## Outline for 11/14

Administrative:

- ACM meeting tonight at 7pm D106 LSRC on why's and how's of going to grad school in CPS
- This weekend is ACM programming contest in LSRC

Objective:

- NTFS – another case study
- Distributed File Systems

## NT File System

Another Case Study

## NTFS Files

- Files consist of multiple “streams”
  - Filename is a stream
  - One long unnamed data stream
  - Other data streams possible
    - Image in unnamed stream
    - Thumbnail in named stream
  - Filename:stream1

## NTFS System Calls

| Win32 API function | UNIX | Description |
|---|---|---|
| CreateFile | open | Create a file or open an existing file; return a handle |
| DeleteFile | unlink | Destroy an existing file |
| CloseHandle | close | Close a file |
| ReadFile | read | Read data from a file |
| WriteFile | write | Write data to a file |
| SetFilePointer | lseek | Set the file pointer to a specific place in the file |
| GetFileAttributes | stat | Return the file properties |
| LockFile | fcntl | Lock a region of the file to provide mutual exclusion |
| UnlockFile | fcntl | Unlock a previously locked region of the file |

- API functions for file I/O in Windows 2000
- Second column gives nearest UNIX equivalent

## File System API Example

```
/* Open files for input and output. */
inhandle = CreateFile("data", GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);
outhandle = CreateFile("newf", GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
    FILE_ATTRIBUTE_NORMAL, NULL);

/* Copy the file. */
do {
    s = ReadFile(inhandle, buffer, BUF_SIZE, &count, NULL);
    if (s && count > 0) WriteFile(outhandle, buffer, count, &ocnt, NULL);
} while (s > 0 && count > 0);

/* Close the files. */
CloseHandle(inhandle);
CloseHandle(outhandle);
```

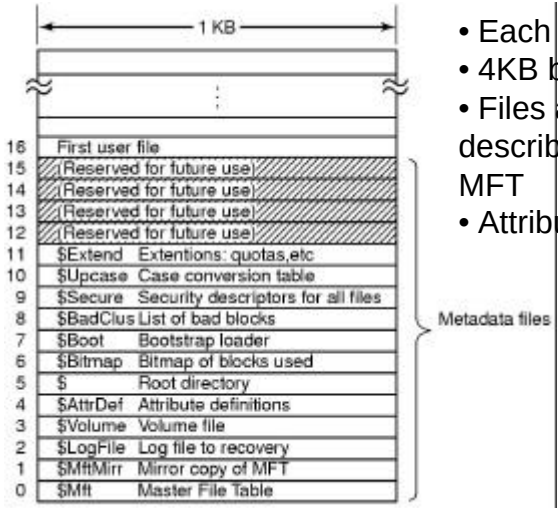
A program fragment for copying a file using the Windows 2000 API functions

## Directory System Calls

| Win32 API function | UNIX | Description |
|---|---|---|
| CreateDirectory | mkdir | Create a new directory |
| RemoveDirectory | rmdir | Remove an empty directory |
| FindFirstFile | opendir | Initialize to start reading the entries in a directory |
| FindNextFile | readdir | Read the next directory entry |
| MoveFile | rename | Move a file from one directory to another |
| SetCurrentDirectory | chdir | Change the current working directory |

- API functions for directory management in Windows 2000
- Second column gives nearest UNIX equivalent, when one exists

# File System Structure




- Each volume has its MFT
- 4KB blocks
- Files and directories described by 1KB records in MFT
- Attribute/value pairs

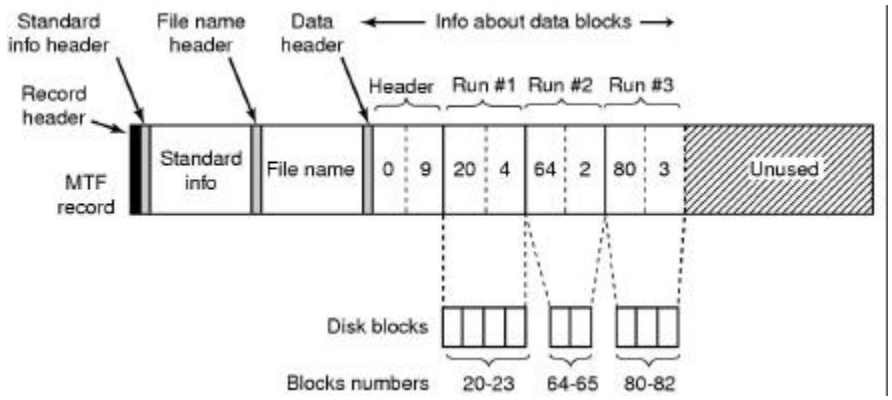
The NTFS master file table

# File System Structure

| Attribute | Description |
|---|---|
| Standard information | Flag bits, timestamps, etc. |
| File name | File name in Unicode; may be repeated for MS-DOS name |
| Security descriptor | Obsolete. Security information is now in $Extend$Secure |
| Attribute list | Location of additional MFT records, if needed |
| Object ID | 64-bit file identifier unique to this volume |
| Reparse point | Used for mounting and symbolic links |
| Volume name | Name of this volume (used only in $Volume) |
| Volume information | Volume version (used only in $Volume) |
| Index root | Used for directories |
| Index allocation | Used for very large directories |
| Bitmap | Used for very large directories |
| Logged utility stream | Controls logging to $LogFile |
| Data | Stream data; may be repeated |

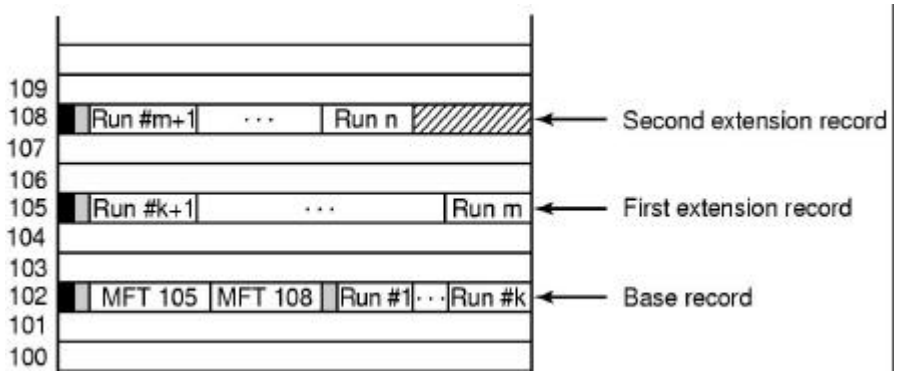
The attributes used in MFT records

# File System Structure



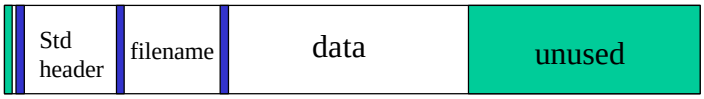

An MFT record for a three-run, nine-block file

# File System Structure



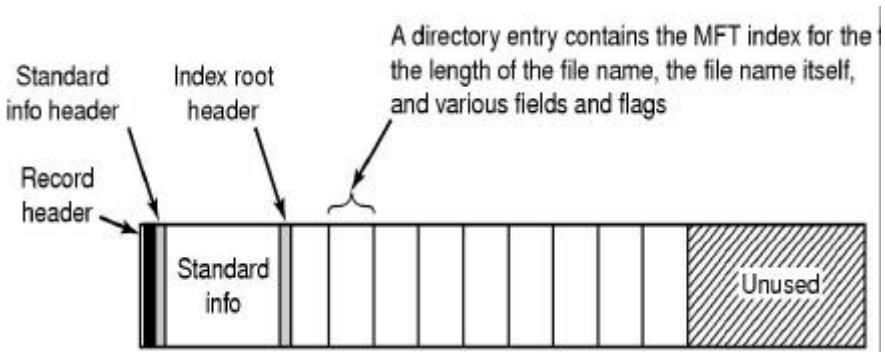

A file that requires three MFT records to store its runs

## Immediate Files



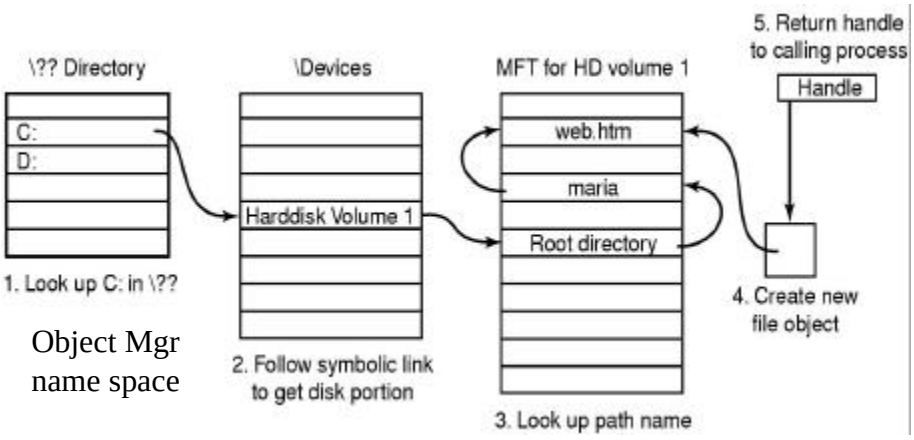

Data resides directly in entry for short file.

## File System Structure



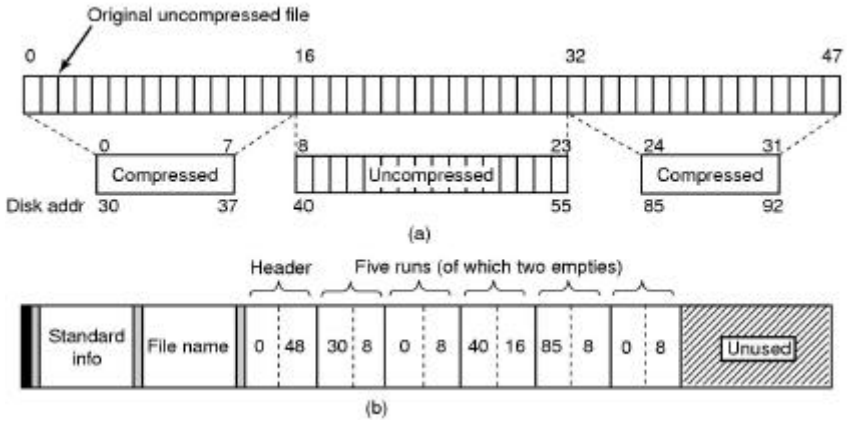

The MFT record for a small directory.

# File Name Lookup




Steps in looking up the file *C:\maria\web.htm*

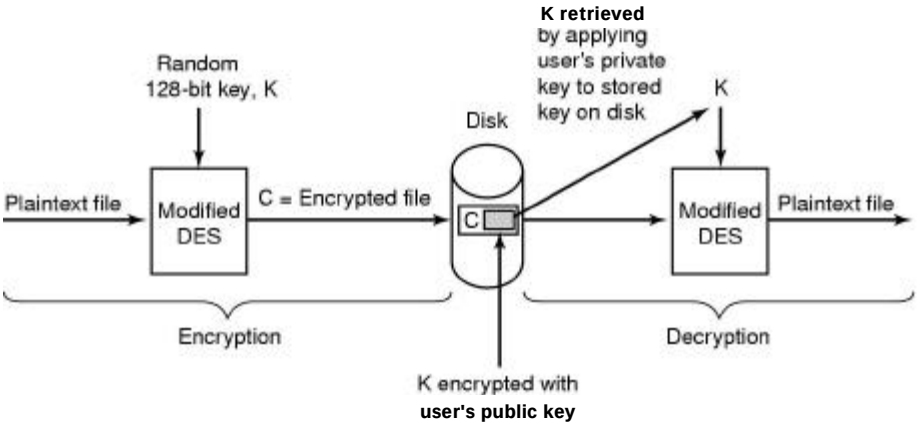
# File Compression




(a) An example of a 48-block file being compressed to 32 blocks

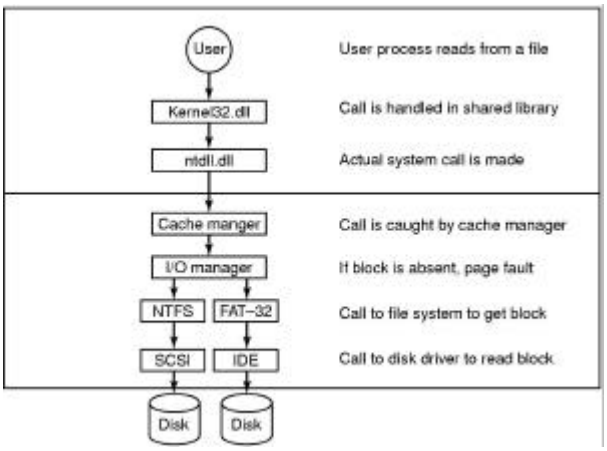
(b) The MTF record for the file after compression

# File Encryption




Operation of the encrypting file system

# Caching in Windows 2000




The path through the cache to the hardware

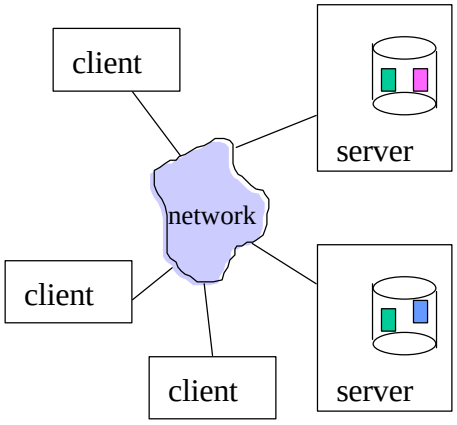
## Comparisons

| | FFS | LFS | NTFS |
|---|---|---|---|
| Data blocks | Clustering, Cylinder grouping | Log: contiguous by temporal ordering | Runs of contiguous blocks possible, immed. |
| Directories | Directory nodes, cylinder grouping | Directory nodes, in log | They *are* MFT entries |
| Block indices | Inodes, specified loc in cylinder group | Inodes in log | In MFT entries for files |

## Distributed File Systems

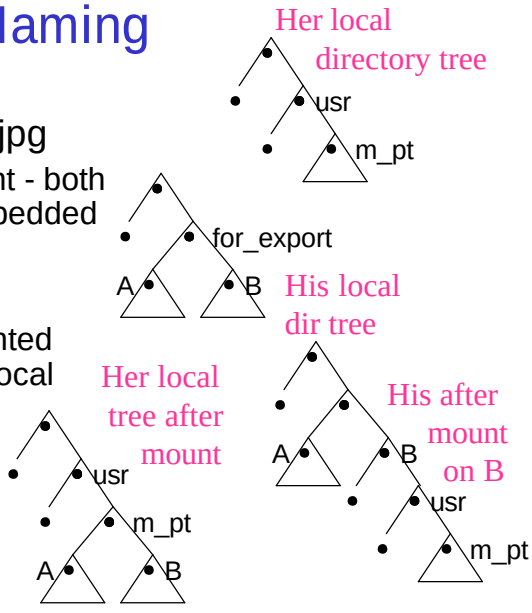
# Distributed File Systems

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



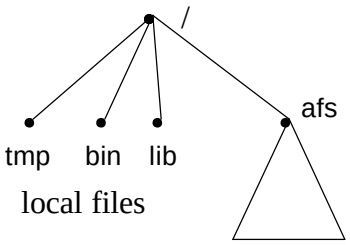

# Naming

- \\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



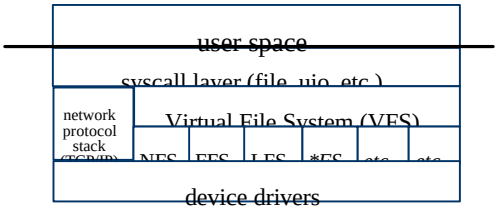

local files

shared files -
looks identical to
all clients

# 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.



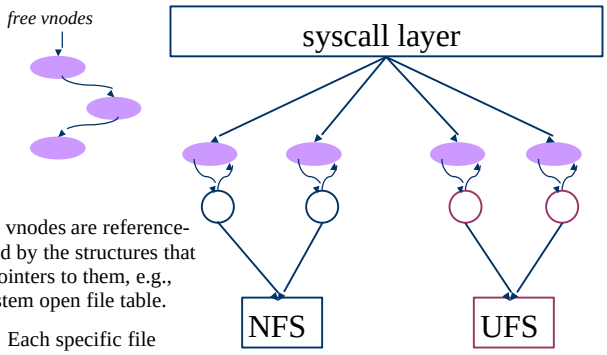

VFS was an internal kernel restructuring with no effect on the syscall interface.

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

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.




Active vnodes are reference-counted by the structures that hold pointers to them, e.g., the system open file table.

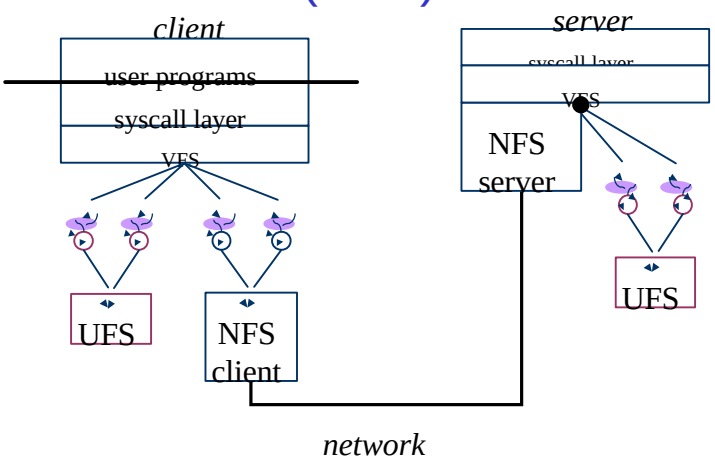
Each specific file system maintains a hash of its resident vnodes.

Each vnode has a standard *file attributes* struct.

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

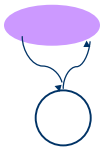
Vnode operations are macros that vector to filesystem-specific procedures.

# 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:
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

# Hints

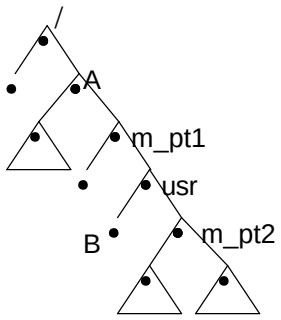
- A valuable distributed systems design technique that can be illustrated in naming.
- Definition: information that is not guaranteed to be correct. If it is, it can improve performance. If not, things will still work OK. Must be able to validate information.
- Example: Sprite prefix tables

# Prefix Tables

~~/A/m_pt1 –> blue~~

~~/A/m_pt1/usr/B –> pink~~

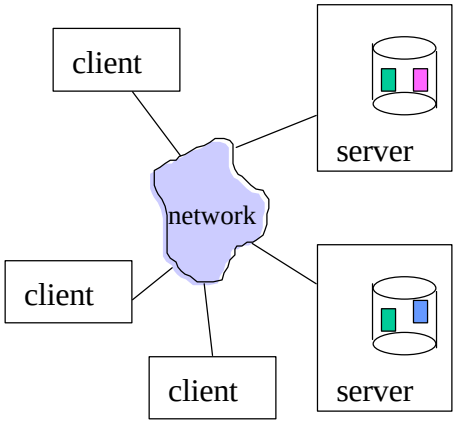
~~/A/m_pt1/usr/m_pt2 –> pink~~




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

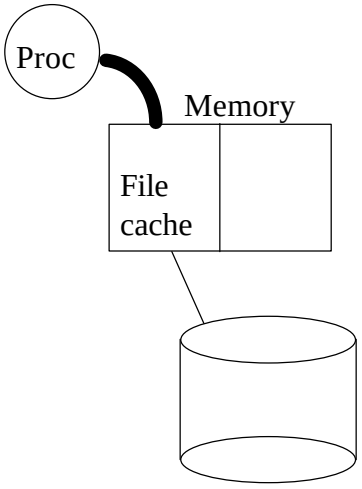
## Distributed File Systems

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



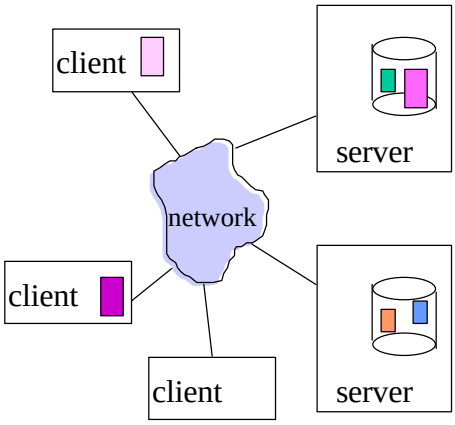

## Caching was "The Answer"

- 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.

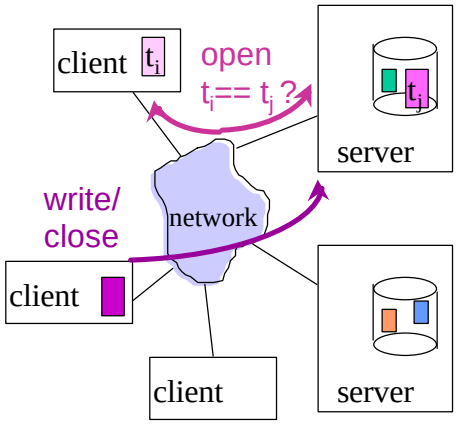
*Callback invalidation* (AFS).

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

*Leases* (NQ-NFS) [Gray&Cheriton89]

## Sun NFS Cache Consistency

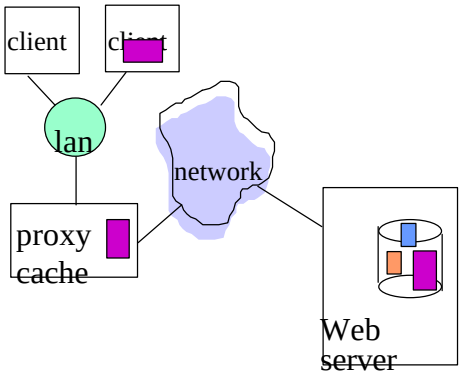
- Server is ***stateless***
- Requests are self-contained.
- ***Blocks*** are transferred and cached in memory.
- Timestamp of last known mod kept with cached file, compared with “true” timestamp at server on Open.
  (Good for an interval)
- Updates delayed but flushed before Close ends.




33

## Cache Consistency for the Web

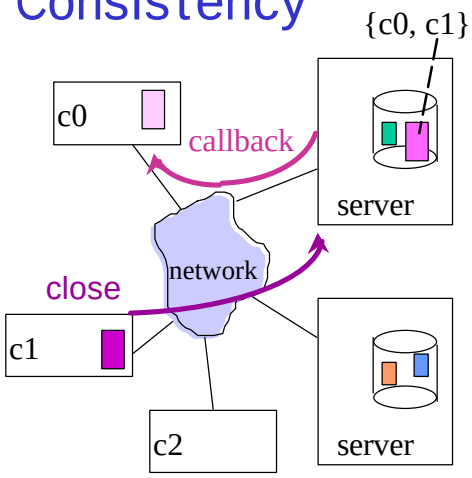
- Time-to-Live (TTL) fields - HTTP “expires” header
- Client polling -HTTP “if-modified-since” request headers
  - polling frequency? possibly *adaptive* (e.g. based on age of object and assumed stability)




34

# 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)

15

# 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.

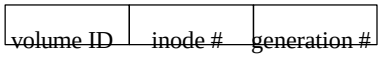
## File Handles

**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

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?