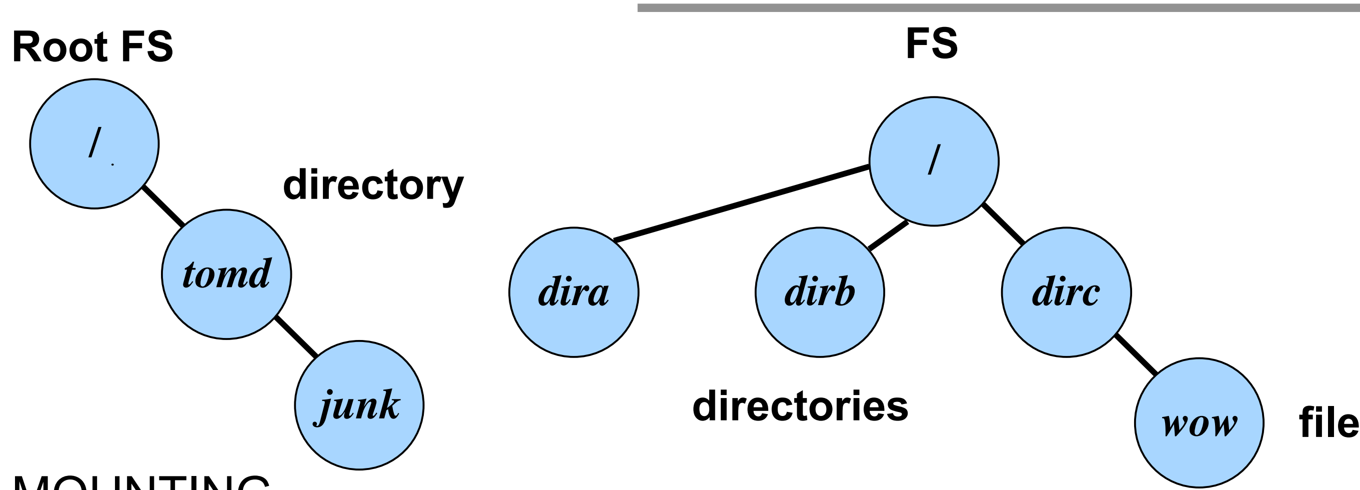


A directory block with three entries

Managing namespace: mount/unmount

- One can have many FSs on many devices
  - … but only one namespace
- So, one must combine the FSs into one namespace
  - starts with a “root file system”
    - the one that has to be there when the system boots
  - “mount” operation attaches one FS into the namespace
    - at a specific point in the overall namespace
  - “unmount” detaches a previously-attached file system

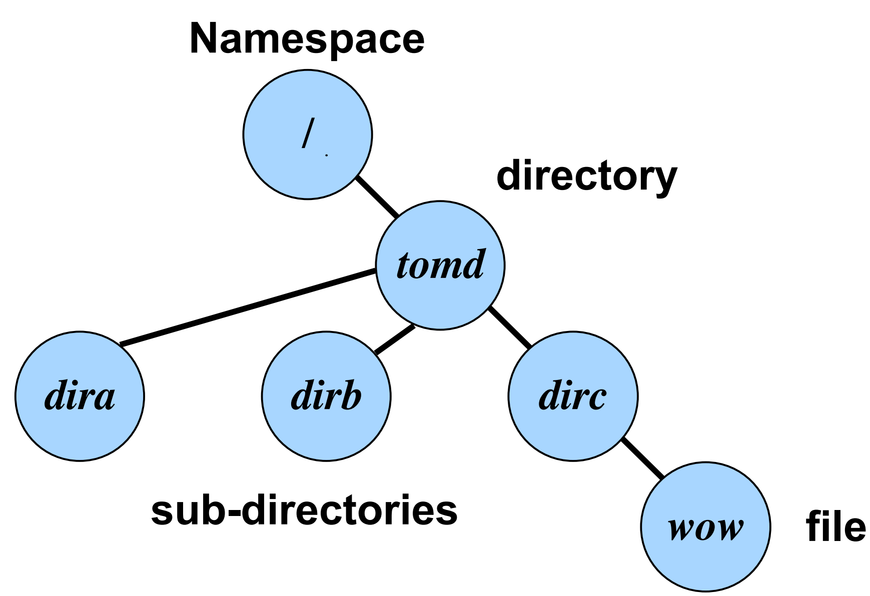
Mounting an FS



VIEW BEFORE MOUNTING

VIEW AFTER MOUNTING

`# mount FS /tomd`



How to find the root directory?

- Need enough information to find key structures
  - allocation structures
  - inode for root directory
  - any other defining information
- Common approach
  - use predetermined locations within file system
    - known locations of (copies of) superblocks
- Alternate approach
  - some external record

Sidebar: multiple FSs on one disk

- How is this possible?
  - divide capacity into multiple "partitions"
- How are the partitions remembered?
  - commonly, via a "partition map" at the 2nd LBN
  - each partition map entry specifies
    - start LBN for partition
    - length of partition (in logical blocks)
- Usually device drivers handle partition map
  - file system requests are relative to their partition
  - device driver shifts these requests relative to partition start

Difficulty with directory hierarchies

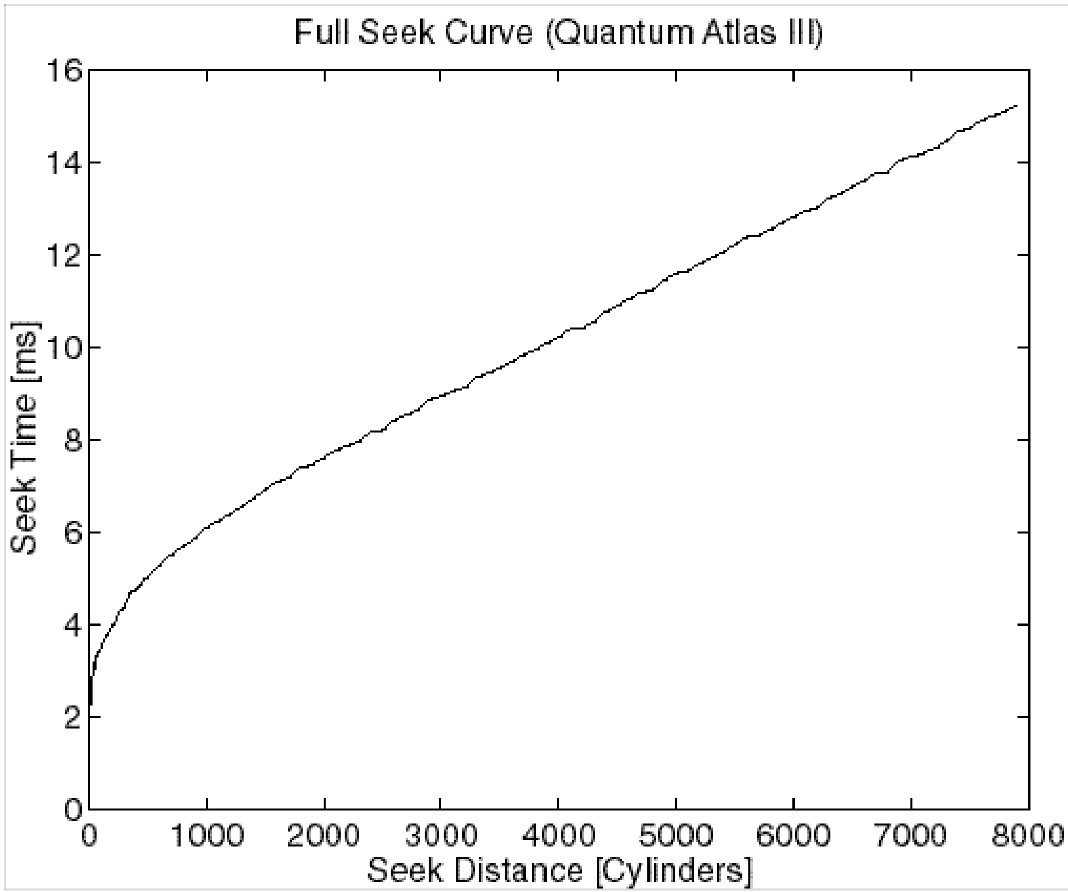
- Can be very difficult to scale to large sizes
  - eventually, the refinements become too fine
  - and they tend to be less distinct
- Problem: what happens when number of entries in directory grows too large??
  - think about having to read through all of those entries
  - possible solution: partition into subdirectories again
- Problem: what happens when data objects could fit into any of several subdirectories??
  - think about having to find something specific
  - possible solution: multiple names for such files

On-disk Data Placement

Key FS design issues

- Application interface and system software
- Data organization and naming
- On-disk data placement
- Cache management
- Metadata integrity and crash recovery
- Access control

Fact – seek time depends on distance

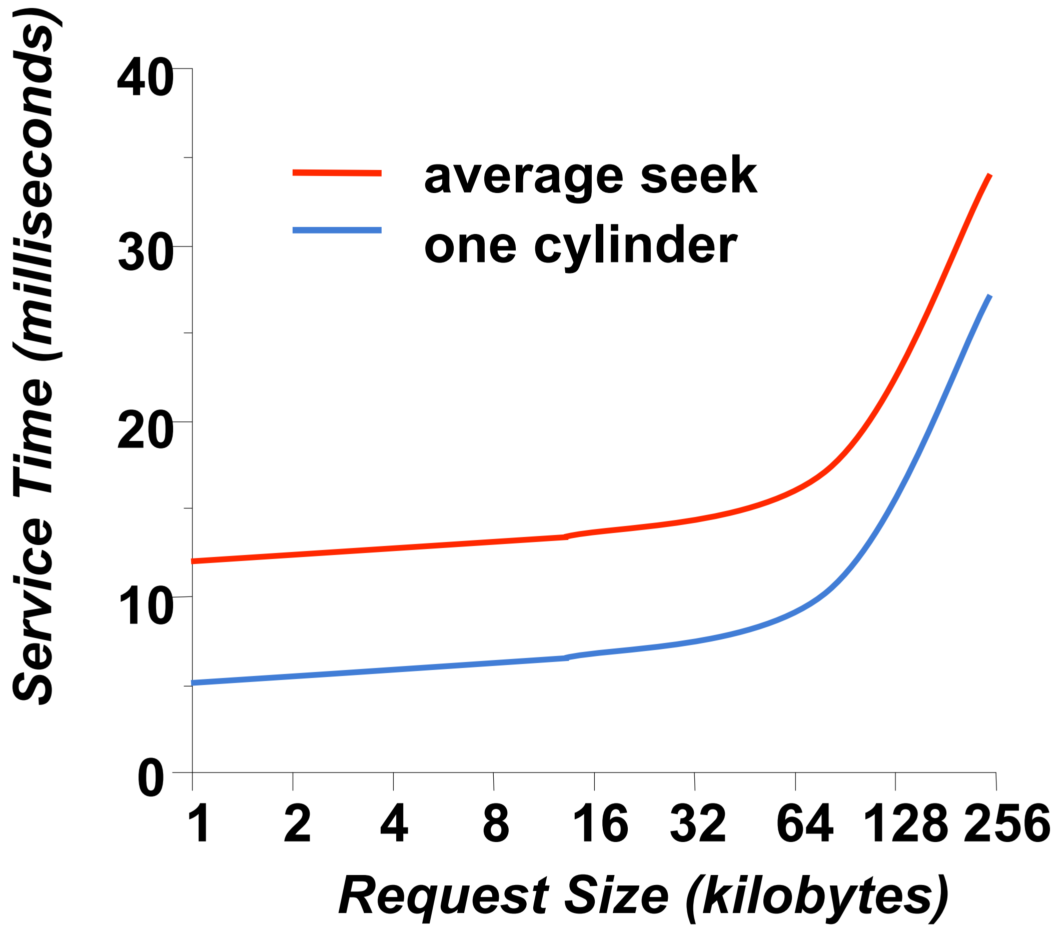


Quantum Atlas III	
year	1998
seek	7.8 ms
single track	0.80 ms
full stroke seek	15 ms

Seagate Cheetah 15K.3	
year	2002
seek	3.6 ms
single track	0.20 ms
full stroke seek	6.5 ms

Goal – requests in sequence physically near one another

Fact –positioning time dominates transfer

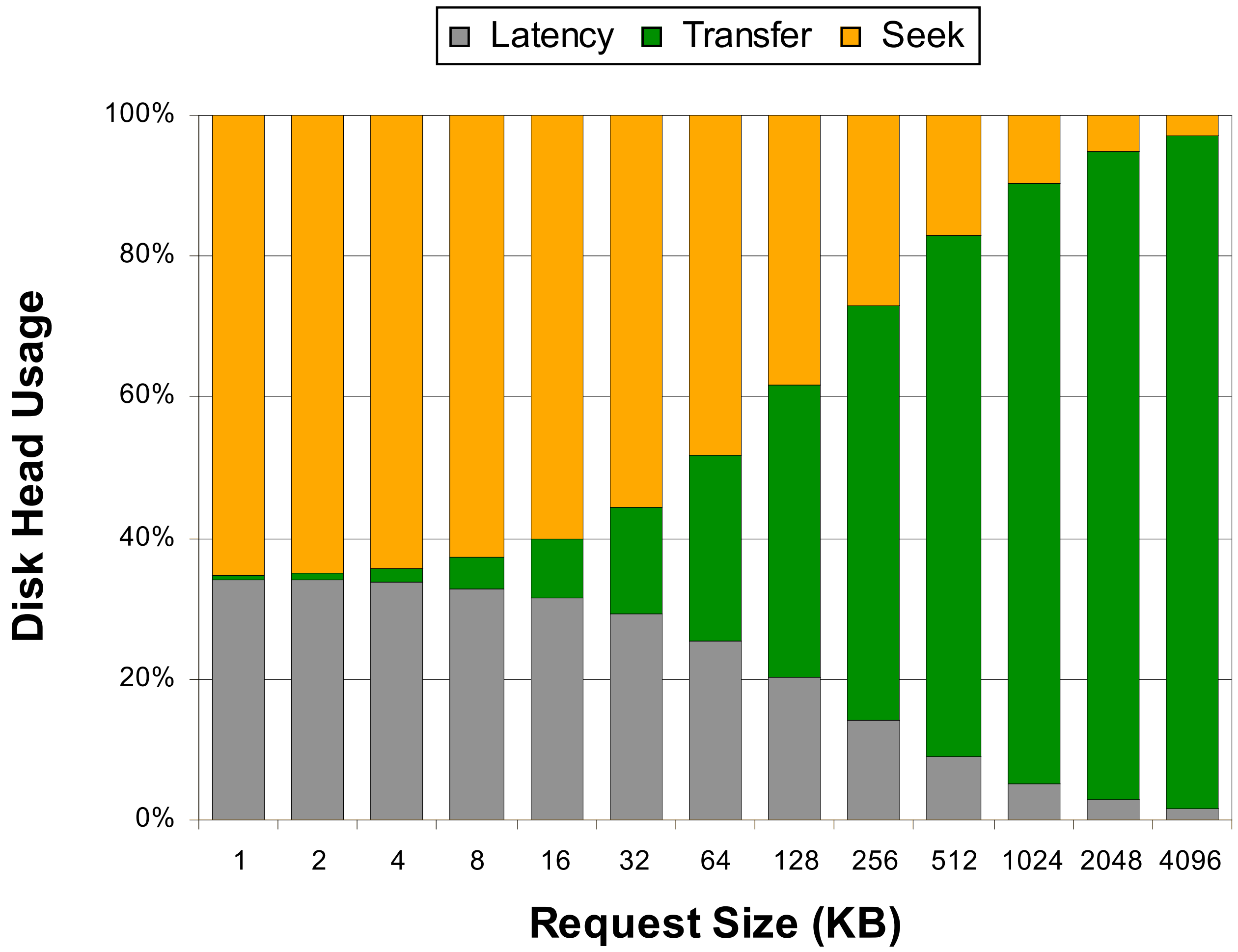


Seagate Cheetah 15K.3	
model	ST373453
capacity	73.4 GB
read disk (seq)	< 20 min
read disk (2KB)	> 30 hours
read disk (128KB)	< 1 hour

time to read the entire disk, sequentially or randomly with varying request size

Goal – fewer, larger requests to amortize positioning costs

Breakdown of disk head time



File System Allocation

- Two issues
  - Keep track of which space is available
  - Pick unused blocks for new data
- Simplest solution – free list
  - maintain a linked list of free blocks
    - using space in unused blocks to store the pointers
  - grab block from this list when new block is needed
    - usually, the list is used as a stack

  - While simple, this approach rarely yields good performance

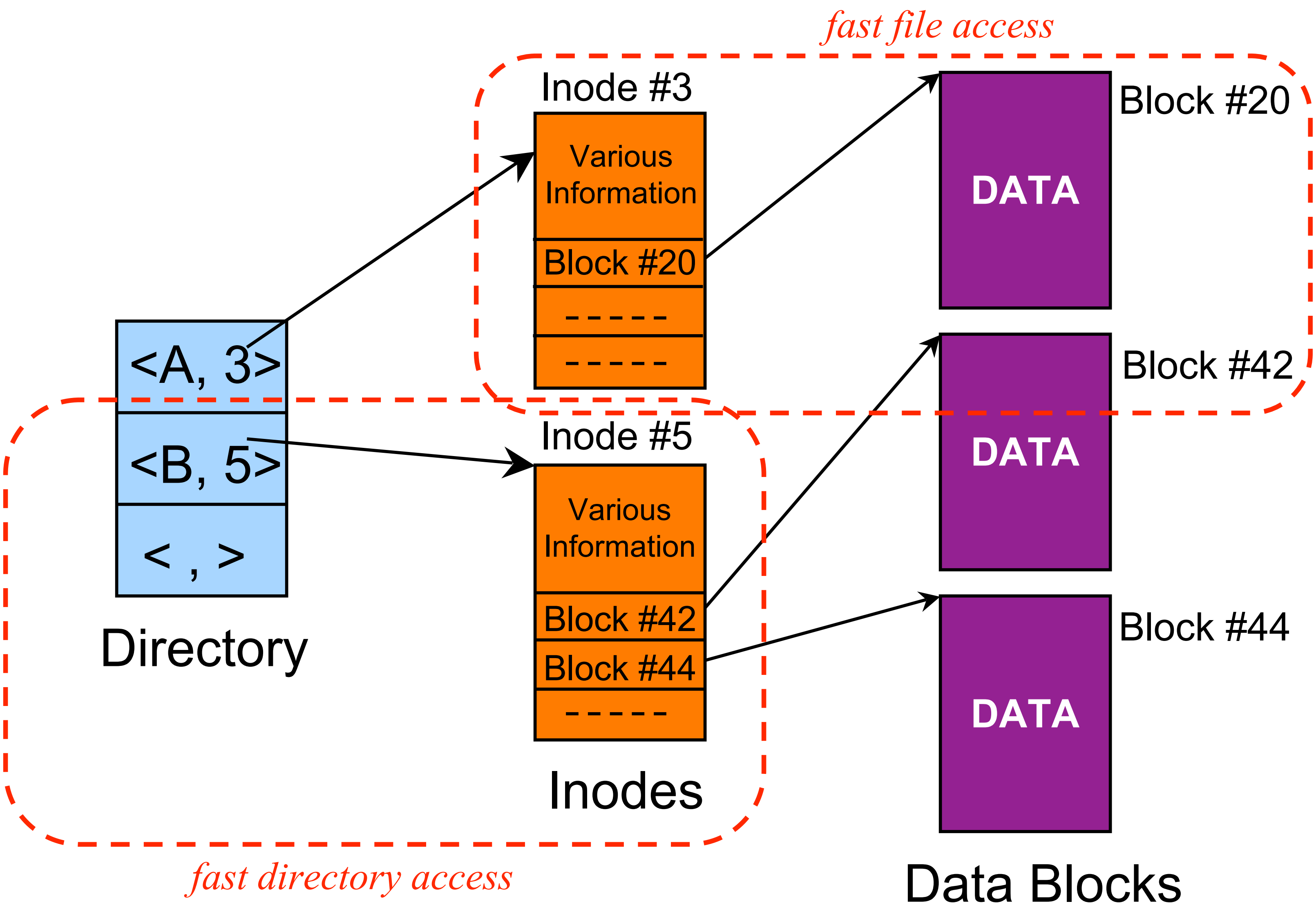
File System Allocation (cont.)

- Most common approach – a bitmap
  - large array of bits, with one bit per allocation unit
    - one value says “free” and the other says “in use”
  - Scan the array when a new block is needed
    - we don’t have to just take first “free” block in the array
    - we can look in particular regions
    - we can look for particular patterns
- Even better way (in some cases) – list of free extents
  - maintain a sorted list of “free” extents of space
    - each extent holds a contiguous range of free space
  - pull space from a part of a specific free extent
    - can start at a specific point
    - can look for a point with significant room for growth

File System Allocation – Summary

- FS performance (largely) dictated by disk performance
  - and optimization starts with allocation algorithms
  - as always, there are exceptions to this rule
- Two technology drivers yield two goals
  - Closeness (locality)
    - reduce seeks by putting related things close to each other
    - generally, benefits can be in the 2x range
  - Amortization (large transfers)
    - amortize each positioning delay by accessing lots of data
    - generally, benefits can reach into the 10x range

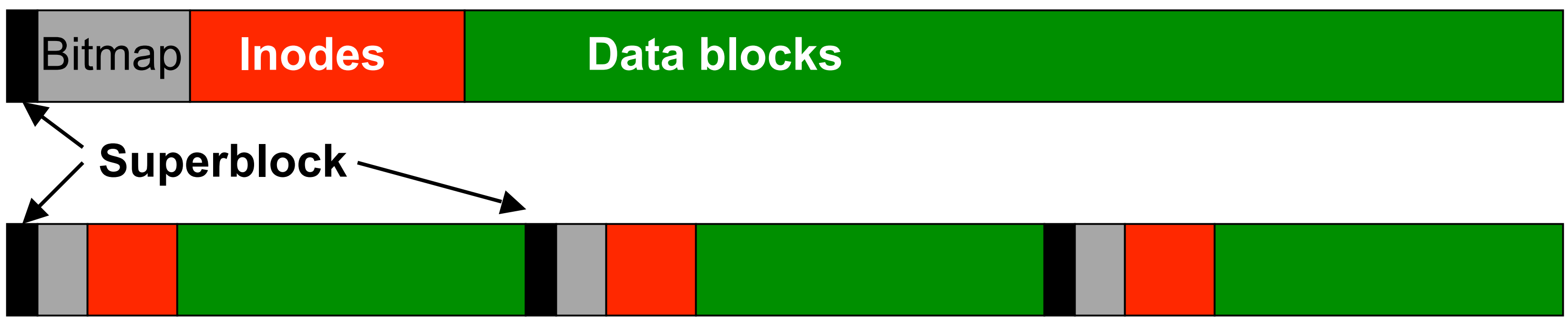
Spatial proximity can yield…



Fast File System (1984)

- Source of many still-popular optimizations
- For locality – cylinder groups
  - called *allocation groups* in many modern file systems

Default usage of LBN space



Organization of an *allocation group*

- allocate inode in cylgroup with directory
- allocate first data blocks in cylgroup with inode

Other ways of enhancing locality

- Disk request scheduling
  - for example, consider all dirty blocks in file cache
- Write anywhere
  - specifically, writing near the disk head [Wang99]
    - assumes space is free and the head's location is known
  - cool idea that nobody currently uses
- Same thing for reads
  - assumes multiple replicas on the disk
  - difficult to keep the metadata consistent

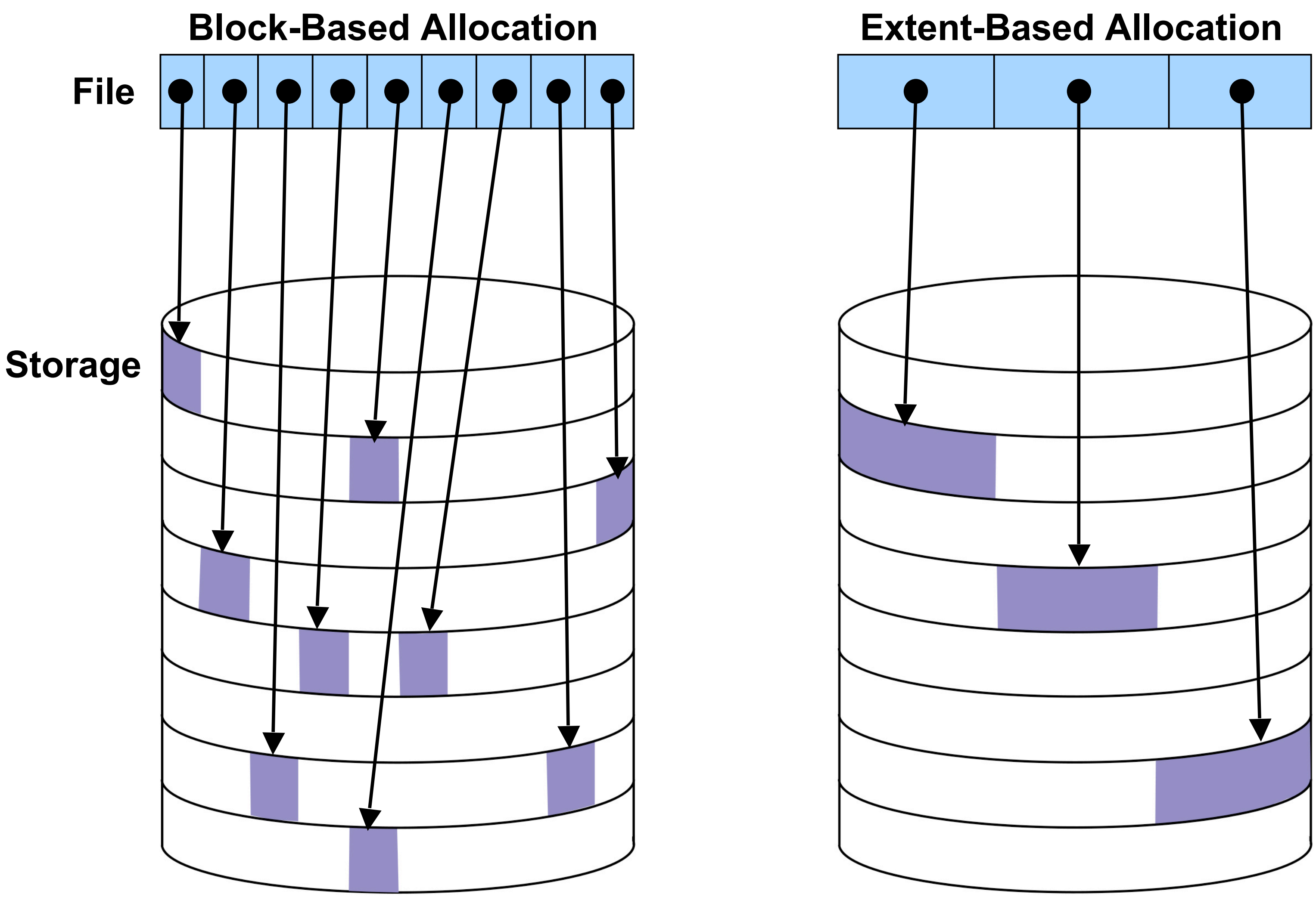
FFS schemes

- To get large transfers
  - larger block size
    - more data per disk read or write
    - use with fragments for small-file space efficiency
  - allocate next block after previous one, if possible
    - do this by starting search at block # just after previous
  - fetch more when sequential access detected
    - so, multiple blocks per seek+rotational latency

Other ways of getting large transfers

- Re-allocation
  - to re-establish sequential allocation when it was not feasible at the time of original allocation
  - Can you give an example?
- Pre-allocation
  - to avoid a failure to allocate sequentially later
- Extents (and extent-like)
  - as a replacement for block lists
  - as a replacement for bitmaps
  - things to consider
    - When does this help?
    - When does it hurt performance?

Block-based vs. Extent-based Allocation



Sidebar: BSD FFS constants

Parameter	*Meaning*
`MAXBPG`	max blocks per file in a cylinder group
`MAXCONTIG`	max contiguous blocks before *rotdelay* gap
`MINFREE`	min percentage of free space
`NSECT`	sectors per track
`ROTDELAY`	rotational delay between contiguous blocks
`RPS`	revs per second
`TRACKS`	tracks per cylinder
`TRACKSKEW`	track skew in sectors

- What their purpose?
- Historical prospective
  - details being pushed down

Object-based Access

OIDs

- Generation of unique ID
- Flat name space (no hierarchy)

- How to remember where things are?
  - Divide and conquer
  - Employ external applications/DATABASES

- Will discuss in the context of Centera

What's next…

- Lecture: 9/26
  - Database structures
  - DB Workloads