# LFS, NFS

# Log-Structured File System

UCR

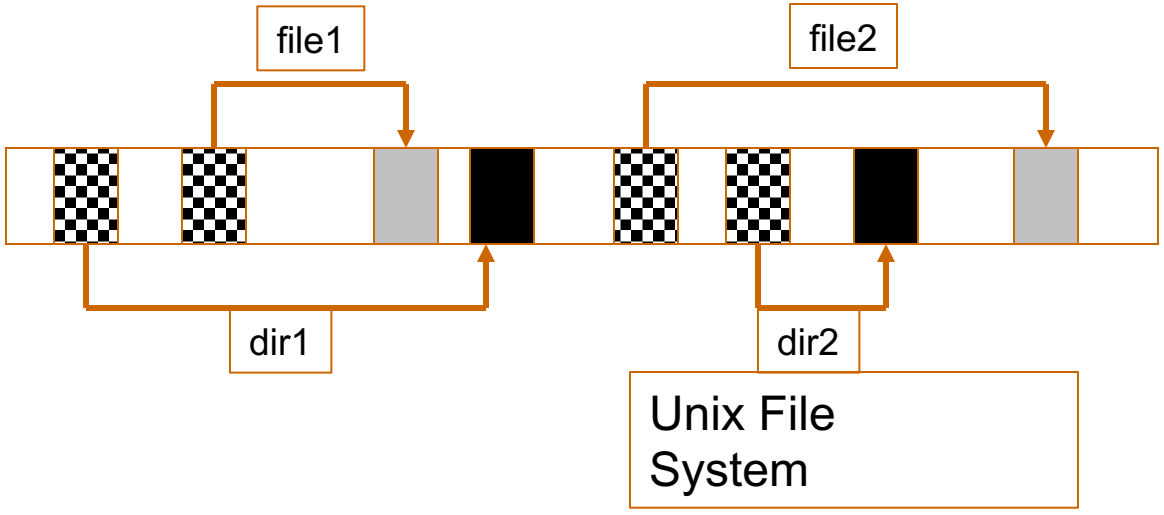
- Radically different file system design
- Technology motivations:
  - CPUs outpacing disks: I/O is bottleneck
  - Most disk traffic writes due to caching
- Problems with (then) current file systems:
  - Lots of little writes
  - Synchronous: wait for disk in too many places – makes it hard to win much from RAIDs, too little concurrency
  - 5 seeks to create a new file: (rough order): i-node create, data write, directory entry, i-node finalize, directory i-node (not to mention bitmap update)

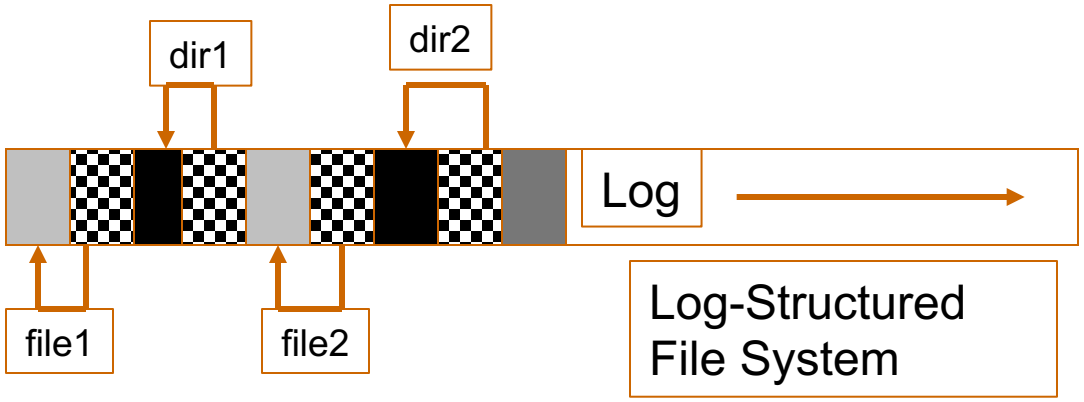
# LFS Basic Idea

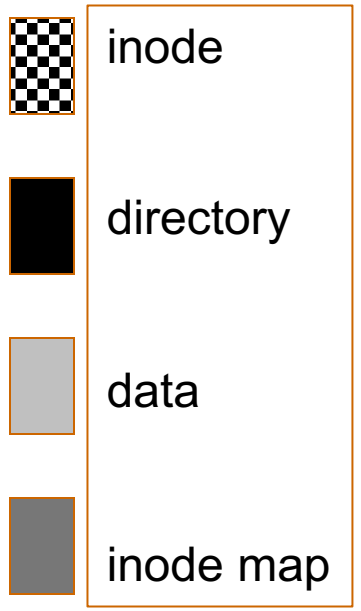
UCR

- Log all data and metadata with efficient, large, sequential writes
  - Do not update blocks in place – just write new versions in the log
- Treat the log as the truth, but keep an index on its contents
- Not necessarily good for reads, but trends help
  - Rely on a large memory to provide fast access through caching
- Data layout on disk has “temporal locality” (good for writing), rather than “logical locality” (good for reading)
  - Why is this a better? Because caching helps reads but not writes!

# LFS vs. UFS



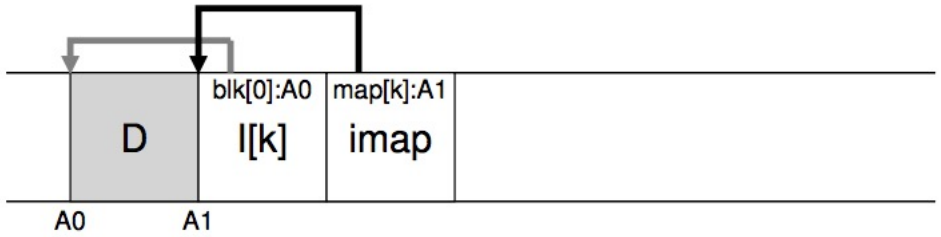




Blocks written to create two 1-block files: dir1/file1 and dir2/file2, in UFS and LFS

# Devil is in the details

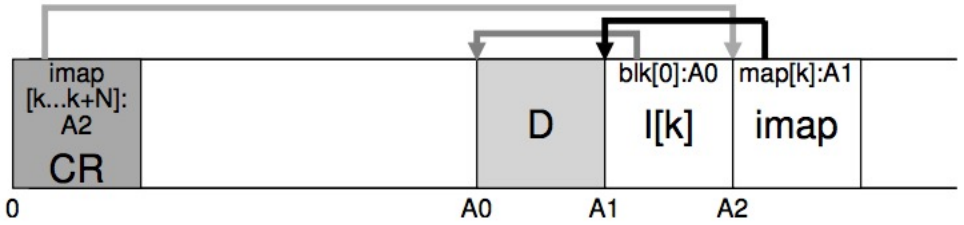
- Two potential problems:
  - Log retrieval on cache misses – how do we find the data?
  - Wrap-around: what happens when end of disk is reached?
    - No longer any big, empty runs available
    - How to prevent fragmentation?

# i-node map



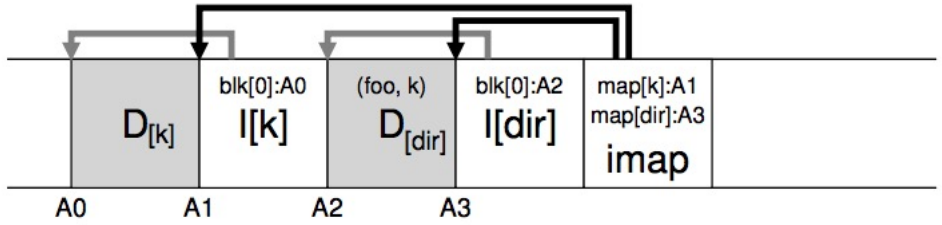

- A map keeping track of the location of i-nodes
- Anytime an i-node is written to disk, the imap is updated
  - But is that any better? In a second
- Most of the time the imap is in memory, so access is fast
- Updated imap is saved as part of the log!
  - but how do we find it!

# Final piece to the solution




- Checkpoint region is written to point to the location of the imap
  - Also serves as an indicator of a stable point in the file system for crash recovery
- So, to read a file from LFS:
  - Read the CR, use it to read and cache the imap
  - After that, it is identical to FFS
  - Are reads fast?

# What about directories?




- When a file is updated, its inode changes (new copy)
  - We need to update the directory inode (also creating a copy)
  - We need to update its parent directory

- Ugh….what to do?
  - Inode map helps with that too – just keep track of inode number and resolve it through inode map

# LFS Disk Wrap-Around/Garbage collection

UCR

- Compact live info to open up large runs of free space
  - Problem: long-lived information gets copied over-and-over
- Thread log through free spaces
  - Problem: disk fragments, causing I/O to become inefficient again

- Solution: *segmented log*
  - Divide disk into large, fixed-size segments
  - Do compaction within a segment; thread between segments
  - When writing, use only clean segments (i.e. no live data)
  - Occasionally clean segments
  - Try to collect long-lived info into segments that never need to be cleaned
  - Note there is not free list or bit map (as in FFS), only a list of clean segments
    - Related: TRIM command for SSDs

# LFS Segment Cleaning

UCR

- Which segments to clean?
  - Keep estimate of free space in each segment to help find segments with lowest utilization
  - Always start by looking for segment with utilization=0, since those are trivial to clean…
  - If utilization of segments being cleaned is U:
    - write cost = (total bytes read & written)/(new data written) = 2/(1-U) (unless U is 0)
    - write cost increases as U increases: U = .9 => cost = 20!
    - Need a cost of less than 4 to 10; => U of less than .75 to .45

# Evaluation Results

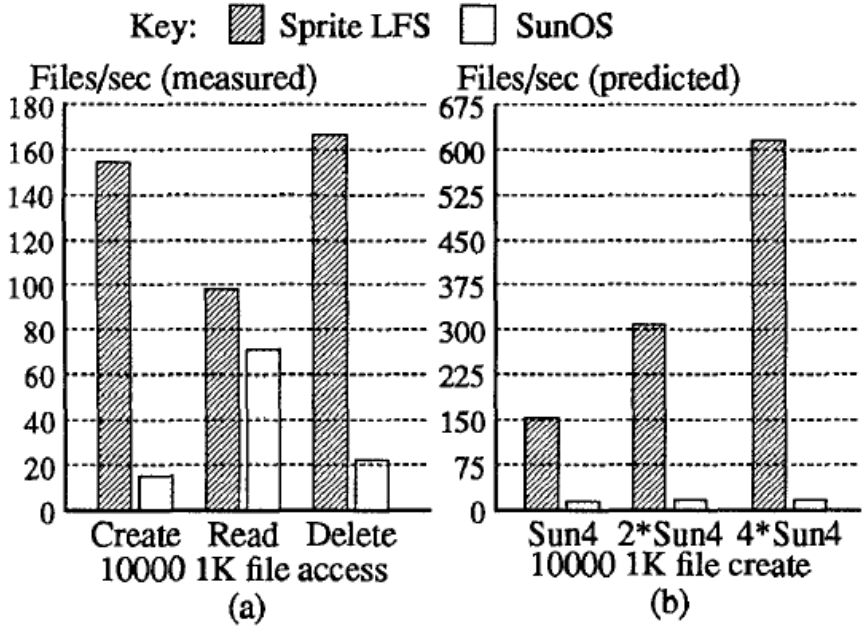



**Figure 8 — Small-file performance under Sprite LFS and SunOS.**
Figure (a) measures a benchmark that created 10000 one-kilobyte files, then read them back in the same order as created, then deleted them. Speed is measured by the number of files per second for each operation on the two file systems. The logging approach in Sprite LFS provides an order-of-magnitude speedup for creation and deletion. Figure (b) estimates the performance of each system for creating files on faster computers with the same disk. In SunOS the disk was 85% saturated in (a), so faster processors will not improve performance much. In Sprite LFS the disk was only 17% saturated in (a) while the CPU was 100% utilized; as a consequence I/O performance will scale with CPU speed.

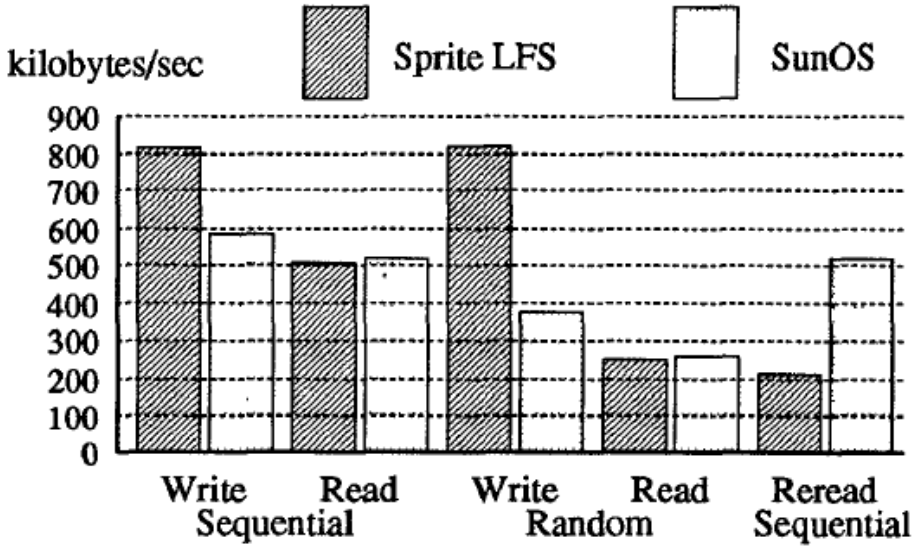



**Figure 9 — Large-file performance under Sprite LFS and SunOS.**
The figure shows the speed of a benchmark that creates a 100-Mbyte file with sequential writes, then reads the file back sequentially, then writes 100 Mbytes randomly to the existing file, then reads 100 Mbytes randomly from the file, and finally reads the file sequentially again. The bandwidth of each of the five phases is shown separately. Sprite LFS has a higher write bandwidth and the same read bandwidth as SunOS with the exception of sequential reading of a file that was written randomly.

# Is this a good paper?

UCR

- What were the authors' goals?
- What about the evaluation/metrics?
- Did they convince you that this was a good system/approach?
- Does the system/approach meet the "Test of Time" challenge?
  - WAFL commercial file system based on LFS
  - Most SSD file systems are log structured
- How would you review this paper today?

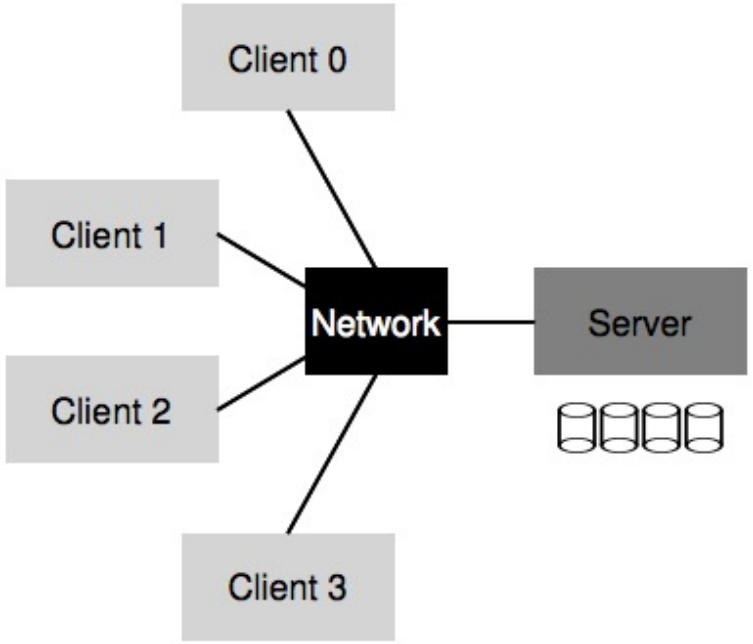
UNIVERSITY OF CALIFORNIA
UC RIVERSIDE

# DESIGN AND IMPLEMENTATION OF THE SUN NETWORK FILESYSTEM

R. Sandberg, D. Goldberg
S. Kleinman, D. Walsh, R. Lyon

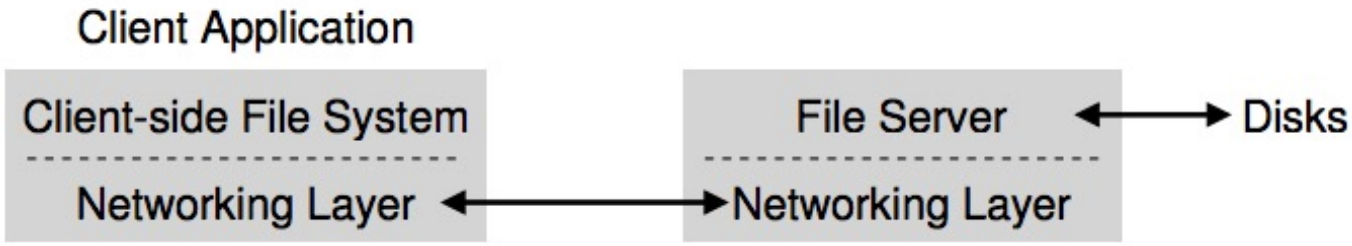
Sun Microsystems

# What is NFS?




- First commercially successful network file system:
  - Developed by Sun Microsystems for their diskless workstations
  - Designed for robustness and “adequate performance”
  - Sun published all protocol specifications
  - Many many implementations

# Overview and Objectives




- Fast and efficient crash recovery
  - Why do crashes occur?
- To accomplish this:
  - NFS is stateless – key design decision
    - All client requests must be self-contained
  - The virtual filesystem interface
    - VFS operations
    - VNODE operations

# Additional objectives

- ***Machine and Operating System Independence***
  - Could be implemented on low-end machines of the mid-80's
- ***Transparent Access***
  - Remote files should be accessed in exactly the same way as local files
- *UNIX semantics should be maintained on client*
  - Best way to achieve transparent access
- *"Reasonable" performance*
  - Robustness and preservation of UNIX semantics were much more important

# Example

```
char buffer[MAX];
int fd = open("foo", O_RDONLY); // get descriptor "fd"
read(fd, buffer, MAX);          // read MAX bytes from foo (via fd)
read(fd, buffer, MAX);          // read MAX bytes from foo
...
read(fd, buffer, MAX);          // read MAX bytes from foo
close(fd);                      // close file
```

- What if the client simply passes the open request to the server?
  - Server has state
  - Crash causes big problems
- Three important parts
  - The protocol
  - The server side
  - The client side

# The protocol (I)

- Uses the Sun RPC mechanism and Sun eXternal Data Representation (XDR) standard
- Defined as a set of remote procedures
- Protocol is ***stateless***
  - Each procedure call contains ***all the information necessary to complete the call***
  - Server maintains no “between call” information

# Advantages of statelessness

- Crash recovery is very easy:
  - When a server crashes, client just resends request until it gets an answer from the rebooted server
  - Client cannot tell difference between a server that has crashed and recovered and a slow server
- Client can always ***repeat any request***

# NFS as a “Stateless” Service

UCR

- A classical NFS server maintains no in-memory hard state.
  - The only hard state is the stable file system image on disk.
  - no record of clients or open files
  - no implicit arguments to requests
    - *E.g., no server-maintained file offsets*: **read** and **write** requests must explicitly transmit the byte offset for each operation.
  - no write-back caching on the server
  - no record of recently processed requests
  - etc., etc....
- *Statelessness makes failure recovery simple and efficient.*

# Consequences of statelessness

- Read and writes must specify their start offset
  - Server does not keep track of current position in the file
  - User still use conventional UNIX reads and writes
- Open system call translates into several lookup calls to server
- No NFS equivalent to UNIX close system call

# Important pieces of protocol

```
NFSPROC_GETATTR
  expects: file handle
  returns: attributes
NFSPROC_SETATTR
  expects: file handle, attributes
  returns: nothing
NFSPROC_LOOKUP
  expects: directory file handle, name of file/directory to look up
  returns: file handle
NFSPROC_READ
  expects: file handle, offset, count
  returns: data, attributes
NFSPROC_WRITE
  expects: file handle, offset, count, data
  returns: attributes
NFSPROC_CREATE
  expects: directory file handle, name of file, attributes
  returns: nothing
NFSPROC_REMOVE
  expects: directory file handle, name of file to be removed
  returns: nothing
NFSPROC_MKDIR
  expects: directory file handle, name of directory, attributes
  returns: file handle
NFSPROC_RMDIR
  expects: directory file handle, name of directory to be removed
  returns: nothing
NFSPROC_READDIR
  expects: directory handle, count of bytes to read, cookie
  returns: directory entries, cookie (to get more entries)
```

# From protocol to distributed file system

- Client side translates user requests to protocol messages to implement the request remotely
- Example:

| Client | Server |
| --- | --- |
| **fd = open("/foo", ...);** | |
| Send LOOKUP (rootdir FH, "foo") | |
| | Receive LOOKUP request<br>look for "foo" in root dir<br>return foo's FH + attributes |
| Receive LOOKUP reply<br>allocate file desc in open file table<br>store foo's FH in table<br>store current file position (0)<br>return file descriptor to application | |

# The lookup call (I)

- Returns a **file handle** instead of a file descriptor
  - File handle specifies unique location of file
    - Volume identifier, inode number and generation number
- **lookup(dirfh, name)** *returns* **(fh, attr)**
  - Returns file handle **fh** and attributes of named file in directory **dirfh**
  - Fails if client has no right to access directory **dirfh**

# Server side (I)

- Server implements a write-through policy
  - Required by statelessness
  - Any blocks modified by a write request (including i-nodes and indirect blocks) must be written back to disk before the call completes

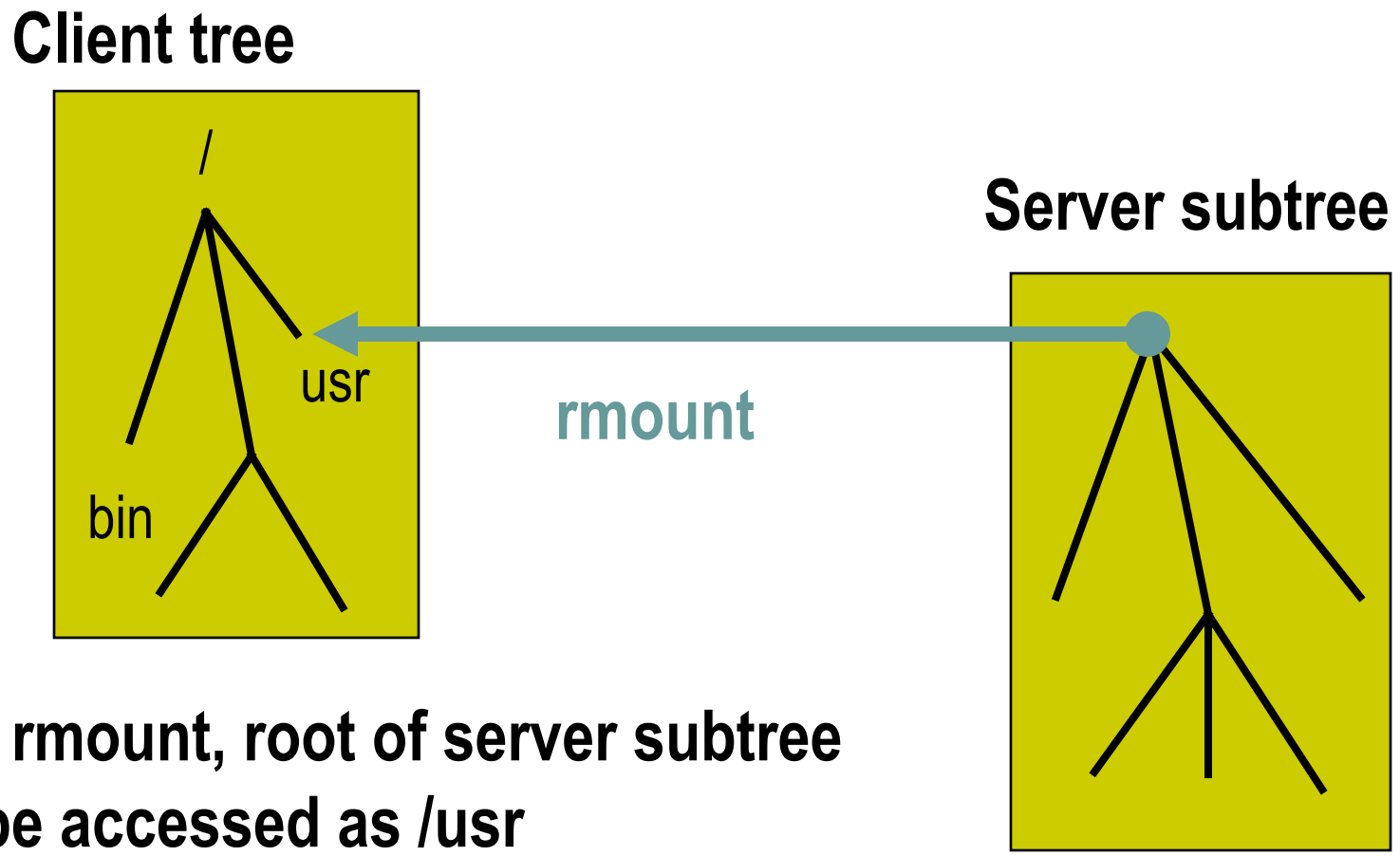
# Server side (II)

- File handle consists of
  - Filesystem id identifying disk partition
  - I-node number identifying file within partition
  - Generation number changed every time i-node is reused to store a new file
- Server will store
  - Filesystem id in filesystem superblock
  - I-node generation number in i-node

# Client side (I)

- Provides transparent interface to NFS
- Mapping between remote file names and remote file addresses is done a server boot time through ***remote mount***
  - Extension of UNIX mounts
  - Specified in a ***mount table***
  - Makes a remote subtree appear part of a local subtree

# Remote mount

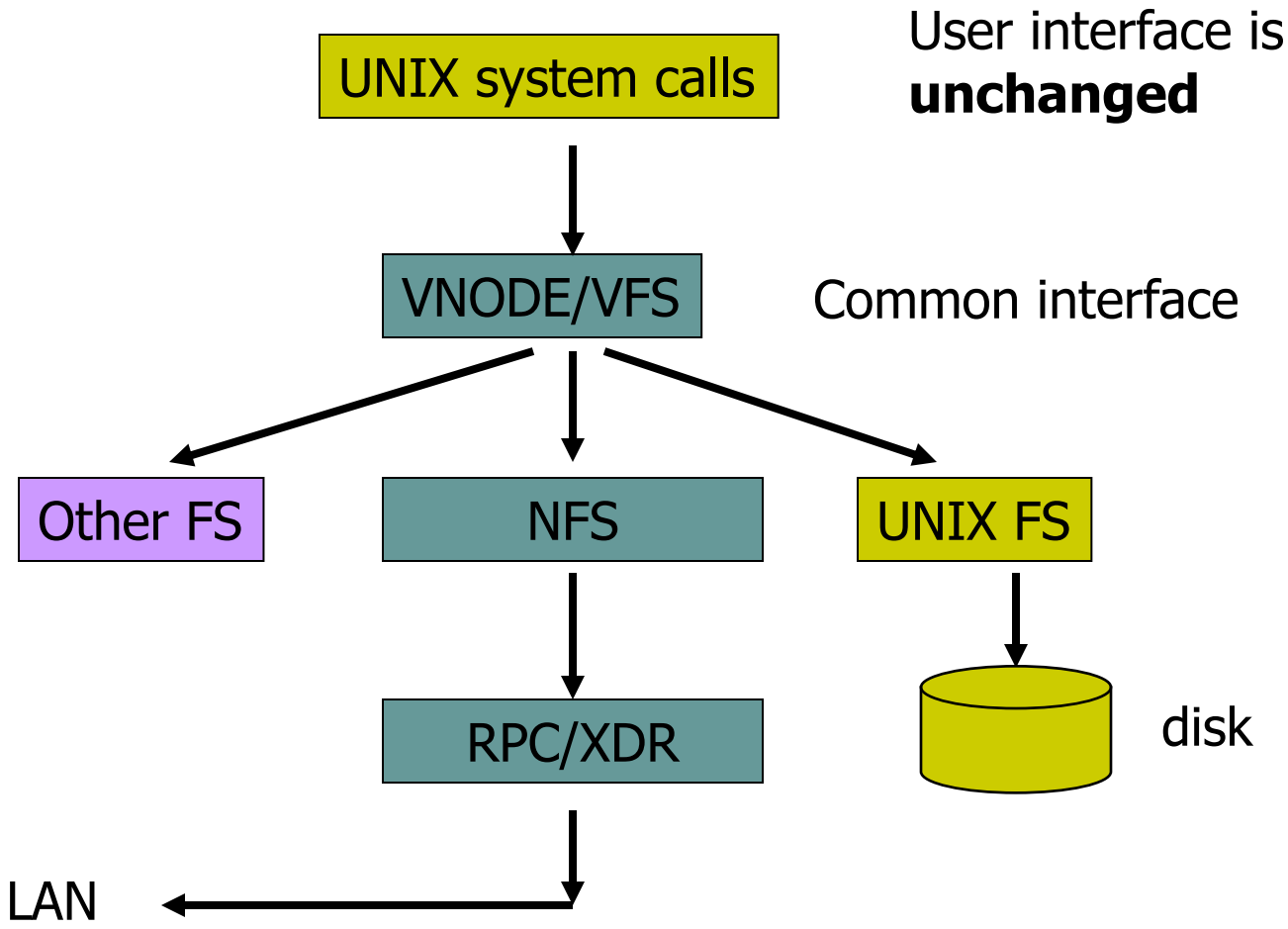



After rmount, root of server subtree can be accessed as /usr

# Client side (II)

- Provides transparent access to
  - NFS
  - Other file systems (including UNIX FFS)
- New virtual filesystem interface supports
  - VFS calls, which operate on whole file system
  - VNODE calls, which operate on individual files
- Treats all files in the same fashion

# Client side (III)



# More examples

**read(fd, buffer, MAX);**
Index into open file table with fd
get NFS file handle (FH)
use current file position as offset
Send READ (FH, offset=0, count=MAX)

Receive READ request
use FH to get volume/inode num
read inode from disk (or cache)
compute block location (using offset)
read data from disk (or cache)
return data to client

Receive READ reply
update file position (+bytes read)
set current file position = MAX
return data/error code to app