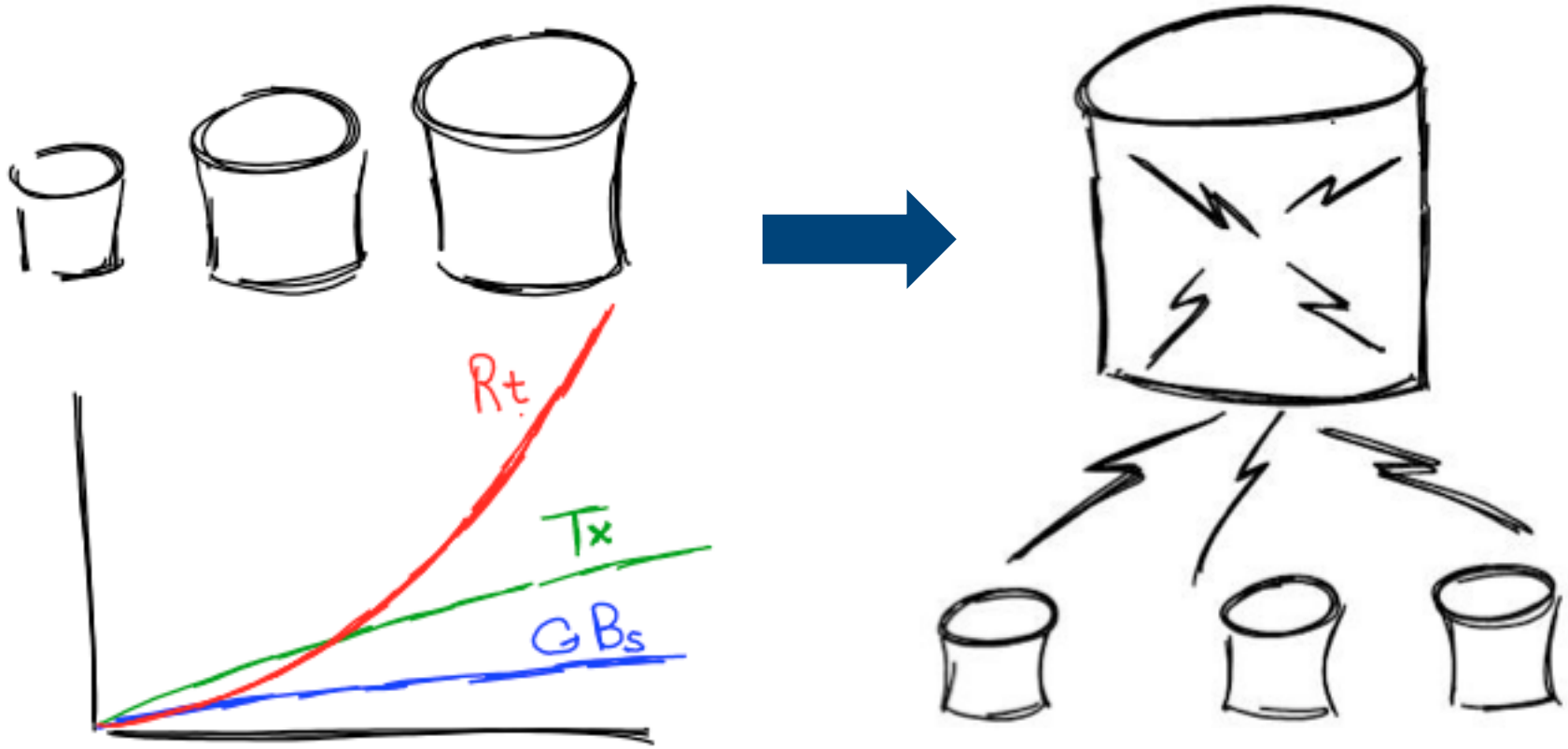
Duke Systems

Storage Etc.

Jeff Chase

Duke University



http://dbshards.com/dbshards/database-sharding-white-paper/

Block storage API

- Multiple storage objects: dynamic create/destroy
- Each object is a sequence of logical blocks
- Blocks are fixed-size
- Read/write whole blocks, or sequential ranges of blocks
- Storage address: object + logical block offset

How to allocate for objects on a disk?

How to map a storage address to a location on disk?

Example: AWS Simple Storage Service

Amazon S3

Create Free Account »

Amazon S3 is storage for the Internet. It is designed to make web-scale computing easier for developers.

Amazon S3 provides a simple web-services interface that can be used to store and retrieve any amount of data, at any time, from anywhere on the web. It gives any developer access to the same highly scalable, reliable, secure, fast, inexpensive infrastructure that Amazon uses to run its own global network of web sites. The service aims to maximize benefits of scale and to pass those benefits on to developers.

AWS Free Tier includes **5GB storage, 20,000 Get Requests, and 2,000 Put Requests** with Amazon S3.

View AWS Free Tier Details »

We are lowering S3 storage prices by 36% to 65%, effective April 1st, 2014.

See New Amazon S3 Prices »

Amazon S3 (Simple Storage Service) Basics

Amazon S3 stores data as objects within **buckets**. An **object** is comprised of a file and optionally any metadata that describes that file.

To store an object in Amazon S3, you upload the file you want to store to a bucket. When you upload a file, you can set permissions on the object as well as any metadata.

Buckets are the containers for objects. You can have one or more buckets. For each bucket, you can control access to the bucket (who can create, delete, and list objects in the bucket), view access logs for the bucket and its objects, and choose the geographical region where Amazon S3 will store the bucket and its contents.

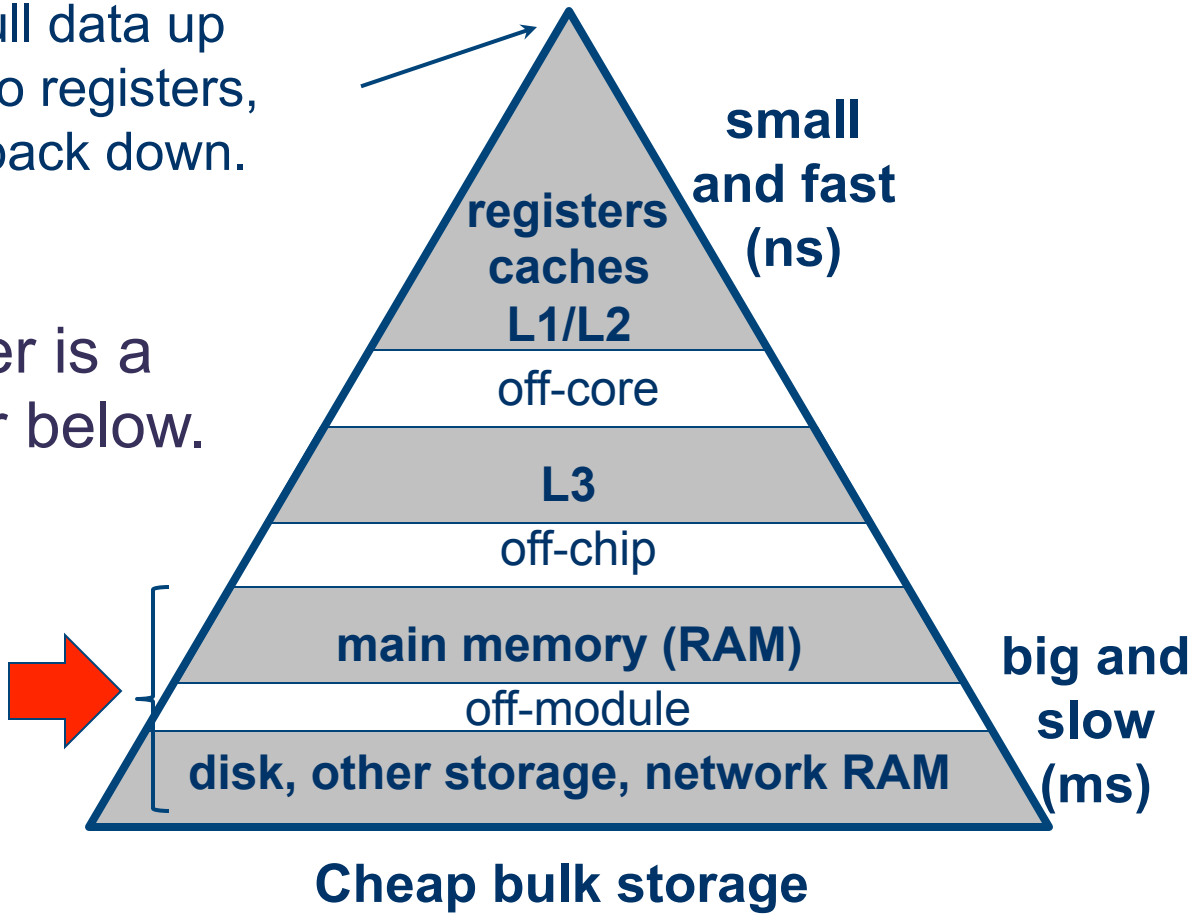
http://docs.aws.amazon.com/AmazonS3/latest/gsg/AmazonS3Basics.html

Memory/storage hierarchy

Computing happens **here**, at the tip of the spear. The cores pull data up through the hierarchy into registers, and then push updates back down.

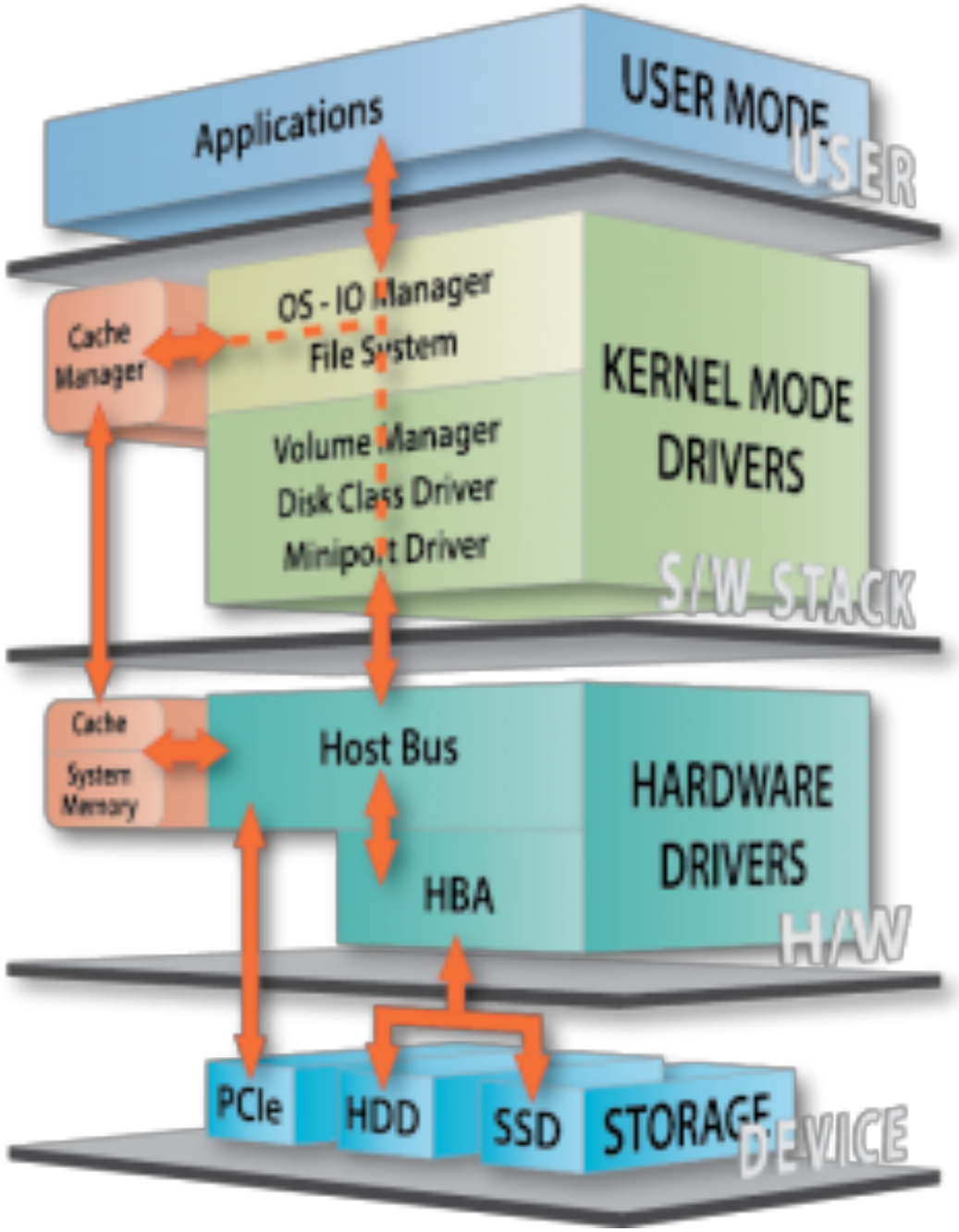
In general, each layer is a **cache** over the layer below.

You are here.



The block storage abstraction

- Read/write **blocks** of size **b** on a **logical** storage device ("disk").
- A disk is a numbered array of these basic blocks. Each block is named by a unique number (e.g., logical BlockID).
- CPU (typically executing kernel code) forms **buffer** in memory and issues read or write command to device queue/driver.
- Device **DMAs** data to/from memory buffer, then interrupts the CPU to signal completion of each request.
- Device I/O is **asynchronous**: the CPU is free to do something else while I/O in progress.
- Transfer size **b** may vary, but is always a multiple of some basic block size (e.g., **sector** size), which is a property of the device, and is always a power of 2.
- Storage blocks containing data/metadata are cached in memory buffers while in active use: called **buffer cache** or **block cache**.

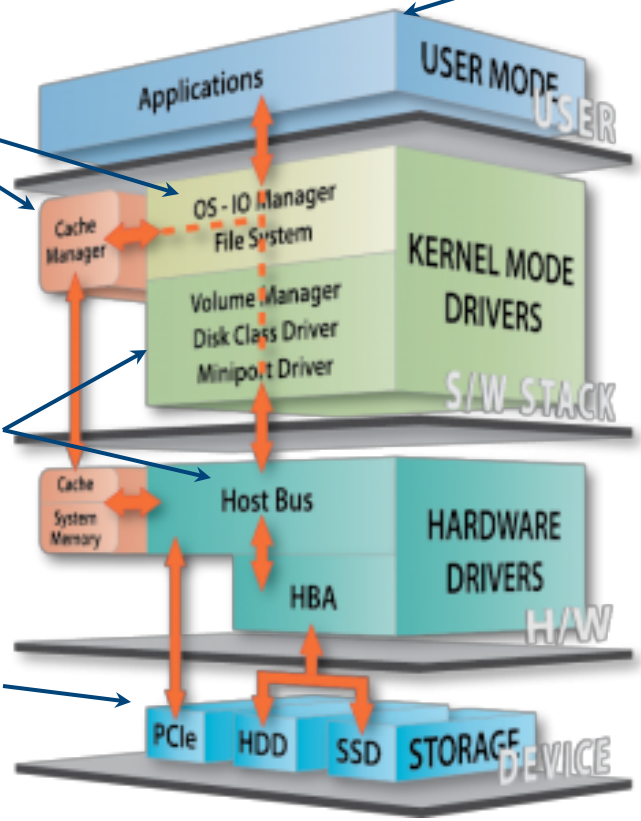

Applications
USER MODE
USER
OS - IO Manager
File System
Cache Manager
Volume Manager
Disk Class Driver
Miniport Driver
KERNEL MODE DRIVERS
S/W STACK
Cache
System Memory
Host Bus
HBA
HARDWARE DRIVERS
H/W
PCIe
HDD
SSD
STORAGE
DEVICE

[Calypso]

Storage stack

Databases, Hadoop, etc.

We care mostly about this stuff.



File system API. Generic, for use over many kinds of storage devices.

Device driver software is a huge part of the kernel, but we mostly ignore it.

Standard block I/O internal interface. Block read/write on numbered blocks on each device/partition. For kernel use only: DMA + interrupts.

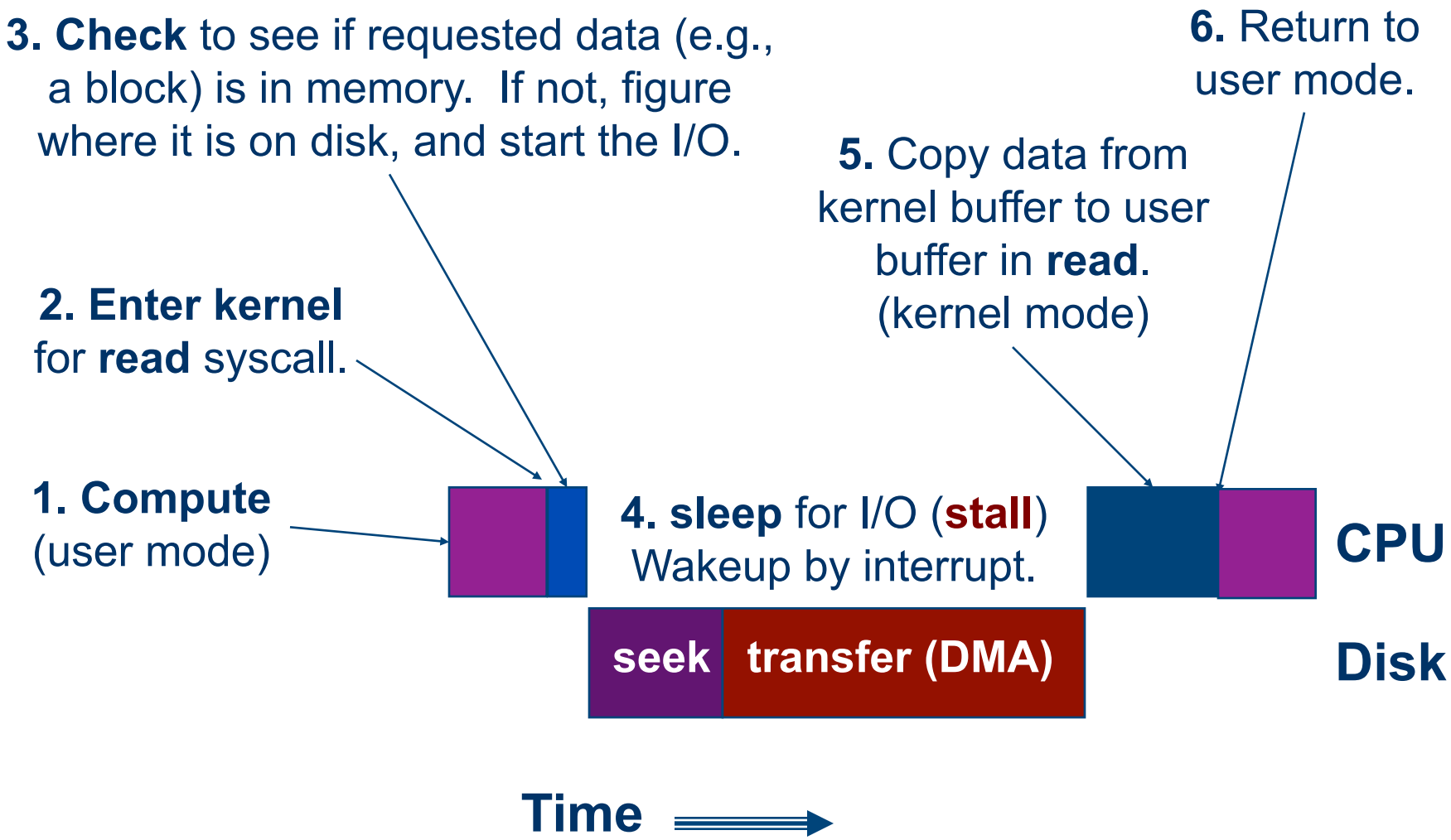
Many storage technologies, advancing rapidly with time.

Rotational disk (HDD): cheap, mechanical, high latency.

Solid-state "disk" (SSD): low latency/power, wear issues, getting cheaper.

[Calypso]

Anatomy of a read



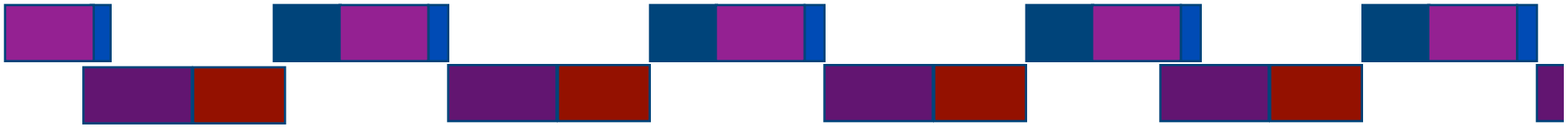
Improving utilization for I/O

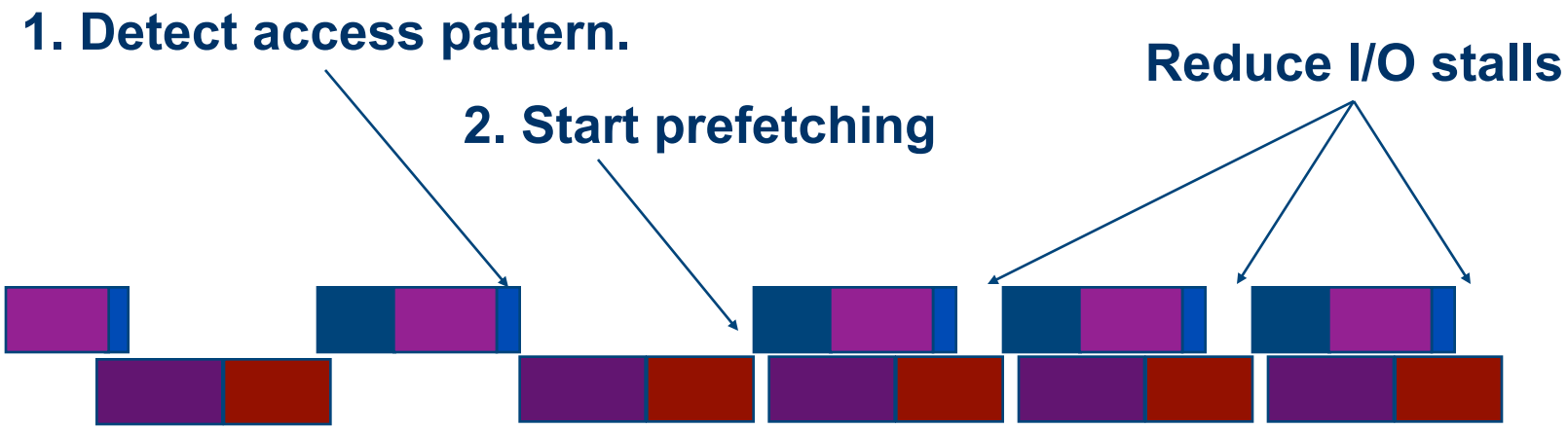
Some things to notice about the “anatomy” fig.

- The CPU is idle when the disk is working.
- The disk is idle when the CPU is working.
- If their service demands are equal, each runs at 50%.
  - Limits throughput! **How to improve this?**
  - How to “hide” the I/O latency?
- If the disk service demand is 10x the CPU service demand, then CPU utilization is at most 10%.
  - Limits throughput! **How to improve this?**
  - How to balance the system?

Prefetching for high read throughput

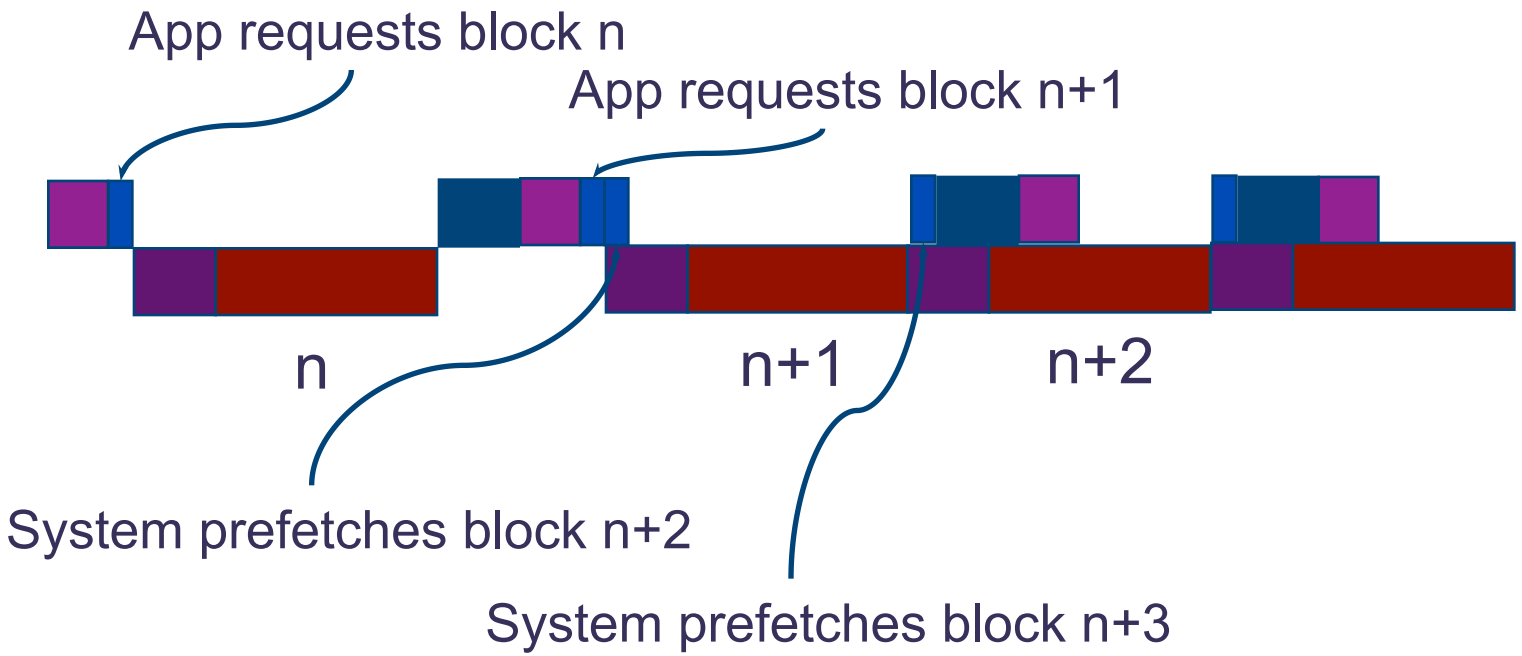
- **Read-ahead** (prefetching)
  - Fetch blocks into the cache in expectation that they will be used.
  - Requires prediction. Common for **sequential** access.





Sequential read-ahead

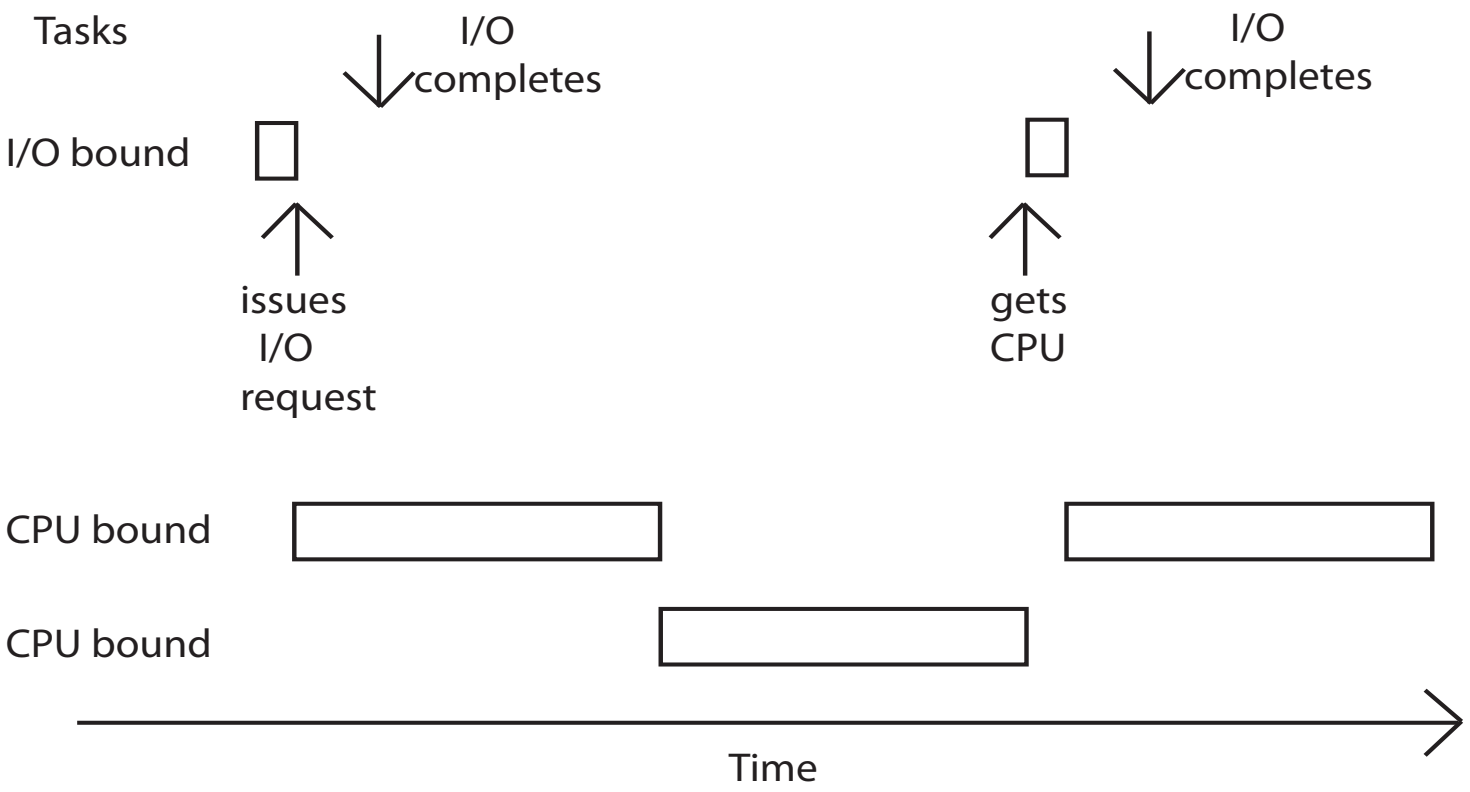
- **Prediction is easy for sequential access.** "Most files are read and written sequentially."
- **Read-ahead also helps reduce seeks by reading larger chunks if data is laid out sequentially on disk.**



Challenge: I/O and scheduling

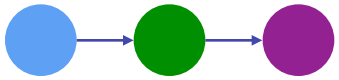
- Suppose thread T does a lot of I/O.
- T blocks while the I/O is in progress.
- When each I/O completes, T gets back on the readyQ.
- Where T waits for threads that use a lot of CPU time.
  - While the disk or other I/O device sits idle!
- T needs only a smidgen of CPU time to get its next I/O started.
- Why not let it jump the queue, and get the disk going so that both the disk and CPU are fully utilized?
- This is a form of **shortest job first (SJF)** scheduling, also known as **shortest processing time first (SPT).**

Mixed Workload

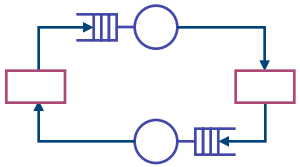


Two Schedules for CPU/Disk

1. Naive **Round Robin**



CPU busy 25/37: U = 67%
Disk busy 15/37: U = 40%



2. Add internal priority boost for I/O completion

CPU busy 25/25: U = 100%
Disk busy 15/25: U = 60%

33% improvement in utilization

When there is work to do, U == efficiency. More U means better throughput.

Estimating Time-to-Yield

How to predict which job/task/thread will have the shortest demand on the CPU?

- If you don't know, then guess.

  Weather report strategy: predict future *D* from the recent past.

We can "guess" well by using **adaptive internal priority**.

- Common technique: **multi-level feedback queue**.
- Set N priority levels, with a timeslice quantum for each.
- If thread's quantum expires, drop its priority **down** one level.
  - "It must be **CPU bound**." (mostly exercising the CPU)
- If a job yields or blocks, bump priority **up** one level.
  - "It must be **I/O bound**." (blocking to wait for I/O)

Example: a recent Linux rev

“Tasks are determined to be I/O-bound or CPU-bound based on an interactivity heuristic. A task's **interactiveness metric** is calculated based on how much time the task executes compared to how much time it sleeps. Note that because I/O tasks schedule I/O and then wait, an I/O-bound task spends more time sleeping and waiting for I/O completion. This increases its interactive metric.”

Key point: interactive tasks get higher priority for the CPU, when they want the CPU (which is not much).

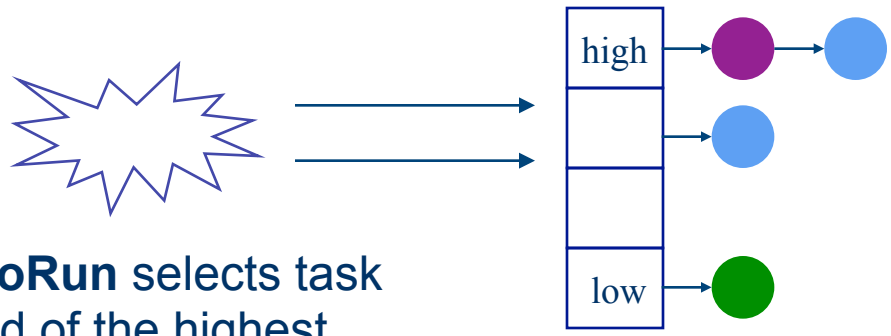
Multilevel Feedback Queue

Many systems (e.g., Unix variants) implement internal priority using a **multilevel feedback queue**.

- **Multilevel**. Separate ready queue for each of *N* priority levels.

  Use RR on each queue; look at queue *i+1* only if queue *i* is empty.

- **Feedback**. Factor a task's previous behavior into its priority.
- Put each ready/awakened task at the tail of the q for its priority.



I/O bound tasks

Tasks holding resouces
Tasks with high external priority

CPU-bound tasks

GetNextToRun selects task at the head of the highest priority queue: constant time, no sorting

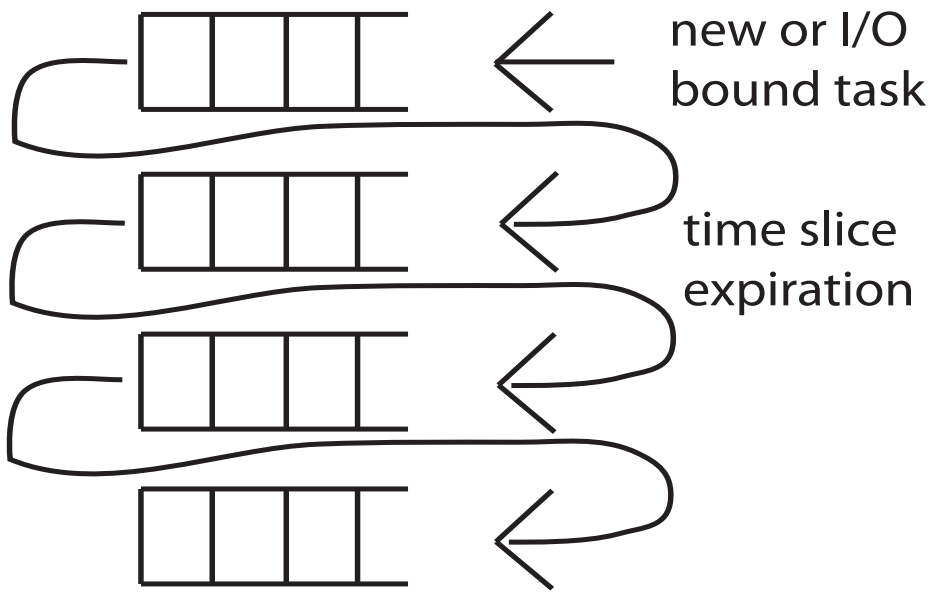
ready queues
indexed by priority

Priority of CPU-bound tasks decays with system load and service received.

MFQ

Priority	Time Slice (ms)
1	10
2	20
3	40
4	80

Round Robin Queues



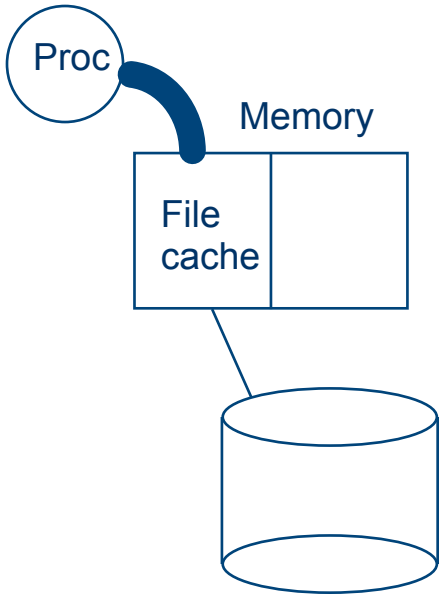
Challenge: data management

- Data volumes are growing enormously.
- Mega-services are “grounded” in data.
- **How to scale the data tier?**
  - Scaling requires **dynamic placement** of data items across data servers, so we can grow the number of servers.
  - **Sharding** divides data across multiple servers or storage units.
  - **Caching** helps to reduce load on the data tier.
  - **Replication** helps to survive failures and balance read/write load.
  - Caching and replication require **careful update protocols** to ensure that servers see a consistent view of the data.

The Buffer Cache

To the user, both reading and writing of files appear to be synchronous and unbuffered. That is immediately after return from a *read* call the data are available, and conversely after a *write* the user's workspace may be reused. In fact the system maintains a rather complicated buffering mechanism which reduces greatly the number of I/O operations required to access a file. Suppose a *write* call is made specifying transmission of a single byte.

UNIX will search its buffers to see whether the affected disk block currently resides in core memory; if not, it will be read in from the device. Then the affected byte is replaced in the buffer, and an entry is made in a list of blocks to be written. The return from the *write* call may then take place, although the actual I/O may not be completed until a later time. Conversely, if a single byte is read, the system determines whether the secondary storage block in which the byte is located is already in one of the system's buffers; if so, the byte can be returned immediately. If not, the block is read into a buffer and the byte picked out.

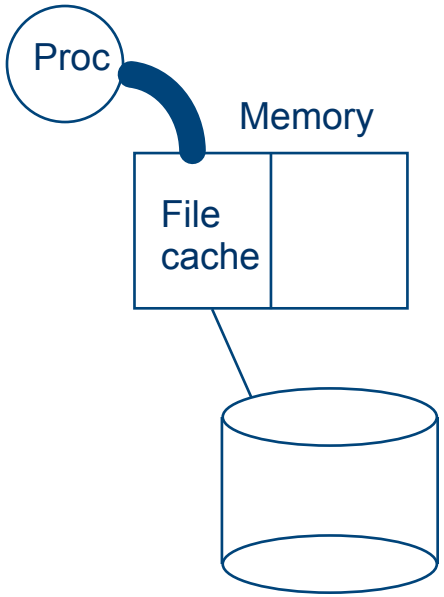


Ritchie and Thompson
The UNIX Time-Sharing System, 1974

Editing Ritchie/Thompson

The system maintains a buffer cache (block cache, file cache) to reduce the number of I/O operations.

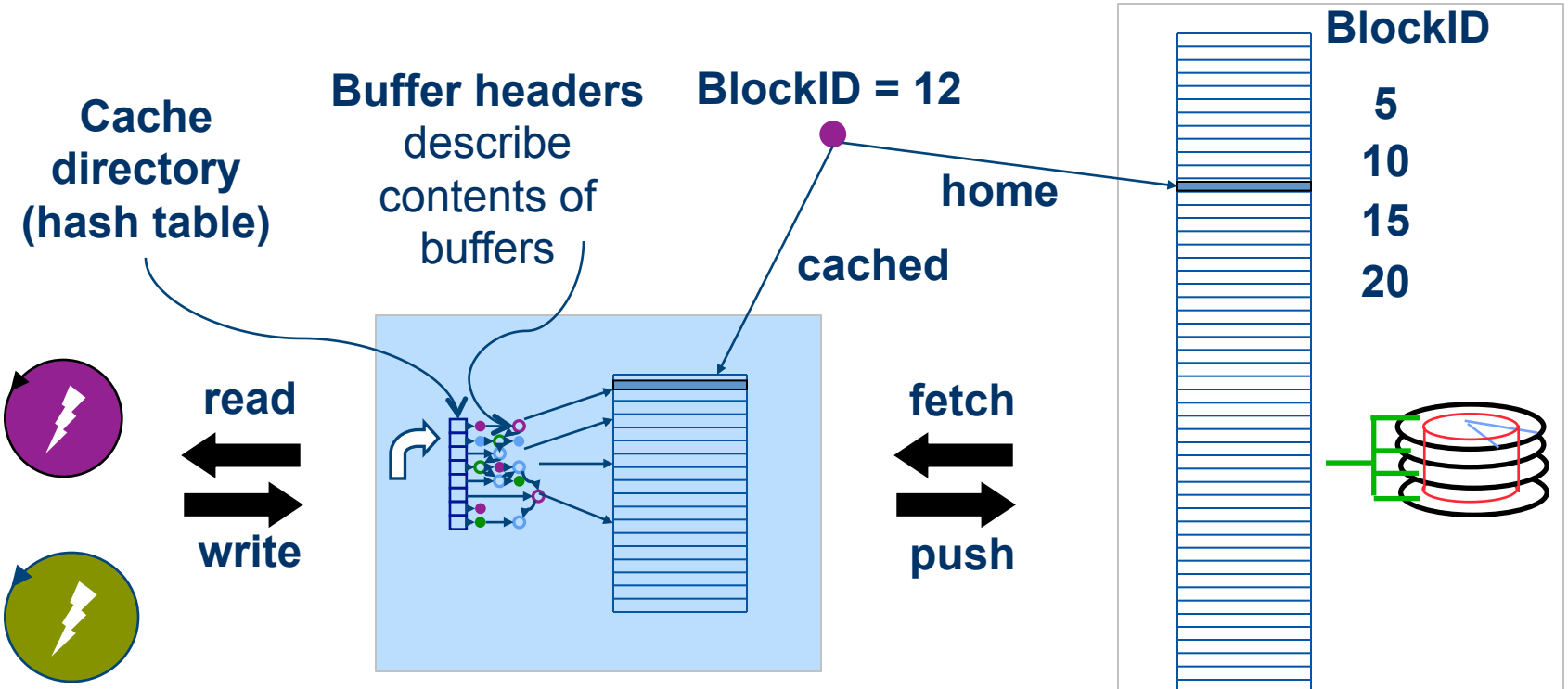
Suppose a process makes a system call to access a **single byte** of a file. UNIX determines the affected disk block, and finds the block if it is resident in the cache. If it is not resident, UNIX allocates a cache buffer and reads the block into the buffer from the disk.



Then, if the op is a **write**, it replaces the affected byte in the buffer. A buffer with modified data is marked **dirty**: an entry is made in a list of blocks to be written. The **write** call may then return. The actual write might not be completed until a later time.

If the op is a **read**, it picks the requested byte out of the buffer and returns it, leaving the block in the cache.

I/O caching



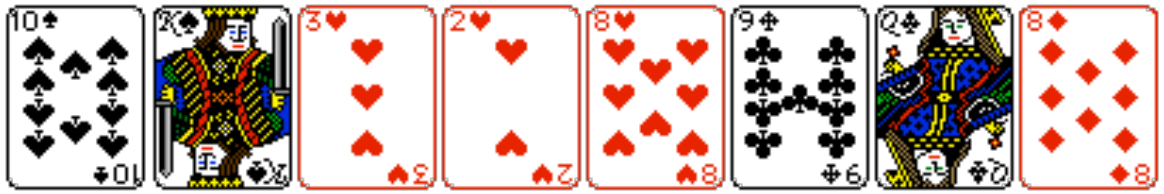
app threads
Request **read** and **write** operations on byte ranges of files

memory
A set of available frames (buffers) for block I/O caching, whose use is controlled by the system

An array of numbered **blocks** on **storage**

Concept: load spreading

- Spread (“deal”) the data across a set of storage units.
  - Make it “look like one big unit”, e.g., “one big disk”.
  - Redirect requests for a data item to the right unit.
- The concept appears in many different settings/contexts.
  - We can spread load across many servers too, to make a **server cluster** look like “one big server”.
  - We can spread out different data items: objects, records, blocks, chunks, tables, buckets, keys….
  - Keep track using maps or a deterministic function (e.g., a hash).
- Also called sharding, declustering, striping, “bricks”.



Example: disk arrays and striping

One way to grow:

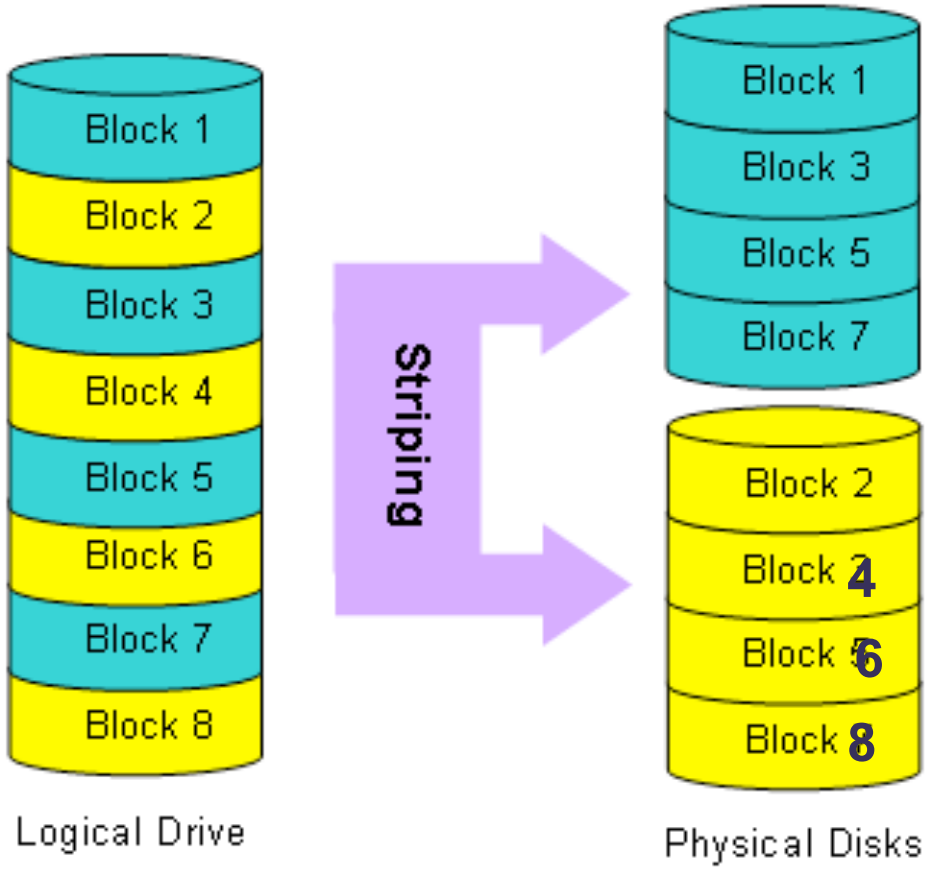
(1) Buy more disks

(2) “Deal the blocks”

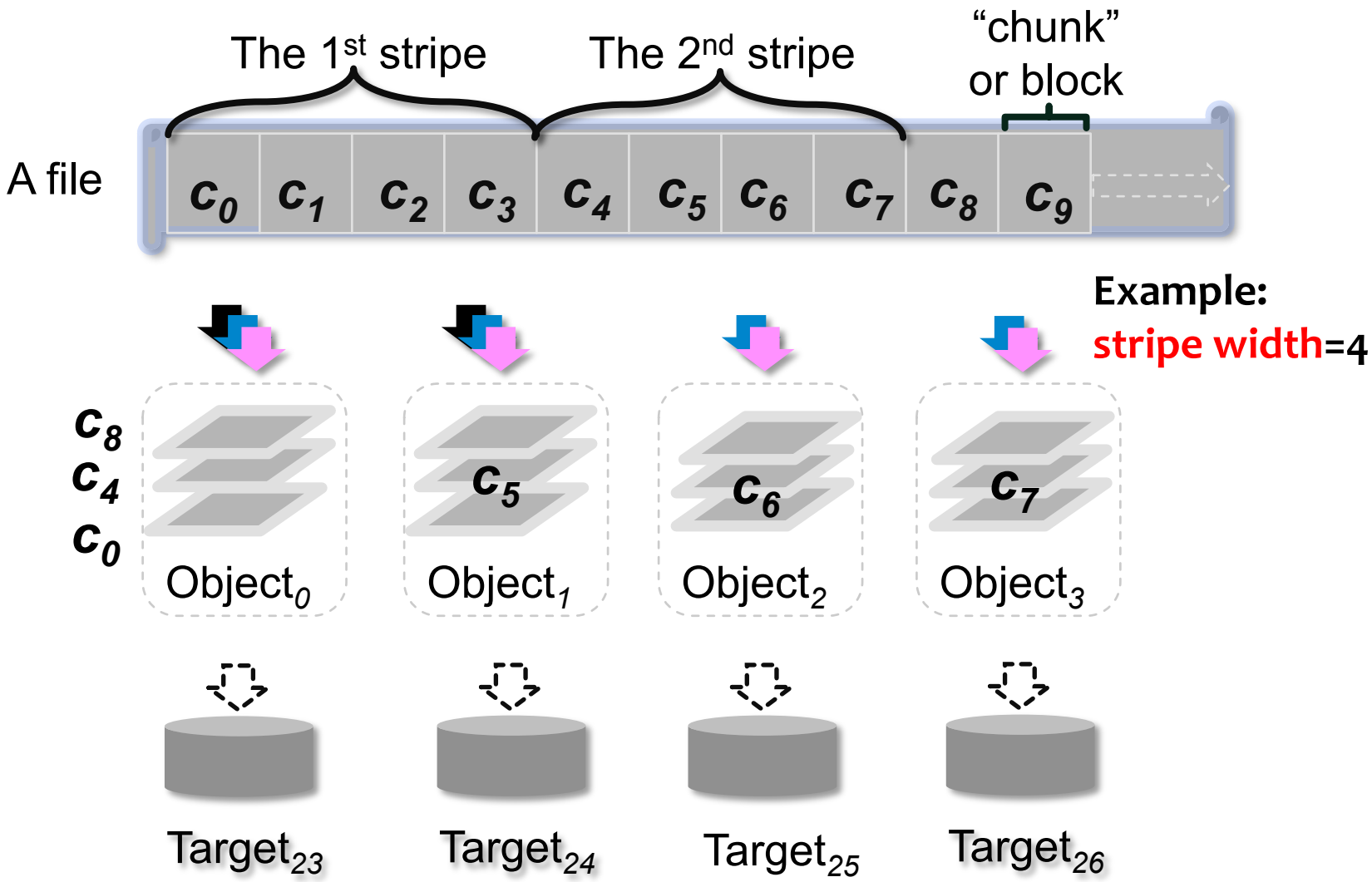
(3) Keep track of them

(4) Spread the I/O

How to keep track of the blocks? Given a request for block n in the “logical drive”, which disk to send it to? Which block on that disk?

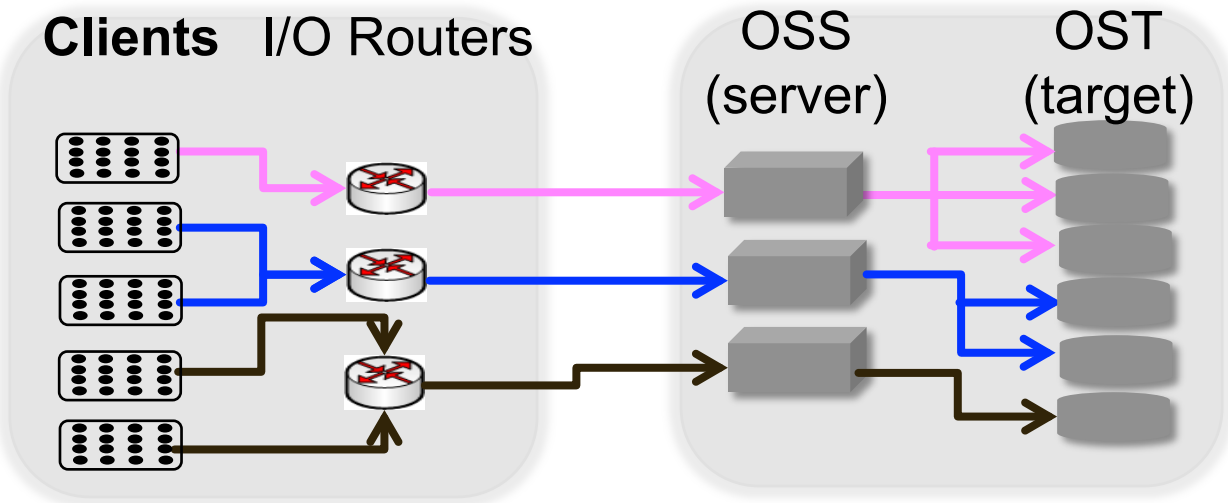


Example: striping in Lustre, a Parallel FS



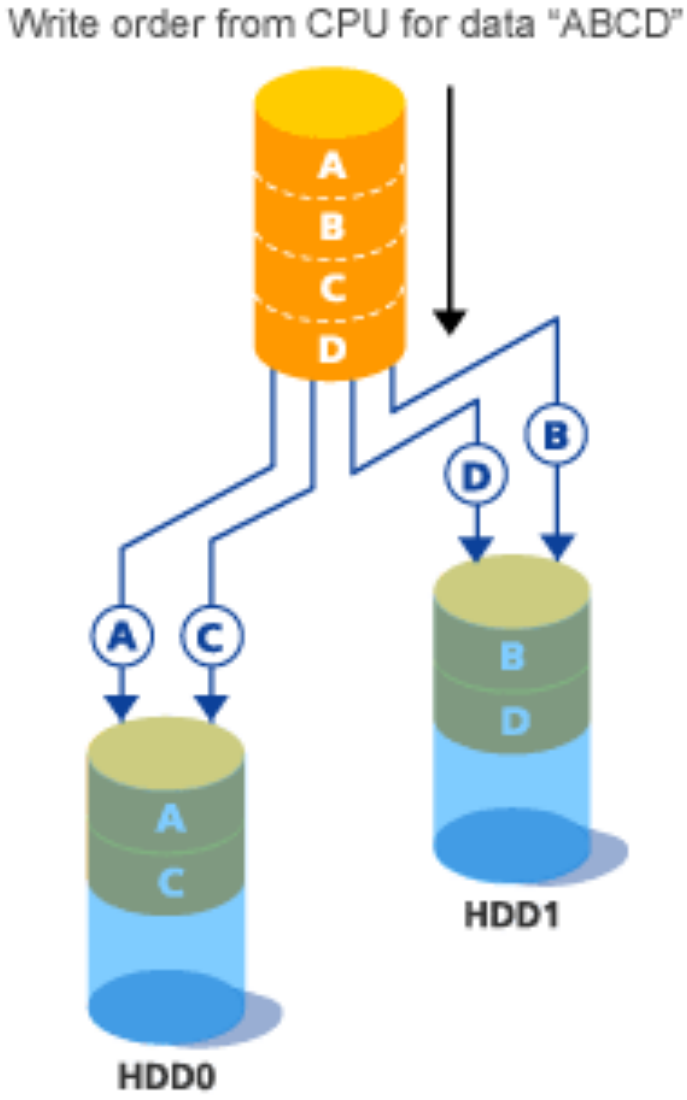
Lustre

- Lustre is an open-source **parallel file system** widely used in supercomputers.
- A Lustre file system contains a collection of many disk servers, each with multiple disks (“targets”).
- Large bursts of reads/writes on large striped files use many targets in parallel.



Load spreading and performance

- What effect does load spreading across N units have on performance, relative to 1 unit?
- What effect does it have on **throughput**?
- What effect does it have on **response time**?
- **How does the workload affect the answers?**
- **E.g., what if striped file access is sequential?**
- **What if accesses are random?**
- What if the accesses follow a **skewed distribution, so some items are more "popular" than others?**



Data is written by spreading it across multiple disks, this is called "striping"

RAID 0

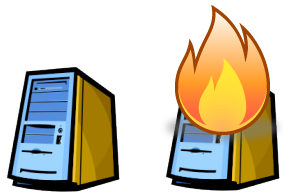
Striping

- Sequential throughput?
- Random throughput?
- Random latency?
- Read vs. write?
- Cost per GB?

Pure declustering

What about failures?

- **Systems fail**. Here's a reasonable set of assumptions about failure properties for servers/bricks (or disks)
  - **Fail-stop** or **fail-fast** fault model
  - Nodes either function correctly or remain silent
  - A failed node may restart, or not
  - A restarted node loses its memory state, and recovers its secondary (disk) state



- If failures are random/independent, the probability of **some** failure is linear with the number of units.
  - Higher scale → less reliable!
- If a disk in a striped storage system fails, we lose the entire file! (It's no good with all those holes.)

What is the probability that a disk fails in an array of N disks at any given time?

Let's make some reasonable assumptions (not always true, but reasonable) about the disks in the array:

- Disk **failures are independent**: a disk doesn't "know" what happens to other disks when it "decides" to fail.

- Disks all have the same probability of failure in any given point in time (or any given interval): no disk is more or less prone to failing to any other.

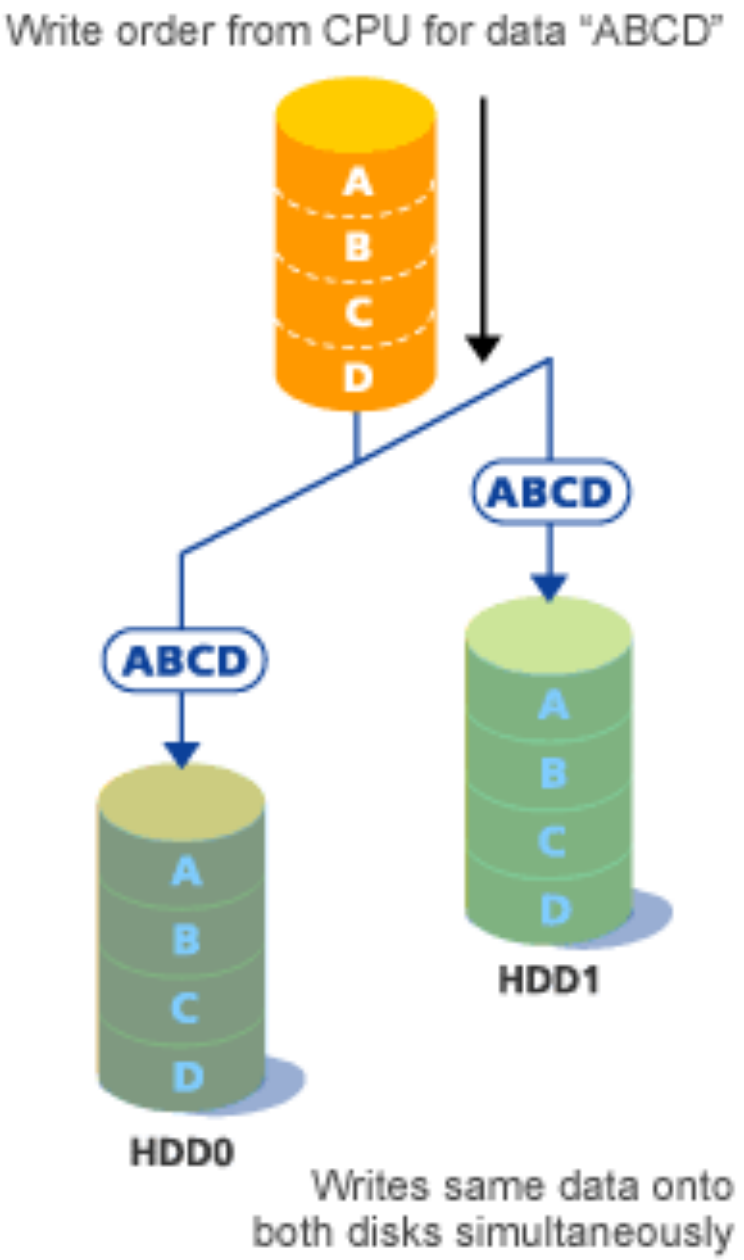
- Disk failures are evenly distributed in time: no interval in time is more prone to disk failures than any other interval.

So the probability of any given disk failing in any given interval (of some fixed length) is a constant. Let us call this constant F.

It is an axiom of elementary probability that:
- The probability of either A or B occurring (but not both) is P(A) + P(B) - P(A and B).
- The probability of both A and B occurring is P(A)*P(B).

Well, if F is a small number (it is), then the probability of two disks failing in the same interval is F-squared, a very small number. So forget about that.

So the probability that **any** one of N disks fails in some given interval is NF.



RAID 1

Mirroring

- Sequential throughput?
- Random throughput?
- Random latency?
- Read vs. write?
- Cost per GB?

Pure replication

Building a better disk: RAID 5

- Market standard
- **Striping** for high throughput for pipelined/batched reads.
- Data redundancy: **parity**
- Parity enables recovery from **one** disk failure.
- Cheaper than mirroring
- Random block write must also update parity for stripe
- Distributes parity: no"hot spot" for random writes

Fujitsu

Parity

Scheme	Character Bits	Parity Bit
Parity Odd	1101101 1000100 1111111	0 1 0
Parity Even	1101101 1000100 1111111	1 0 1

Simple Parity Generation

A parity bit (or parity block) is redundant information stored with a string of data bits (or a stripe of data blocks). Parity costs less than full replication (e.g., mirroring), but allows us to reconstruct the data string/stripe if any single bit/block is lost.

Just to spell it out: a **stripe** is an array of blocks, one block per disk in the disk array. Each block is an array of bits ("memory/storage is fungible"). In RAID-5, one of the blocks of each stripe is a **parity block**. Each bit in the parity block contains parity over the corresponding bits in the data blocks (i.e., the result of a bitwise XOR, exclusive-or).

Q: **If two bits/blocks are lost, your even/odd state may not be changed. Therefore, how do you recover from a failure in this situation?**
A: A classic RAID 5 can survive only a single disk failure. That is why many deployments are using alternatives with more redundancy, e.g., "RAID-6" (two parity disks) or "RAID-10" (essentially a mirrored pair of RAID-5 units, 2x5=10).

Q: **How do you know which bit/block was lost?**
A: It is presumed that disk drive failure is detectable by external means. One possibility is that the disk drive is "fail stop": if it fails, it simply stops serving requests. Or the failed disk reports errors when you try to use it. It is presumed that a disk doesn't ever lie about the stored data (essentially a "byzantine" failure). Fortunately, this assumption is almost always true.

So: in a classic RAID-5, if you know which drive has failed, it is easy to rebuild each stripe from the contents of the remaining disks. For each stripe, the failed disk either contains the parity block of that stripe, or it contains some data block of the stripe. These cases are symmetric: either way the array can rebuild the contains of the missing block.

Note that recovery is very expensive. The recovery process must read the contents of all surviving disks, and write parity data to a replacement disk. That means that a disk failure must be repaired quickly: if a drive fails in your RAID, you should replace it right away.

Q: Can you specify the differences between read and write on that same slide for the different raids? I know you said in class that for raid 1 reads get faster and writes get slower? Could you explain that and also for the other raids?

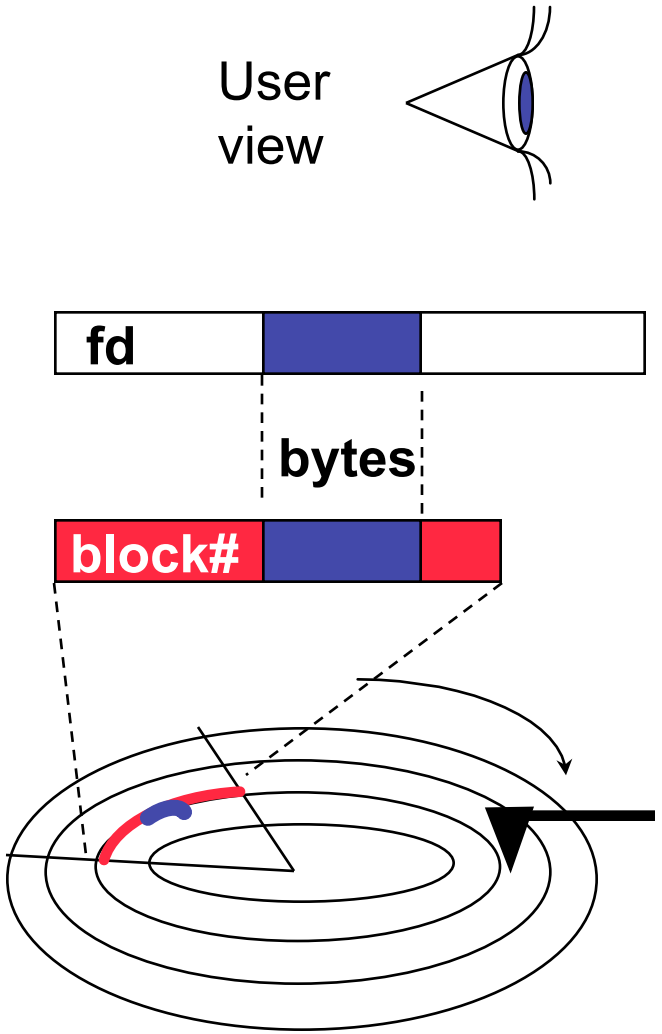
A: Once you understand the RAID 0, 1, 5 patterns (striping, mirroring, and parity), all of these differences are "obvious". The differences stem from different levels of redundancy. Redundancy increases $$$ cost but it also improves reliability. And redundancy tends to make writes more expensive than reads: redundant data must be kept up to date in a write, but it may allow more copies to choose from for load-balancing reads.

Always be careful and precise about "faster" and "slower". For example, for random accesses, throughput goes up with RAIDs but latency (minimum response time, not counting queuing delays) is unchanged: you still have to reach out with a disk arm to get to that data. This example underscores that “faster” and “slower” also depend on characteristics of the workload.

Also, “faster” and “slower” are relative, so you must specify a baseline. Writes on RAID-1 are slower than for RAID-0, because writes in RAID-1 have to go to all disks. In RAID-0 or RAID-5 you can write different data to multiple disks in parallel, but not in RAID-1. But RAID-1 is no slower for writes than a single disk---just more expensive. Also, RAID-1s can load-balance better than RAID-0 or RAID-5 for random read workloads. For any given block, you have N disks to choose from that have a copy of the block, so you can choose the least loaded. So read throughput on a RAID-1 is N times better than a single disk.

RAID-5 suffers for random write throughput because any write of a data chunk that is smaller than a whole stripe must also write a parity block. For a stream of random single-block writes, that means RAID-5 is doing twice as much work as RAID-0. So throughput will be lower.

Names and layers



notes in notebook file

Application

notefile: fd, byte range*

File System

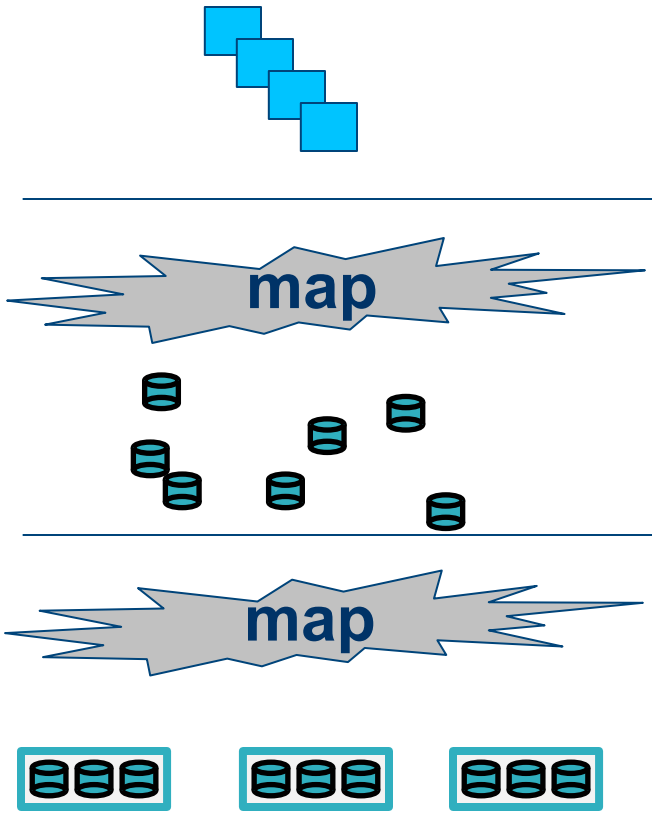
device, block #

Disk Subsystem

surface, cylinder, sector

Add more layers as needed.

More (optional) layers of mapping



Files

Logical disk volumes or objects

Physical disks

There could be many more layers than this!

For “storage virtualization”…

It’s turtles all the way down.

Which block?

When an app requests to read/write a file at offset **i**, how to know which block contains that byte **i** on storage?

We need a map!

We know which **logical block** it is in the file (simple arithmetic), but how do we know which block it is on disk?

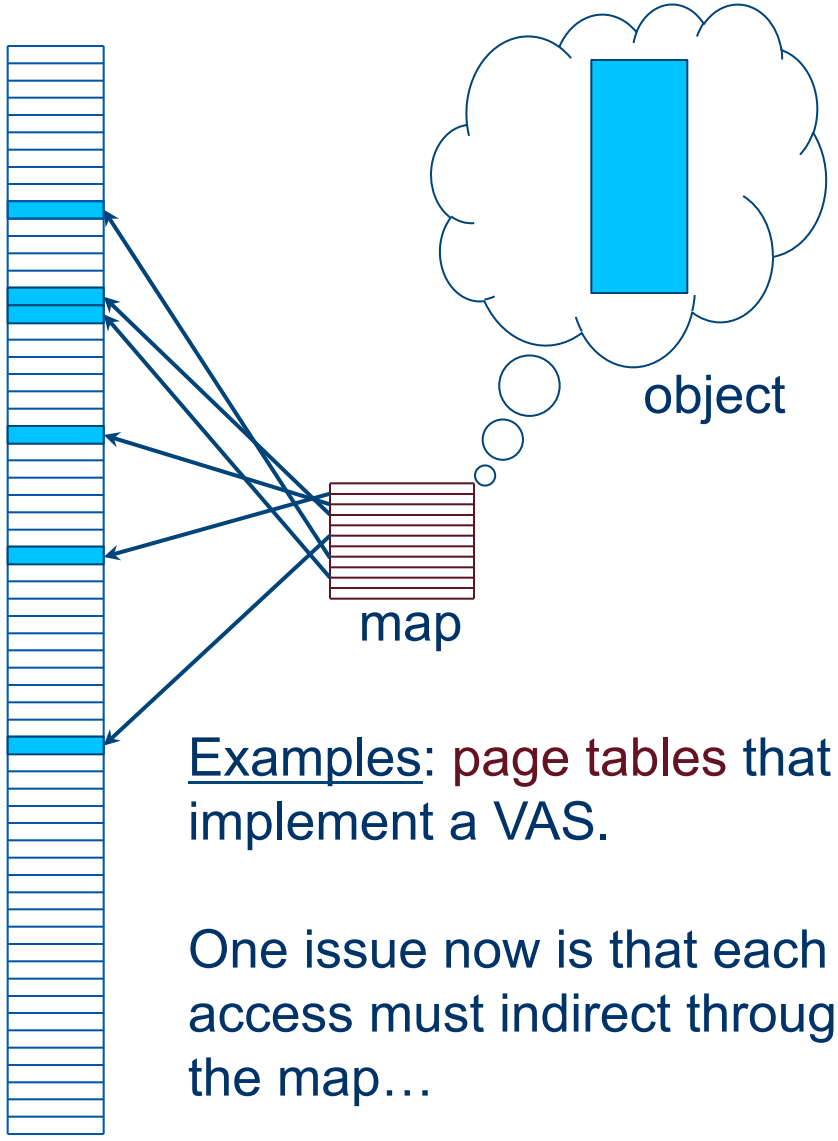
We need a map!

Block maps

Large storage objects (e.g., files, segments) may be **mapped** so they don't have to be stored contiguously in memory or on disk.

Idea: use a level of indirection through a **map** to assemble a storage object from "scraps" of storage in different locations.

The "scraps" can be fixed-size slots: that makes allocation easy because the slots are interchangeable (**fixed partitioning**). Fixed-size chunks of data or storage are called **blocks** or **pages**.

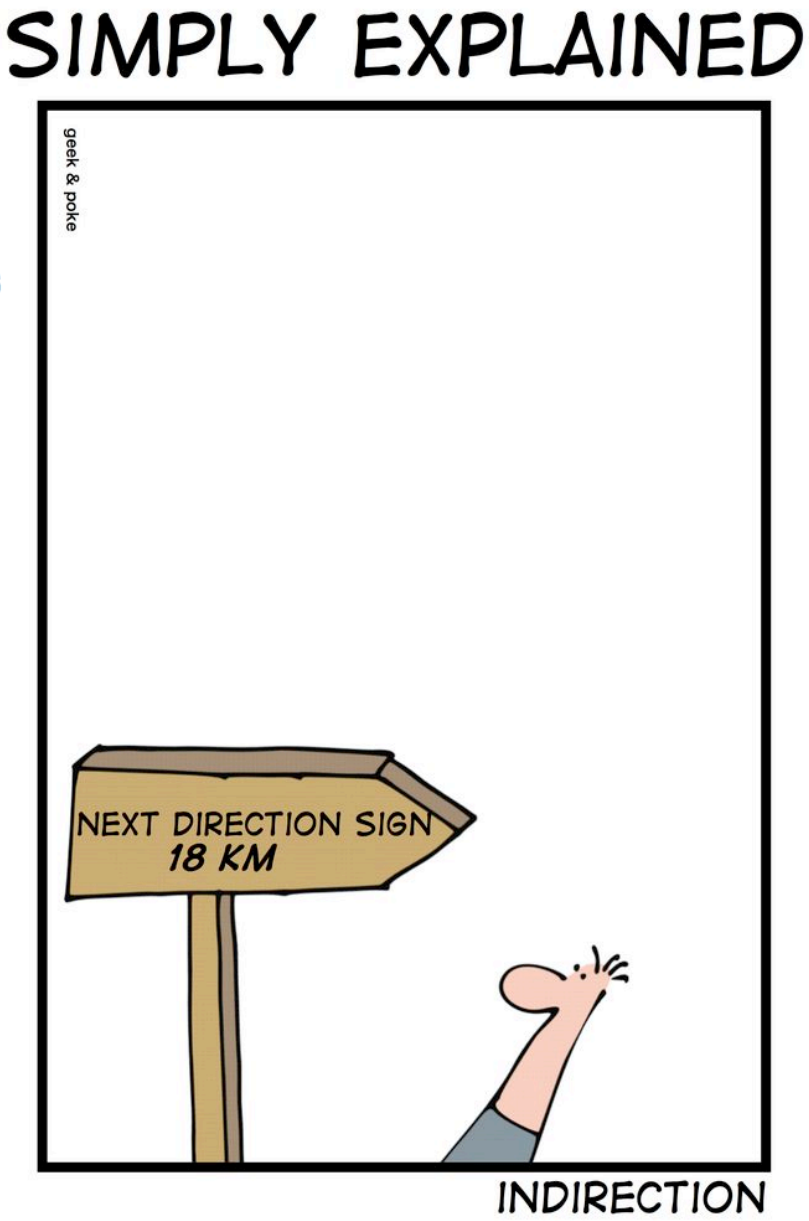


Examples: page tables that implement a VAS.

One issue now is that each access must indirect through the map…

Indirection

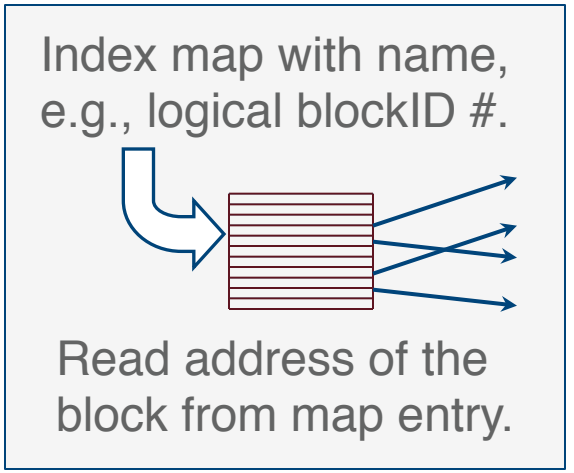
A famous aphorism of David Wheeler goes: "All problems in computer science can be solved by another level of indirection";[1] this is often deliberately mis-quoted with "abstraction layer" substituted for "level of indirection". Kevlin Henney's corollary to this is, "...except for the problem of too many layers of indirection."



Using block maps

- Files are accessed through e.g. **read/write** syscalls: the kernel can chop them up, allocate space in pieces, and reassemble them.
- Allocate in units of fixed-size **logical blocks** (e.g., 4KB, 8KB).
- Each logical block in the object has an address (**logical block number** or **logical blockID**): a block offset within the object.
- Use a **block map** data structure.
  - **Index by logical blockID, return underlying address**
  - **Example**: inode indexes file blocks on disk
  - Maps file+logical blockID to disk block #
  - **Example**: page table indexes pages in memory
  - Maps VAS+page VPN to machine frame PFN

Note: the addresses (block # or PFN) might themselves be blockIDs indexing another level of virtual map!



To put it another way

- Variable partitioning is a pain. We need it for heaps, and for other cases (e.g., address space layout).
- But for files/storage we can break the objects down into “pieces”.
  - When access to files is through an API, we can add some code behind that API to represent the file contents with a dynamic linked data structure (a map).
  - If the pieces are fixed-size (called pages or logical blocks), we can use fixed partitioning to allocate the underlying storage, which is efficient and trivial.
  - With that solution, internal fragmentation is an issue, but only for small objects. (Why?)
- That approach can work for VM segments too: we have VM hardware to support it (since the 1970s).

Representing files: inodes

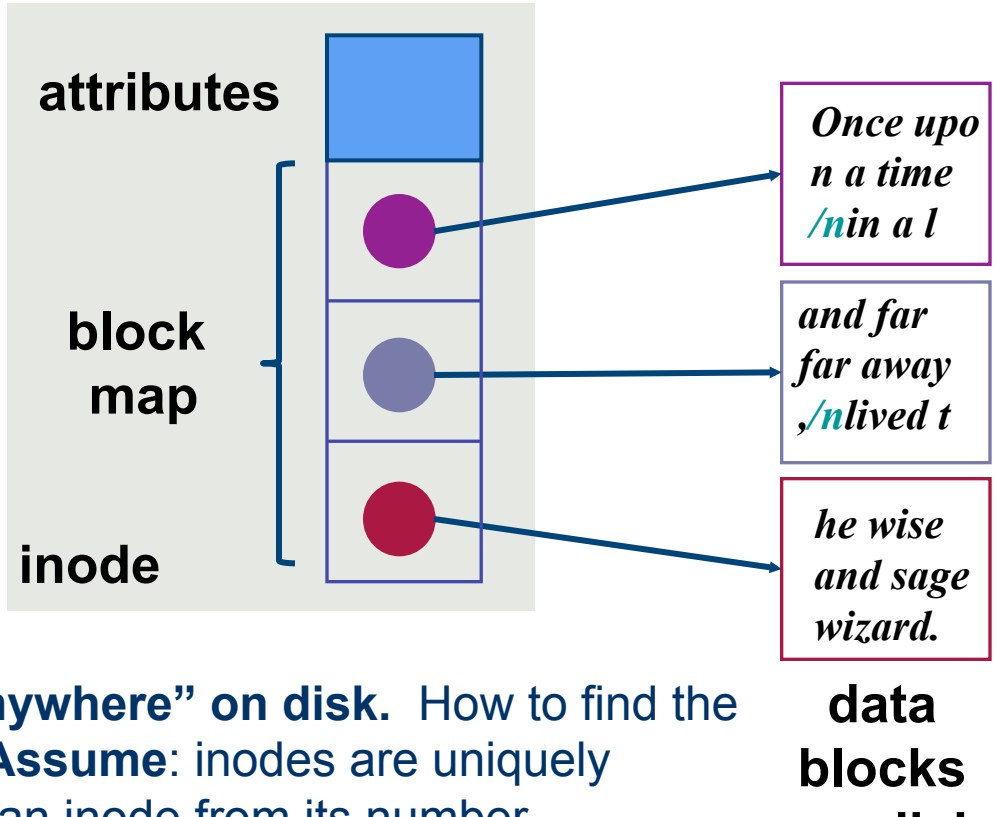
- There are many many file system implementations.
- Most of them use a block map to represent each file.
- Each file is represented by a corresponding data object, which is the root of its block map, and holds other information about the file (the file's "**metadata**").
- In classical Unix and many other systems, this per-file object is called an **inode**. ("index node")
- The inode for a file is stored "on disk": the OS/FS reads it in and keeps it in memory while the file is in active use.
- When a file is modified, the OS/FS writes any changes to its inode/maps back to the disk.

Inodes

A file's data blocks could be "anywhere" on disk. The file's **inode** maps them. Each entry of the map gives the disk location for the corresponding logical block.

A fixed-size inode has a fixed-size block map.

How to represent large files that have more logical blocks than can fit in the inode's map?



An inode could be "anywhere" on disk. How to find the inode for a given file? **Assume**: inodes are uniquely numbered: we can find an inode from its number.

Classical Unix inode

A classical Unix **inode** has a set of **file attributes** (below) in addition to the root of a hierarchical block map for the file. The inode structure size is fixed, e.g., total size is 128 bytes: 16 inodes fit in a 4KB block.

```
/* Metadata returned by the stat and fstat functions */
struct stat {
    dev_t         st_dev;      /* device */
    ino_t         st_ino;      /* inode */
    mode_t        st_mode;     /* protection and file type */
    nlink_t       st_nlink;    /* number of hard links */
    uid_t         st_uid;      /* user ID of owner */
    gid_t         st_gid;      /* group ID of owner */
    dev_t         st_rdev;     /* device type (if inode device) */
    off_t         st_size;     /* total size, in bytes */
    unsigned long st_blksize;  /* blocksize for filesystem I/O */
    unsigned long st_blocks;   /* number of blocks allocated */
    time_t        st_atime;    /* time of last access */
    time_t        st_mtime;    /* time of last modification */
    time_t        st_ctime;    /* time of last change */
};
```

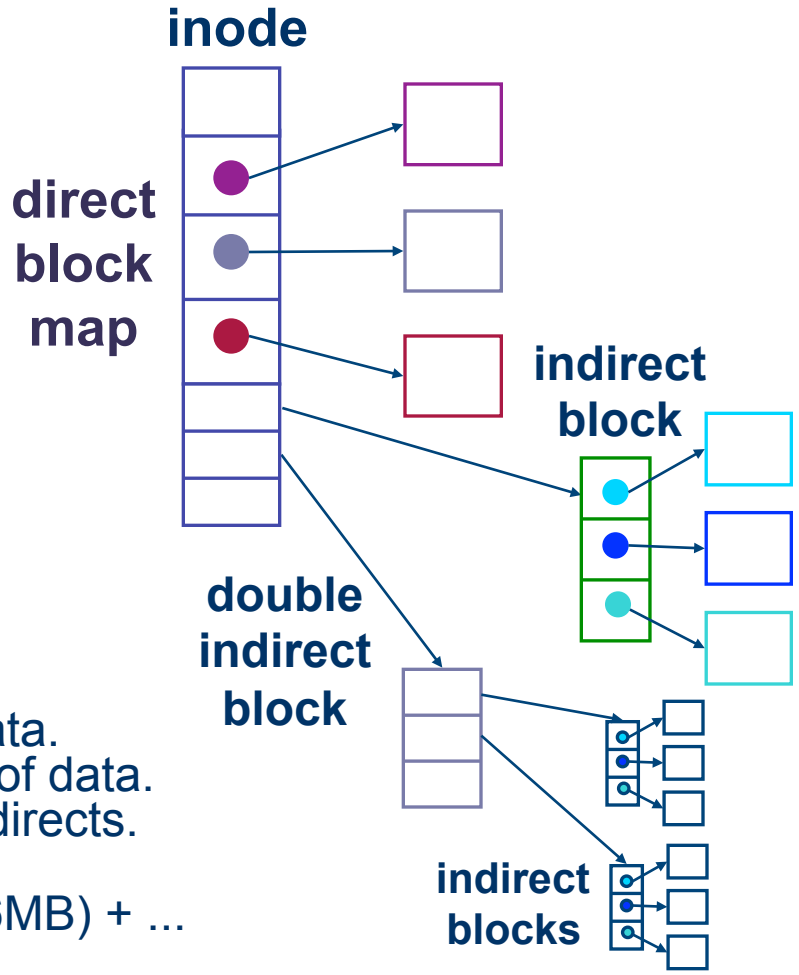
Not to be tested

Representing Large Files

Classical Unix file systems

inode == 128 bytes

Each inode has 68 bytes of attributes and 15 block map entries that are the root of a tree-structured block map.



Suppose block size = 8KB
12 direct block map entries: map 96KB of data.
One indirect block pointer in inode: + 16MB of data.
One double indirect pointer in inode: +2K indirects.

Maximum file size is 96KB + 16MB + (2K*16MB) + ...

The numbers on this slide are for illustration only.

Skewed tree block maps

- Inodes are the root of a tree-structured block map.
  - Like hierarchical page tables, **but**
- These maps are skewed.
  - Low branching factor at the root.
  - "The further you go, the bushier they get."
  - Small files are cheap: just need the inode to map it.
  - …and most files are small.
- Use **indirect blocks** for large files.
  - Requires another fetch for another level of map block
  - But the shift to a high branching factor covers most large files.
- **Double indirect blocks** allow very large files.

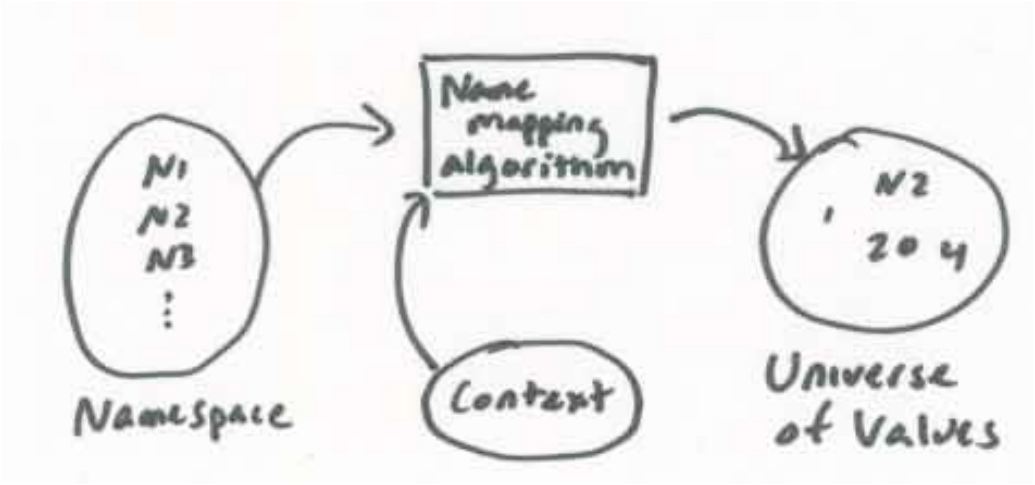
Inodes on disk

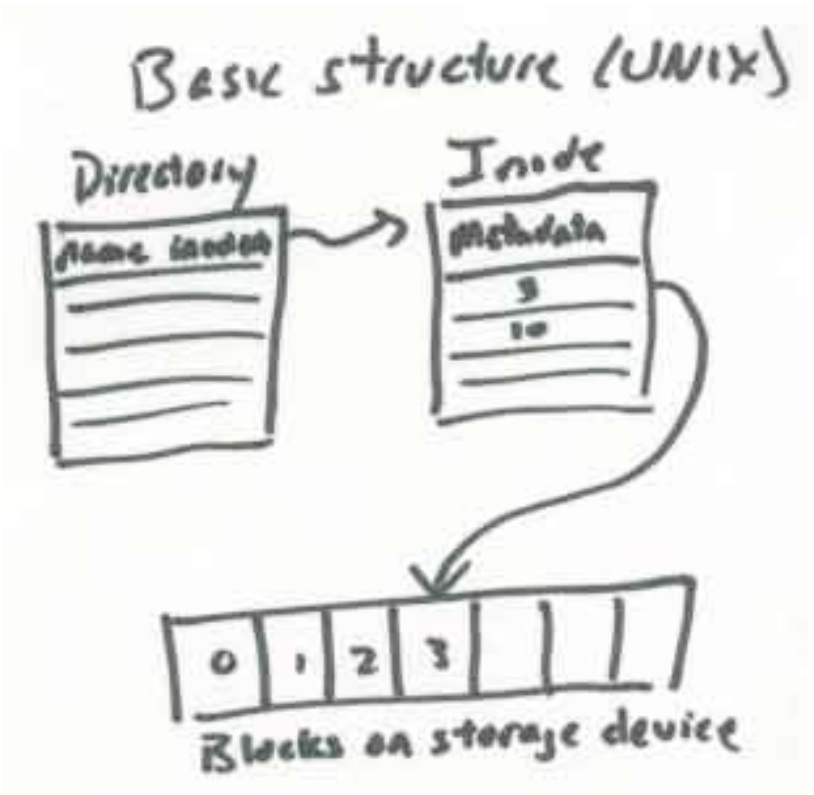
Where should inodes be stored on disk?

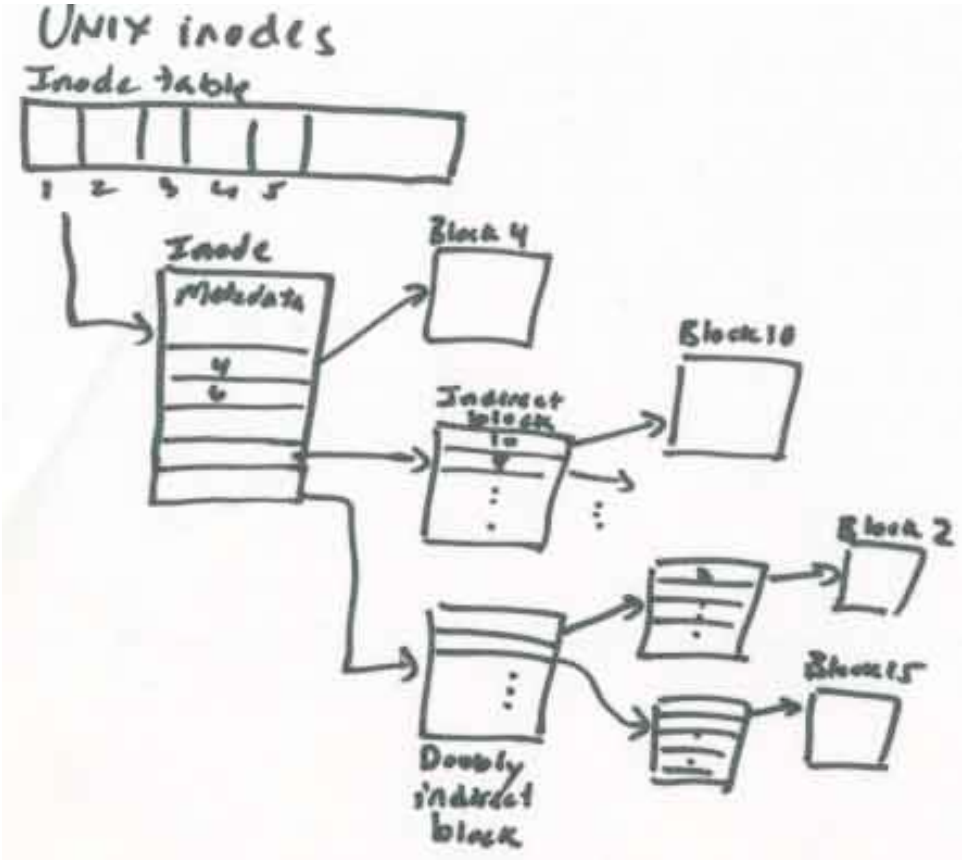
- They're a fixed size, so we can dense-pack them into blocks. We can find them by **inode number**. But where should the blocks be?
- Early Unix reserved a fixed array of inodes at the start of the disk.
  - But how many inodes will we need? And don't we want inodes to be stored close to the file data they describe?
- Second-gen file systems (FFS) reserve a fixed set of blocks at known locations distributed throughout the storage volume.
- Newer file systems add a **level of indirection**: make a system **inode file** ("ifile") in the volume, and store inodes in the inode file.
  - That allows a variable number of inodes (ifile can grow), and they can be anywhere on disk: the ifile is itself a file indexed by an inode.
  - Originated with Berkeley's Log Structured File System (LFS) and NetApp's Write Anywhere File Layout (WAFL).

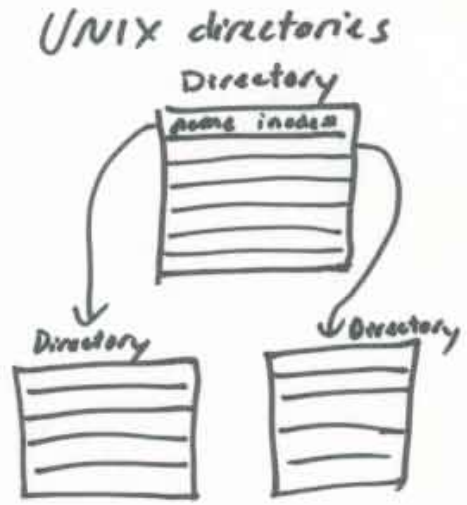
File systems today: “Filers”

- Network-attached (IP)
- RAID appliance
- Multiple protocols
  - iSCSI, NFS, CIFS
- Admin interfaces
- Flexible configuration
- Lots of virtualization: dynamic volumes
- Volume cloning, mirroring, snapshots, etc.
- NetApp technology leader since 1994 (WAFL)



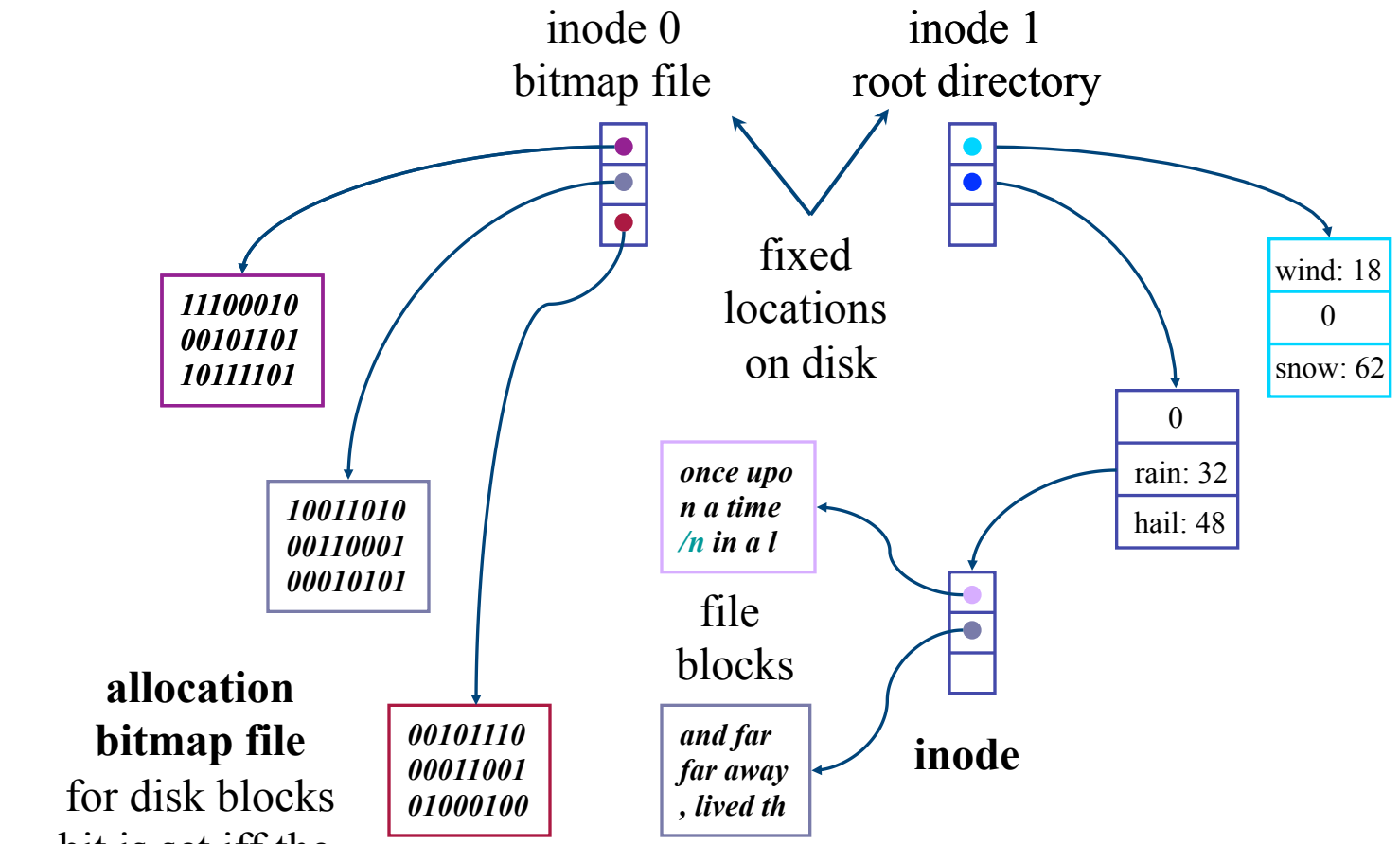






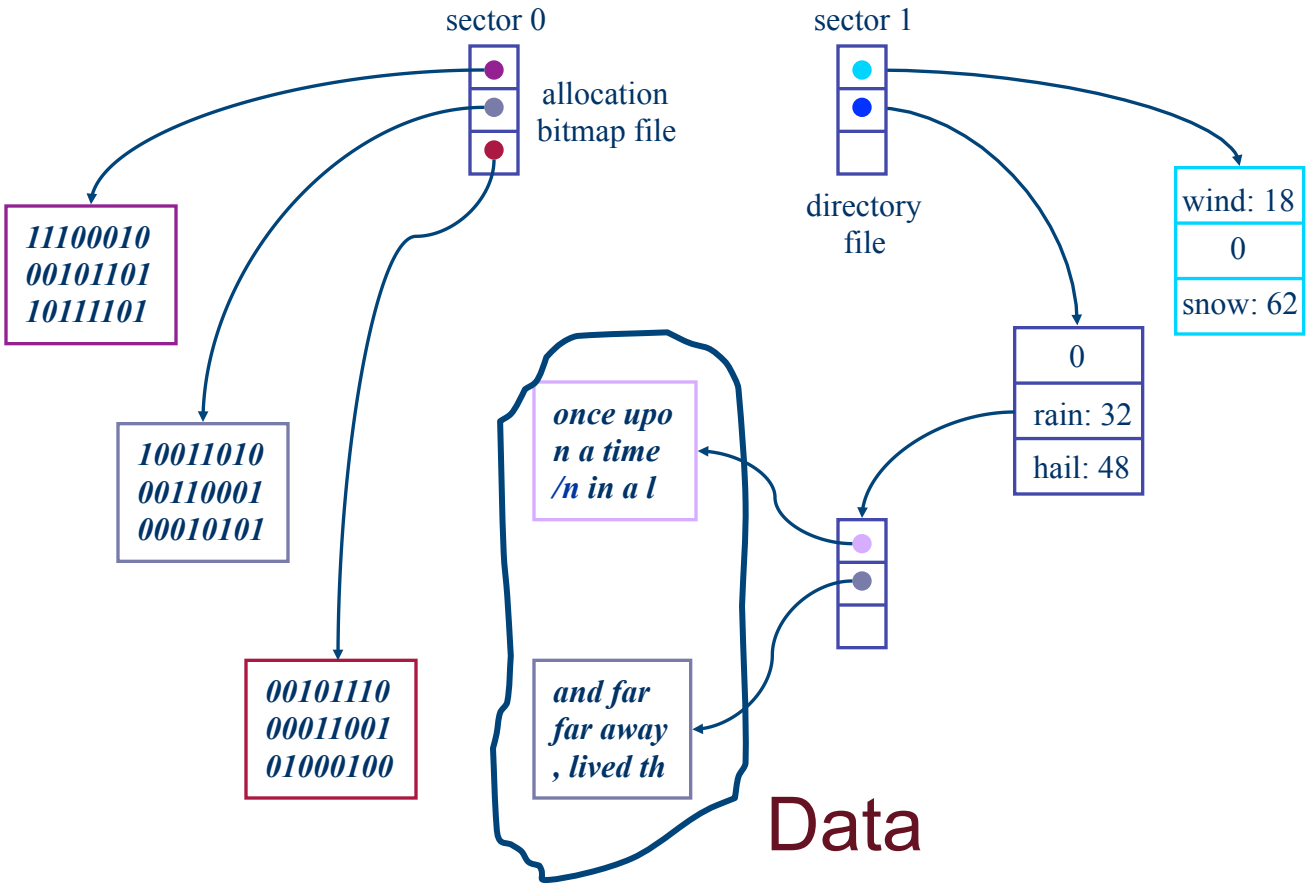
http://web.mit.edu/6.033/2001/wwwdocs/handouts/naming_review.html

Filesystem layout on disk

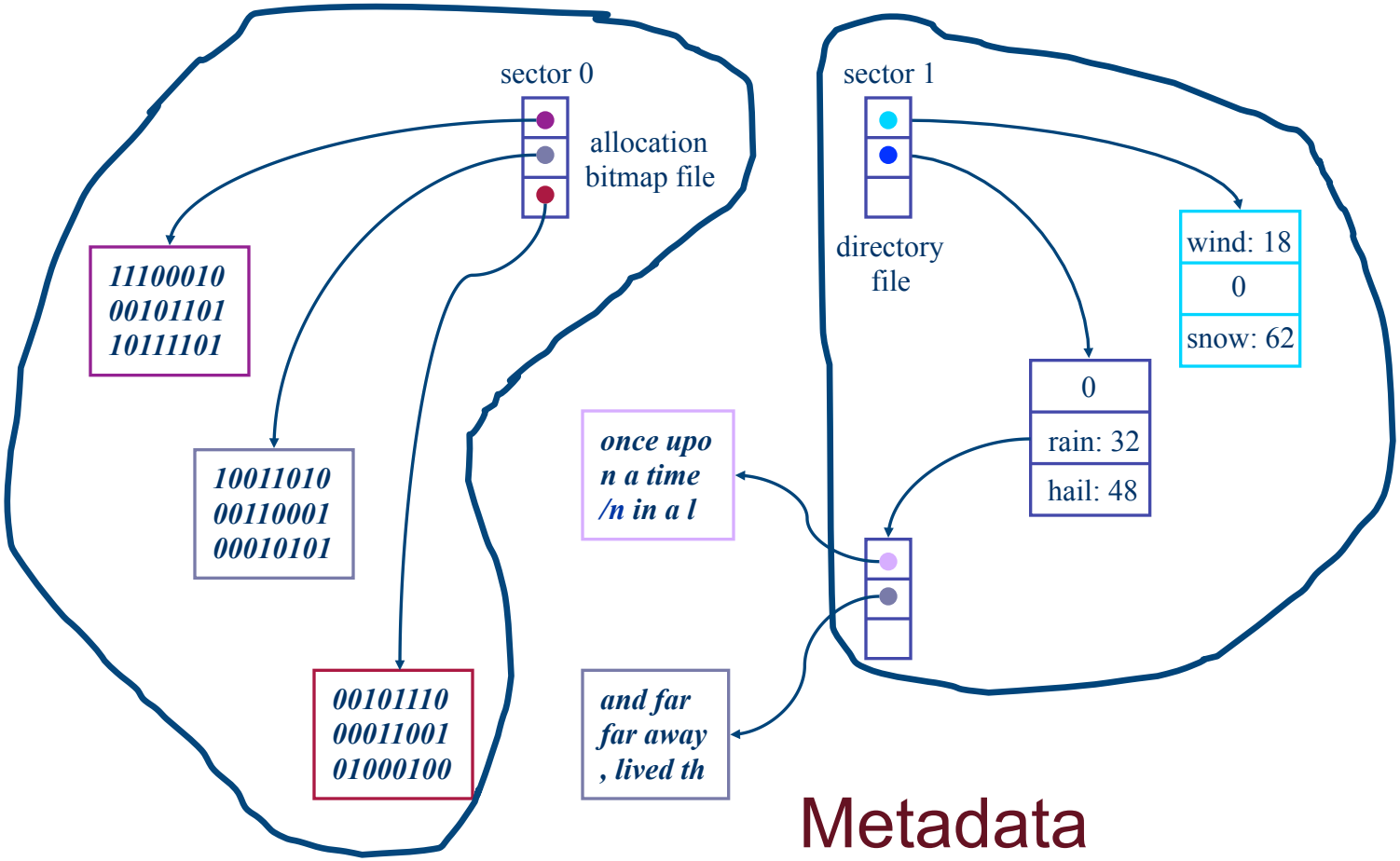


This is a toy example (Nachos).

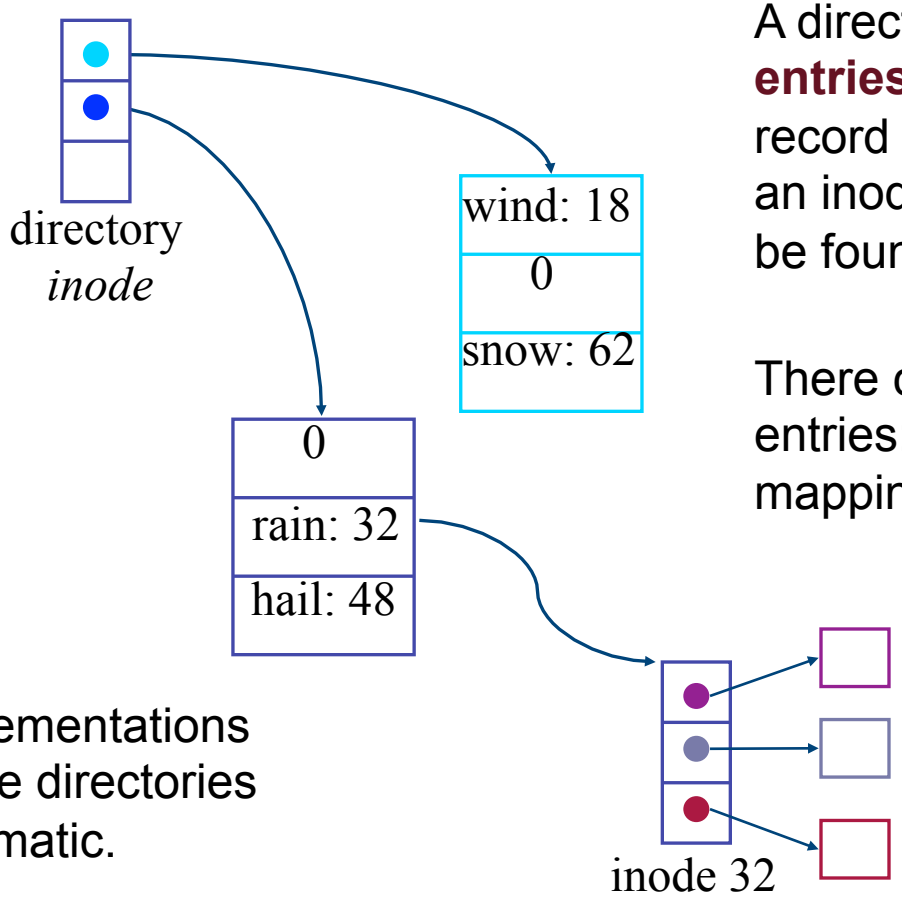
A Filesystem On Disk



A Filesystem On Disk



Directories



A directory contains a set of **entries**. Each directory entry is a record mapping a symbolic name to an inode number. The inode can be found on disk from its number.

There can be no duplicate name entries: the name-to-inode mapping is a function.

Note: implementations vary. Large directories are problematic.

A creat or mkdir operation must scan the directory to ensure that creates are **exclusive**.

Entries or free slots are typically found by a linear scan.

Q: Can you characterize the difference between page table maps and inode maps?

A: Page tables and inode maps are similar in that they are both block maps for locating data given block offsets in a logical storage object. A VAS is a logical storage object: a space of sequentially numbered pages/blocks that could be stored anywhere in memory. A file is a logical storage object: a space of sequentially numbered blocks that could be stored anywhere on disk.

Both kinds of logical storage objects can be large, and both kinds can be sparse. And, no surprise, the data structures they use are almost identical: a tree-structured map. There are some differences too, and you should understand why they exist.

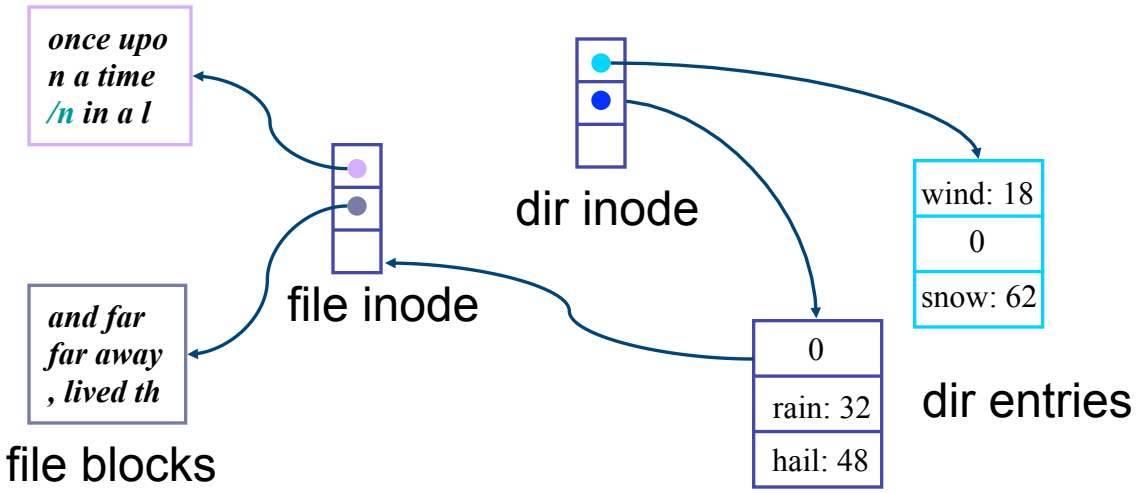
One key difference: the **pointers in an inode block map are disk addresses**. The map exists to find data on “disk”. Note: that is true for all of the file system metadata structures. File system code must read the metadata structures from disk into memory in order to find and access files, and then modify the metadata and write it back as files are created and destroyed and shrink and grow, to keep track of files and their names, locations, and properties.

In contrast, the pointers in a page table are "physical" memory addresses. The map exists to find data in memory.

Inode block maps are “skewed” because they are optimized for small files and dense files. Most files are written sequentially (they are “dense” with no “holes”), and most files are small. For small files the inode map is very compact, yet it can grow (by adding indirect blocks) as the file grows. Page tables are optimized for sparseness of VAS: a VAS is a collection of segments that may be widely separated, with empty regions (“holes”) between them. The tree structure is compact because we don’t need to allocate maps for the holes: just leave the branch empty.

Safety of metadata

- **How to protect integrity of the metadata structures?**
  - Metadata is a complex linked data structure, e.g., a tree.
  - Must be "**well-formed**" after a crash/restart, even if writes are lost.
  - …or, must be possible to restore metadata to a consistent state with a scrub (file system check or "fsck") on restart after a crash.

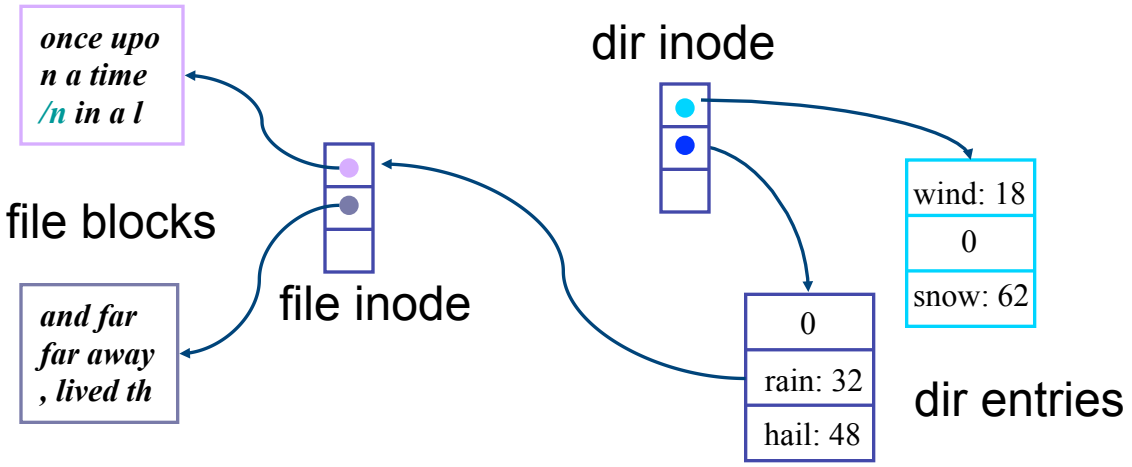



The disk was not ejected properly. If possible, always eject a disk before unplugging it or turning it off.
To eject a disk, select it in the Finder and choose File > Eject. The next time you connect the disk, Mac OS X will attempt to repair any damage to the information on the disk.
OK

Atomic updates: the recovery problem

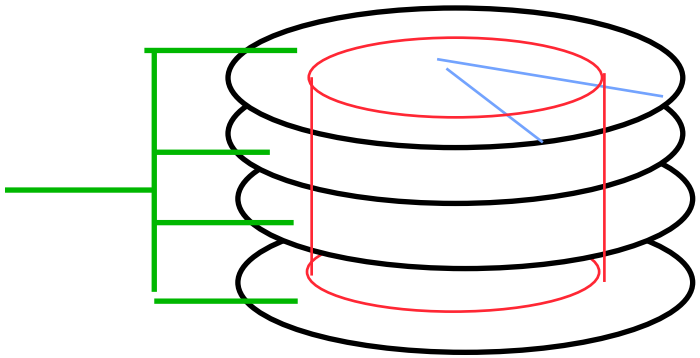
The safe metadata update problem in file systems is a simplified form of the **atomic update and recovery problem** for databases.

- We want to make a group of related updates to a complex linked data structure, e.g., to create a new file. The updates could be all over the disk.
- But we could crash at any time, e.g., in the middle of the group of updates.
- We need some way to do **atomic commit**: either all of the updates in each group complete, or none of them do. And we want it to be fast.
- The concern is similar to concurrency control: we don't want software to "see" an inconsistent state that violates structural invariants.



Disk write behavior (cartoon version)

- Disk may reorder pending writes.
  - Limited ordering support ("do X before Y").
  - Host software can enforce ordering by writing X **synchronously**: wait for write of X to complete before issuing Y.
- Writes at **sector** grain are atomic (512 bytes?).
- Writes of larger blocks may fail "in the middle".
- Disk may itself have a writeback cache.
  - Even "old" writes may be lost.
  - (The cache can be disabled.)



Atomic commit: shadowing

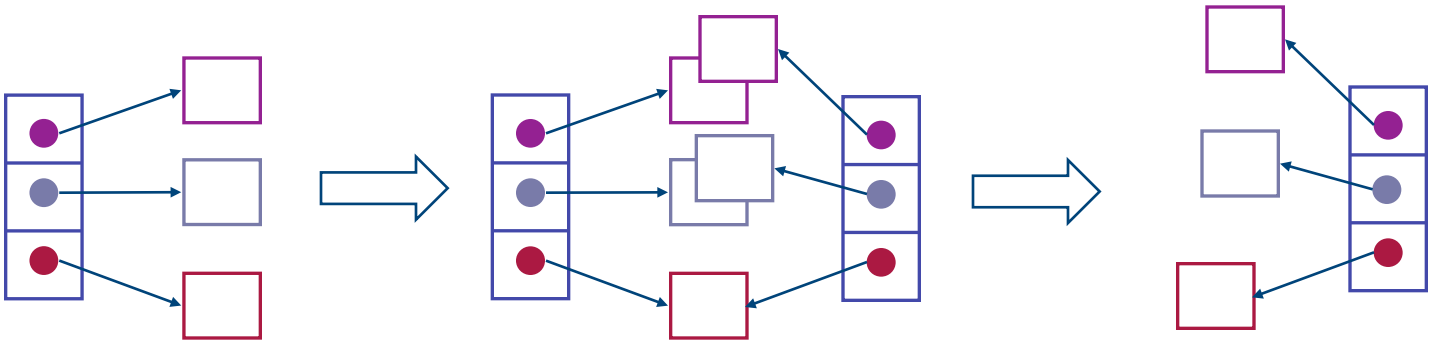
Shadowing is used in NetApp WAFL:

Write Anywhere File Layout

1. Write each modified block to a new location.
2. Update block maps to point to the new locations.
3. Write the new block map to disk with a single disk write.

Shadowing presumes that the data is mapped by a block map on disk, that the disk is large enough to store both the old version and the modified data, and that we can update the block map on disk with a single (atomic) disk write. To grow the block map we can make it hierarchical.

Shadowing



1. starting point
modify purple/grey blocks

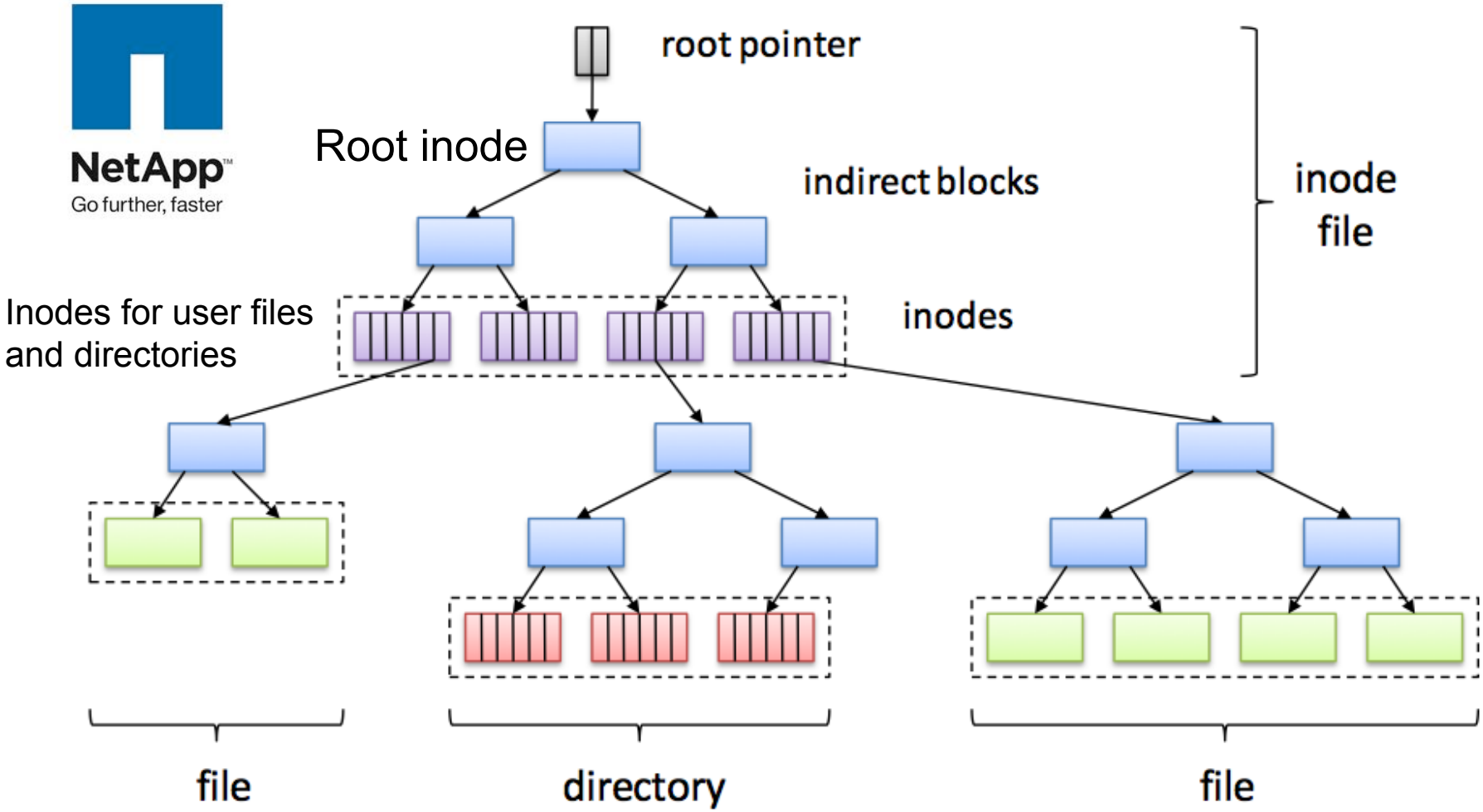
2. write new blocks to disk
prepare new block map

3. write new block map
(**atomic commit**)
and free old blocks
(optional)

Shadowing is a basic technique for atomic commit and recovery. It is used in WAFL.

Just to spell it out: if the system crashes before step 3, then the update fails, but the previous version is still intact. To abort the failed update we just need to free any blocks written in step 2. Step 3 completes the update: it replaces the old map with the new. Because it is a single disk write, the system cannot fail “in the middle”: it either completes or it does not: it is atomic. Once it is complete, the new data is safe.

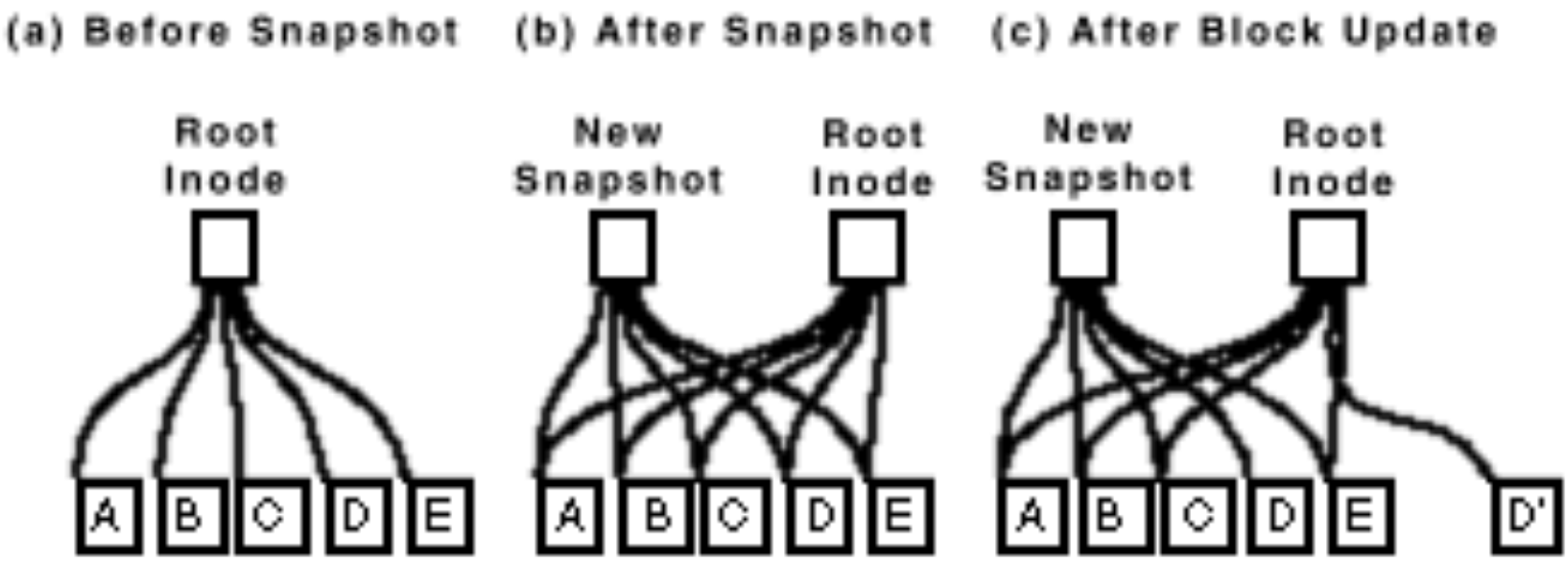
On-disk metadata structures
Write Anywhere File Layout (WAFL)



WAFL and Writes

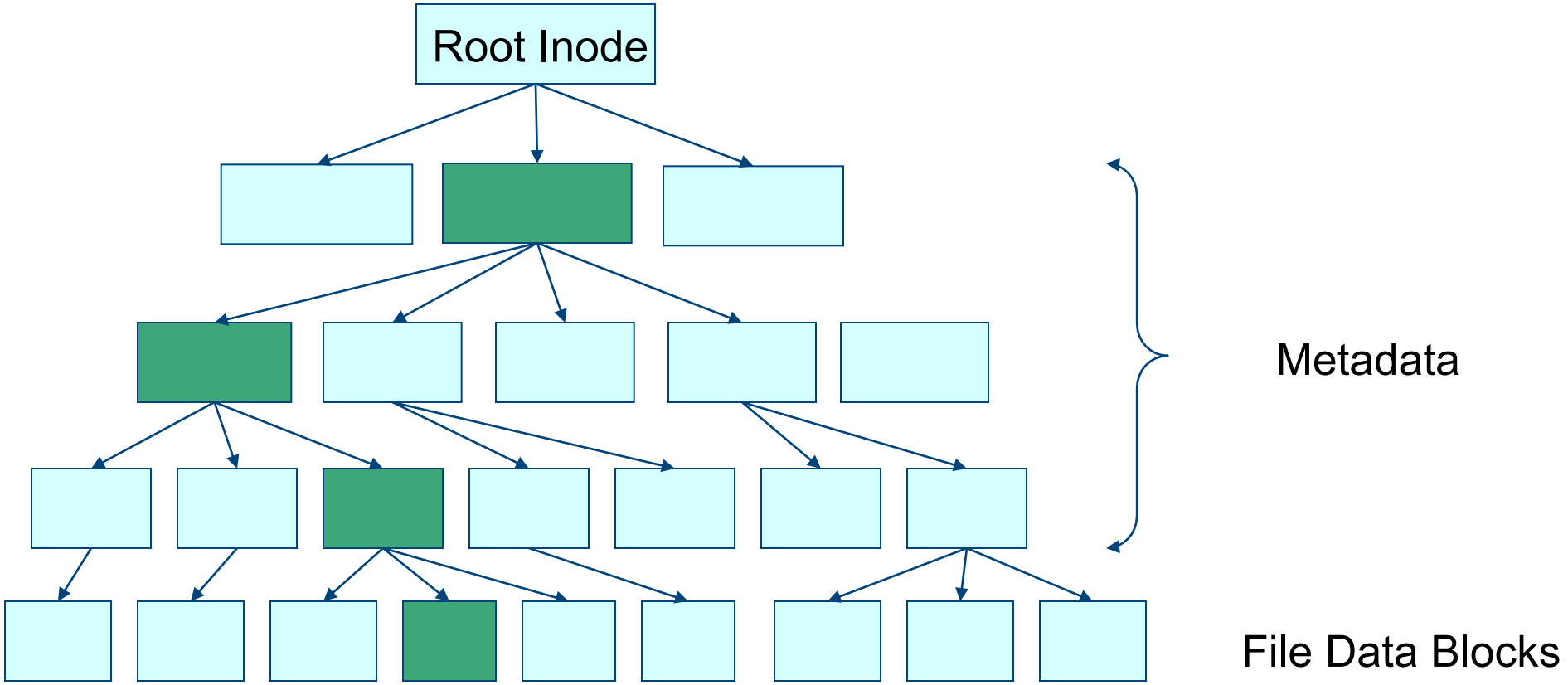
- Any modified data/metadata can go anywhere on the disk.
  - The WAFL metadata structure assures this: every piece of metadata is linked in a tree rooted in the root pointer.
- An arbitrary stream of updates can be installed atomically.
  - Retain the old copy: “no overwrite”
  - Switch to new copy with a single write to the root (shadowing).
- WAFL’s design naturally maintains multiple point-in-time consistent snapshots of each file volume.
  - Old copy lives on as a point-in-time snapshot.

WAFL Snapshots



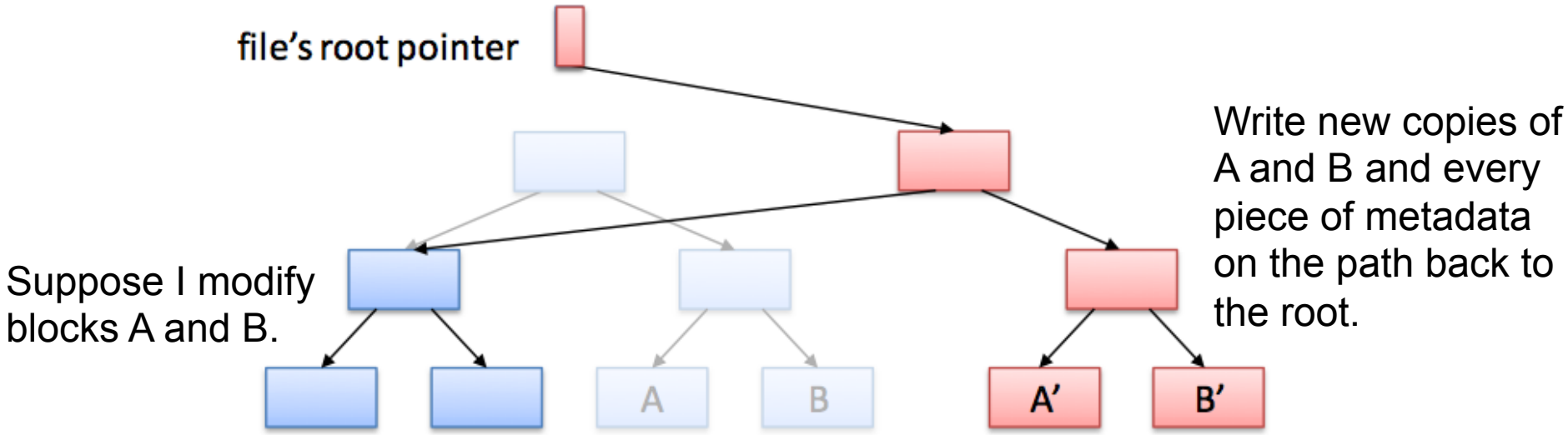
The snapshot mechanism is used for user-accessible snapshots and for transient “consistency points”.

WAFL's on-disk structure (high level)



Another Look

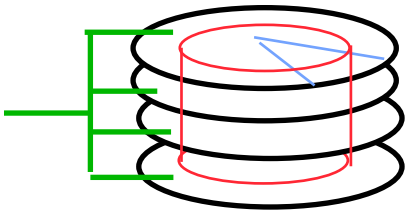
- Use copy-on-write up to root of file system



- Any change requires bubbling to the FS root

Storage system performance

- **How to get good storage performance?**
  - Build better disks: new technology, SSD hybrids.
  - Gang disks together into arrays (RAID logical devices).
  - Smart disk head scheduling (when there is a pool of pending requests to choose from).
  - Smarter caching: better victim selection policies
  - Asynchronous I/O: prefetching, read ahead, “write behind”
  - Location, location, location: smart block placement
- **It’s a big part of the technology of storage systems.**



Building better file systems

- The 1990s was a period of experimentation with new strategies for high-performance file system design.
- The new file systems generally used the FFS mechanisms and data structures, but changed the policies for block allocation.
  - Block allocation policy: where to place new data (or modified old data) on the storage volume? Which block number to choose?
  - *"File system design is 99% block allocation."* - Larry McVoy
  - Example: Group large-file data into big contiguous chunks called clusters or extents that can be read or written as a unit (larger b). [McVoy91] and [Smith/Seltzer96]
  - Example: Write modified data and metadata wherever convenient to minimize seeking: e.g., "log-structured" file systems (LFS) [Rosenblum91] and NetApp's WAFL [Hitz95]. Note: requires a level of indirection so the FS can write each version of an inode to a different location on the disk. (See WAFL's inode file.)

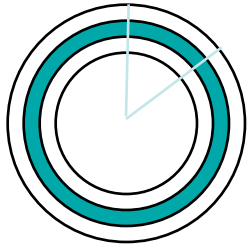
WAFL and the disk system

- WAFL generates a continuous stream of large-chunk contiguous writes to the disk system.
  - WAFL does not overwrite the old copy of a modified structure: it can write a new copy anywhere. So it gathers them together.
- Large writes minimize seek overhead and deliver the full bandwidth of the disk.
- WAFL gets excellent performance by/when using many disks in tandem ("**RAID**")…
- …and writing the chunks in interleaved fashion across the disks ("**striping**").
- Old copies of the data and metadata survive on the disk and are accessible through point-in-time "**snapshots**".

Block placement and layout

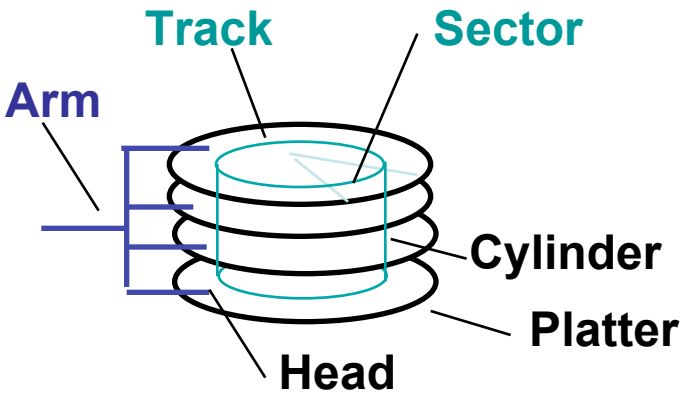
- One key assumption: **“seeks waste time”**.
  - Blocks whose addresses (logical block numbers) are close together are cheaper to access together.
  - “Sequentialize!”
- Location, location, location:
  - Place data on disk carefully to keep related items close together (smart block allocation).
  - Use larger b (larger blocks, clustering, extents, etc.)
  - Smaller s (placement / ordering, sequential access, logging, etc.)

Access time

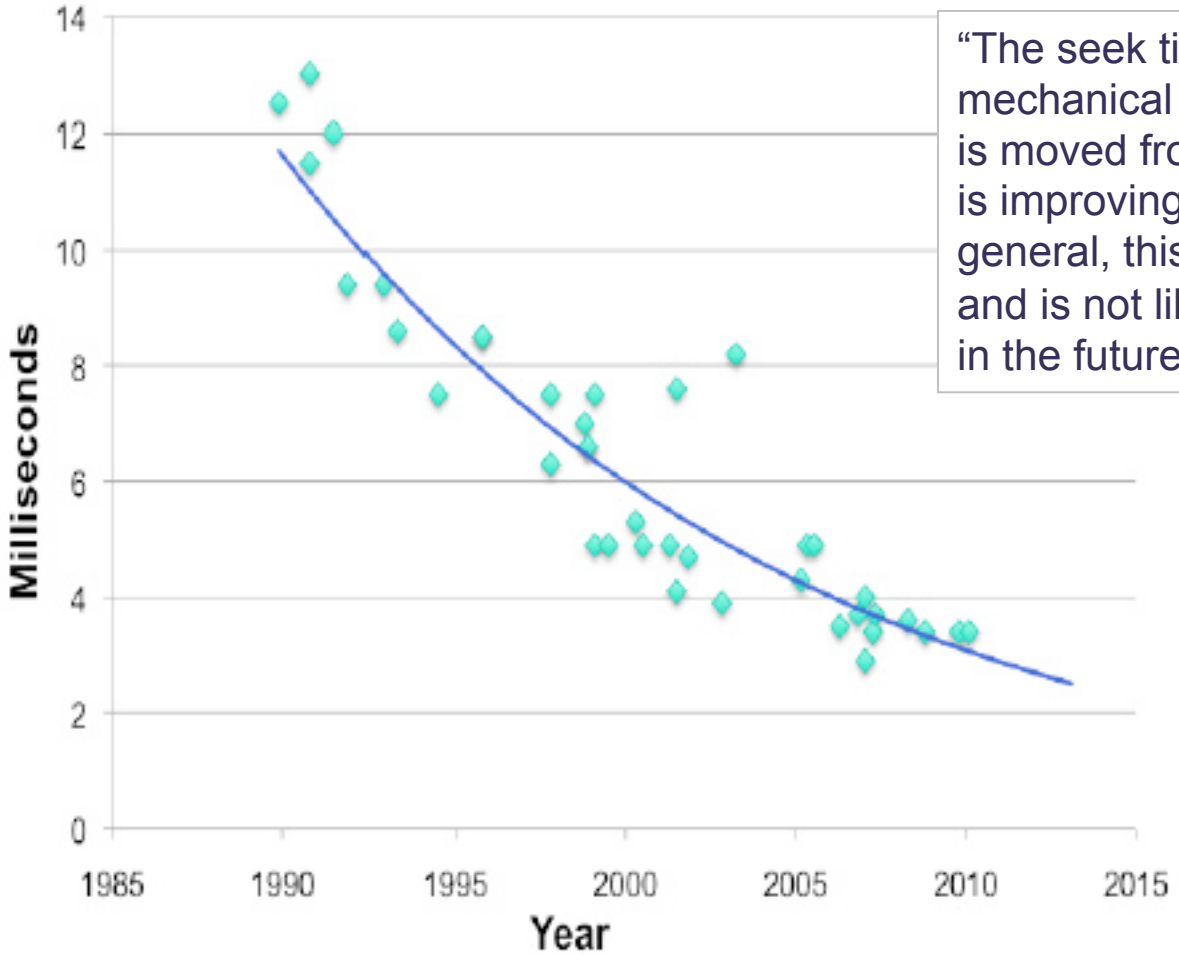


How long to access data on disk?

- 5-15 ms on average for access to random location
- Includes **seek time** to move head to desired track
  - Roughly linear with radial distance
- Includes **rotational delay**
  - Time for sector to rotate under head
- These times depend on drive model:
  - platter width (e.g., 2.5 in vs. 3.5 in)
  - rotation rate (5400 RPM vs. 15K RPM).
  - Enterprise drives use more/smaller platters spinning faster.
- These properties are mechanical and improve slowly as technology advances over time.



Average seek time



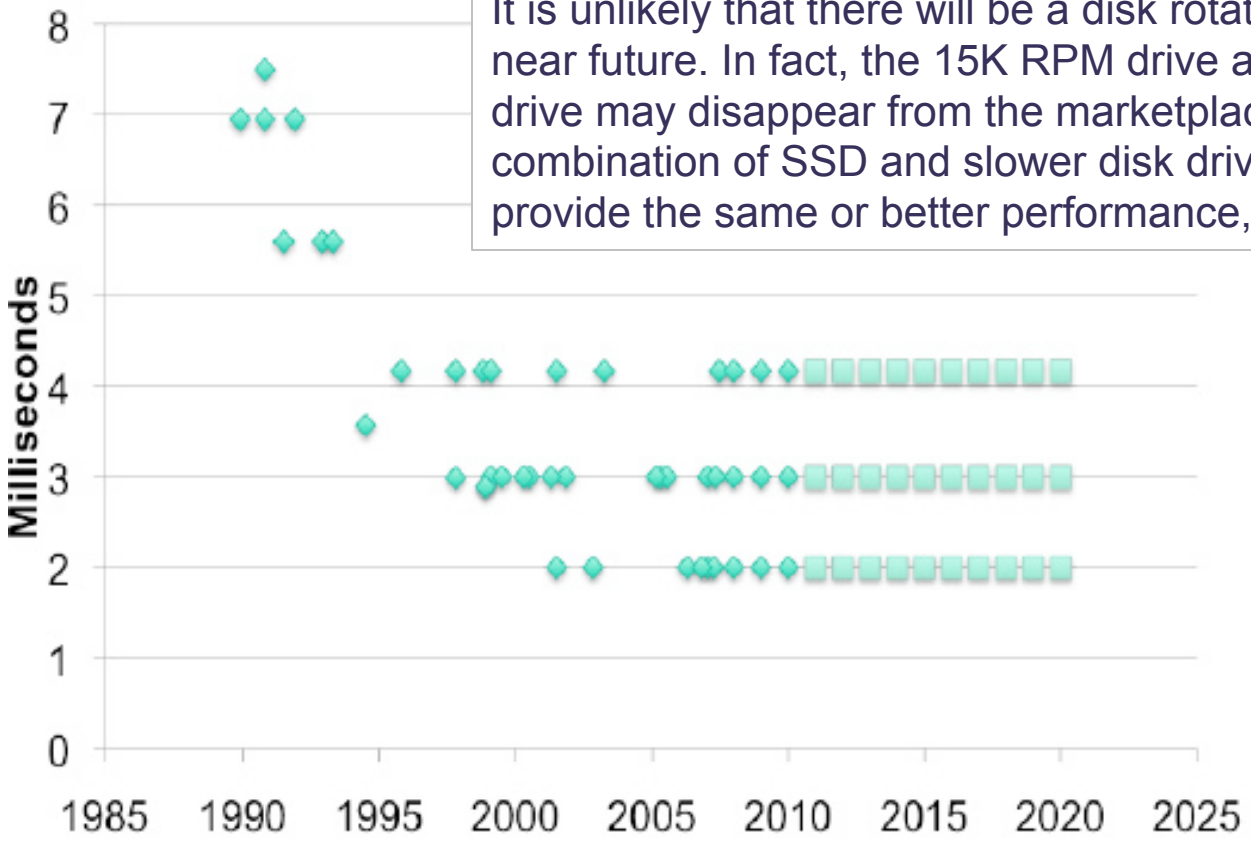
"The seek time is due to the mechanical motion of the head when it is moved from one track to another. It is improving by about 5% CAGR. In general, this is a mature technology and is not likely to change dramatically in the future. "

IBM Research Report 2011
GPFS Scans 10 Billion
Files in 43 Minutes

Rotational latency

The average disk latency is ½ the rotational time of the disk drive. As you can see from its recent history…[it] has settled down to three values 2, 3 and 4.1 milliseconds. These are ½ the inverses of 15,000, 10,000 and 7,200 revolutions per minute (RPM), respectively.

It is unlikely that there will be a disk rotational speed increase in the near future. In fact, the 15K RPM drive and perhaps the 10K RPM drive may disappear from the marketplace…driven by the successful combination of SSD and slower disk drives into storage systems that provide the same or better performance, cost and power.



Drives spin at a fixed constant RPM. (A few can “shift gears” to save power, but the gains are minimal.)

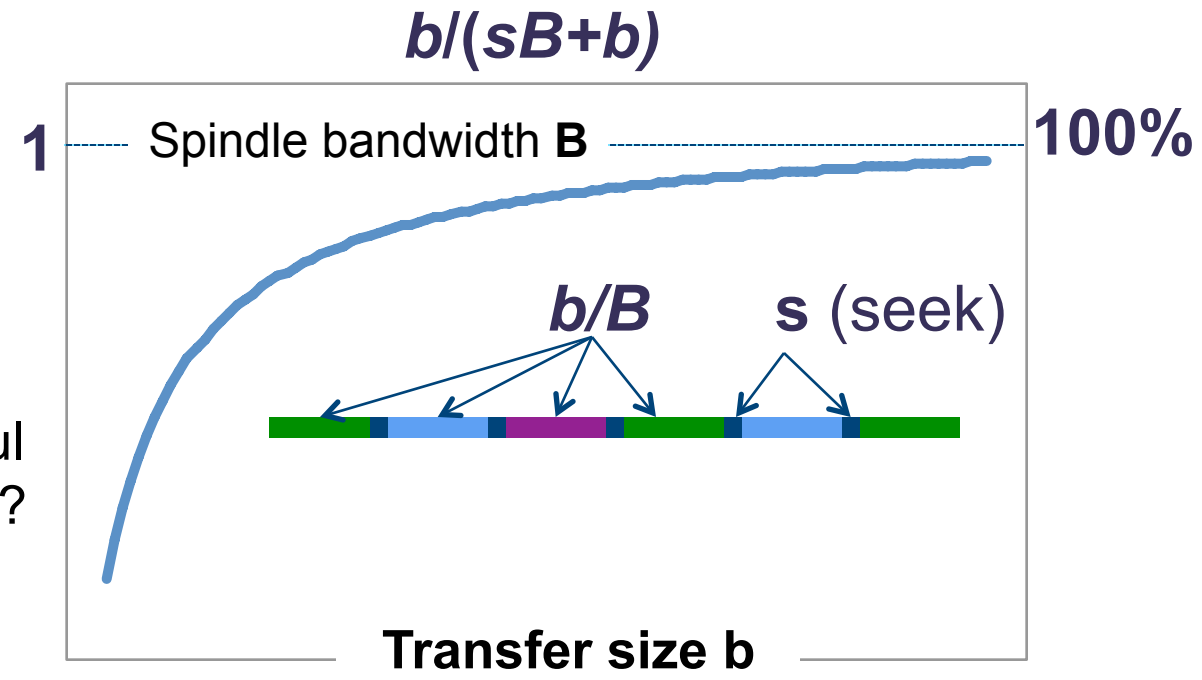
IBM Research Report 2011
GPFS Scans 10 Billion Files in 43 Minutes

Effective bandwidth

Seeks are overhead: "wasted effort". It is a cost **s** that the device imposes to get to the data. It is not actually transferring data.

This graph is obvious. It applies to so many things in computer systems and in life.

Effective bandwidth is efficiency or goodput
What percentage of the time is the busy resource (the disk head) doing useful work, i.e., transferring data?



Effective bandwidth

Effective bandwidth or **bandwidth utilization** is the share or percentage of potential bandwidth that is actually delivered. *E.g., what percentage of time is the disk actually transferring data, vs. seeking etc.?*

Define

b Block size

B Raw disk bandwidth ("spindle speed")

s Average access (seek+rotation) delay per block I/O

Then

Transfer time per block = b/B

I/O completion time per block = s + (b/B)

Delivered bandwidth for I/O request stream = bytes/time = b/(s + (b/B))

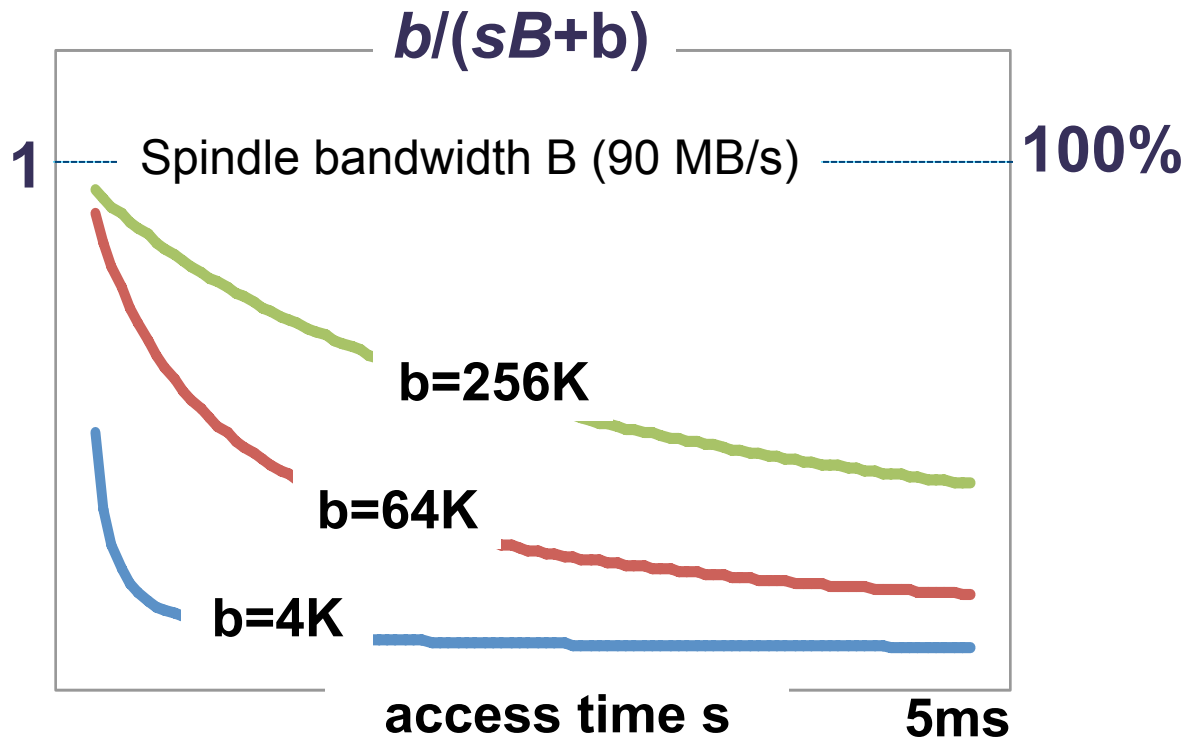
Bandwidth wasted per I/O: sB

So

Effective bandwidth: bandwidth utilization/efficiency (%): b/(sB + b)

[bytes transferred over the "byte time slots" consumed for the transfer]

Effective bandwidth by access time



Bigger is better. Other things being equal, effective bandwidth is higher when access costs can be amortized over larger transfers. High access cost is the reason we use tapes primarily for backup! As **B** grows and **s** is unchanged, disks are looking more and more like tapes! (Jim Gray)