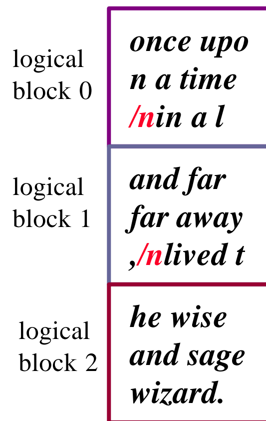
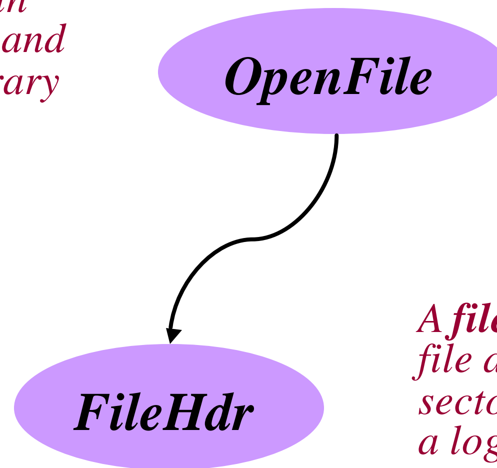
File Systems and NFS

Representing Files On Disk: Nachos

*An **OpenFile** represents a file in active use, with a seek pointer and read/write primitives for arbitrary byte ranges.*

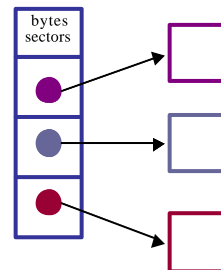




OpenFile(sector)
Seek(offset)
Read(char data, bytes)*
Write(char data, bytes)*

*A **file header** describes an on-disk file as an ordered sequence of sectors with a length, mapped by a logical-to-physical block map.*

OpenFile ofd = filesys->Open("tale");*
ofd->Read(data, 10) gives 'once upon '
ofd->Read(data, 10) gives 'a time/nin '

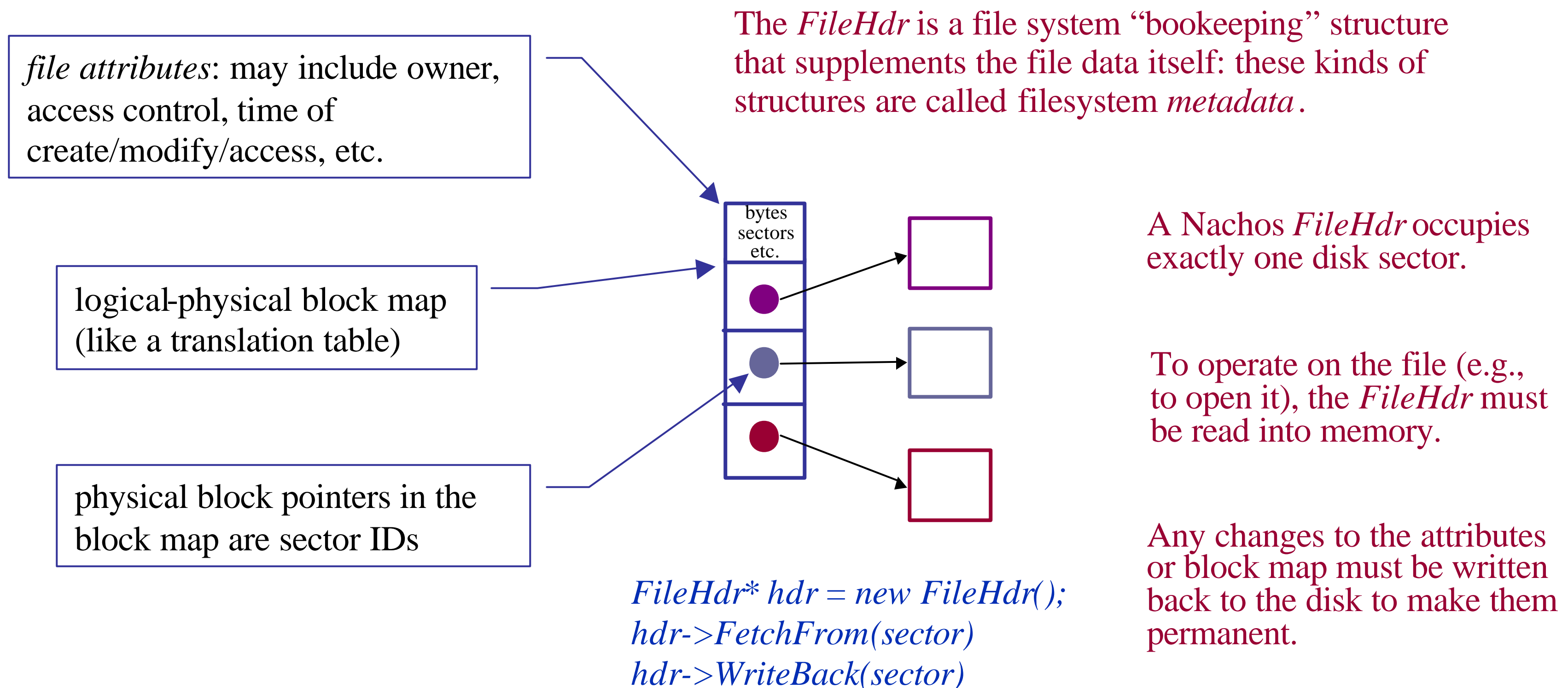


Allocate(...,filesize)
length = FileLength()
sector = ByteToSector(offset)

File Metadata

On disk, each file is represented by a *FileHdr* structure.

The *FileHdr* object is an in-memory *copy* of this structure.



Representing Large Files

The Nachos *FileHdr* occupies exactly one disk sector, limiting the maximum file size.

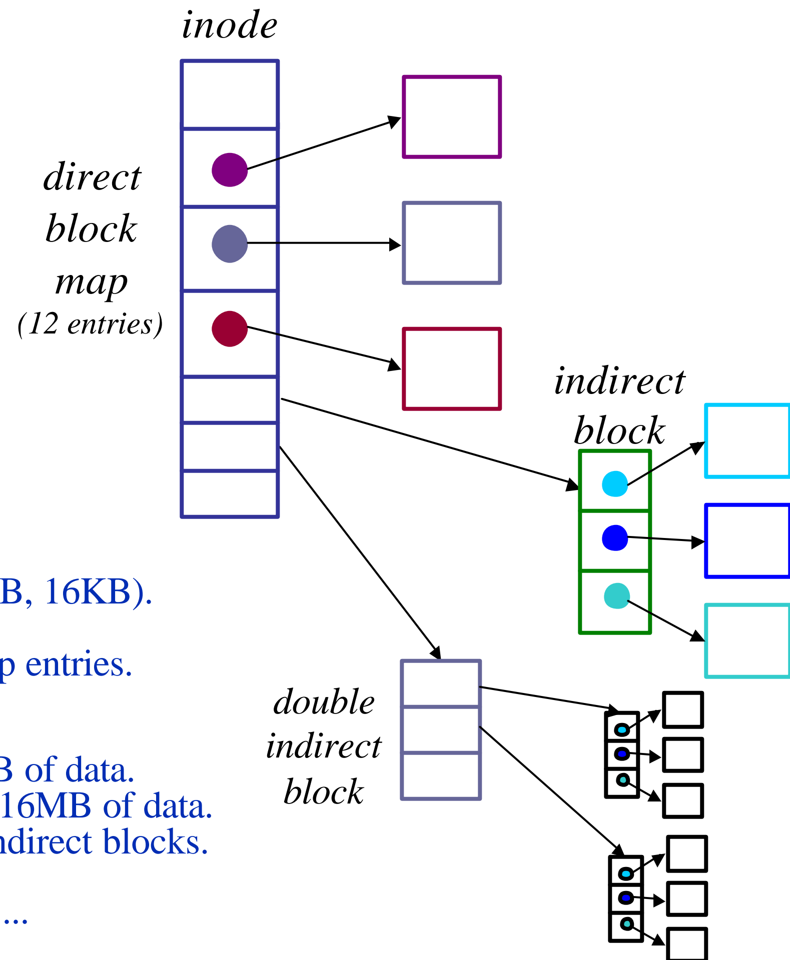
sector size = 128 bytes
120 bytes of block map = 30 entries
each entry maps a 128-byte sector
max file size = 3840 bytes

In Unix, the *FileHdr* (called an index-node or *inode*) represents large files using a hierarchical block map.

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



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).

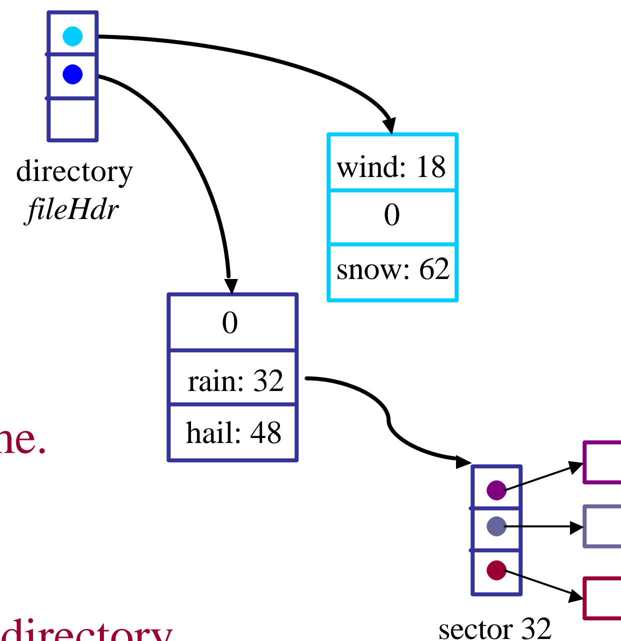
  If multiple fragments they must be contiguous.

Basics of Directories

A *directory* is a set of file names, supporting lookup by symbolic name.

In Nachos, each directory is a file containing
a set of mappings from name->*FileHdr*.

Directory(entries)
sector = Find(name)
Add(name, sector)
Remove(name)



Each directory entry is a fixed-size
slot with space for a *FileNameMaxLen* byte name.

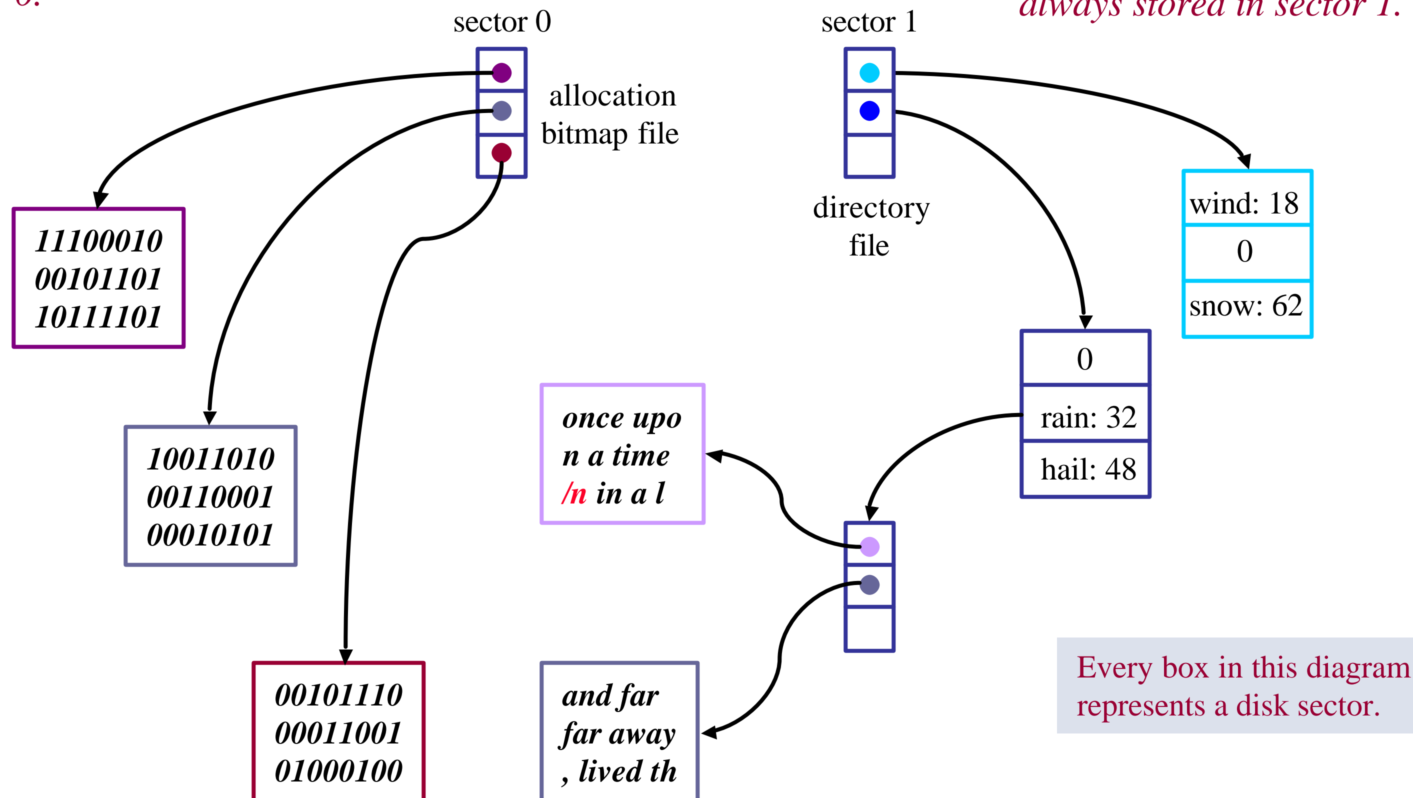
Entries or slots are found by a linear scan.

A directory entry may hold a pointer to another directory,
forming a hierarchical name space.

A Nachos Filesystem On Disk

An allocation bitmap file maintains free/allocated state of each physical block; its ***FileHdr*** *is always stored in sector 0.*

A directory maintains the name->FileHdr mappings for all existing files; its ***FileHdr*** *is always stored in sector 1.*

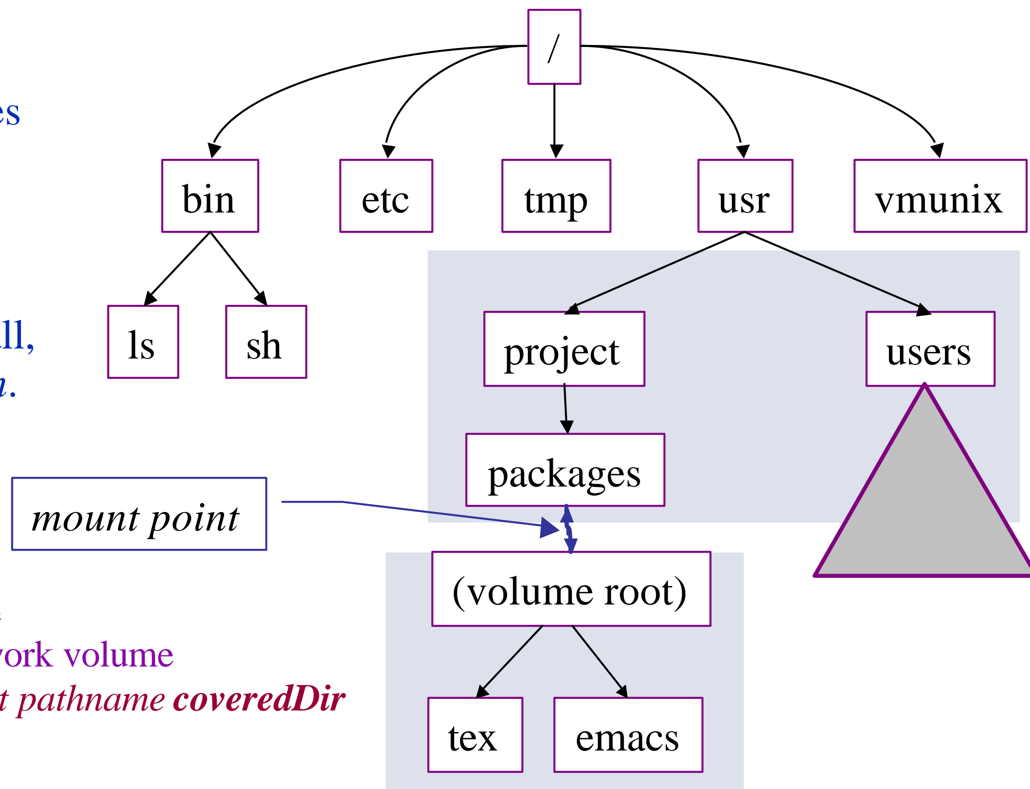


A Typical Unix File Tree

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***

DUKE Systems & Architecture

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.

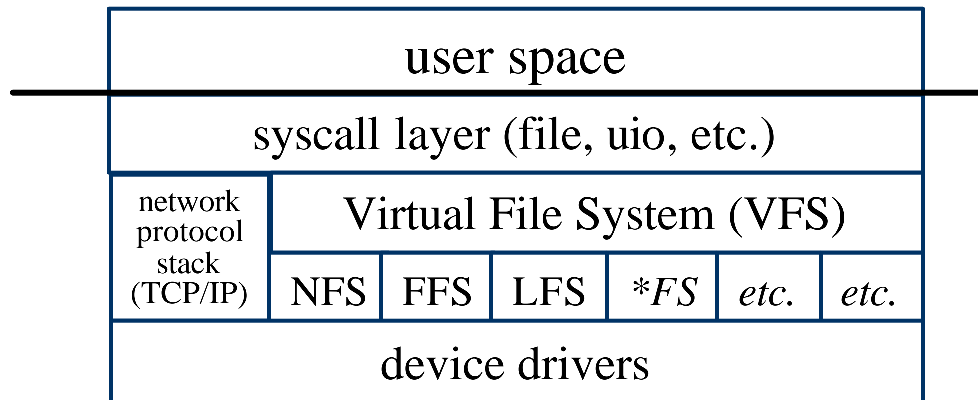
- *procfs*: the */proc* filesystem allows access to process internals.
- *mfs*: the *memory file system* is a memory-based scratch store.

Processes access filesystems through common system calls.

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.



Other abstract interfaces in the kernel: device drivers, file objects, executable files, memory objects.

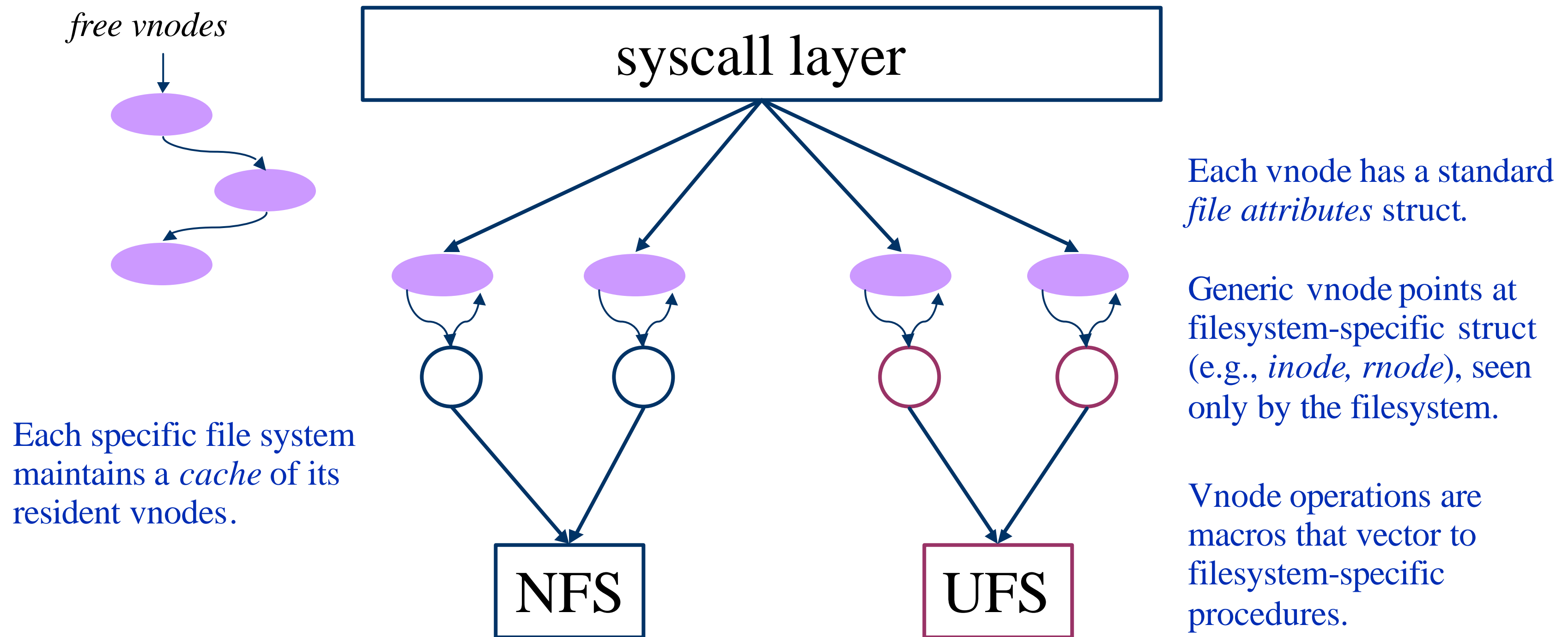
VFS was an internal kernel restructuring with no effect on the syscall interface.

Incorporates object-oriented concepts: a generic procedural interface with multiple implementations.

Based on abstract objects with dynamic method binding by type...in C.

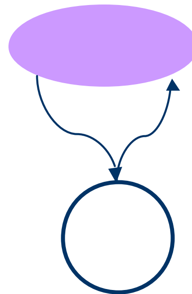
Vnodes

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



Vnode Operations and Attributes

vnode attributes (*vattr*)
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

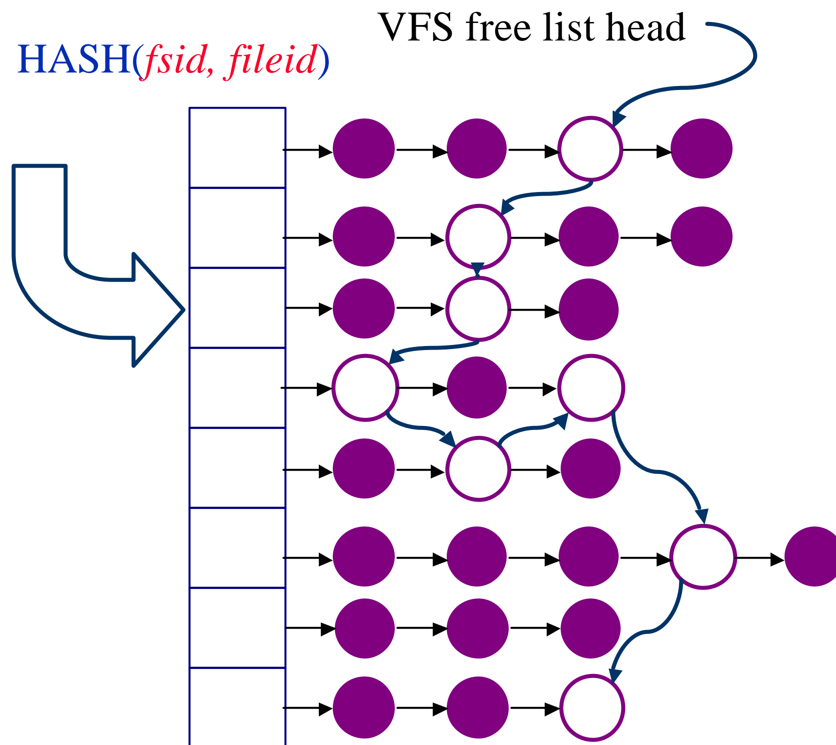


generic operations
vop_getattr (vattr)
vop_setattr (vattr)
vhold()
vholdrele()

directories only
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, contents)
vop_readdir (uio, cookie)
vop_readlink (uio)

files only
vop_getpages (page**, count, offset)
vop_putpages (page**, count, sync, offset)
vop_fsync ()

V/Inode Cache



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

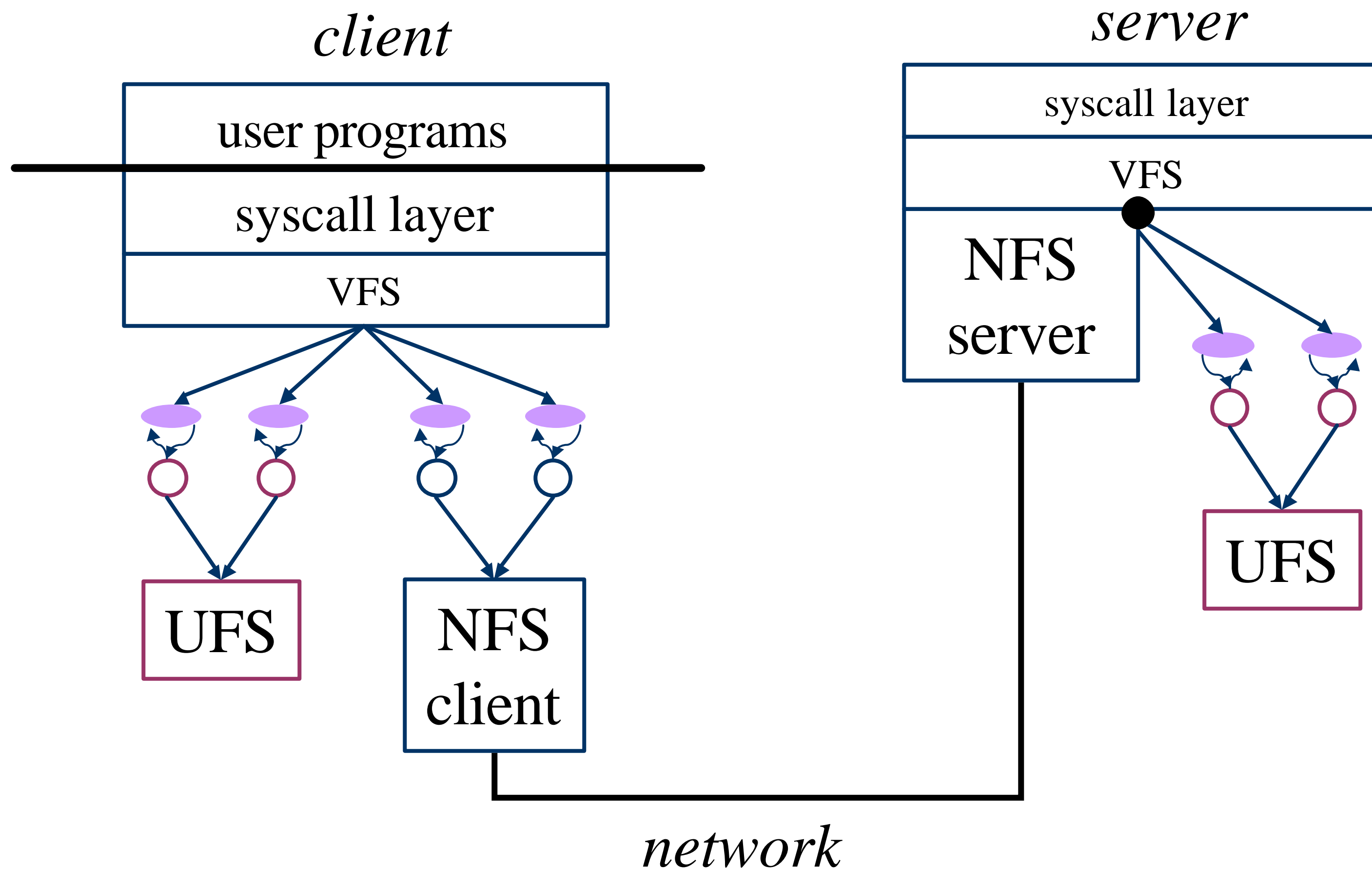
- system open file table
- process current directory
- file system mount points
- etc.

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

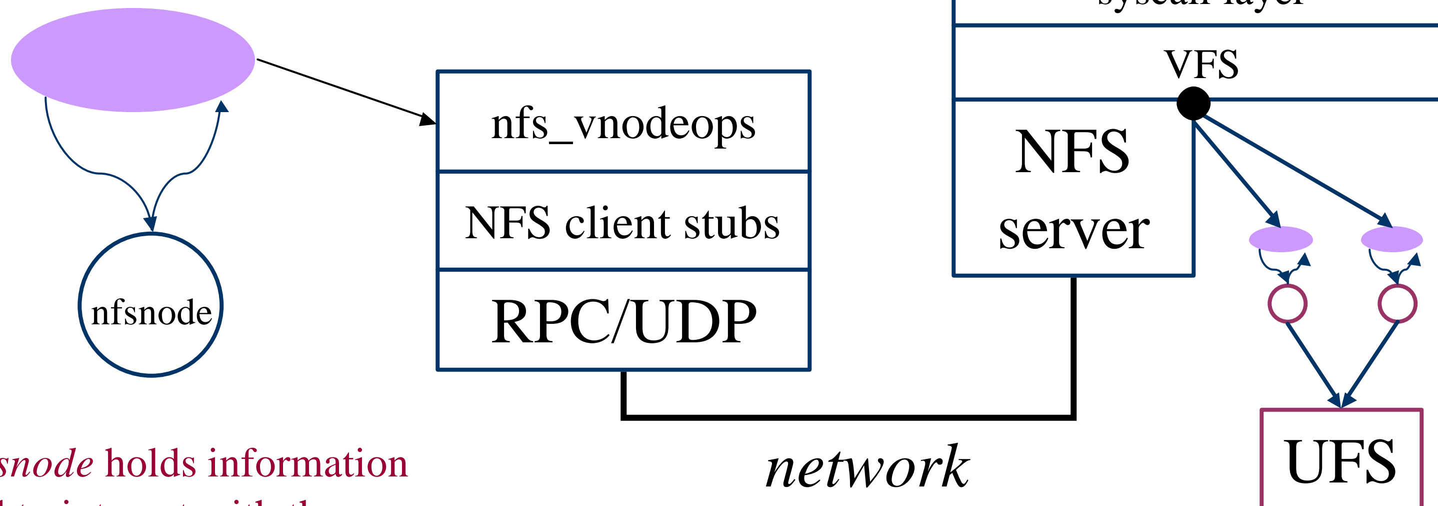
NFS is a network protocol layered above TCP/IP.

- 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.
  - Some newer implementations use TCP as a transport.
- The NFS protocol is a set of message formats and types.
  - Client issues a *request* message for a service operation.
  - Server performs requested operation and returns a *reply* message with status and (perhaps) requested data.

NFS Vnodes

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

struct nfsnode* np = VTONFS(vp);



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

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*?

  More generally, how to pass a pointer or an object reference as an argument/result of an RPC call?

In NFS, the reference is a *file handle* or *fhandle*, a 32-byte 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 #

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

From Servers to Services

Are Web servers and RPC servers scalable? Available?

A single server process can only use one machine.

Upgrading the machine causes interruption of service.

If the process or machine fails, the service is no longer reachable.

We improve scalability and availability by replicating the functional components of the service.

(May need to replicate data as well, but save that for later.)

- View the *service* as made up of a collection of *servers*.
- Pick a convenient server: if it fails, find another (*fail-over*).

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 file system semantics in the presence of failures and/or sharing by multiple clients?

  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.

Drawbacks of a Stateless Service

The stateless nature of NFS has compelling design advantages (simplicity), but also some key drawbacks:

- Update operations are disk-limited because they *must be committed synchronously* at the server.
- NFS cannot (quite) preserve local *single-copy semantics*.

    Files may be removed while they are open on the client.

    Idempotent operations cannot capture full semantics of Unix FS.

- Retransmissions can lead to correctness problems and can quickly saturate an overloaded server.
- Server keeps no record of blocks held by clients, so cache consistency is problematic.

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*).

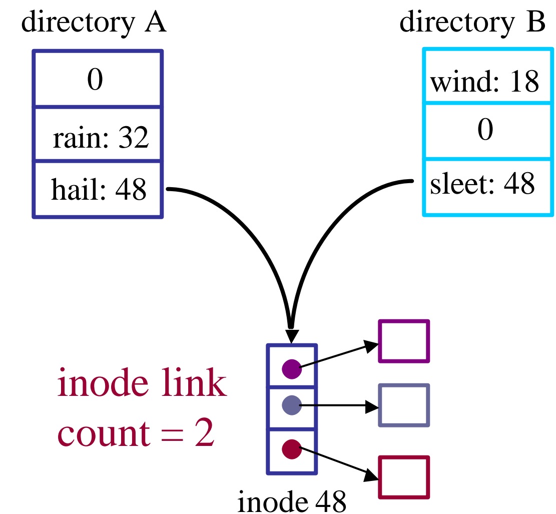
Speeding Up NFS Writes

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).

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.

link system call
link (existing name, new name)
create a new name for an existing file
increment inode link count

unlink system call ("remove")
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

Unix Symbolic (Soft) Links

Unix files may also be named by *symbolic (soft) links*.

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

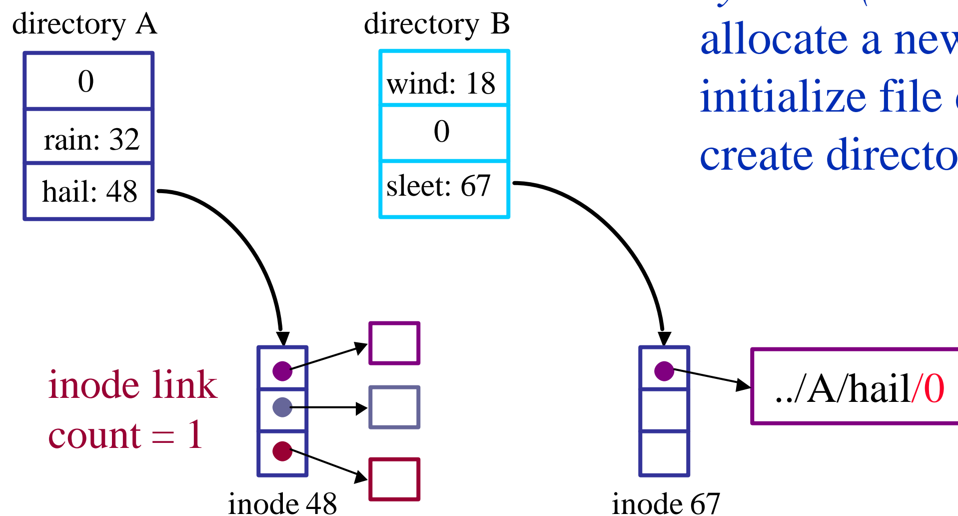
symlink system call

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?