

Outline for Today's Lecture

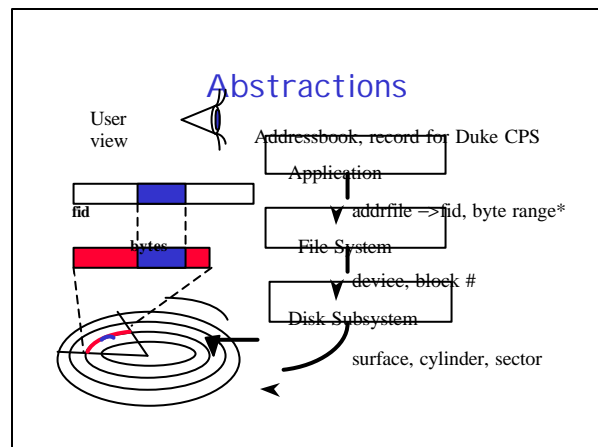
- Administrative
 - Nachos assignment 4 due Thursday
- Objective:
 - File system issues

File System Issues

- What is the *role* of files?
What is the file abstraction?
- File naming. How to find the file we want?
Sharing files. Controlling access to files.
- Performance issues - how to deal with the bottleneck of disks?
What is the "right" way to optimize file access?

Role of Files

- Persistence – long-lived – data for posterity
 - non-volatile storage media
 - semantically meaningful (memorable) names



*File Abstractions

- UNIX-like files
 - Sequence of bytes
 - Operations: open (create), close, read, write, seek
- Memory mapped files
 - Sequence of bytes
 - Mapped into address space
 - Page fault mechanism does data transfer
- Named, Possibly typed

Functions of File System

- (Directory subsystem) Map filenames to fileids - open (create) syscall. Create kernel data structures. Maintain naming structure (unlink, mkdir, rmdir)
- Determine layout of files and metadata on disk in terms of blocks. Disk block allocation. Bad blocks.
- Handle read and write system calls
- Initiate I/O operations for movement of blocks to/from disk.
- Maintain buffer cache

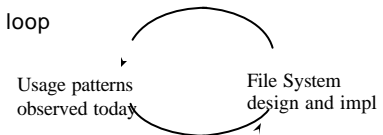
Functions of Device Subsystem

In general, deal with device characteristics

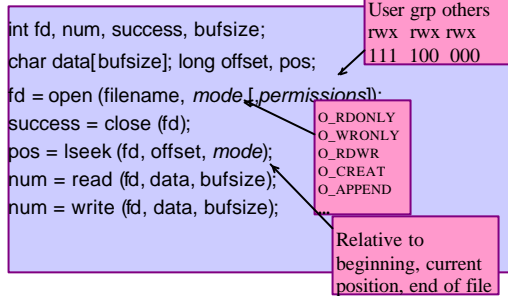
- Translate block numbers (the abstraction of device shown to file system) to physical disk addresses. Device specific (subject to change with upgrades in technology) intelligent placement of blocks.
- Schedule (reorder?) disk operations

Know your Workload!

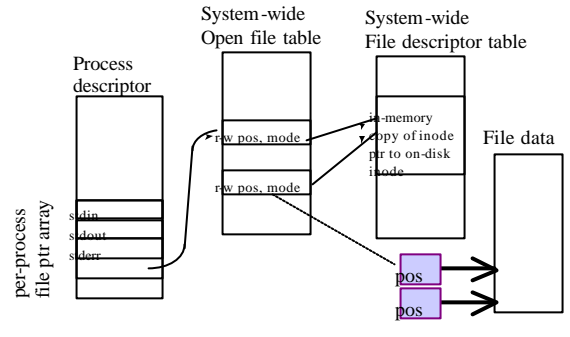
- File usage patterns should influence design decisions. Do things differently depending:
 - How large are most files? How long-lived?
 - Read vs. write activity. Shared often?
 - Different levels “see” a different workload.
- Feedback loop



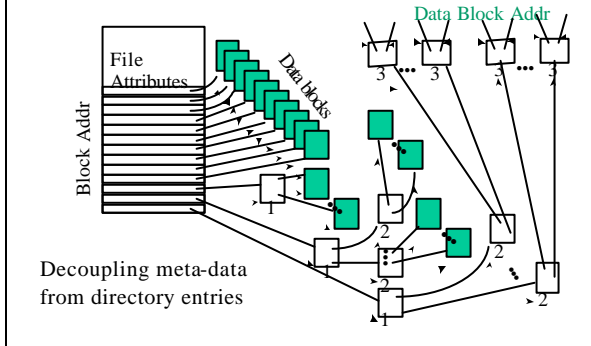
Unix File Syscalls



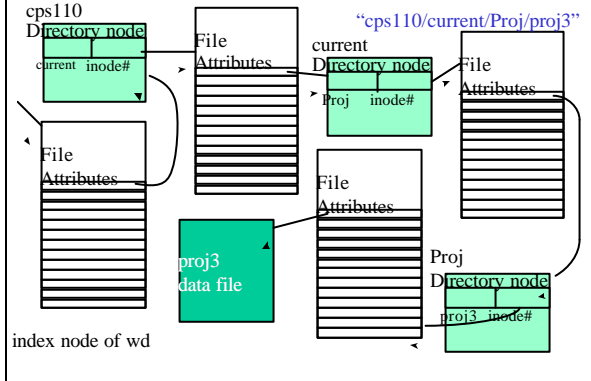
File System Data Structures



UNIX Inodes



Pathname Resolution



File Sharing Between Parent/Child

```

main(int argc, char *argv[]) {
    char c;
    int fdrd, fdwt, fdpriv;

    if ((fdrd = open(argv[1], O_RDONLY)) == -1)
        exit(1);
    if ((fdwt = creat(argv[2], 0666)) == -1)
        exit(1);

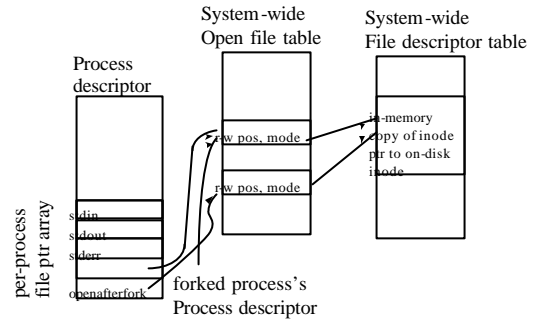
    fork();

    if ((fdpriv = open(argv[3], O_RDONLY)) == -1)
        exit(1);

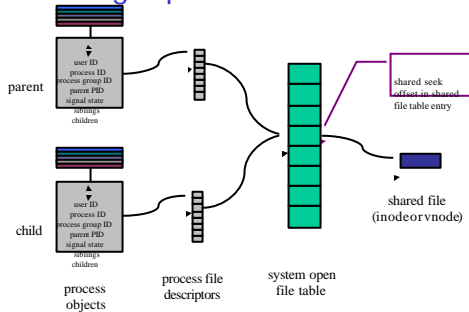
    while (TRUE) {
        if (read(fdrd, &c, 1) != 1)
            exit(0);
        write(fdwt, &c, 1);
    }
}

```

File System Data Structures



Sharing Open File Instances

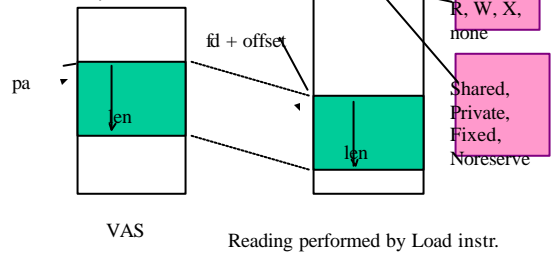


Memory Mapped Files

```

fd = open (somefile, consistent_mode);
pa = mmap(addr, len, prot, flags, fd,
offset);

```



Nachos File Syscalls/Operations

```
Create("zot");
```

```
OpenFileId fd  
fd = Open("zot");  
Close(fd);
```

```
char data[bufsize];  
Write(data, count, fd);  
Read(data, count, fd);
```

Limitations:

1. small, fixed-size files and directories
2. single disk with a single directory
3. stream files only: no seek syscall
4. file size is specified at creation time
5. no access control, etc.

Goals of File Naming

- Foremost function - to find files (e.g., in `open()`), Map file name to file object.
- To store **meta-data** about files.
- To allow users to choose their own file names without undue **name conflict** problems.
- To allow **sharing**.
- Convenience: **short** names, groupings.
- To avoid implementation complications

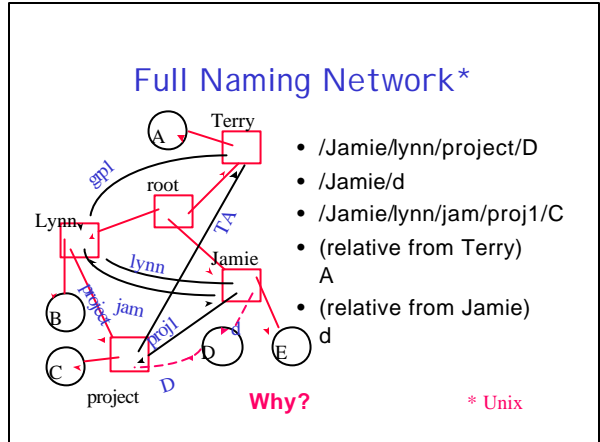
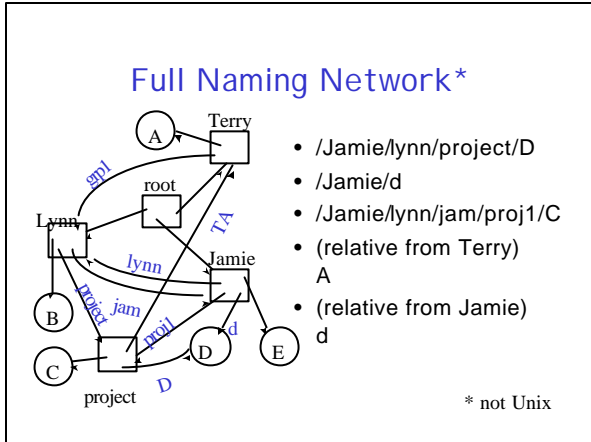
Naming Structures

- Flat name space - 1 system-wide table,
 - Unique naming with multiple users is hard. Name conflicts.
 - Easy sharing, need for protection
- Per-user name space
 - Protection by isolation, no sharing
 - Easy to avoid name conflicts
 - Register identifies with directory to use to resolve names, possibility of user-settable (`cd`)

Naming Structures

Naming network

- Component names - **pathnames**
 - **Absolute** pathnames - from a designated *root*
 - **Relative** pathnames - from a **working directory**
 - Each name carries how to resolve it.
- Short names to files anywhere in the network produce cycles, but convenience in naming things.



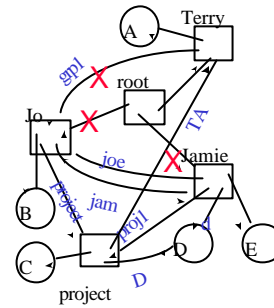
- ### Meta-Data
- | | |
|--|--|
| <ul style="list-style-type: none"> • File size • File type • Protection - access control information • History: creation time, last modification, last access. | <ul style="list-style-type: none"> • Location of file - which device • Location of individual blocks of the file on disk. • Owner of file • Group(s) of users associated with file |
|--|--|

- ### Restricting to a Hierarchy
- Problems with full naming network
 - What does it mean to “delete” a file?
 - Meta-data interpretation

Operations on Directories (UNIX)

- link (oldpathname, newpathname) - make entry pointing to file
- unlink (filename) - remove entry pointing to file
- mknod (dirname, type, device) - used (e.g. by mkdir utility function) to create a directory (or named pipe, or special file)
- getdents(fd, buf, structsized) - reads dir entries

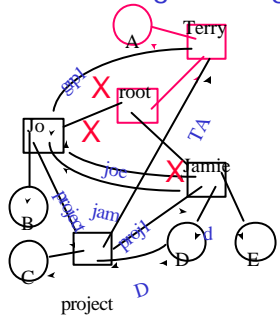
Reclaiming Storage



Series of unlinks

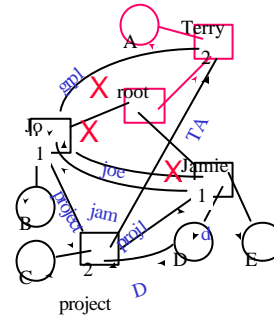
What should be dealloc?

Reclaiming Storage



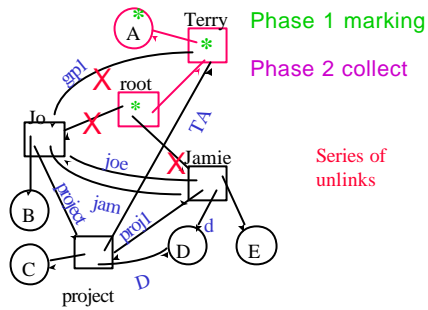
Series of unlinks

Reference Counting?



Series of unlinks

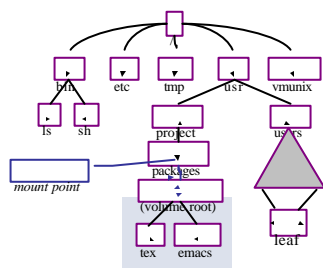
Garbage Collection



Restricting to a Hierarchy

- Problems with full naming network
 - What does it mean to “delete” a file?
 - Meta-data interpretation
- Eliminating cycles
 - allows use of *reference counts* for reclaiming file space
 - avoids *garbage collection*

Given: Naming Hierarchy (because of implementation issues)

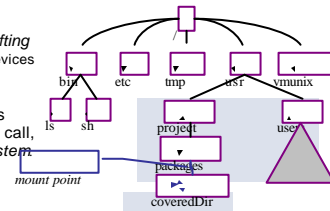


A Typical Unix File Tree

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 devices 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

volume root contents become visible at *pathname*

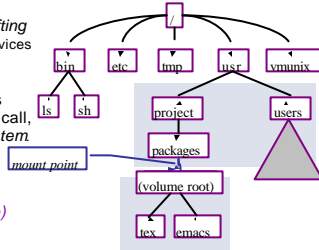
coveredDir

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 devices or from network servers.

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



`mount (coveredDir, volume)`

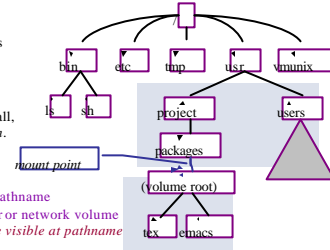
`/usr/project/packages/coveredDir/emacs`

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 devices 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*

`/usr/project/packages/coveredDir/emacs`

Reclaiming Convenience

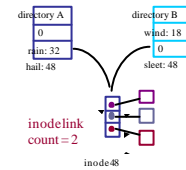
- Symbolic links - indirect files filename maps, not to file object, but to another pathname
 - allows short aliases
 - slightly different semantics
- Search path rules

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.

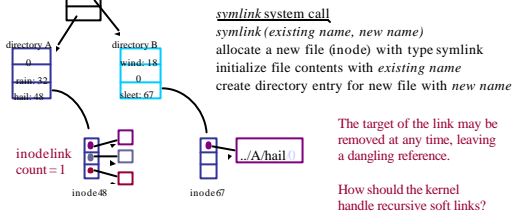


`linkssystem call`
link (existing name, new name)
 create a new name for an existing file
 increment inode link count

`unlinkssystem 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.



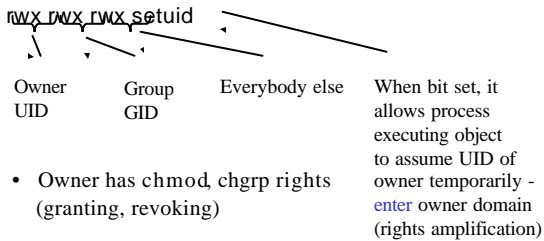
Convenience, but *not* performance!

Access Control for Files

- Access control lists - detailed list attached to file of users allowed (denied) access, including kind of access allowed/denied.
- UNIX RWX - owner, group, everyone

UNI X access control

- Each file carries its access control with it.



The Access Model

- Authorization problems can be represented abstractly by of an *access model*.
 - each row represents a subject/principal/domain
 - each column represents an object
 - each cell: accesses permitted for the {*subject, object*} pair
 - read, write, delete, execute, search, control, or any other method
- In real systems, the access matrix is sparse and dynamic.
 - need a flexible, efficient representation

Two Representations

- ACL - Access Control Lists
 - Columns of previous matrix
 - Permissions attached to Objects
 - ACL for file hotgossip: Terry, rw; Lynn, rw
- Capabilities
 - Rows of previous matrix
 - Permissions associated with Subject
 - Tickets, Namespace (what it is that one can name)
 - Capabilities held by Lynn: luvltr, rw; hotgossip,rw

42

Access Control Lists

- *Approach*: represent the access matrix by storing its columns with the objects.
 - Tag each object with an *access control list* (ACL) of authorized subjects/principals.
- To authorize an access requested by *S* for *O*
 - search *O*'s ACL for an entry matching *S*
 - compare requested access with permitted access
 - access checks are often made only at bind time

Capabilities

- *Approach*: represent the access matrix by storing its rows with the subjects.
 - Tag each subject with a list of *capabilities* for the objects it is permitted to access.
 - A *capability* is an unforgeable object reference, like a pointer.
 - It endows the holder with permission to operate on the object
 - e.g., permission to invoke specific methods
 - Typically, capabilities may be passed from one subject to another.
 - Rights propagation and confinement problems

Dynamics of Protection Schemes

- How to endow software modules with appropriate privilege?
 - What mechanism exists to bind principals with subjects?
 - e.g., setuid syscall, setuid bit
 - What principals should a software module bind to?
 - privilege of creator: but may not be sufficient to perform the service
 - privilege of owner or system: dangerous

Dynamics of Protection Schemes

- How to revoke privileges?
- What about adding new subjects or new objects?
- How to dynamically change the set of objects accessible (or vulnerable) to different processes run by the same user?
 - Need-to-know principle / Principle of minimal privilege
 - How do subjects change identity to execute a more privileged module?
 - protection domain, protection domain switch (*enter*)

46

Protection Domains

- Processes execute in a protection domain, initially inherited from subject
- Goal: to be able to change protection domains
- Introduce a level of indirection
- Domains become protected objects with operations defined on them: **owner**, **copy**, **control**

	gradefile	solutions	proj1	luv/tr	hotgossip	Domain0
TA	rw	rwo	xc	r		ctl
grp		r	rwx			enter
Terry					rw	
Lynn			rw	rw		
Domain0			r			

47

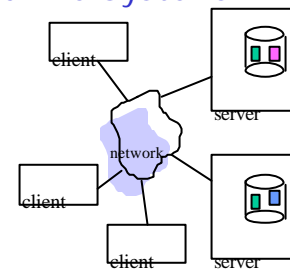
- If domain contains **copy** on right to some object, then it can *transfer* that right to the object to another domain.
- If domain is **owner** of some object, it can *grant* that right to the object, with or without **copy** to another domain
- If domain is **owner** or has **ctl** right to a domain, it can *remove* right to object from that domain
- Rights propagation.

	gradefile	solutions	proj1	luv/tr	hotgossip	Domain0
TA	rw	rwo	xc	r		ctl
grp		r	rwo			
Terry		rc			rw	
Lynn			rw	rw		enter
Domain0			r			

48

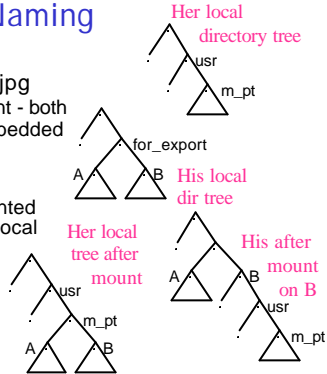
Distributed File Systems

- Naming
 - Location transparency/independence
- Caching
 - Consistency
- Replication
 - Availability and updates



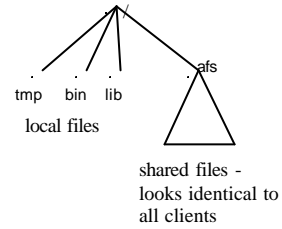
Naming

- \\His\d\pictures\castle.jpg
 - Not location transparent - both machine and drive embedded in name.
- NFS mounting
 - Remote directory mounted over local directory in local naming hierarching.
 - /usr/m_pt/A
 - No global view



Global Name Space

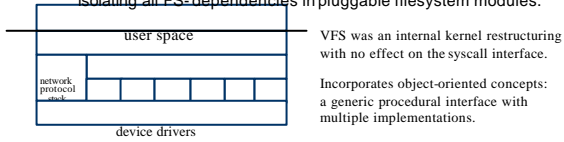
Example: Andrew File System



VFS: the Filesystem Switch

Sun Microsystems introduced the *virtual file system* framework in 1985 to accommodate the Network File System cleanly.

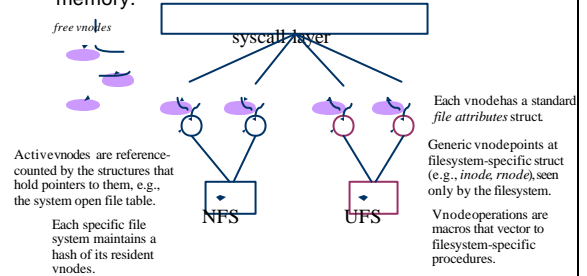
- VFS allows diverse *specific file systems* to coexist in a file tree, isolating all FS-dependencies in pluggable filesystem modules.

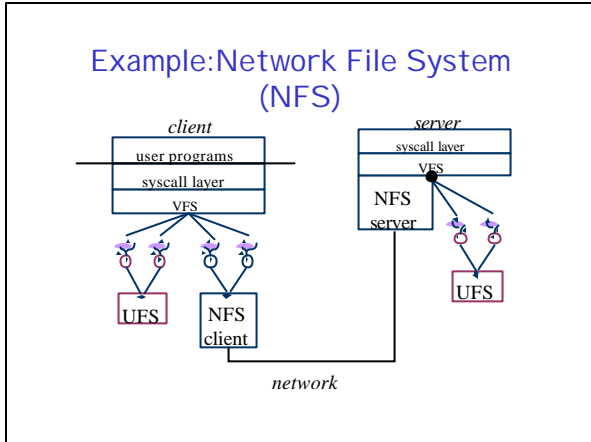


Other abstract interfaces in the kernel: device drivers, file objects, executable files, memory objects.

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 file attributes (vattr or fattr)
 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

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_readdir (uio, cookie)
 vop_symlink (OUT vpp, name, vattr, contents)
 vop_readlink (uio)

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

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

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:

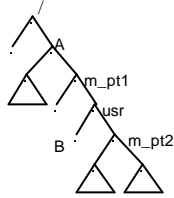
- crossing mount points
- obtaining root vnode (or current dir)
- finding resident vnodes in memory
- caching name->vnode translations
- symbolic (soft) links
- disk implementation of directories
- locking/referencing to handle races with name create and delete operations

Hints

- A valuable distributed systems design technique that can be illustrated in naming.
- Definition: information that is not guaranteed to be correct. If it is, it can improve performance. If not, things will still work OK. Must be able to validate information.
- Example: Sprite prefix tables

Prefix Tables

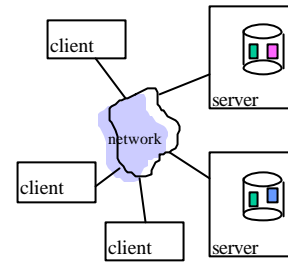
/A/m_pt1 -> blue
/A/m_pt1/usr/B -> pink
/A/m_pt1/usr/m_pt2 -> pink



/A/m_pt1/usr/m_pt2/stuff.below

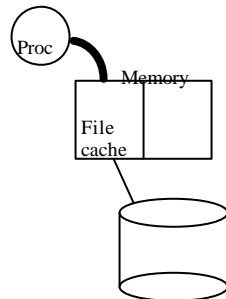
Distributed File Systems

- Naming
 - Location transparency/ independence
- Caching
 - Consistency
- Replication
 - Availability and updates



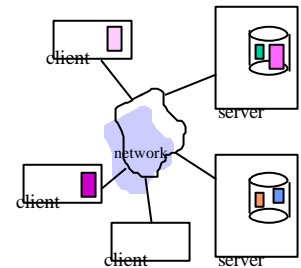
Caching was "The Answer"

- Avoid the disk for as many file operations as possible.
- Cache acts as a filter for the requests seen by the disk - reads served best.
- Delayed writeback will avoid going to disk at all for temp files.



Caching in Distributed F.S.

- Location of cache on client - disk or memory
- Update policy
 - write through
 - delayed writeback
 - write-on-close
- Consistency
 - Client does validity check, contacting server
 - Server call-backs



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.

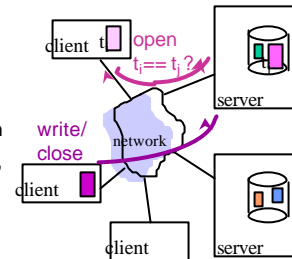
Callback invalidation (AFS).

Request notification (callback) from the server if the file changes; invalidate cache on callback.

Leases (NQ-NFS) [Gray&Cheriton89]

Sun NFS Cache Consistency

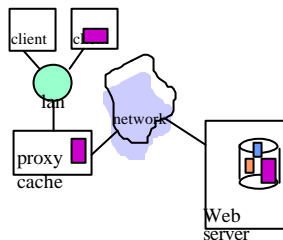
- Server is **stateless**
- Requests are self-contained.
- **Blocks** are transferred and cached in memory.
- Timestamp of last known mod kept with cached file, compared with "true" timestamp at server on Open. (Good for an interval)
- Updates delayed but flushed before Close ends.



63

Cache Consistency for the Web

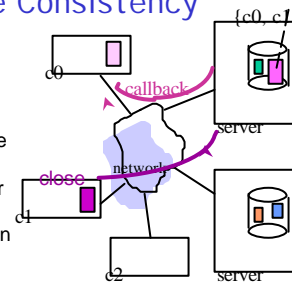
- Time-to-Live (TTL) fields - HTTP "expires" header
- Client polling - HTTP "if-modified-since" request headers
 - polling frequency? possibly *adaptive* (e.g. based on age of object and assumed stability)



64

AFS Cache Consistency

- Server keeps state of all clients holding copies (copy set)
- **Callbacks** when cached data are about to become stale
- Large units (whole files or 64K portions)
- Updates propagated upon close
- Cache on local disk & memory
- If client crashes, revalidation on recovery (lost callback possibility)



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.

Leases may be revoked by the server if another client requests a conflicting operation (server sends *eviction notice*).

Since leases expire, losing "state" of leases at server is OK.

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.

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.

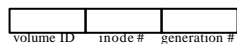
File Handles

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.



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?

Nachos File Syscalls/Operations

```

Create("zot");

OpenFile fd;
fd = Open("zot");
Close(fd);

char data[bufsize];
Write(data, count, fd);
Read(data, count, fd);

```

FileSystem class internal methods:
 Create(name, size)
 OpenFile = Open(name)
 Remove(name)
 List()

BitMap
 Bitmap indicates whether each disk block is in use or free.

Directory
 A single 10-entry directory stores names and disk locations for all currently existing files.

Filesystem data structures reside on disk, with a copy in memory.

Limitations:

1. small, fixed-size files and directories
2. single disk with a single directory
3. stream files only: no seek syscall
4. file size is specified at creation time
5. no access control, etc.

Representing A File in Nachos

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

OpenFile
 OpenFile(sector)
 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.

FileHdr

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

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

File Metadata

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

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

file attributes may include owner, access control, time of create/modify/access, etc.

The **FileHdr** is a file system "bookkeeping" structure that supplements the file data itself; these kinds of structures are called filesystem metadata.

logical-physical block map (like a translation table)

physical block pointers in the block map are sector IDs

A Nachos **FileHdr** occupies exactly one disk sector.

To operate on the file (e.g., to open it), the **FileHdr** must be read into memory.

Any changes to the attributes or block map must be written back to the disk to make them permanent.

FileHdr* hdr = new FileHdr();
 hdr->FetchFrom(sector)
 hdr->WriteBack(sector)

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) + ...

Nachos Directories

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

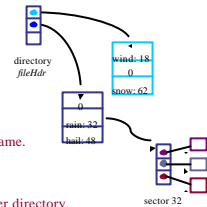
Each directory is a file containing a set of mappings from *name* → *FileHdr*.

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

In Nachos, each directory entry is a fixed-size slot with space for a *FileNameMaxLen* byte name.

Entries or slots are found by a linear scan.

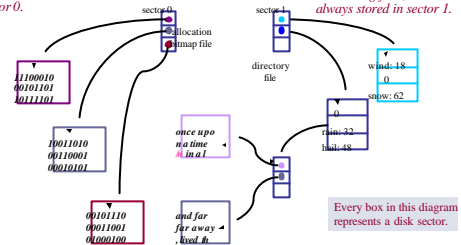
A directory entry may hold a pointer to another directory, forming a hierarchical name space.



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.



Nachos File System Classes

