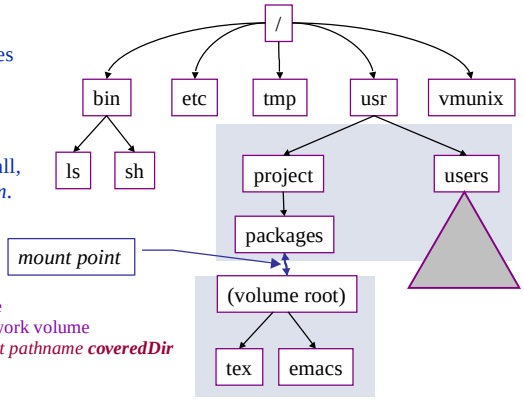
Sun's VFS and NFS

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

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.

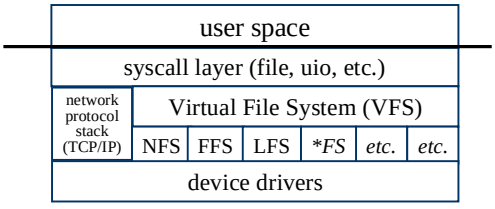
- *procfs*: the */proc* filesystem allows access to process internals.
- *mfs*: the *memory file system* is a memory-based scratch store.

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.

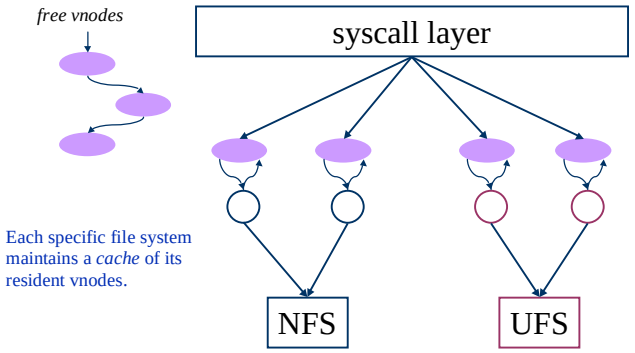
VFS was an internal kernel restructuring with no effect on the syscall interface.

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.



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

Each vnode has a standard *file attributes* struct.

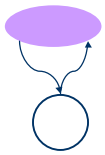
Generic vnode points at filesystem-specific struct (e.g., *inode*, *rnode*), seen only by the filesystem.

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

CPS 210

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

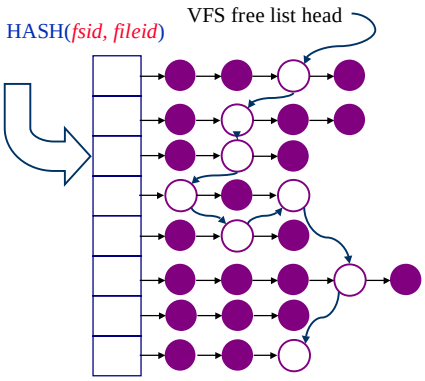


generic operations
vop_getattr (vattr)
vop_setattr (vattr)
vhold()
vholdrele()

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



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)

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

Continuum of Distributed Systems

Issues:
naming and sharing
performance and scale
resource management

small
fast

Parallel
Architectures
CPS 221

Networks
CPS 214

big
slow

?

?

Multiprocessors

clusters
(GMS)
fast network
trusting hosts
coordinated

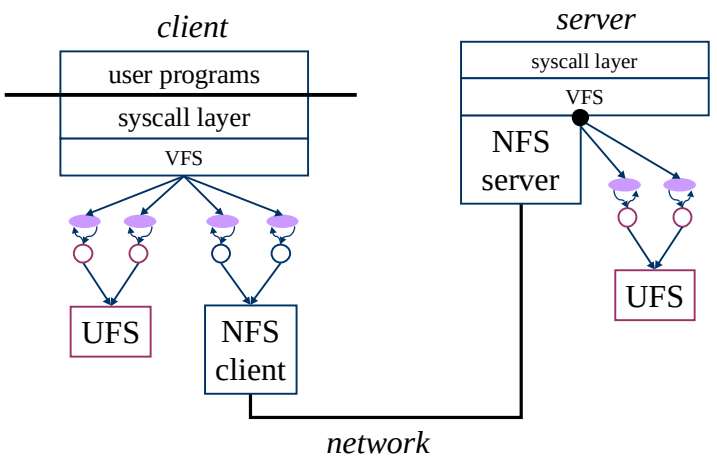
LAN
(NFS)
slow network
untrusting hosts
autonomy

Global
Internet

low latency
high bandwidth
secure, reliable interconnect
no independent failures
coordinated resources

high latency
low bandwidth
autonomous nodes
unreliable network
fear and distrust
independent failures
decentralized administration

Network File System (NFS)



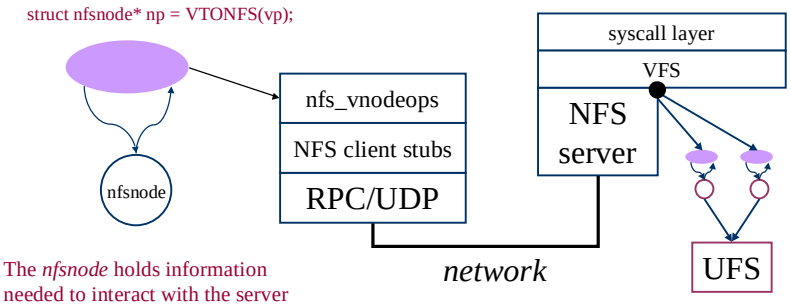
NFS Protocol

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.



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).

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 to the Retransmission Problem

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]