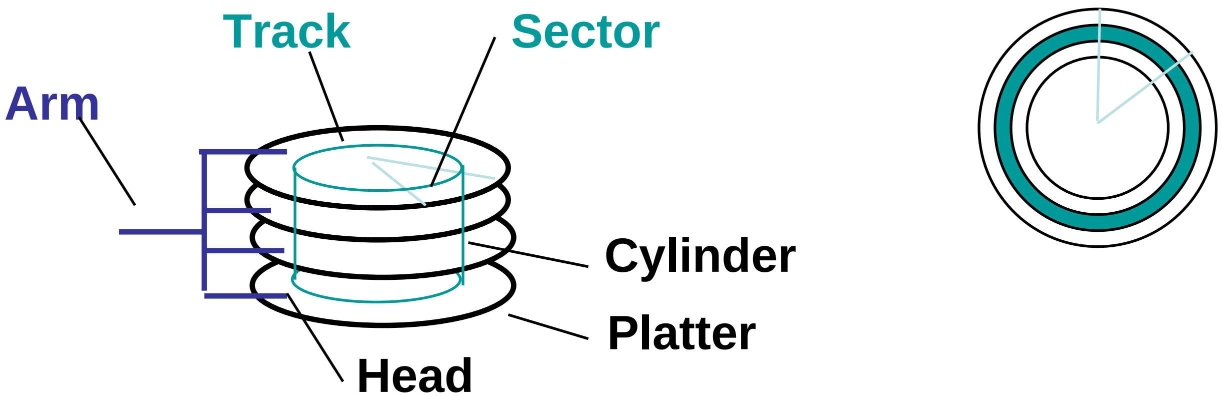
Storage

Jeff Chase
Duke University

Storage: The Big Issues

1. Disks are rotational media with mechanical arms.
   - High access cost →caching and prefetching
   - Cost depends on previous access → careful block placement and scheduling.
2. Stored data is *hard state*.
   - Stored data *persists* after a restart.
   - Data corruption and poor allocations also persist.
   - →Allocate for longevity, and write carefully.
3. Disks fail.
   - Plan for failure → redundancy and replication.
   - RAID: integrate redundancy with striping across multiple disks for higher throughput.

Rotational Media

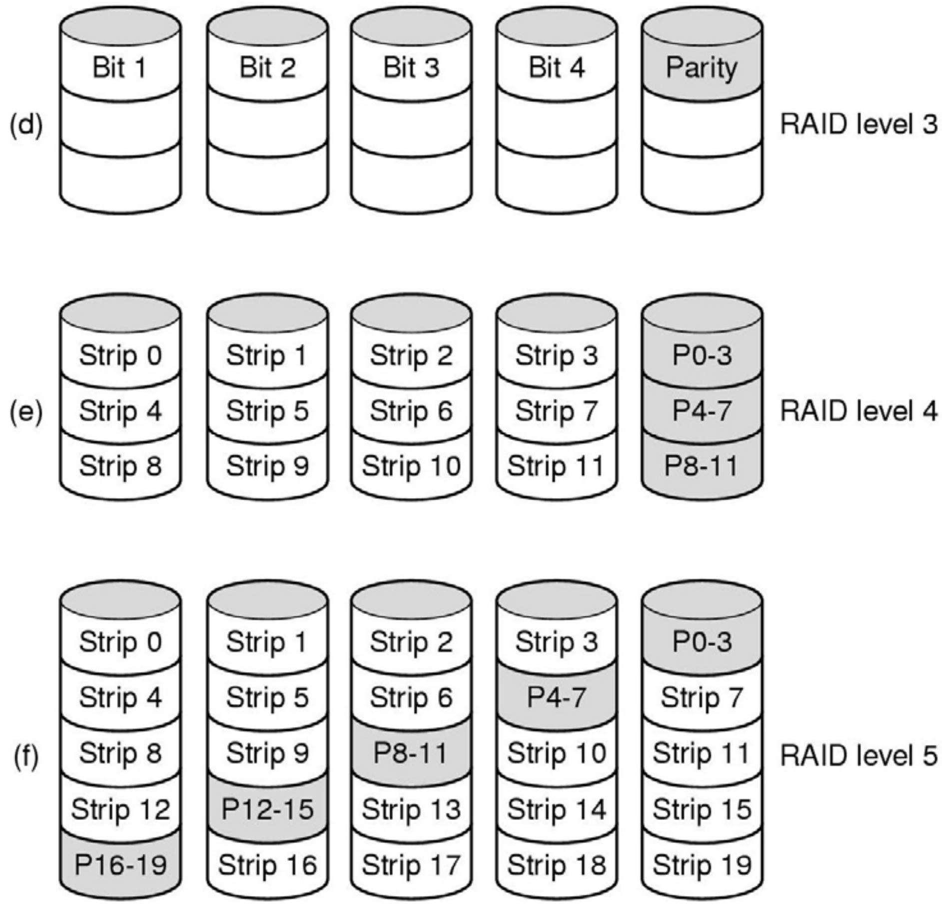


<u>Access time = seek time + rotational delay + transfer time</u>

seek time = 5-15 milliseconds to move the disk arm and settle on a cylinder
rotational delay = 8 milliseconds for full rotation at 7200 RPM: average delay = 4 ms
transfer time = 1 millisecond for an 8KB block at 8 MB/s

Bandwidth utilization is less than 50% for any noncontiguous access at a block grain.

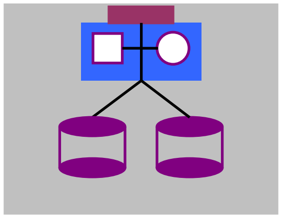
RAID



- Raid levels 3 through 5
- Backup and parity drives are shaded

Disks and Drivers

- Disk hardware and driver software provide foundational support for *block devices*.
- OS views the block devices as a collection of *volumes*.
  - A logical volume may be a *partition* of a single disk or a *concatenation* of multiple physical disks (e.g., RAID).
    - volume == LUN
- Each volume is an array of fixed-size *sectors*.
  - Name sector/block by *(volumeID, sector ID)*.
  - *Read/write* operations DMA data to/from physical memory.
- Device interrupts OS on I/O completion.
  - ISR wakes process, updates internal records, etc.

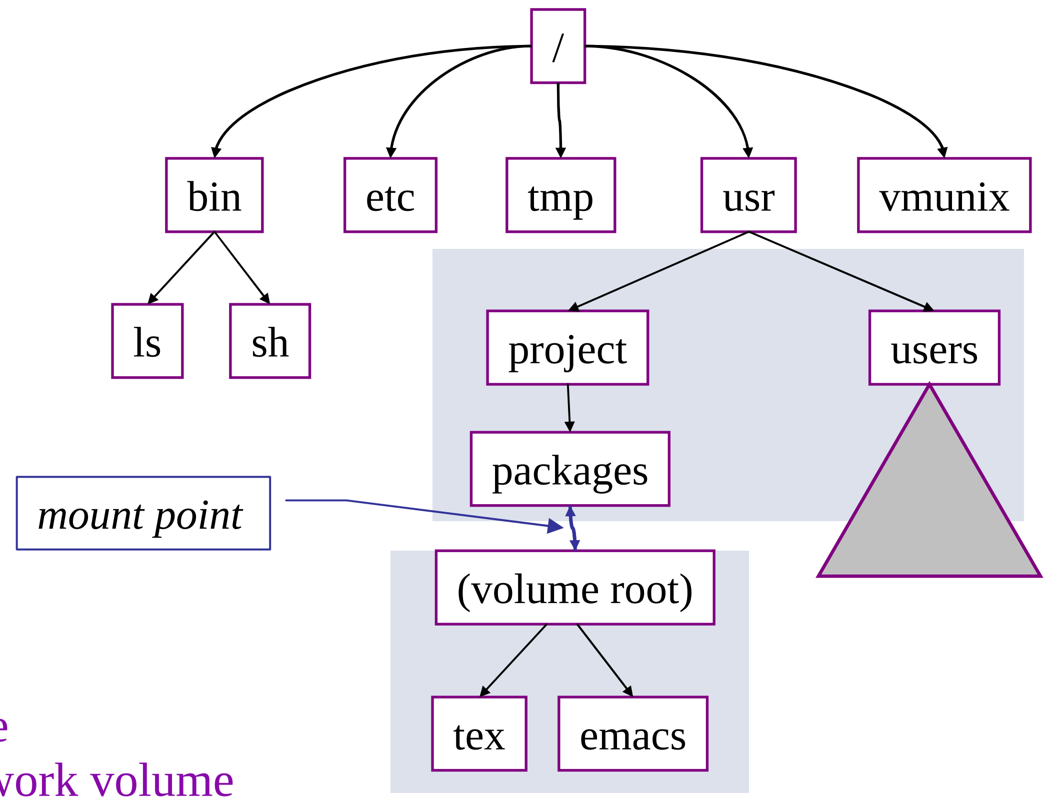


A Typical Unix File Tree

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

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

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



mount (coveredDir, volume)
coveredDir: directory pathname
volume: device specifier or network volume
volume root contents become visible at pathname ***coveredDir***

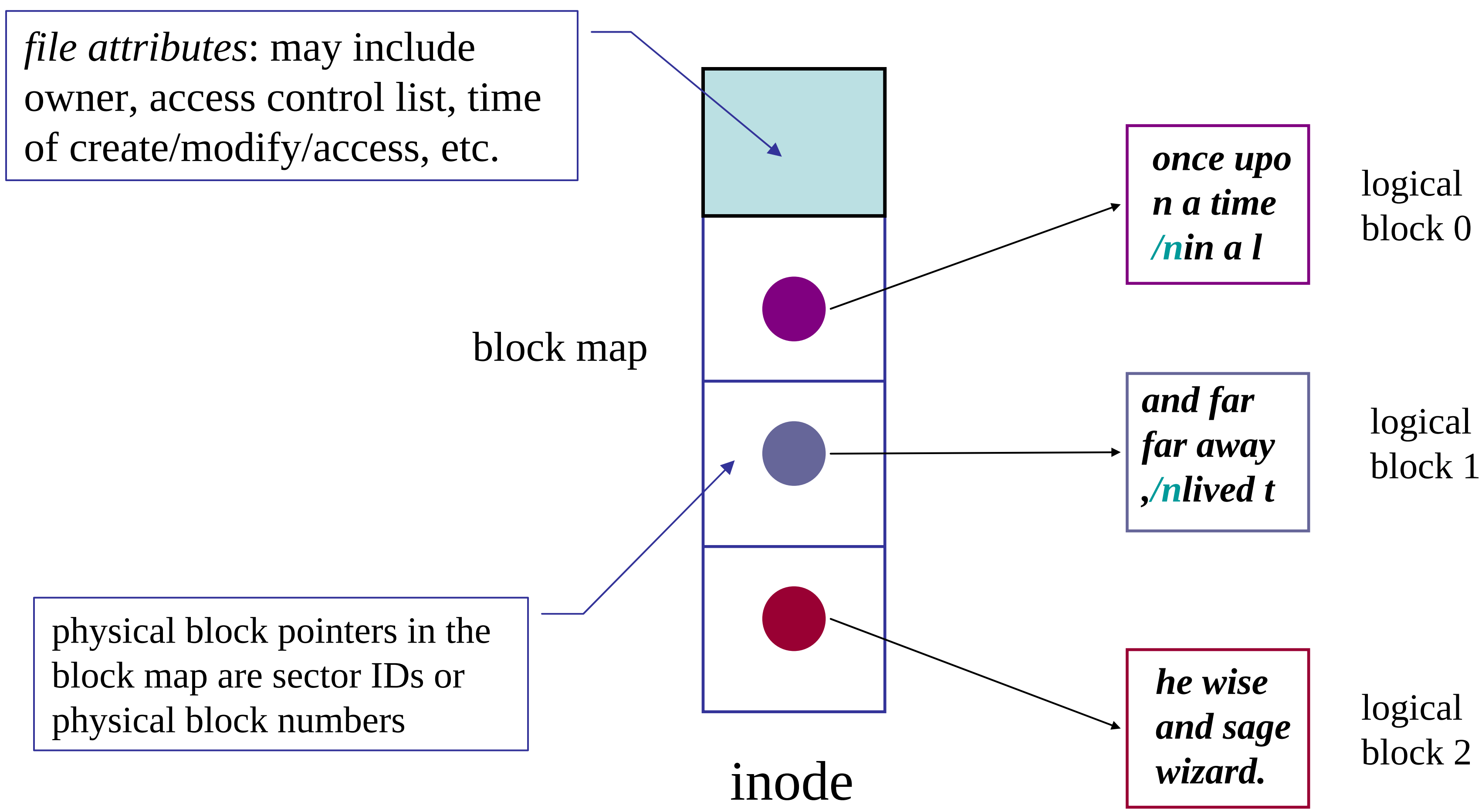
Filesystems

- Files
  - Sequentially numbered bytes or *logical blocks*.
  - Metadata stored in on-disk data object
    - e.g, Unix "inode"
- Directories
  - A special kind of file with a set of name mappings.
    - E.g., name to inode
  - Pointer to parent in rooted hierarchy: .., /
- System calls
  - Unix: *open, close, read, write, stat, seek, sync, link, unlink, symlink, chdir, chroot, mount, chmod, chown.*

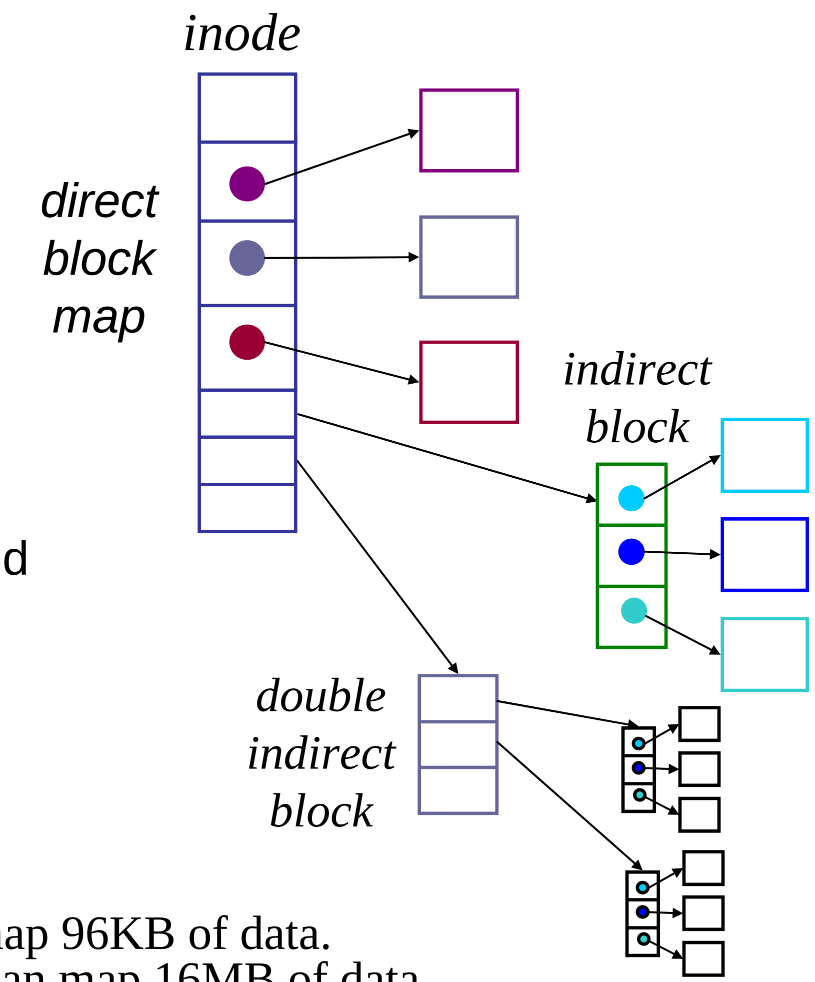
File Systems: The Big Issues

- Buffering disk data for access from the processor.
  - Block I/O (DMA) needs aligned physical buffers.
  - Block update is a read-modify-write.
- Creating/representing/destroying independent files.
- Allocating disk blocks and scheduling disk operations to deliver the best performance for the I/O stream.
  - What are the patterns in the request stream?
- Multiple levels of name translation.
  - Pathname→inode, logical→physical block
- Reliability and the handling of updates.

Representing a File On Disk



Representing Large Files



Classical Unix

Each file system block is a clump of sectors (4KB, 8KB, 16KB).

Inode == 128 bytes, packed into blocks.

Each inode has 68 bytes of attributes and 15 block map entries.

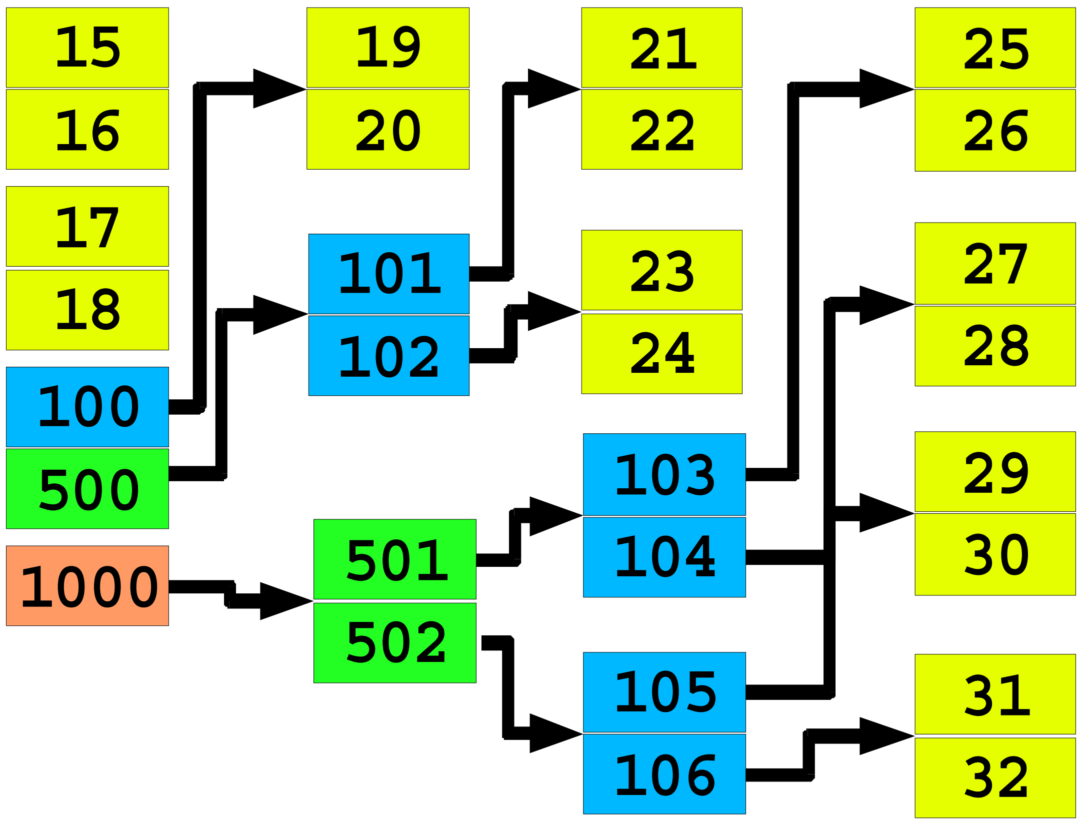
Suppose block size = 8KB
12 direct block map entries in the inode can map 96KB of data.
One indirect block (referenced by the inode) can map 16MB of data.
One double indirect block pointer in inode maps 2K indirect blocks.

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

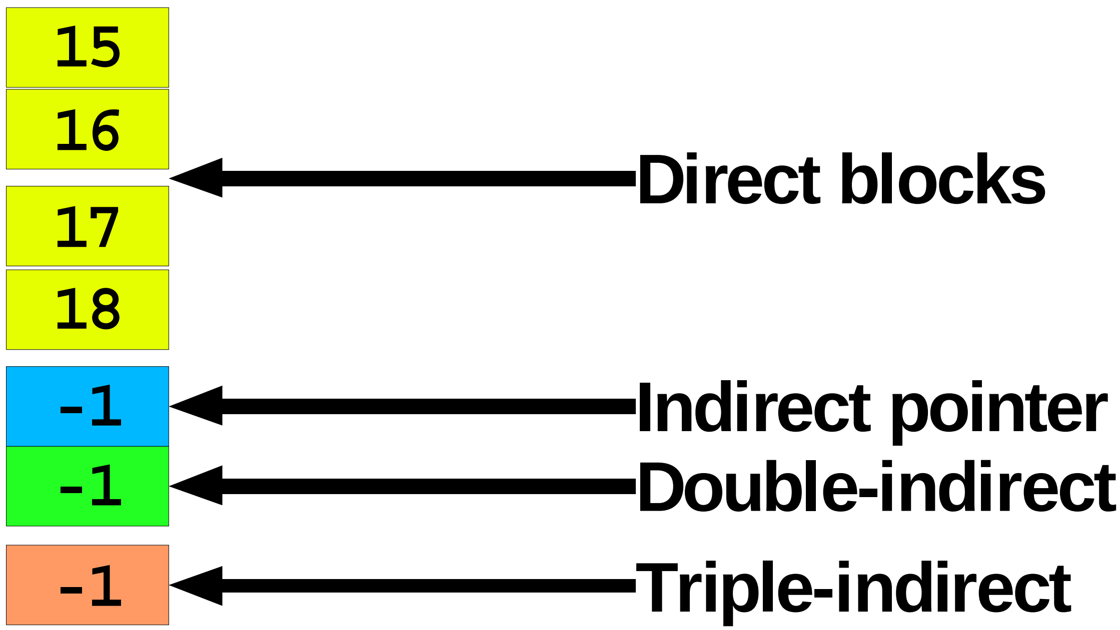
Unix index blocks

- Intuition
  - *Many* files are small
    - Length = 0, length = 1, length < 80, ...
  - Some files are *huge* ***(3 gigabytes)***
- "Clever heuristic" in Unix FFS inode
  - 12 (direct) block pointers: 12 * 8 KB = 96 KB
    - Availability is "free" - you need inode to open() file anyway
  - 3 indirect block pointers
    - single, double, triple

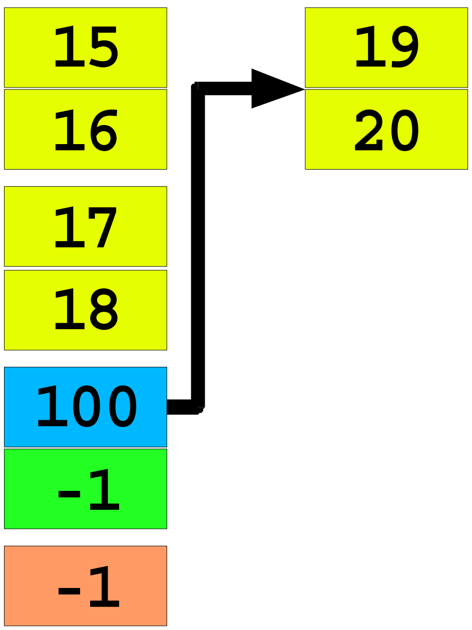
Unix index blocks



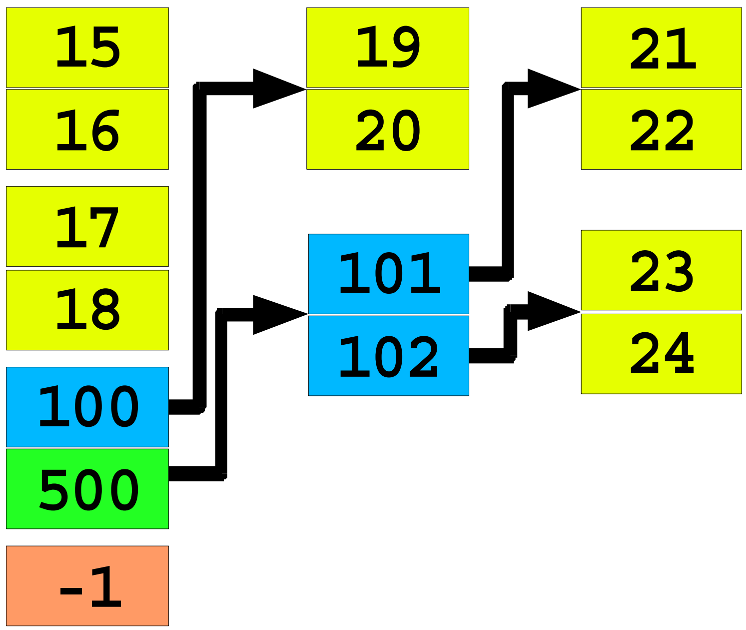
Unix index blocks



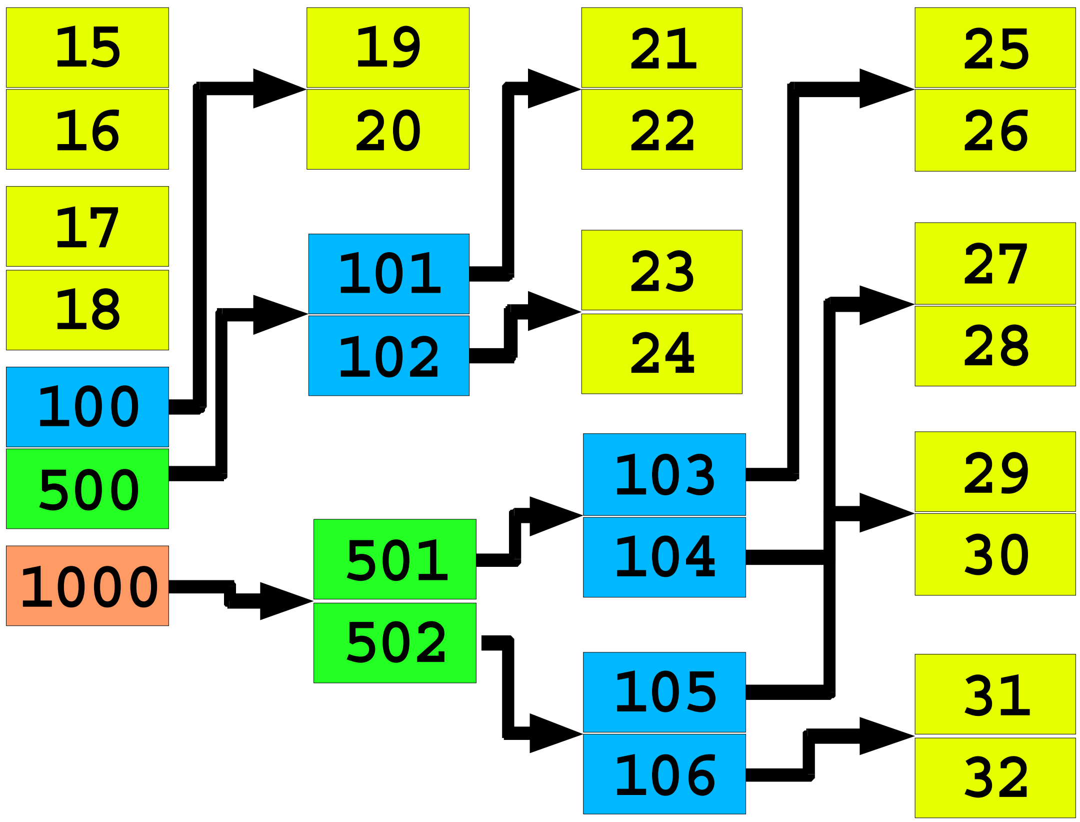
Unix index blocks



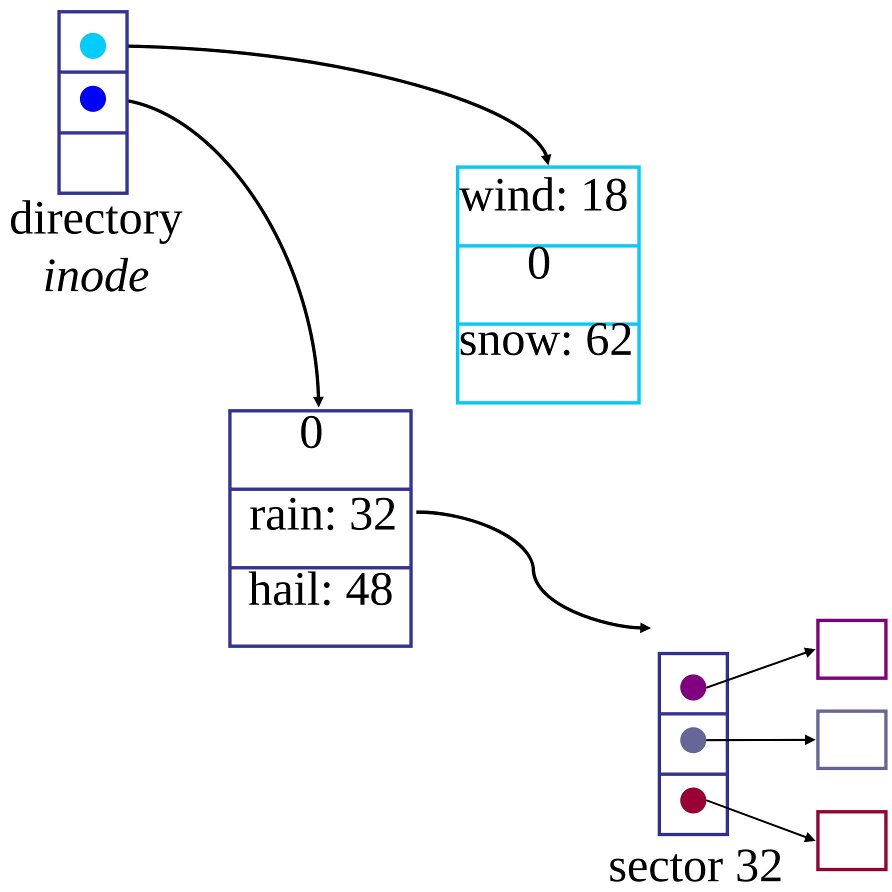
Unix index blocks



Unix index blocks

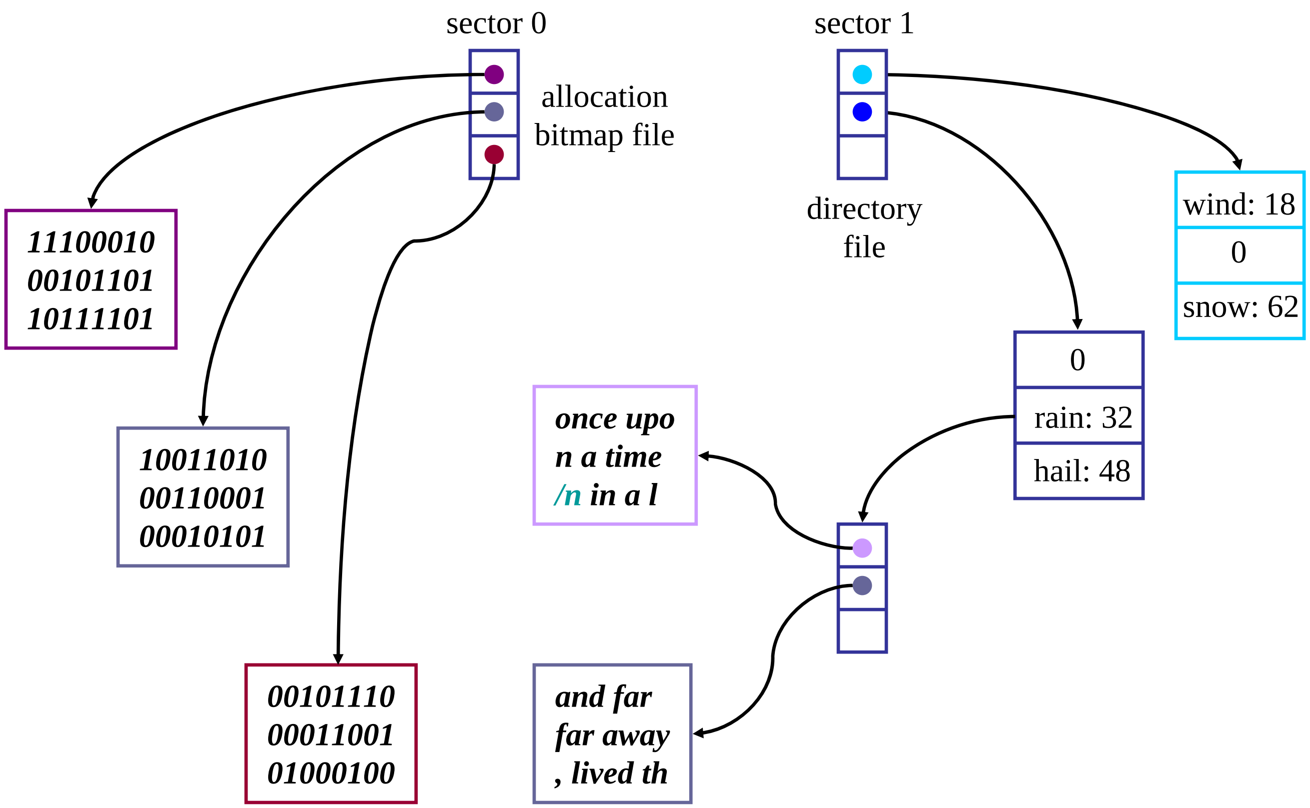


Directories



Entries or slots are found by a linear scan.

A Filesystem On Disk



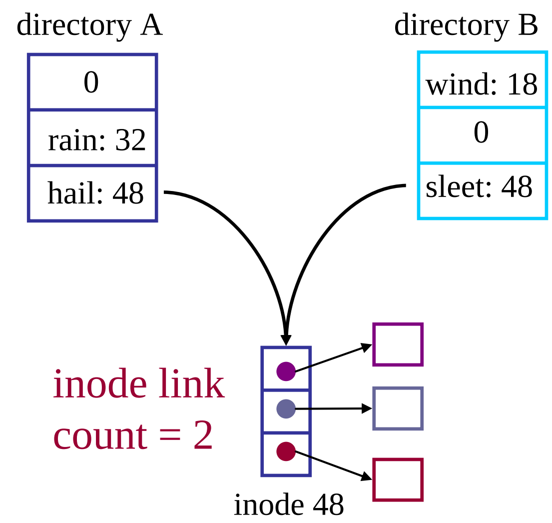
This is just an example (Nachos)

Unix File Naming (Hard Links)

A Unix file may have multiple names.

Each directory entry naming the file is called a *hard link*.

Each inode contains a *reference count* showing how many hard links name it.



<u>*link* system call</u>
link (existing name, new name)
create a new name for an existing file
increment inode link count

<u>*unlink* system call (“remove”)</u>
unlink(name)
destroy directory entry
decrement inode link count
if count == 0 and file is not in active use
free blocks (recursively) and on-disk inode

Illustrates: garbage collection by reference counting.

Unix Symbolic (Soft) Links

- A soft link is a file containing a pathname of some other file.

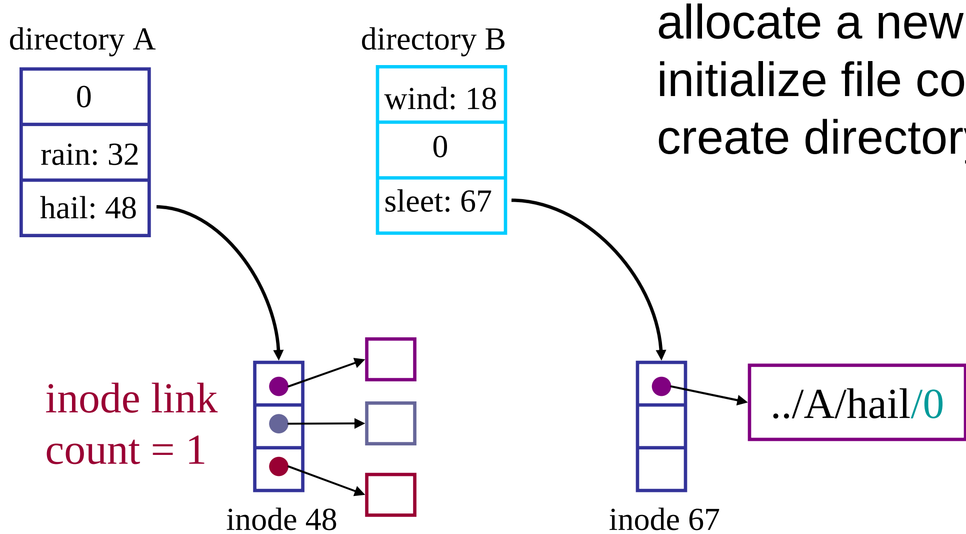
<u>*symlink* system call</u>

symlink (existing name, new name)

allocate a new file (inode) with type symlink

initialize file contents with *existing name*

create directory entry for new file with *new name*



The target of the link may be removed at any time, leaving a dangling reference.

How should the kernel handle recursive soft links?

Failures, Commits, Atomicity

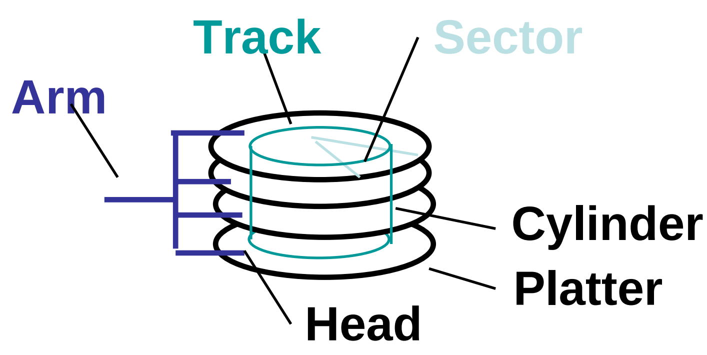
- What guarantees does the system offer about the hard state if the system fails?
  - Durability
    - Did my writes *commit*, i.e., are they on the disk?
  - Atomicity
    - Can an operation "partly commit"?
    - Also, can it interleave with other operations?
  - Recoverability and Corruption
    - Is the metadata well-formed on recovery?

Unix Failure/Atomicity

- File writes are not guaranteed to commit until close.
  - A process can force commit with a *sync*.
  - The system forces commit every (say) 30 seconds.
  - Failure could lose an arbitrary set of writes.
- Reads/writes to a shared file interleave at the granularity of system calls.
- Metadata writes are atomic/synchronous.
- Disk writes are carefully ordered.
  - The disk can become corrupt in well-defined ways.
  - Restore with a scrub ("fsck") on restart.
  - Alternatives: logging, shadowing
- Want better reliability? Use a database.

The Problem of Disk Layout

- The level of indirection in the file block maps allows flexibility in file layout.
    - "File system design is 99% block allocation." [McVoy]
- Competing goals for block allocation:
  - *allocation cost*
  - *bandwidth* for high-volume transfers
  - *stamina/longevity*
  - *efficient directory operations*
- **<u>Goal</u>**: reduce disk arm movement and seek overhead.
    - metric of merit: *bandwidth utilization*



Bandwidth utilization

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)

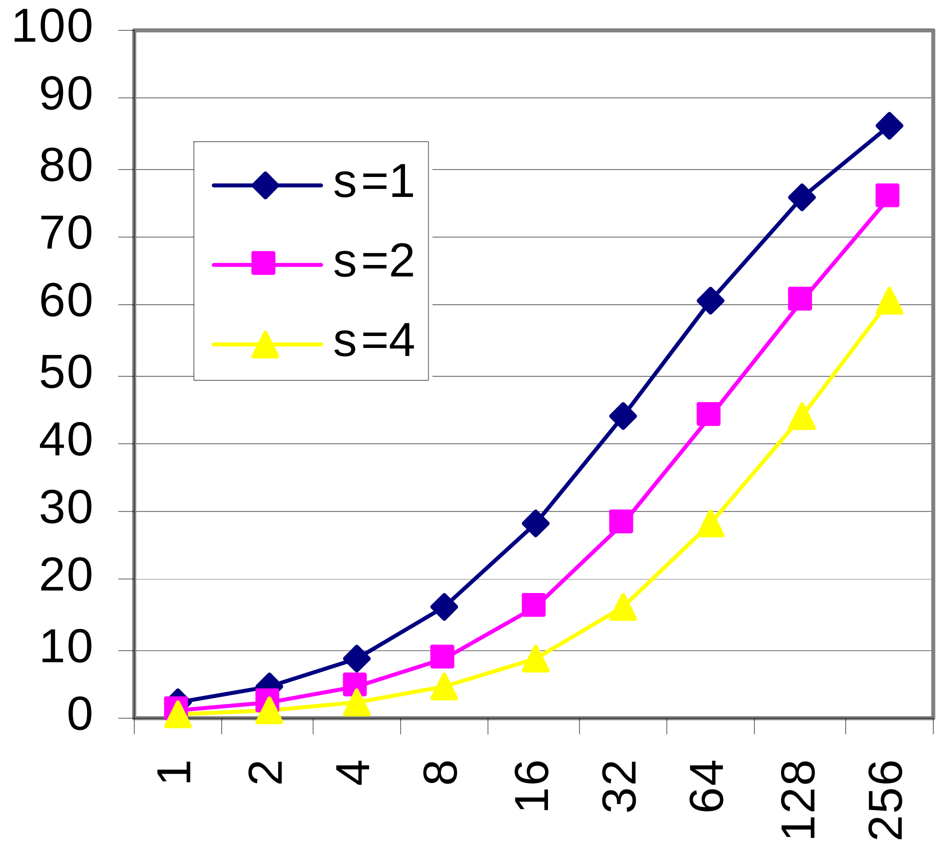
Effective disk bandwidth for I/O request stream = b/(s + (b/B))

Bandwidth wasted per I/O: sB

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

How to get better performance?

- Larger b (larger blocks, clustering, extents, etc.)
- Smaller s (placement / ordering, sequential access, logging, etc.)



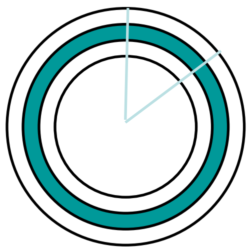
Effective bandwidth (%), B = 40 MB/s

Example: BSD FFS

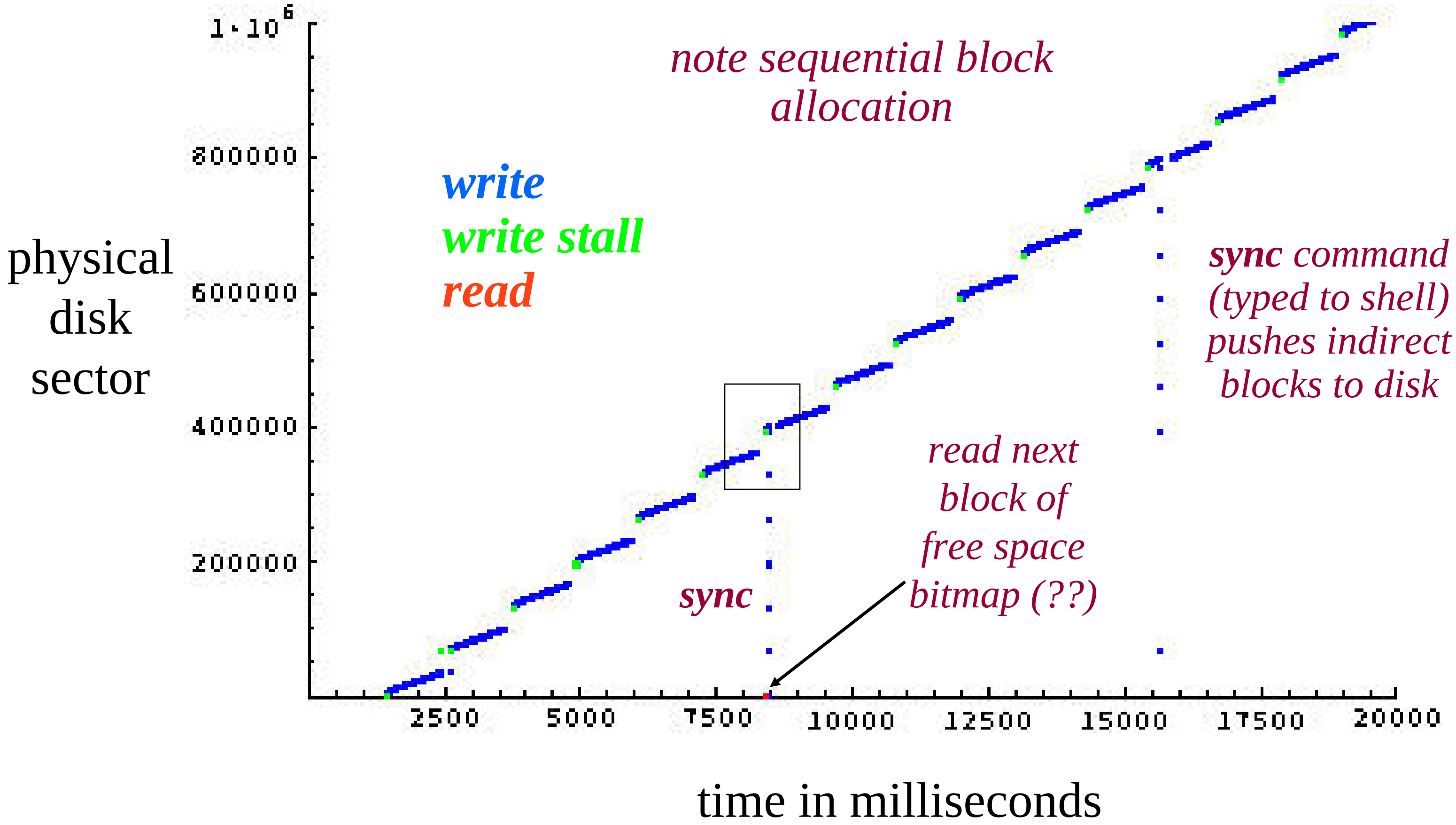
- Fast File System (FFS) [McKusick81]
  - Clustering enhancements [McVoy91], and improved cluster allocation [McKusick: Smith/Seltzer96]
  - FFS can also be extended with metadata logging [e.g., Episode]

FFS Cylinder Groups

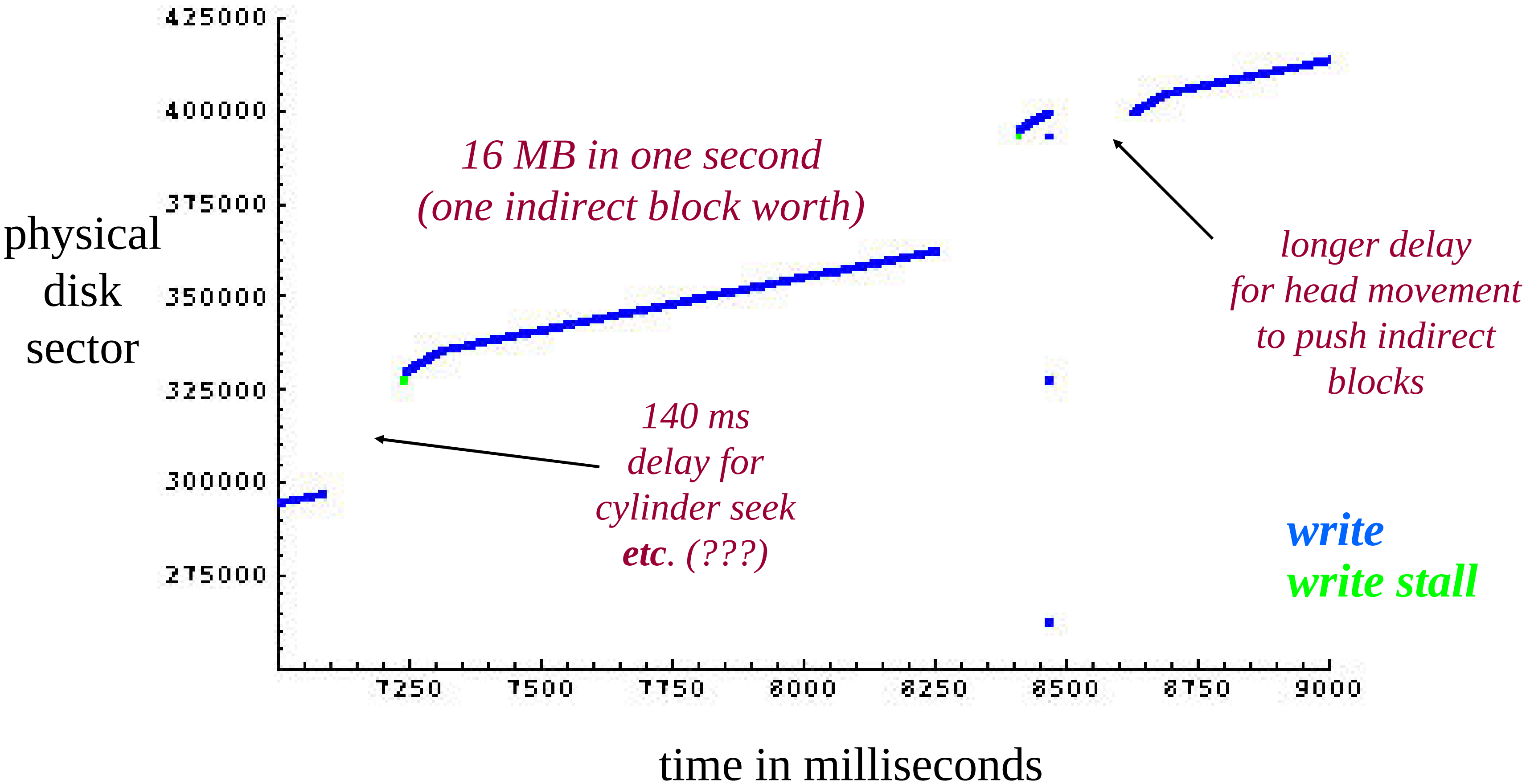
- FFS defines *cylinder groups* as the unit of disk locality, and it factors locality into allocation choices.
  - typical: thousands of cylinders, dozens of groups
  - Strategy: place "related" data blocks in the same cylinder group whenever possible.
    - seek latency is proportional to seek distance
  - Smear large files across groups:
    - Place a run of contiguous blocks in each group.
  - Reserve inode blocks in each cylinder group.
    - This allows inodes to be allocated close to their directory entries and close to their data blocks (for small files).



Sequential File Write



Sequential Writes: A Closer Look



The Problem of Metadata Updates

- Metadata updates are a second source of FFS seek overhead.
  - Metadata writes are poorly localized.
    - E.g., extending a file requires writes to the inode, direct and indirect blocks, cylinder group bit maps and summaries, and the file block itself.
- Metadata writes can be delayed, but this incurs a higher risk of file system corruption in a crash.
  - If you lose your metadata, you are dead in the water.
  - FFS schedules metadata block writes carefully to limit the kinds of inconsistencies that can occur.
    - Some metadata updates must be synchronous on controllers that don't respect order of writes.

FFS Failure Recovery

FFS uses a two-pronged approach to handling failures:

1. Carefully order metadata updates to ensure that no dangling references can exist on disk after a failure.
   - Never recycle a resource (block or inode) before zeroing all pointers to it (*truncate, unlink, rmdir*).
   - Never point to a structure before it has been initialized.
     - E.g., sync inode on *creat* before filling directory entry, and sync a new block before writing the block map.
2. Run a file system *scavenger* (*fsck*) to fix other problems.
   - Free blocks and inodes that are not referenced.
   - Fsck will never encounter a dangling reference or double allocation.

Alternative: Logging and Journaling

- *Logging* can be used to localize synchronous metadata writes, and reduce the work that must be done on recovery.
    - Universally used in database systems.
    - Used for metadata writes in journaling file systems
- Key idea: group each set of related updates into a single log record that can be written to disk *atomically* ("all-or-nothing").
    - Log records are written to the log file or log disk *sequentially*.
        - No seeks, and preserves temporal ordering.
    - Each log record is trailed by a marker (e.g., checksum) that says "this log record is complete".
    - To recover, scan the log and reapply updates.

Metadata Logging

Here's one approach to building a fast filesystem:

1. Start with FFS with clustering.
2. Make all metadata writes asynchronous.

But, that approach cannot survive a failure, so:

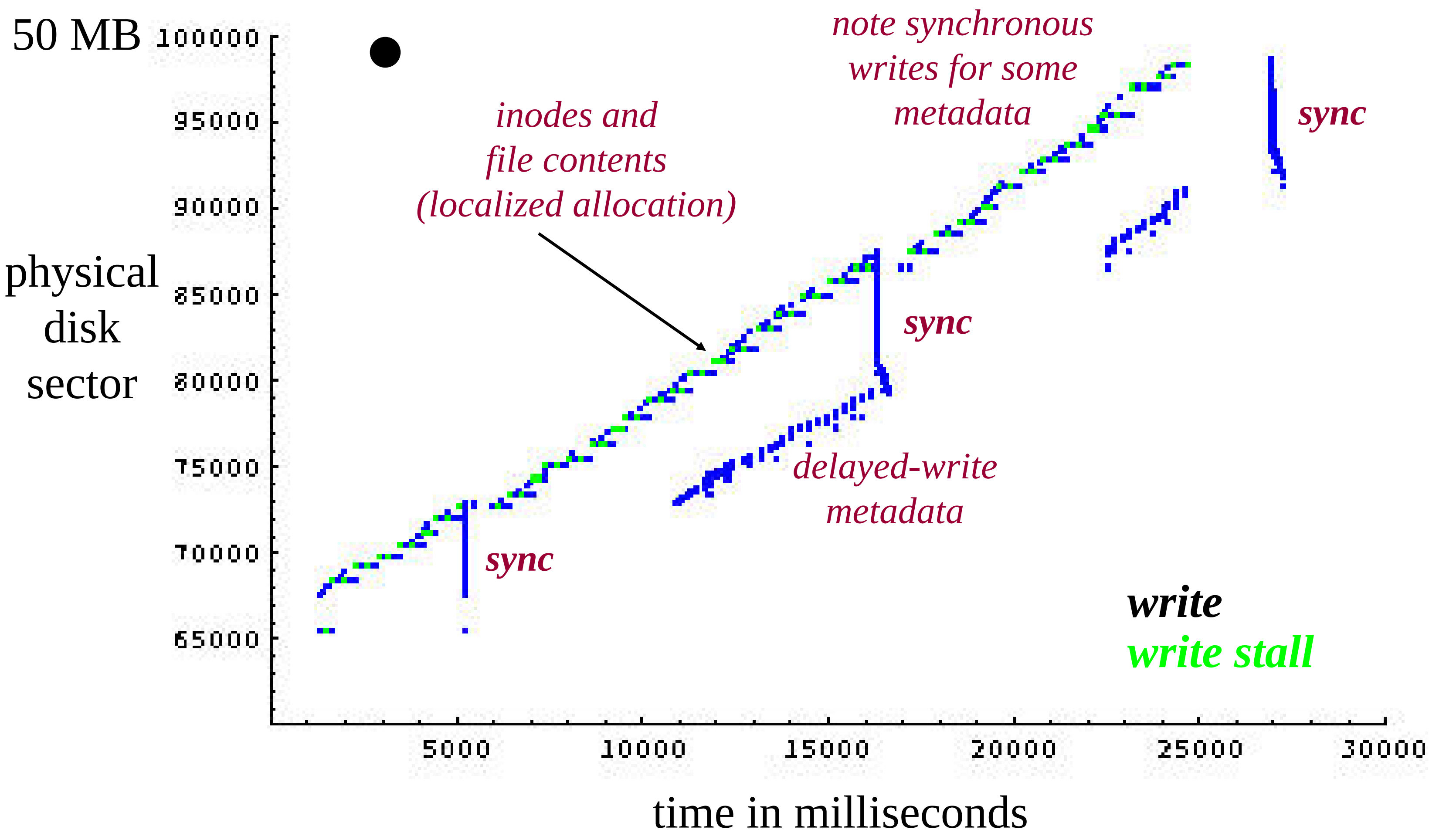
3. Add a supplementary log for modified metadata.
4. When metadata changes, write new versions immediately to the log, *in addition to* the asynchronous writes to "home".
5. If the system crashes, recover by scanning the log.
   - Much faster than scavenging (*fsck)* for large volumes.
6. If the system does not crash, then discard the log.

Representing Small Files

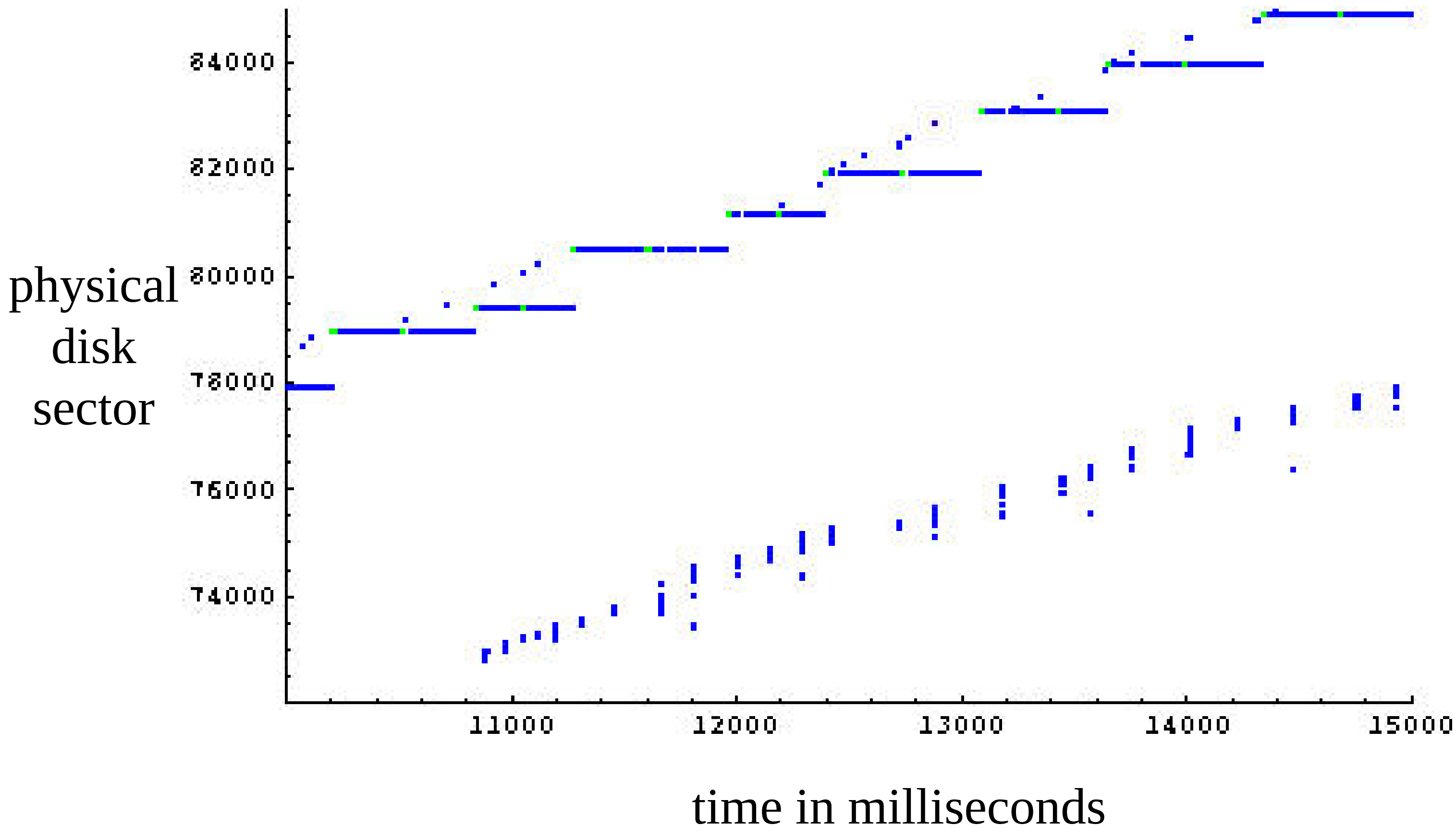
- Internal fragmentation in the file system blocks can waste significant space for small files.
  - E.g., 1KB files waste 87% of disk space (and bandwidth) in a naive file system with an 8KB block size.
  - Most files are small: one study [Irlam93] shows a median of 22KB.
- FFS solution: optimize small files for space efficiency.
  - Subdivide blocks into 2/4/8 *fragments* (or just *frags*).
  - Free block maps contain one bit for each fragment.
    - To determine if a block is free, examine bits for all its fragments.
  - The last block of a small file is stored on fragment(s).
    - If multiple fragments they must be contiguous.

[Provided for completeness]

Small-File Create Storm



Small-File Create: A Closer Look



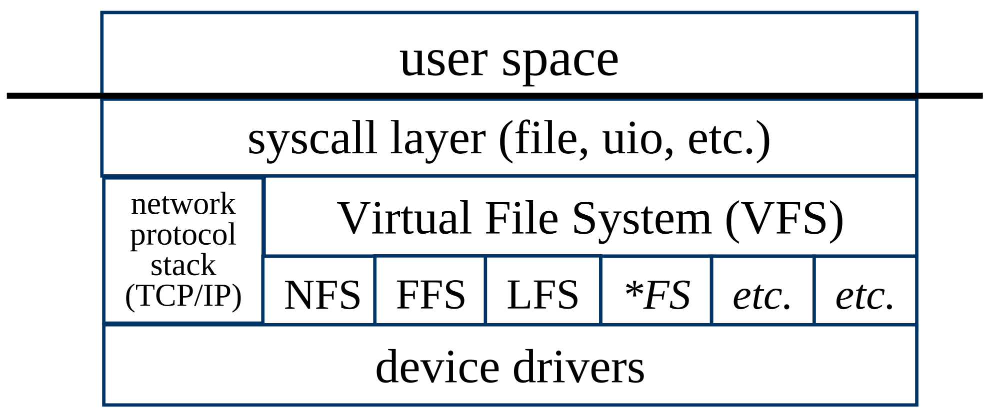
Filesystems

- Each file volume (*filesystem*) has a *type*, determined by its disk layout or the network protocol used to access it.
  - *ufs (ffs), lfs, nfs, rfs, cdfs*, etc.
  - Filesystems are administered independently.
- Modern systems also include “logical” pseudo-filesystems in the naming tree, accessible through the file syscalls.
  - *procfs*: the */proc* filesystem allows access to process internals.
  - *mfs*: the *memory file system* is a memory-based scratch store.
- Processes access filesystems through common syscalls

[Provided for completeness]

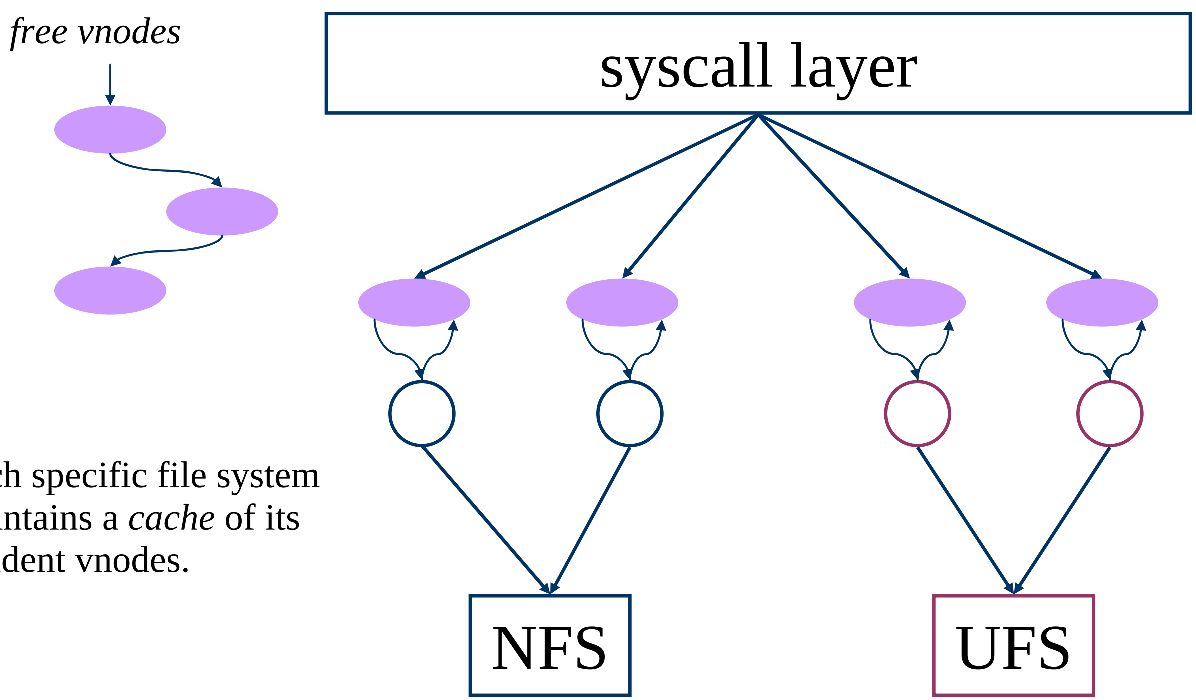
VFS: the Filesystem Switch

- Sun Microsystems introduced the *virtual file system* interface in 1985 to accommodate diverse filesystem types cleanly.
  - VFS allows diverse *specific file systems* to coexist in a file tree, isolating all FS-dependencies in pluggable filesystem modules.



Vnodes

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



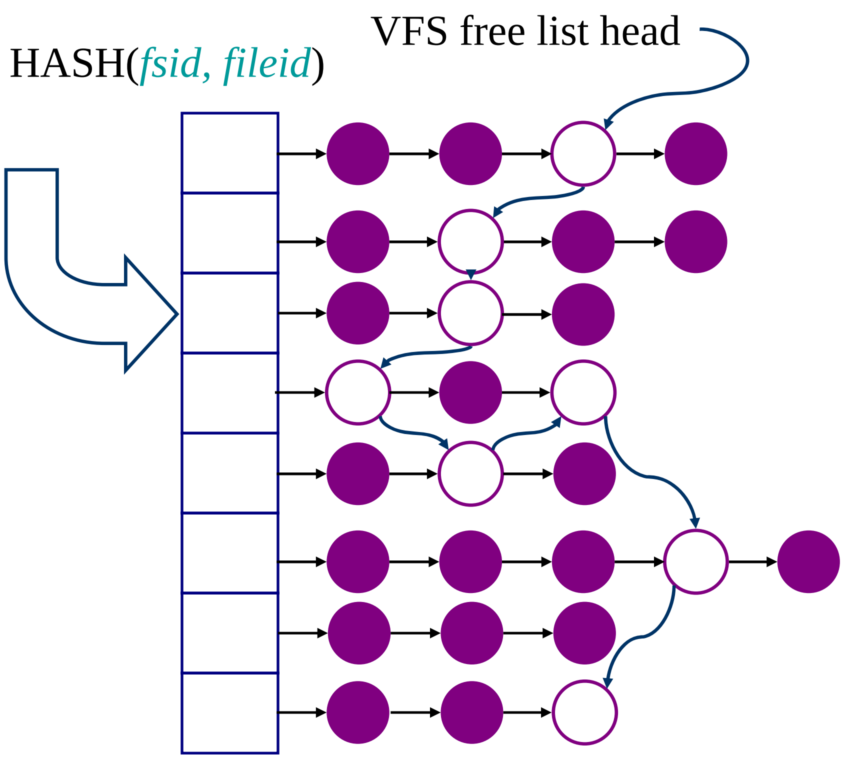
Each vnode has a standard *file attributes* struct.

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

Vnode operations are macros that vector to filesystem-specific procedures.

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

V/Inode Cache



Active vnodes are *reference- counted* by the structures that hold pointers to them.

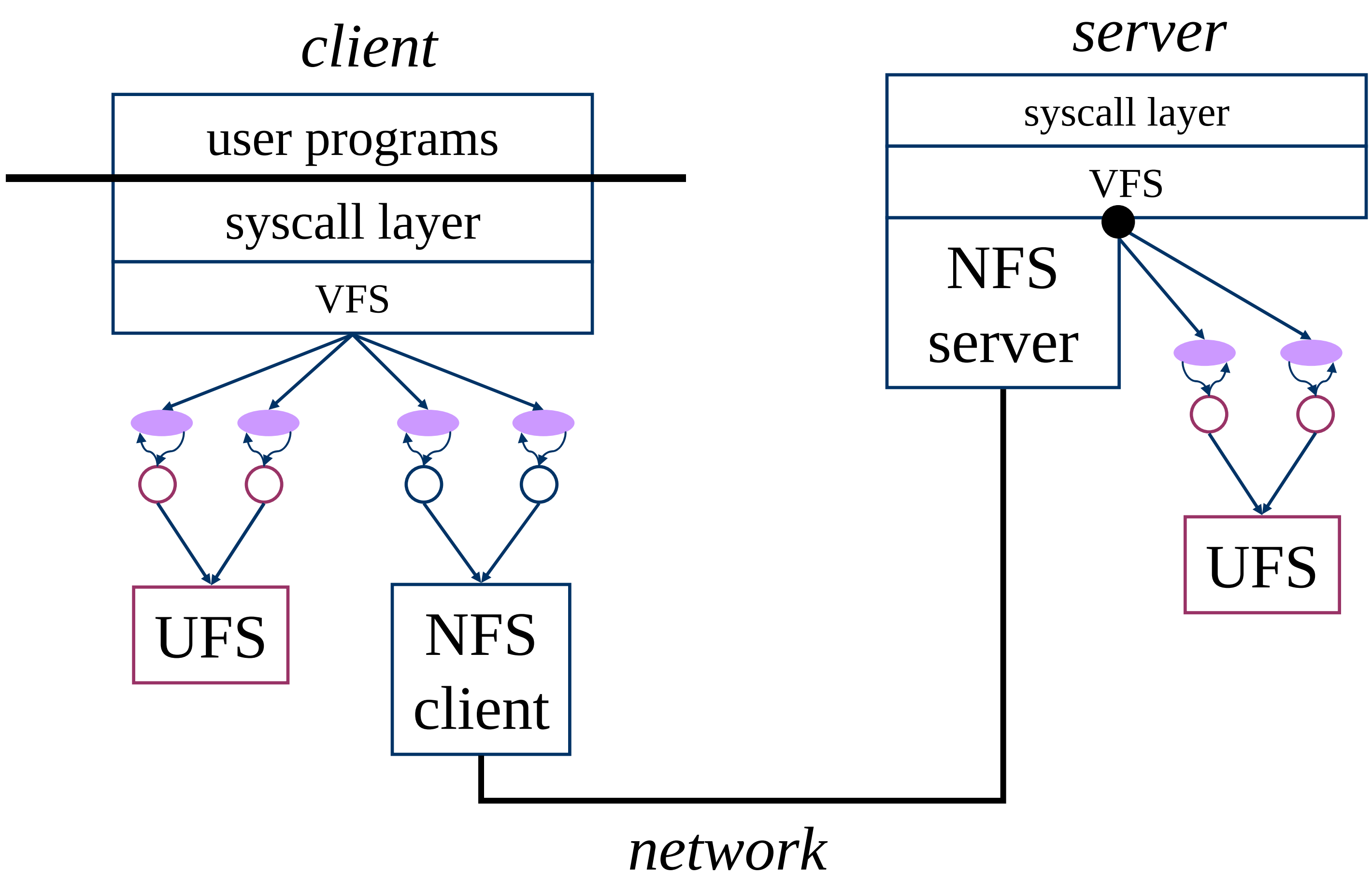
- system open file table
- process current directory
- file system mount points
- etc.

Each specific file system maintains its own hash of vnodes (BSD).

- specific FS handles initialization
- free list is maintained by VFS

vget(vp): reclaim cached inactive vnode from VFS free list
vref(vp): increment reference count on an active vnode
vrele(vp): release reference count on a vnode
vgone(vp): vnode is no longer valid (file is removed)

Network File System (NFS)

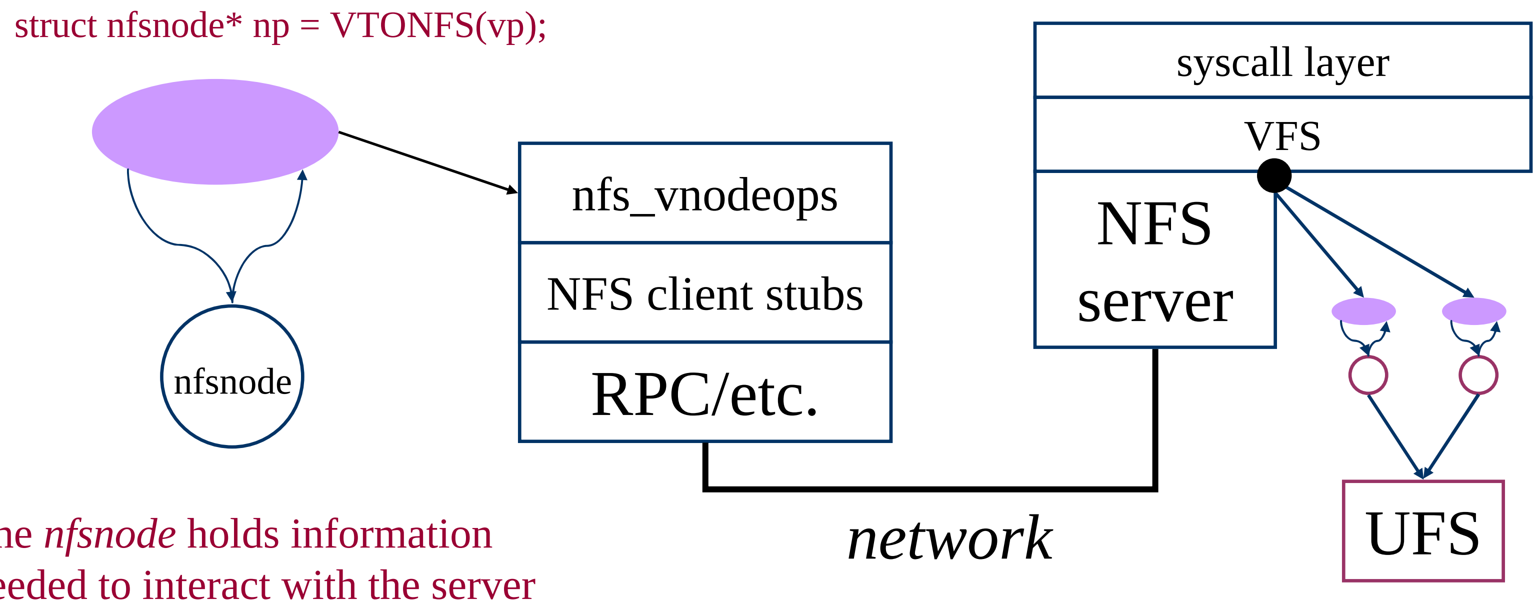


NFS Protocol

- NFS is a network protocol layered above TCP/IP.
  - Original implementations (and most today) use UDP datagram transport for low overhead.
    - Maximum IP datagram size was increased to match FS block size, to allow send/receive of entire file blocks.
    - Newer implementations use TCP as a transport.
  - The NFS protocol is a set of message formats and types for request/response (RPC) messaging.

NFS Vnodes

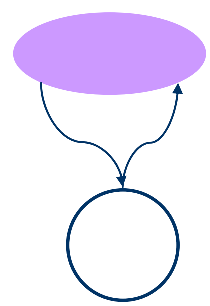
- The NFS protocol has an operation type for (almost) every vnode operation, with similar arguments/results.



The *nfsnode* holds information needed to interact with the server to operate on the file.

Vnode Operations and Attributes

<u>vnode attributes (*vattr*)</u>
type (VREG, VDIR, VLNK, etc.)
mode (9+ bits of permissions)
nlink (hard link count)
owner user ID
owner group ID
filesystem ID
unique file ID
file size (bytes and blocks)
access time
modify time
generation number



<u>generic operations</u>
vop_getattr (vattr)
vop_setattr (vattr)
vhold()
vholdrele()

<u>directories only</u>
vop_lookup (OUT vpp, name)
vop_create (OUT vpp, name, vattr)
vop_remove (vp, name)
vop_link (vp, name)
vop_rename (vp, name, tdvp, tvp, name)
vop_mkdir (OUT vpp, name, vattr)
vop_rmdir (vp, name)
vop_symlink (OUT vpp, name, vattr)
vop_readdir (uio, cookie)
vop_readlink (uio)

<u>files only</u>
vop_getpages (page**, count, offset)
vop_putpages (page**, count, sync, offset)
vop_fsync ()

Not to be tested

Pathname Traversal

- When a pathname is passed as an argument to a system call, the syscall layer must "convert it to a vnode".
  - Pathname traversal is a sequence of ***vop_lookup*** calls to descend the tree to the named file or directory.

```
open("/tmp/zot")
vp = get vnode for / (rootdir)
vp->vop_lookup(&cvp, "tmp");
vp = cvp;
vp->vop_lookup(&cvp, "zot");
```

Issues:

1. crossing mount points
2. obtaining root vnode (or current dir)
3. finding resident vnodes in memory
4. caching name->vnode translations
5. symbolic (soft) links
6. disk implementation of directories
7. locking/referencing to handle races with name create and delete operations

File Handles

- Question: how does the client tell the server which file or directory the operation applies to?
  - Similarly, how does the server return the result of a *lookup*?
- In NFS, the reference is a *file handle* or *fhandle*, a token/ticket whose value is determined by the server.
  - Includes all information needed to identify the file/object on the server, and get a pointer to it quickly.

volume ID	inode #	generation #

Typical NFSv3

NFS: Identity/Security

- "Classic NFS" was designed for a LAN under common administrative control.
  - Common uid/gid space
  - All client kernels are trusted to properly represent the local user identity.
  - Kernels trusted to control access to cached data.
- Volume export (mount privilege) control
  - Access control list at server
  - Subjects are nodes (e.g., DNS name or IP address)
- Mount just gives you a root filehandle: those file handles are capabilities.

NFS: From Concept to Implementation

- Now that we understand the basics, how do we make it work in a real system?
  - How do we make it fast?
    - Answer: caching, read-ahead, and write-behind.
  - How do we make it reliable? What if a message is dropped? What if the server crashes?
    - Answer: client retransmits request until it receives a response.
  - How do we preserve the failure/atomicity model?
    - Answer: well, we don't, at least not completely.
  - What about security and access control?

NFS as a “Stateless” Service

The NFS server maintains no transient information about its clients; there is no state other than the file data on disk.

Makes failure recovery simple and efficient.

- *no record of open files*
- *no server-maintained file offsets*
  - **Read** and **write** requests must explicitly transmit the byte offset for the operation.
- *no record of recently processed requests*
  - Retransmitted requests may be executed more than once.
  - Requests are designed to be ***idempotent*** whenever possible.
  - E.g., no append mode for writes, and no exclusive create.

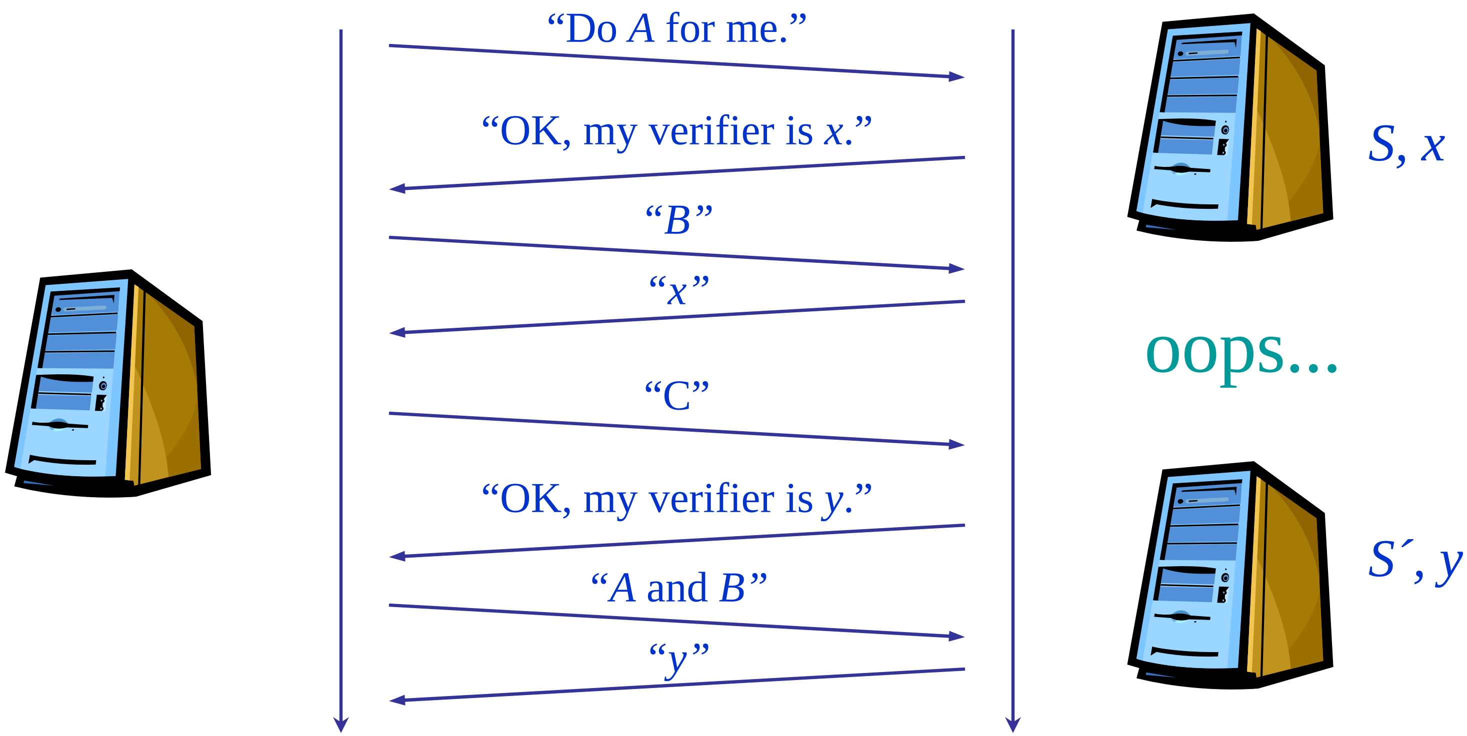
The Synchronous Write Problem

- Stateless NFS servers must commit each operation to stable storage before responding to the client.
  - Interferes with FS optimizations, e.g., clustering, LFS, and disk write ordering (seek scheduling).
    - Damages bandwidth and scalability.
  - Imposes disk access latency for each request.
    - Not so bad for a logged write; much worse for a complex operation like an FFS file write.
- The synchronous update problem occurs for any storage service with reliable update (*commit*).

Speeding Up NFS Writes

- Interesting solutions to the synchronous write problem, used in high-performance NFS servers:
- Delay the response until convenient for the server.
  - E.g., NFS *write-gathering* optimizations for clustered writes (similar to *group commit* in databases).
    - [NFS V3 commit operation]
  - Relies on write-behind from NFS I/O daemons (*iods*).
- Throw hardware at it: non-volatile memory (NVRAM)
  - Battery-backed RAM or UPS (uninterruptible power supply).
  - Use as an operation log (Network Appliance WAFL)...
  - ...or as a non-volatile disk write buffer (Legato).
- Replicate server and buffer in memory (e.g., MIT Harp).

Detecting Server Failure with a Session Verifier



What if $y == x$?

How to guarantee that $y \mathrel{!=} x$?

What is the implication of re-executing *A* and *B*, and after *C*?

Some uses: NFS V3 write commitment, RPC sessions, NFS V4 and DAFS (client).

The Retransmission Problem

- Sun RPC (and hence NFS) masks network errors by retransmitting each request after a timeout.
  - Handles dropped requests or dropped replies easily, but an operation may be executed more than once.
  - Sun RPC has *execute-at-least-once* semantics, but we need execute-at-most-once semantics for non-idempotent operations.
  - Retransmissions can radically increase load on a slow server.

Solutions

1. Use TCP or some other transport protocol that produces reliable, in-order delivery.
   – higher overhead, overkill
2. Implement an execute-at-most once RPC transport.
   – sequence numbers and timestamps
3. Keep a *retransmission cache* on the server.
   – Remember the most recent request IDs and their results, and just resend the result....does this violate statelessness?
4. Hope for the best and smooth over non-idempotent requests.
   - Map ENOENT and EEXIST to ESUCCESS.

File Cache Consistency

- Caching is a key technique in distributed systems.
  - The *cache consistency problem*: cached data may become *stale* if cached data is updated elsewhere in the network.
- Solutions:
  - *Timestamp invalidation* (NFS).
    - Timestamp each cache entry, and periodically query the server: "has this file changed since time *t*?"; invalidate cache if stale.
  - *Callback invalidation* (AFS).
    - Request notification (callback) from the server if the file changes; invalidate cache on callback.
  - *Leases* (NQ-NFS) [Gray&Cheriton89]

Recovery in Stateless NFS

- If the server fails and restarts, there is no need to rebuild in-memory state on the server.
  - Client reestablishes contact (e.g., TCP connection).
  - Client retransmits pending requests.
- Classical NFS uses a connectionless transport (UDP).
  - Server failure is transparent to the client
  - No connection to break or reestablish.
  - Sun/ONC RPC masks network errors by retransmitting a request after an adaptive timeout.
    - Crashed server is indistinguishable from a slow server.
    - Dropped packet is indistinguishable from a crashed server.

Drawbacks of a Stateless Service

- The stateless nature of classical NFS has compelling design advantages (simplicity), but also some key drawbacks:
  - Recovery-by-retransmission constrains the server interface.
    - ONC RPC/UDP has *execute-mostly-once* semantics ("send and pray"), which compromises performance and correctness.
  - Update operations are disk-limited.
    - Updates *must commit synchronously* at the server.
  - NFS cannot (quite) preserve local *single-copy semantics.*
    - Files may be removed while they are open on the client.
    - Server cannot help in client cache consistency.
- Let's look at the consistency problem...

Timestamp Validation in NFS [1985]

- NFSv2/v3 uses a form of timestamp validation like today's Web
  - Timestamp cached data at file grain.
  - Maintain per-file expiration time (TTL)
  - Probe for new timestamp to revalidate if cache TTL has expired.
    - Get attributes (*getattr*)
- NFS file cache and access primitives are block-grained, and the client may issue many operations in sequence on the same file.
  - Clustering: File-grained timestamp for block-grained cache
  - Piggyback file attributes on each response
  - Adaptive TTL
- What happens on server failure? Client failure?

AFS [1985]

- AFS is an alternative to NFS developed at CMU.
    - Duke still uses it.
- Designed for wide area file sharing:
    - Internet is large and growing exponentially.
    - Global name hierarchy with local naming contexts and location info embedded in fully qualified names.
        - Much like DNS
    - Security features, with per-domain authentication / access control.
    - Whole file caching or 64KB chunk caching
        - Amortize request/transfer cost
    - Client uses a disk cache
        - Cache is preserved across client failure.
        - Again, it looks a lot like the Web.

[Provided for completeness]

Callback Invalidations in AFS-2

- AFS-1 uses timestamp validation like NFS; AFS-2 uses *callback invalidations*.
- Server returns "callback promise" *token* with file access.
  - Like ownership protocol, confers a right to cache the file.
  - Client caches the token on its disk.
- Token states: {**valid**, **invalid**, **cancelled**}
- On a sharing collision, server *cancels* token with a callback.
  - Client invalidates cached copy of the associated file.
  - Detected on client write to server: *last writer wins*.
  - (No distinction between read/write token.)

[Provided for completeness]

Issues with Callback Invalidations

- What happens after a failure?
  - Client invalidates its tokens on client restart.
    - Invalid tokens may be revalidated, like NFS *getattr* or WWW.
  - Server must remember tokens across restart.
  - Can the client distinguish a server failure from a network failure?
  - Client invalidates tokens after a timeout interval *T* if the client has no communication with the server.
    - Weakens consistency in failures.
- Then there's the problem of update semantics: two clients may be actively updating the same file at the same time.

[Provided for completeness]

NQ-NFS Leases

- In NQ-NFS, a client obtains a *lease* on the file that permits the client's desired read/write activity.
    - "A lease is a ticket permitting an activity; the lease is valid until some expiration time."
  - A *read-caching lease* allows the client to cache clean data.
    - **Guarantee**: no other client is modifying the file.
  - A *write-caching lease* allows the client to buffer modified data for the file.
    - **Guarantee**: no other client has the file cached.
    - Allows *delayed writes*: client may delay issuing writes to improve write performance (i.e., client has a writeback cache).

[Provided for completeness]

Using NQ-NFS Leases

1. Client NFS piggybacks lease requests for a given file on I/O operation requests (e.g., read/write).
   - NQ-NFS leases are *implicit* and distinct from file locking.
2. The server determines if it can safely grant the request, i.e., does it conflict with a lease held by another client.
   - *read leases* may be granted simultaneously to multiple clients
   - *write leases* are granted exclusively to a single client
3. If a conflict exists, the server may send an *eviction* notice to the holder.
   - Evicted from a write lease? Write back.
   - *Grace period*: server grants extensions while client writes.
   - Client sends *vacated* notice when all writes are complete.

[Provided for completeness]

NQ-NFS Lease Recovery

- Key point: the bounded lease term simplifies recovery.
  - Before a lease expires, the client must *renew* the lease.
  - What if a client fails while holding a lease?
    - Server waits until the lease expires, then unilaterally reclaims the lease; client forgets all about it.
    - If a client fails while writing on an eviction, server waits for *write slack* time before granting conflicting lease.
  - What if the server fails while there are outstanding leases?
    - Wait for *lease period + clock skew* before issuing new leases.
  - Recovering server must absorb lease renewal requests and/or writes for vacated leases.

[Provided for completeness]

NQ-NFS Leases and Cache Consistency

- Every lease contains a file version number.
  - Invalidation cache iff version number has changed.
- Clients may disable client caching when there is concurrent write sharing.
  - *no-caching* lease (Sprite)
- What consistency guarantees do NQ-NFS leases provide?
  - Does the server eventually receive/accept all writes?
  - Does the server accept the writes in order?
  - Are groups of related writes atomic?
  - How are write errors reported?
  - What is the relationship to NFS V3 *commit*?

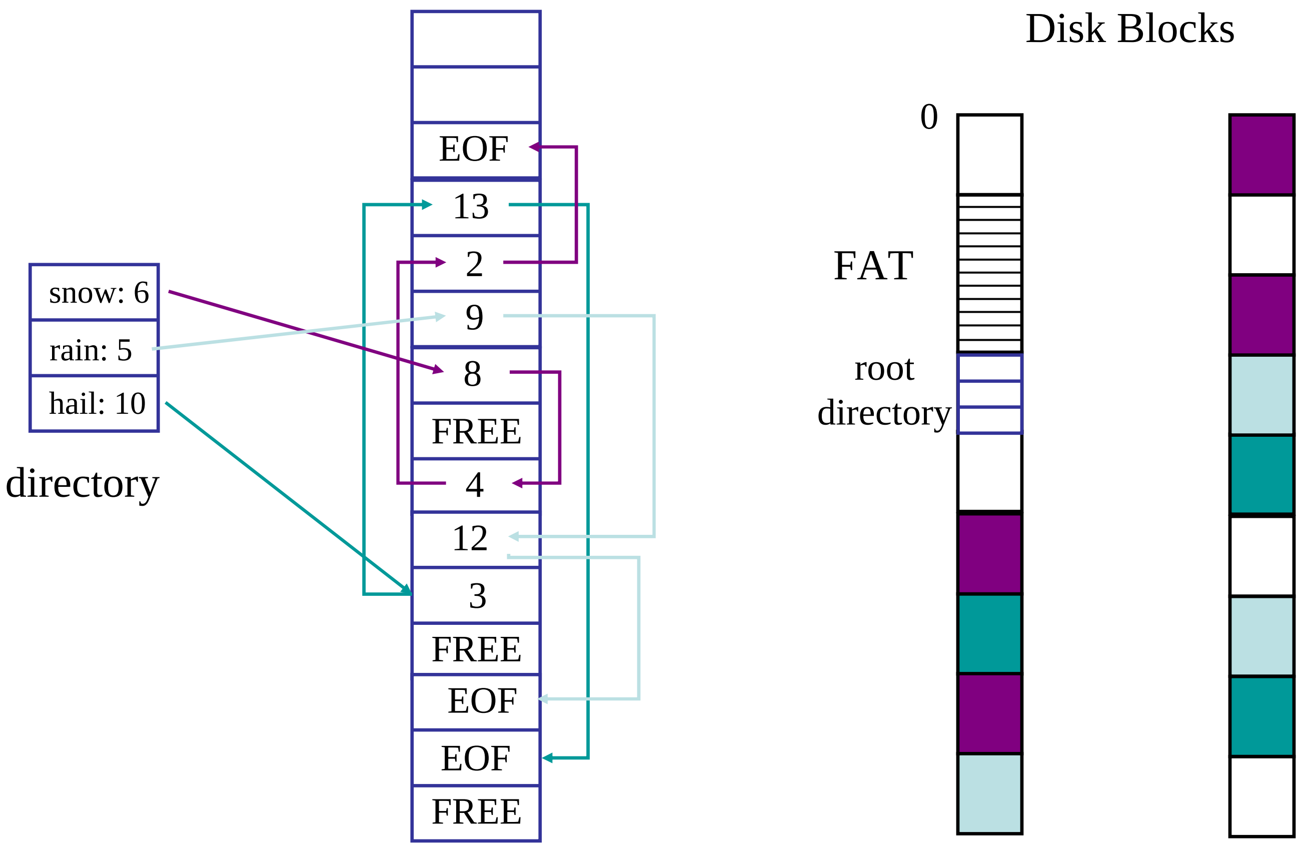
[Provided for completeness]

Using Disk Storage

- Typical operating systems use disks in three different ways:
- System calls allow user programs to access a "raw" disk.
  - Unix: special *device file* identifies volume directly.
  - Any process that can *open* the device file can read or write any specific sector in the disk volume.
- OS uses disk as *backing storage* for virtual memory.
  - OS manages volume transparently as an "overflow area" for VM contents that do not "fit" in physical memory.
- 3. OS provides syscalls to create/access *files* residing on disk.
  - OS *file system* modules virtualize physical disk storage as a collection of logical files.

[Provided for completeness]

Alternative Structure: DOS FAT



[Provided for completeness]