

Outline for Today's Lecture

Administrative:

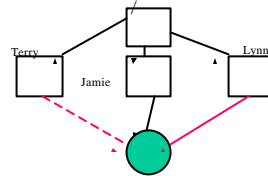
- TAs expect to be in Teer tonight
- Demo signups

Objective:

- File system naming issues continued
- Distributed file systems (naming)
- Down a level - files themselves

Soft vs. Hard Links

What's the difference in behavior?



Unix File Naming (Hard Links)

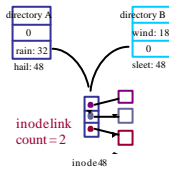
A Unix file may have multiple names.

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

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

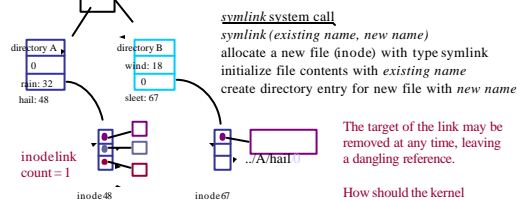
linksystem call
link (existing name, new name)
create a new name for an existing file
increment inode link count

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

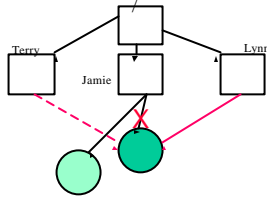
- Unix files may also be named by *symbolic (soft) links*.
- A *soft link* is a file containing a pathname of some other file.



Convenience, but *not* performance!

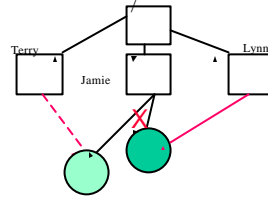
Soft vs. Hard Links

What's the difference in behavior?



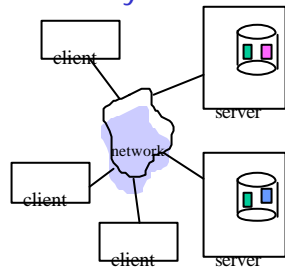
Soft vs. Hard Links

What's the difference in behavior?



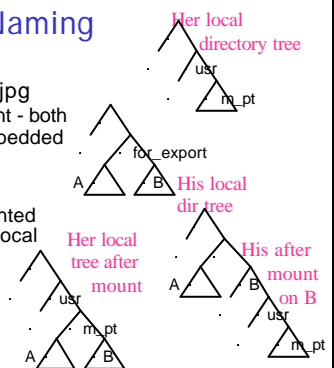
Distributed File Systems

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



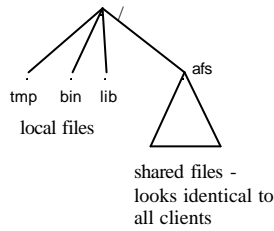
Naming

- `\\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

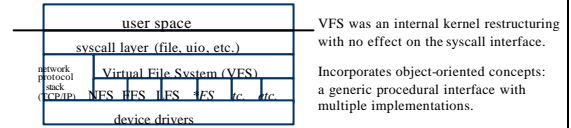
Example: Andrew File System



VFS: the Filesystem Switch

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

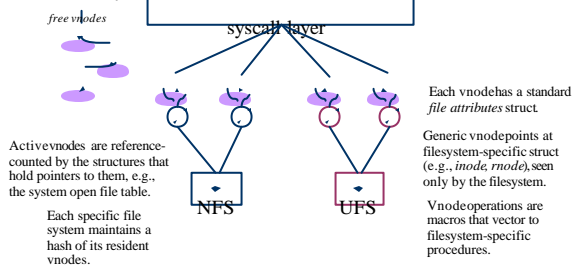
- VFS allows diverse *specific file systems* to coexist in a file tree, isolating all FS- dependencies in pluggable filesystem modules.



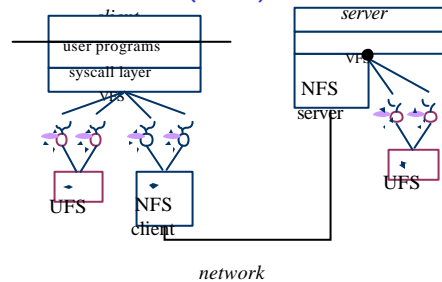
Other abstract interfaces in the kernel: device drivers, file objects, executable files, memory objects.

Vnodes

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



Example: Network File System (NFS)



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



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_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 ()

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");
```

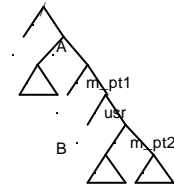
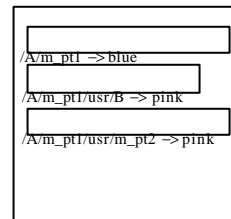
Issues:

- 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/usr/m_pt2/stuff.below

Performance Issue re:Naming What to do about long paths?

- Make long lookups cheaper – cluster inodes and data on disk to make each component resolution step somewhat cheaper
 - Immediate files – meta-data and first block of data co-located
- Collapse prefixes of paths – hash table
 - Prefix table
- “Cache it” – in this case, directory info

Meta-Data

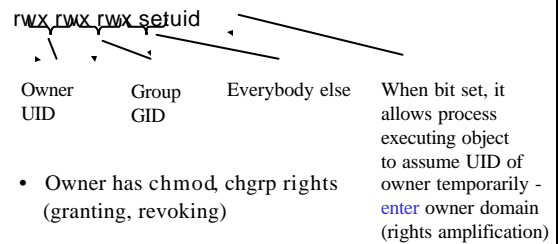
- File size
- File type
- Protection - access control information
- History: creation time, last modification, last access.
- Location of file - which device
- Location of individual blocks of the file on disk.
- Owner of file
- Group(s) of users associated with file

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

UNIX 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

Access Control Matrix

- Processes execute in a protection domain, initially inherited from subject

	gradefile	solutions	proj	luvltr	hotgossip
TA	rw	rw	r	r	
grp		r	rw		
Terry					rw
Lynn			rw	rw	

22

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

23

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

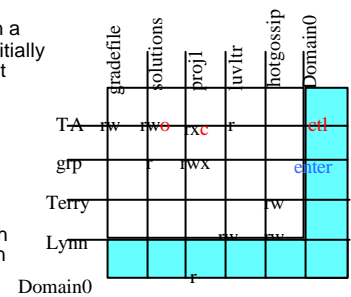
Dynamics of Protection Schemes

- 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**)

27

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**



28

- 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	rc	r		ctl
grp		r	rwo			
Terry		rc			rw	
Lynn				rw	rw	enter
Domain0			r			

29

Finally Arrive at File

- What do users *seem* to want from the file abstraction?
- What do these usage patterns mean for file structure and implementation decisions?
 - What operations should be optimized 1st?
 - How should files be structured?
 - Is there temporal locality in file usage?
 - How long do files really live?

Generalizations from UNIX Workloads

- Standard Disclaimers that you can't generalize...but anyway...
- Most files are small (fit into one disk block) although most bytes are transferred from longer files.
- Most opens are for read mode, most bytes transferred are by read operations
- Accesses tend to be sequential and 100%

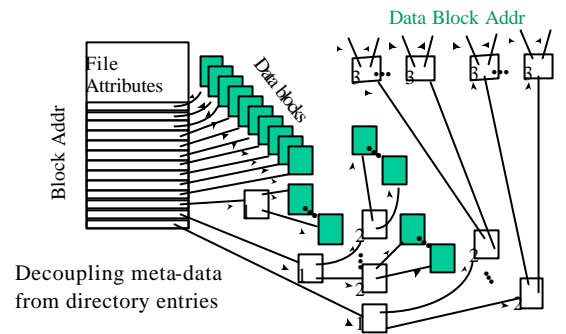
More on Access Patterns

- There is significant reuse (re-opens) – most opens go to files repeatedly opened & quickly. Directory nodes and executables also exhibit good temporal locality.
 - Looks good for caching!
- Use of temp files is significant part of file system activity in UNIX – very limited reuse, short lifetimes (less than a minute).

File Structure Implementation: Mapping File → Block

- Contiguous
 - 1 block pointer, causes fragmentation, growth is a problem.
- Linked
 - each block points to next block, directory points to first, OK for sequential access
- Indexed
 - index structure required, better for random access into file.

UNIX Inodes



File Allocation Table (FAT)

