

# Distributed File Systems

CS432: Distributed Systems  
Spring 2017

# Reading

- Chapter 12 (12.1-12.4) [Coulouris '11]
- Chapter 11 [Tanenbaum '06]
- Section 4.3, “Modern Operating Systems, Fourth Ed.”, Andrew S. Tanenbaum
- Section 11.4, “Operating Systems Concept, Ninth Ed.”, Abraham Silberschatz, et al.

# Objectives

- Learn about the following:
  - Review file systems and the main requirements for designing a distributed file system
  - Famous architecture models of distributed file systems
- Study the design of three file systems NFS, AFS, and GFS (done!)

# Outline

- Introduction
  - Non-Distributed File System (Review)
  - File System Mounting
- Distributed File System Requirements
- Architecture of Distributed File Service
- Case Studies:
  - Sun Network File System (NFS)
  - Andrew File System (AFS)
  - ✓ • Google File System (GFS)


# File Systems

- File systems, in centralized computer systems, provide a convenient programming interface to disk storage
  - blocks of disks → files, directories, ..
  - storage allocation and layout
- Components:
  - Disk management: gathering disk blocks into files
  - Naming: help users find files by their name instead of block identifiers
  - Security: layers of permissions to access and modify files
  - Durability: data written to files should not be tampered with in case of failures

# File Components

- A file contains:
  - Data: sequence of data items that are accessible through read and write operations
  - Attributes: a single record containing information about the file

Managed by the file system  
Users do not typically update them



File length
Creation timestamp
Read timestamp
Write timestamp
Attribute timestamp
Reference count
Owner
File type
Access control list

# Reading and Writing

- Reading from the file system (e.g. `getc()`):
  - Fetch a block containing the required character
  - Return the requested character from the block
- Writing to the file system (e.g. `putc()`):
  - Modify existing data: fetch block, modify, and write.
  - Append data: buffer data until a block size is completed, then write

# Non-distributed File System Modules

- Each module depends only on the layers below it
- Note: the implementation of a distributed file service also requires additional components to deal with:  
**client-server communication, distributed naming, and location of files**

Directory module:	relates file names to file IDs
File module:	relates file IDs to particular files
Access control module:	checks permission for operation requested
File access module:	reads or writes file data or attributes
Block module:	accesses and allocates disk blocks
Device module:	disk I/O and buffering



# Unix File System Operations

<code>filedes = open(name, mode)</code>	Opens an existing file with the given <code>name</code> .
<code>filedes = creat(name, mode)</code>	Creates a new file with the given <code>name</code> .
	Both operations deliver a file descriptor referencing the open file. The <code>mode</code> is <code>read</code> , <code>write</code> or both.
<code>status = close(filedes)</code>	Closes the open file <code>filedes</code> .
<code>count = read(filedes, buffer, n)</code>	Transfers <code>n</code> bytes from the file referenced by <code>filedes</code> to <code>buffer</code> .
<code>count = write(filedes, buffer, n)</code>	Transfers <code>n</code> bytes to the file referenced by <code>filedes</code> from <code>buffer</code> .
	Both operations deliver the number of bytes actually transferred and advance the read-write pointer.
<code>pos = lseek(filedes, offset, whence)</code>	Moves the read-write pointer to <code>offset</code> (relative or absolute, depending on <code>whence</code> ).
<code>status = unlink(name)</code>	Removes the file <code>name</code> from the directory structure. If the file has no other names, it is deleted.
<code>status = link(name1, name2)</code>	Adds a new name ( <code>name2</code> ) for a file ( <code>name1</code> ).
<code>status = stat(name, buffer)</code>	Puts the file attributes for file <code>name</code> into <code>buffer</code> .

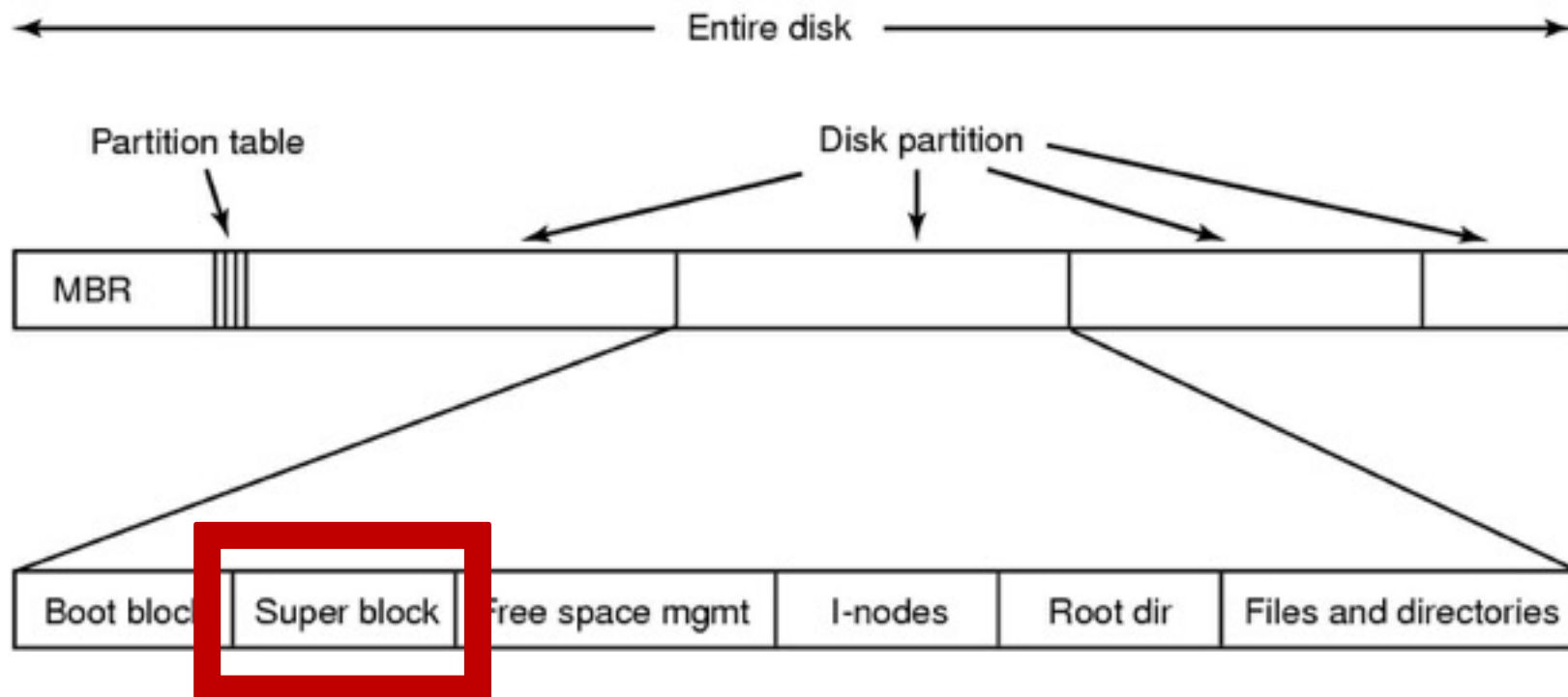
# File System Layout

- File systems are stored on disk.
- Disks are divided into one or more partitions, independent file system on each partition.
- Master Boot Record (MBR)
  - Sector 0 of the disk.
  - Used to boot the computer.
  - The end of the MBR contains the partition table.

# System Booting

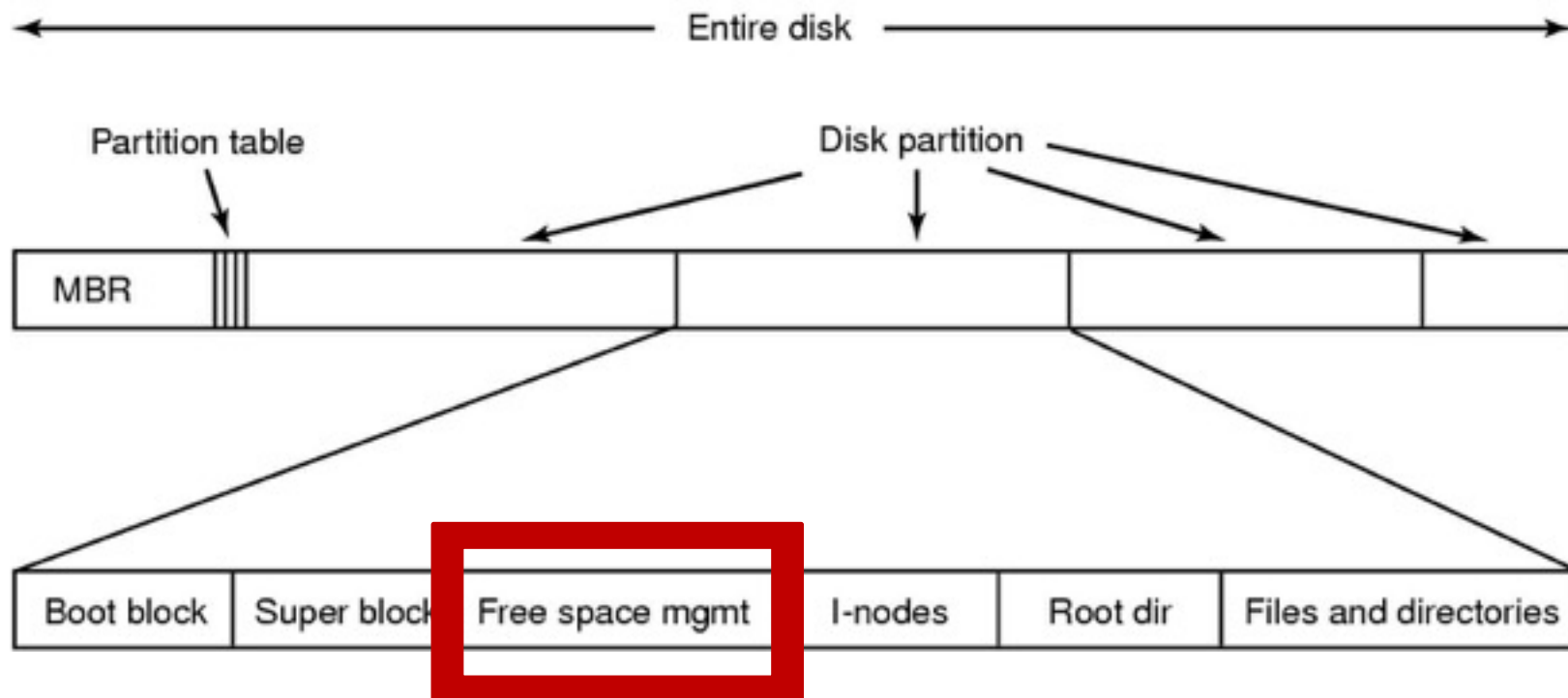
- The partition table gives the starting and ending addresses of each partition.
- One of the partitions in the table is marked as active.
- When the computer is booted, the BIOS reads it and executes the MBR.
- The MBR program locates the active partition, read in its first block (*boot block*) and executes it.
- The program in the boot block loads the operating system contained in that partition.
- Note: each partition starts with a boot block, even if it does not contain a bootable operating system.

# Example: File System Layout



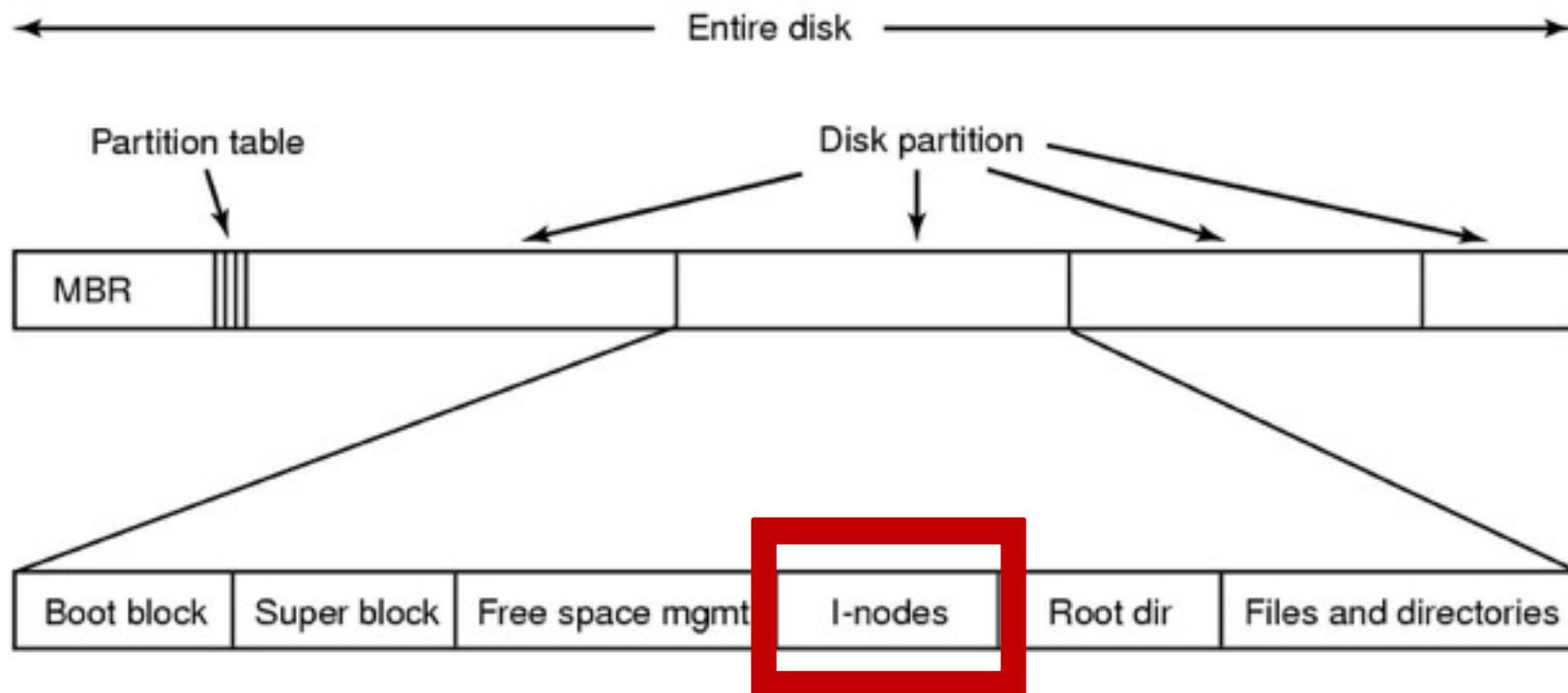
- Super block contains all the key parameters about the file system
- Read into memory when the computer is booted or the file system is first touched.
- Typical information: magic number to identify the file system type, the number of blocks in the file system, and other key administrative information.

# Example: File System Layout



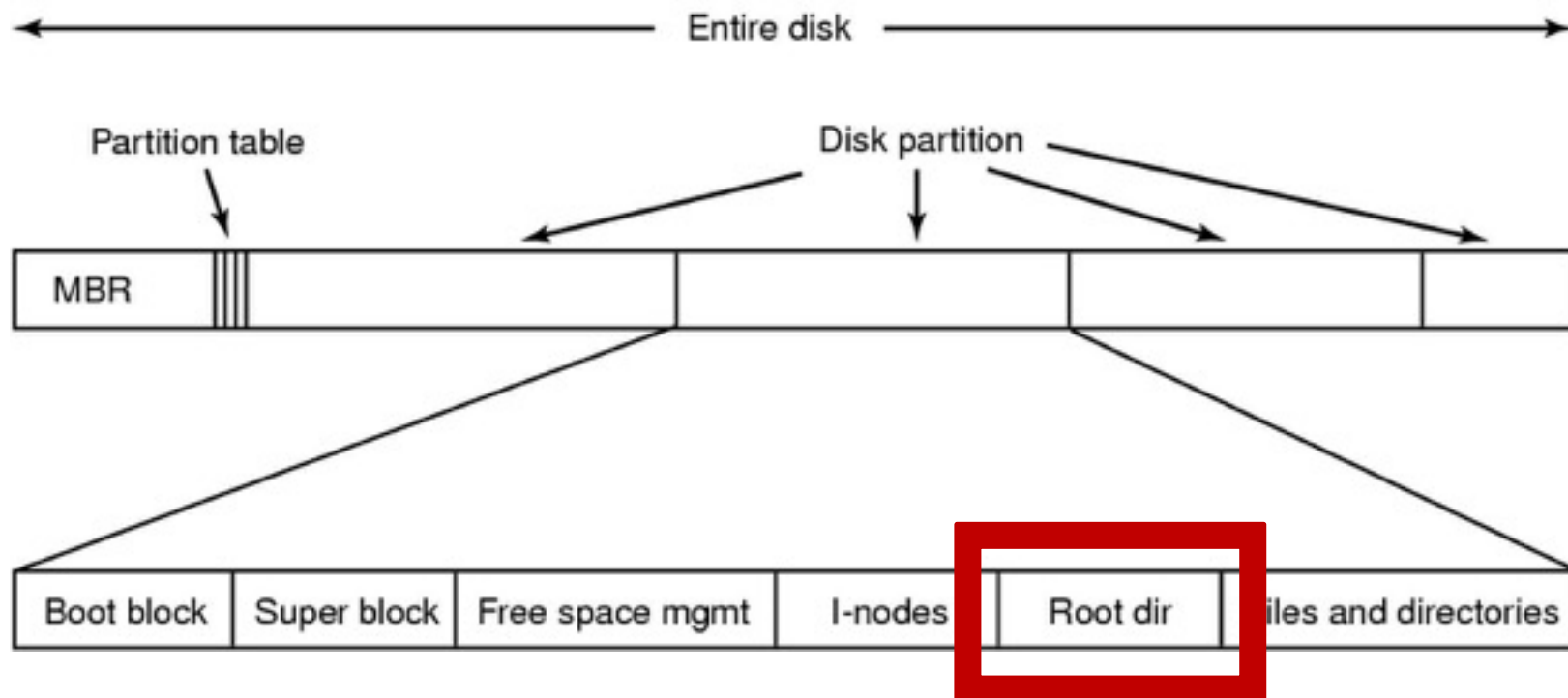
- Information about the free blocks in the file system.
- Example: bitmap, file pointers.

# Example: File System Layout



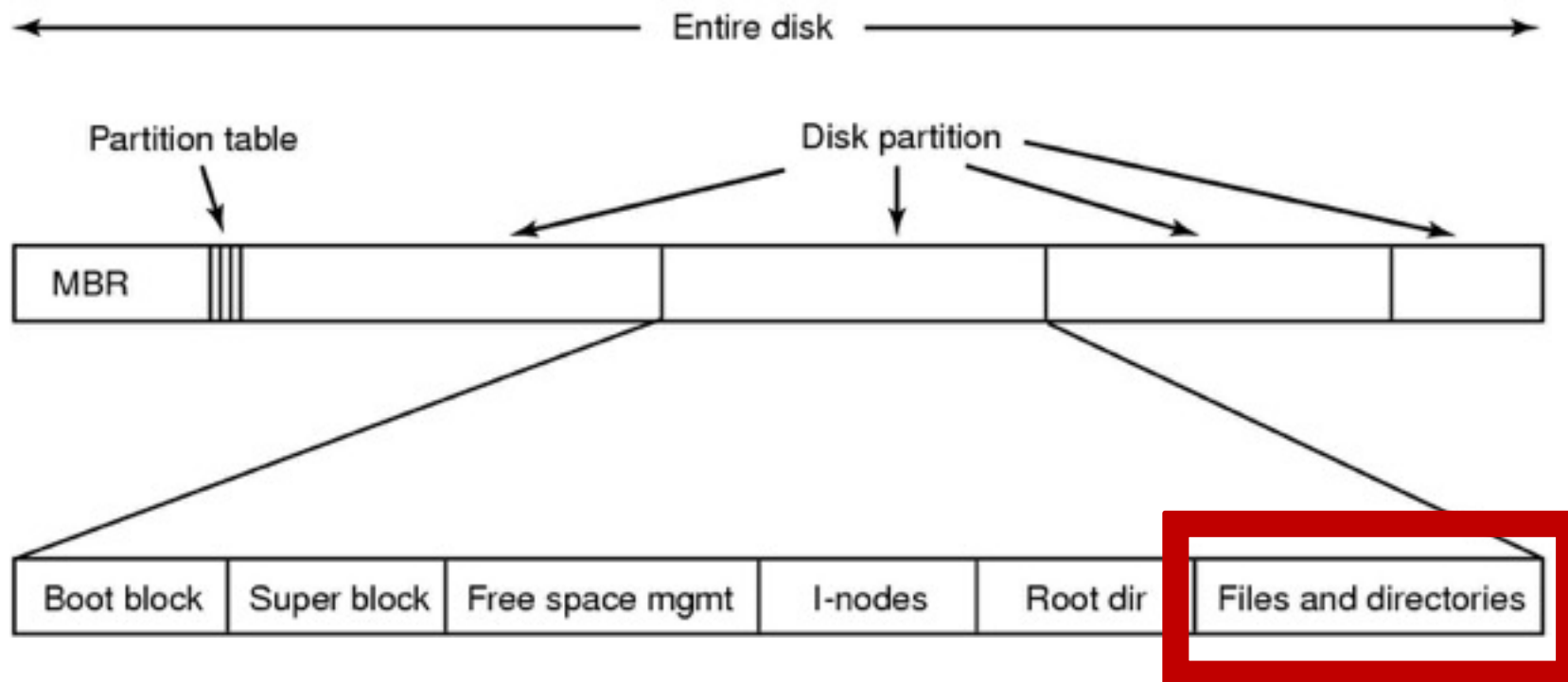
- I-node: a data structure used to represent information about a file system object (file, directory)

# Example: File System Layout



- The top of the file-system tree

# Example: File System Layout



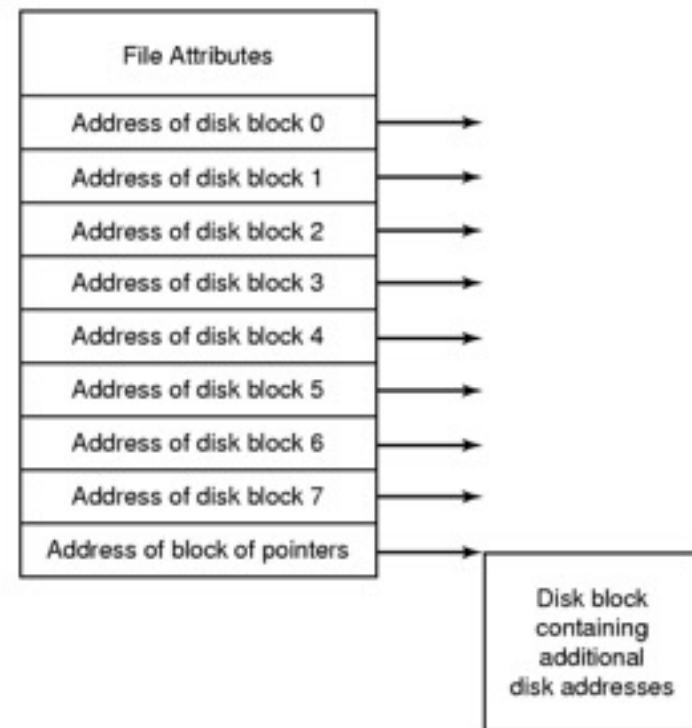
- Directories and files contained in this partition



# Implementing Files

## I-nodes

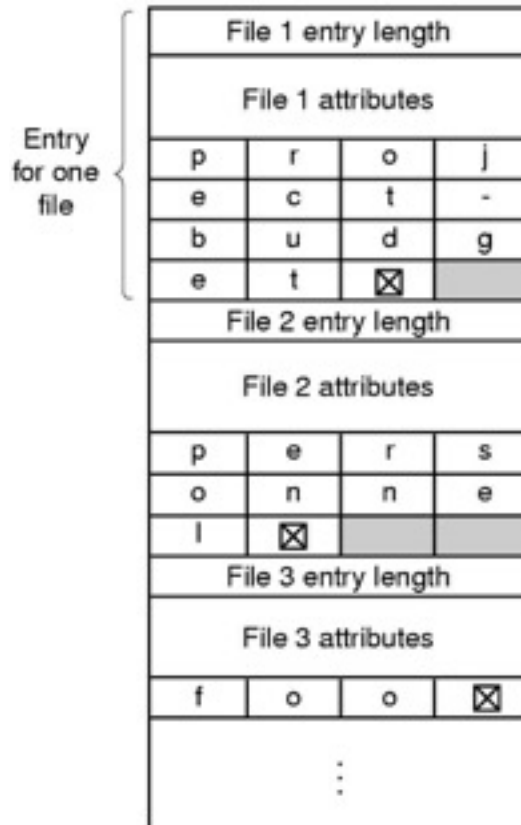
- Associate with each file a data structure called an *i-node* (*index-node*), which lists the attributes and disk addresses of the file's blocks
- The i-node need to be in memory only when the corresponding file is open



# Implementing Directories

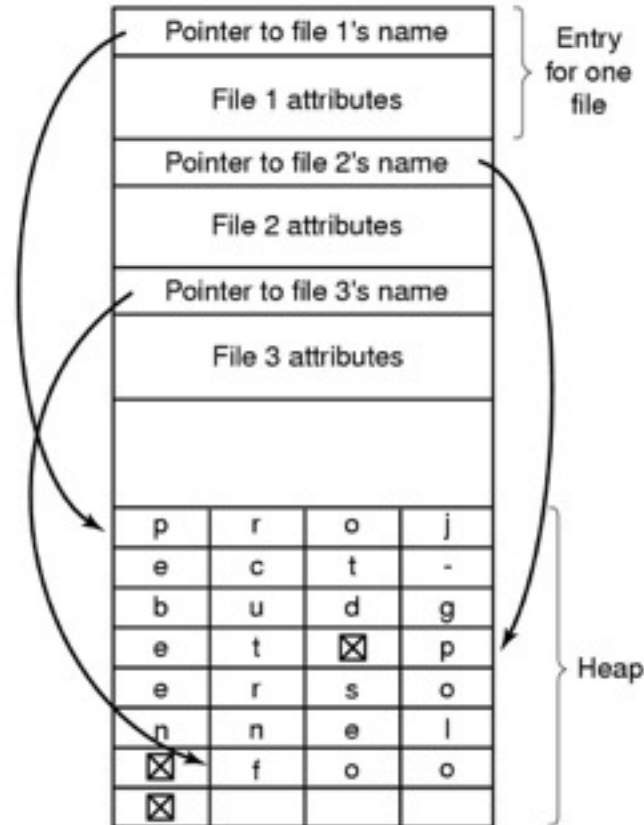
- The main function of the directory system is to map the ASCII name of the file onto the information needed to locate the data
- The directory entry provides the information needed to find the disk blocks
  - Number of the I-node
- Storing files attributes:
  - Directly in the directory entry
  - Store the attributes in the i-node. The directory entry can be a file name and i-node number

# Implementing Directories: File Names



(a)

In-line



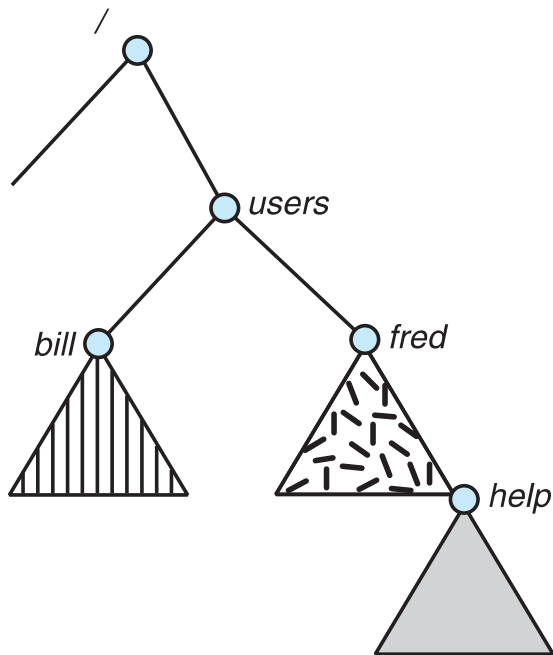
(b)

In a heap

# File System Mounting

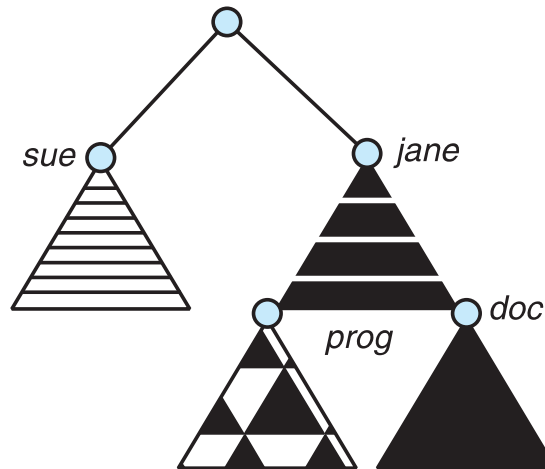
- Just as a file must be opened before it is used, a file system must be mounted before it can be available to processes on the system
- Mount point: the location within the file structure where the file system is to be attached
  - Usually an empty directory
- Mounting procedure: OS is given the name of the device and the mount point, once mounted, it will be able to traverse its directory structure
- Example, mounting home directories in unix

# Mount Point Example



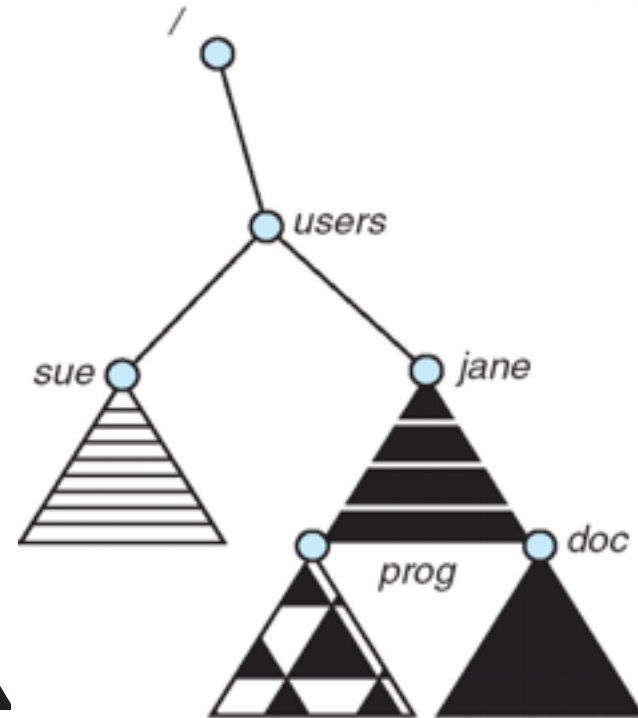
(a)

Existing file  
system



(b)

Unmounted volume



After mounting

# Outline

- Introduction.
- **Distributed File System Requirements**
  - Transparency
  - Concurrent File Updates
  - File Replication
  - Hardware and Operating System Heterogeneity
  - Fault Tolerance
  - Consistency
  - Security
  - Efficiency
- File Service Architecture
- Case Studies

# Distributed File Systems

- Distributed file systems allow multiple processes to share data over long periods of time in a secure and reliable way
- A well designed file service provides access to files stored at a server with performance and reliability similar to, and in some cases better than, files stored on local disks
- A file service enables programs to store and access remote files exactly as they do local ones, allowing users to access their files from any computer (in an intranet)
- A client-server architecture is typically used

# File System vs. Distributed File System

	File System	Distributed File System
Sharing	✗	✓
Persistence	✓	✓
Distributed cache/ replicas	✗	✓
Consistency	Strict-one-copy	Weak guarantees



# Transparency

- **Access transparency:** Client programs should be unaware of the distribution of files
- **Location transparency:** Client programs should see a uniform file name space
- **Mobility transparency:** Neither client programs nor system administration tables in client nodes need to be changed when files are moved
- **Performance transparency:** Client programs should continue to perform satisfactorily while the load on the service varies within a specified range
- **Scaling transparency:** The service can be expanded by incremental growth to deal with a wide range of loads and network sizes

# Concurrent File Updates

- Concurrency Control
- Changes to a file by one client should not interfere with the operation of other clients simultaneously accessing or changing the same file
- Levels of locking are required
- Techniques that provide concurrency control have high costs

# Fault Tolerance

- The file service continues to operate in the face of communication and server failures
- Coping with communication failures:
  - At-most-once invocation semantics
  - At-least-once invocation semantics with a server protocol designed in terms of idempotent operations. This semantic ensures that duplicated requests do not result in invalid updates to files
- Stateless servers: servers can be restarted and the service restored after a failure without needing to recover previous state
- File replication is required

# File Replication

- Several copies of the same file at different locations
- Advantages:
  - Scalability of a service: multiple servers share the load of providing a service to clients accessing the same set of files
  - Fault tolerance: clients are able to locate another server that holds a copy of the file when one has failed
- Caching files (fully or partially) at clients can be considered as a limited form of replication

# Hardware and Operating System Heterogeneity

- Services allowing file access are accessible from different operating systems and computers
- File system server can be deployed on any operating systems or hardware

# Consistency

- One-copy update semantics (e.g. Unix):
  - All of the processes accessing or updating a given file see identical contents as if only a single copy of the file existed
- When files are replicated or cached at different sites:
  - Modifications are propagated to all of the other sites that hold copies
  - This causes inevitable delay ==> deviates from the one-copy semantics

# Security

- Access control mechanism:
  - Uses access control lists
- Authentication:
  - Access control at the server is based on correct user identities
- Encryption can be used to protect the contents of request and reply messages

# Efficiency

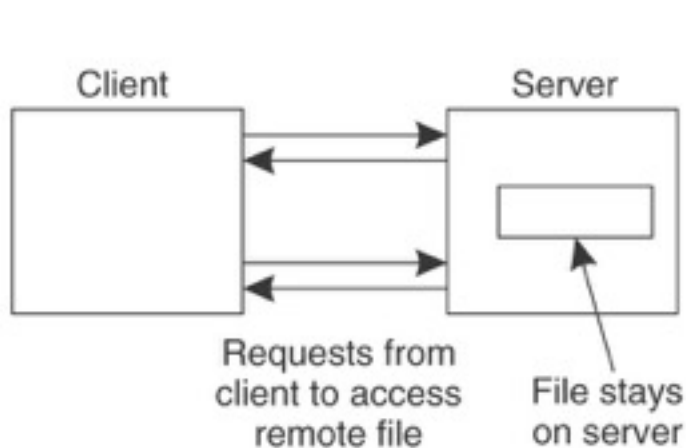
- A distributed file service:
  - Offers facilities that are of at least the same power and generality as those found in conventional file systems
  - Achieve a comparable level of performance
- Trade-off:
  - Scalability, reliability, availability, ...
  - Latency because of accessing remote files



# Outline

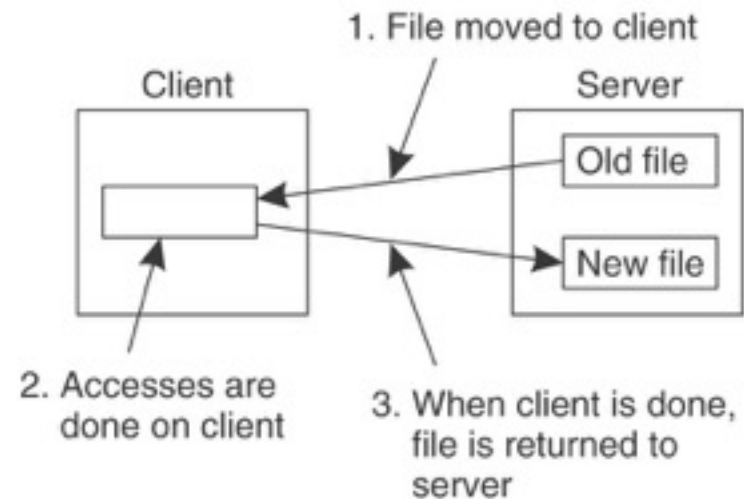
- Introduction
- Distributed File System Requirements
- **File Service Architecture**
- Case Studies:
  - Sun Network File System (NFS)
  - Andrew File System (AFS)
  - Google File System (GFS)

# Distributed File System Access Models



(a)

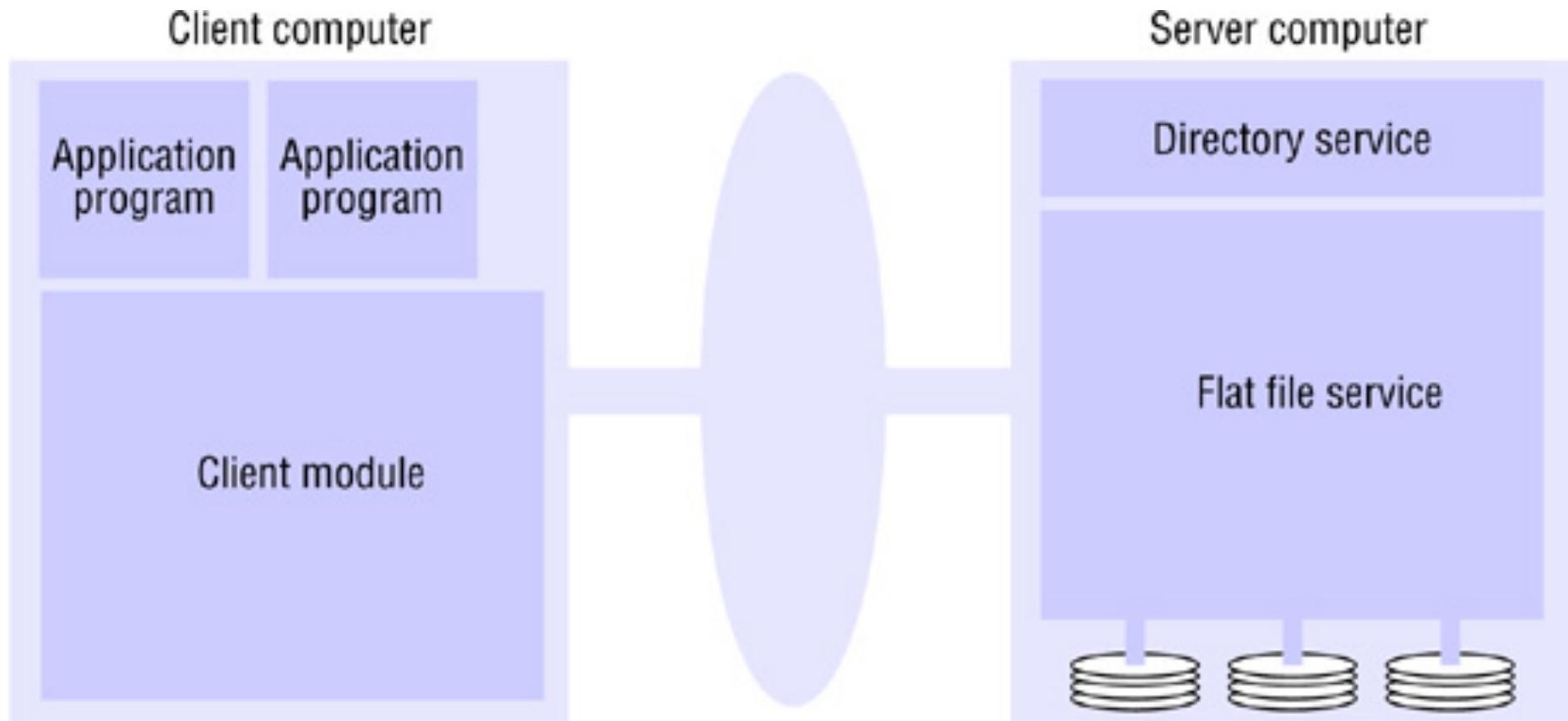
The remote access model



(b)

The upload/download model

# File Service Architecture



# Flat File Service

- Implements operations on files
- *Unique file identifiers (UFIDs)* are used to refer to files. A UFID uniquely identifies a file in a distributed file system
- RPC interface provides a comprehensive set of operations for access to files

# Directory Service

- Provides a mapping between *text names* for files and their UFIDs
- Provide the following services:
  - Generate directories
  - Add new file names to directories
  - Obtain UFIDs from directories
- Can be considered as a client to the flat file service

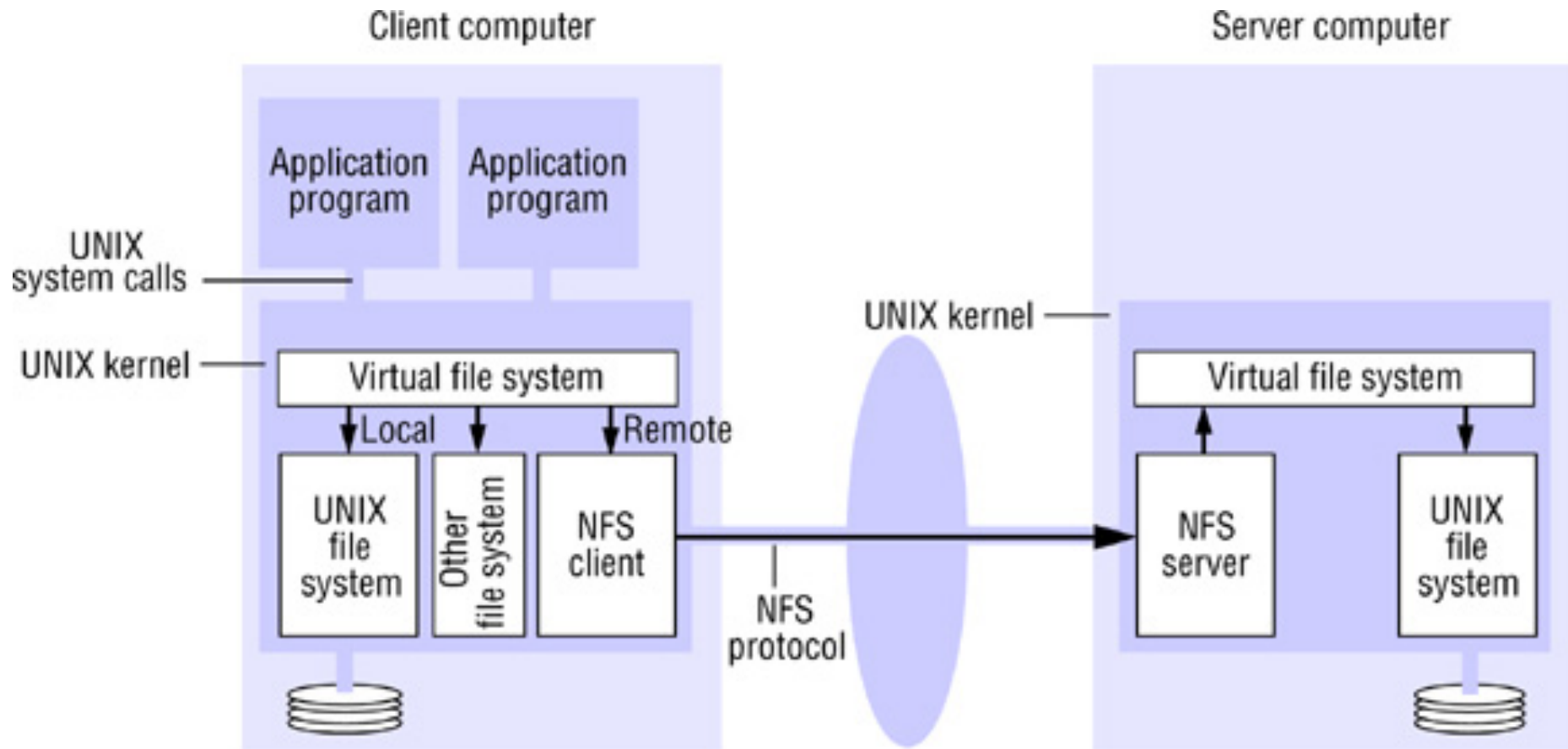
# Client Module

- A client module runs at each client
- Extends the operations of the flat file service and the directory service under a single application programming interface that is available to user-level programs in client computers
- Holds information about the network locations of the flat file server and directory server processes
- Can manage a cache of recently used file blocks at the client

# Outline

- Introduction
- Distributed File System Requirements
- File Service Architecture
- **Case Studies:**
  - **Sun Network File System (NFS)**
  - Andrew File System (AFS)
  - Google File System (GFS)

# Sun Network File System





# NFS V3 Architecture

- **NFS protocol:** a set of remote procedure calls that provide the means for clients to perform operations on a remote file store Sun's RPC
- **NFS server module:** resides in the kernel on each computer that acts as an NFS server
- **NFS client module:** resides in the kernel on each client computer
- NFS client module translates client requests referring to remote files to NFS protocol operations and then passes them to the NFS server module

# Access Transparency in NFS

- Virtual file system (VFS) is used at the client:
  - Users can access local or remote files without distinction
  - VFS is part of the UNIX kernel
- Function of VFS:
  - Keeps track of the file systems that are currently available both locally and remotely
  - Distinguishes between local and remote files
  - Translates between the UNIX-independent file identifiers used by NFS and the internal file identifiers normally used in UNIX and other file systems

# File Identifiers (File Handles)

- A file handle is opaque to clients and contains whatever information the server needs to distinguish an individual file
- Fields of the file handles:
  - File system identifier: a unique number that is allocated to each file system when it is created
  - i-node: a number that identifies and locates the file within the file system
  - i-node generation number: needed because i-node numbers are reused after a file is removed
- VFS structure: for each mounted file system
- v-node: for each opened file
  - Indicates whether the file is local or remote
  - local → reference to i-node
  - remote → the file handle of the remote file

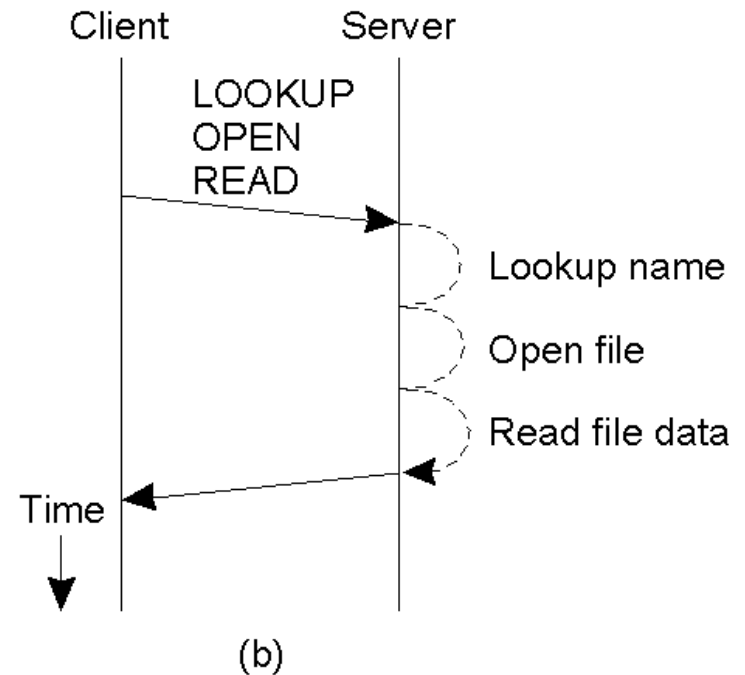
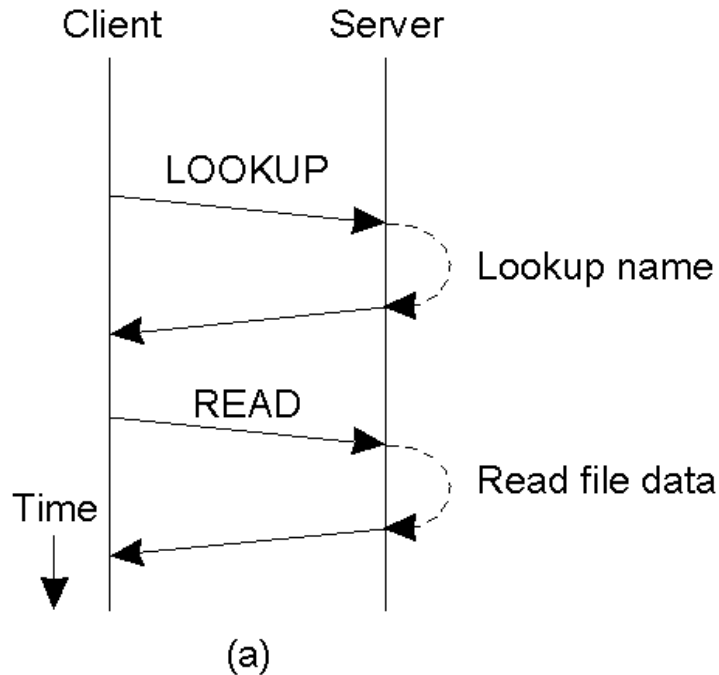
# Client Integration

- The NFS client module is integrated into the kernel:
  - Files are accessed via UNIX system calls
- A shared cache of recently used blocks
  - Same buffer cache is shared with local file system
  - Consistency problem because of copies cached at clients
- Encryption key used to authenticate user IDs passed to the server is retained in the kernel

# Access Control and Authentication

- NFS server is stateless:
  - it does not keep files open on behalf of its clients
  - user's identity is checked on each request
- Sun RPC calls are used to send requests to the NFS server
  - User authentication information (user ID, group ID)
  - Checked against the access permission in the file attributes
- Security loophole:
  - Conventional RPC interface at a well-known port on each host
  - Client can modify the RPC calls to include the user ID of any user
  - Solution:
    - DES encryption of the user's authentication information
    - Integration of Kerberos authentication protocol

# Communication via RPC



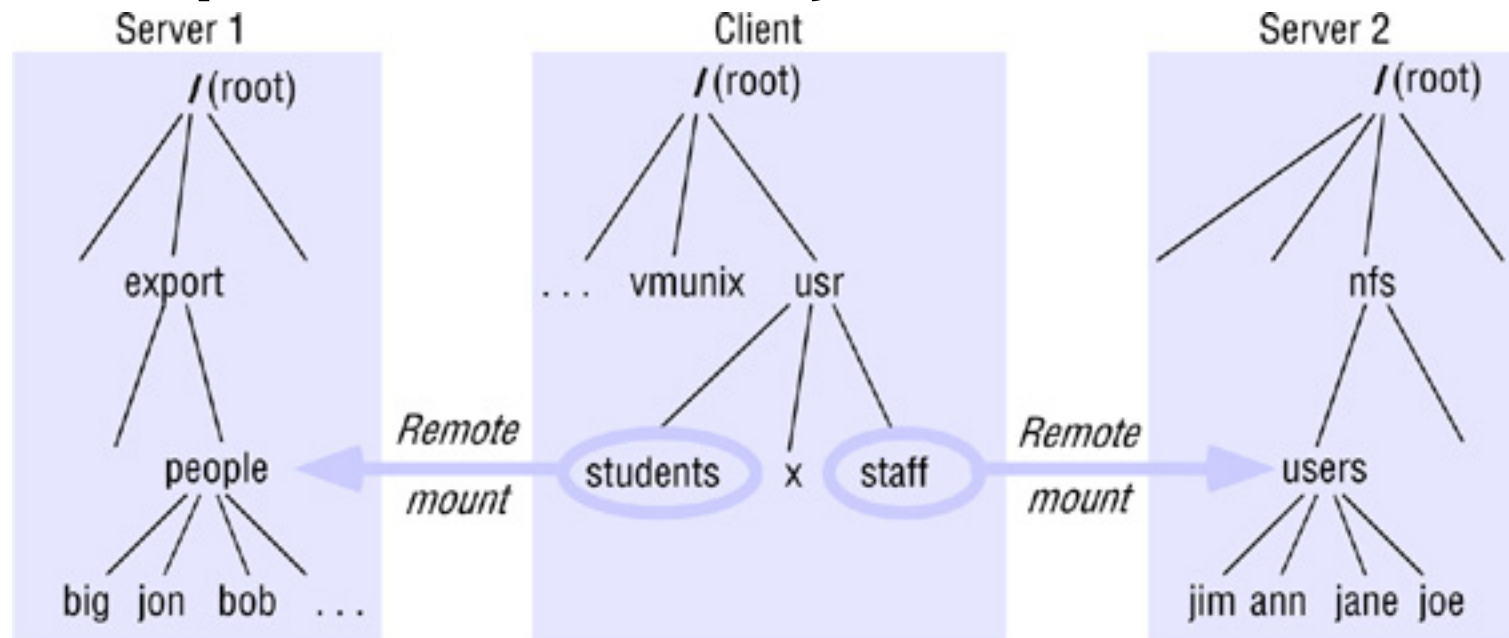
(a) Reading data from a file in NFS version 3.

(b) Reading data using a compound procedure in version 4. (RPCs can be grouped into a single request)

# File System Mounting

- A **mount service** process at each NFS server
- The file `/etc/exports` provides names of local file systems that can be **remotely** mounted
- Access list: hosts that are permitted to mount the file system
- Clients use a modified mount command: remote host's name, the pathname of a directory in the remote file system, and the local name with which it is to be mounted
- Mount Protocol (RPC-based):
  - Given a directory pathname, the file handle of the specified directory is returned (permissions are checked)
  - Location (IP address and port number) of the server and the file handle for the remote directory are passed on to the VFS layer and the NFS client

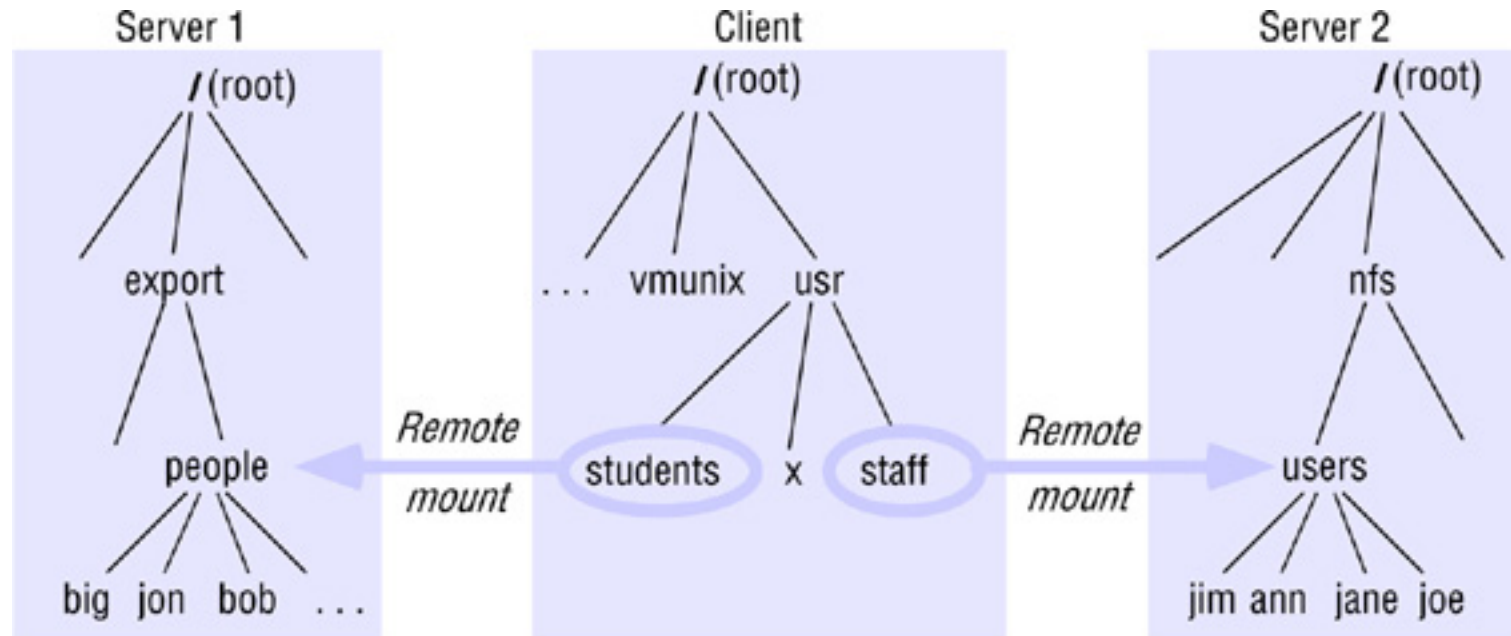
# Example of File System Mounting



**Note:** The file system mounted at `/usr/students` in the client is actually the sub-tree located at `/export/people` in Server 1; the file system mounted at `/usr/staff` in the client is actually the sub-tree located at `/nfs/users` in Server 2

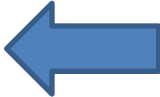


# Example of File System Mounting

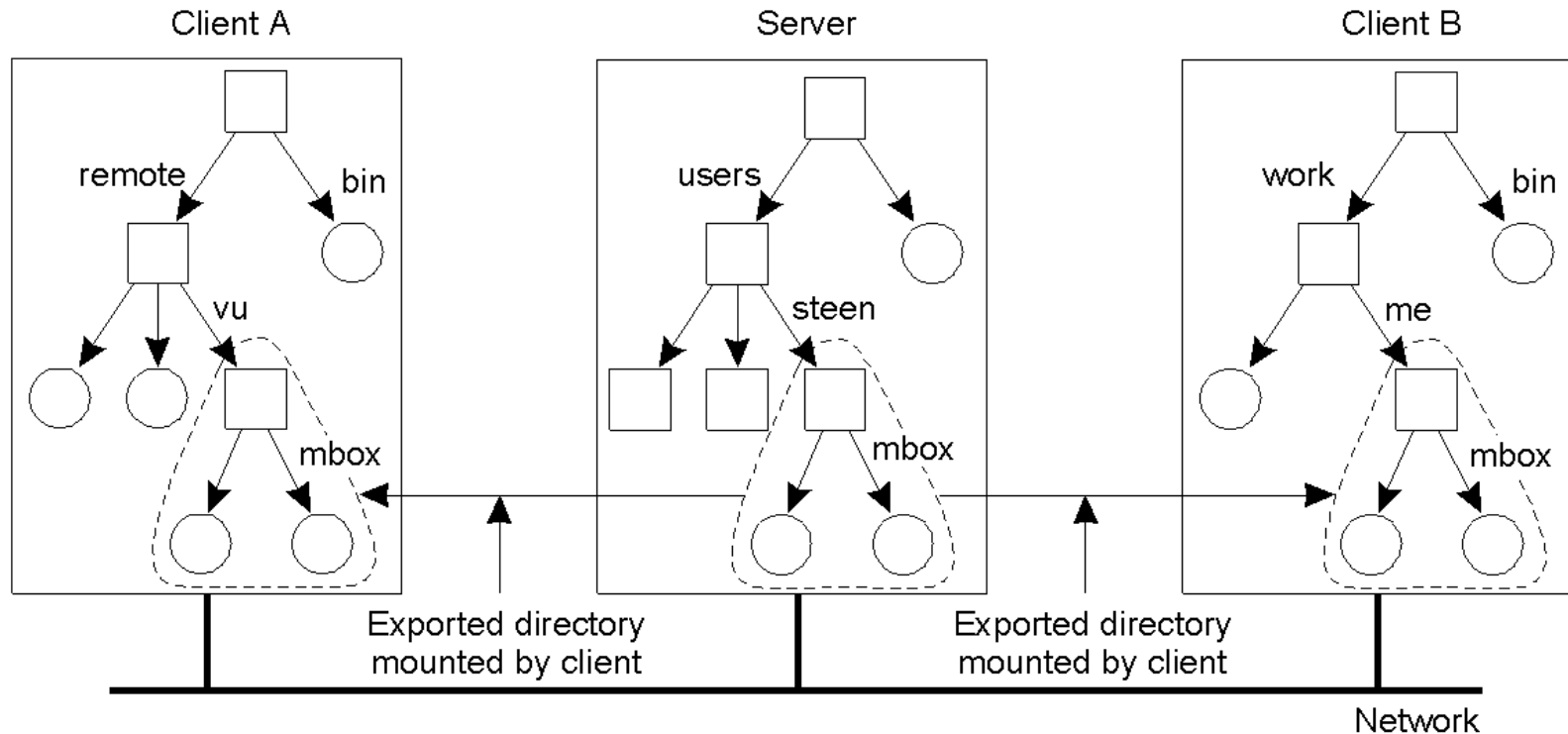


- Client mounts remote file system in its local file system
- Mount part of the remote file system

# Hard-Mounted vs. Soft-Mounted File System

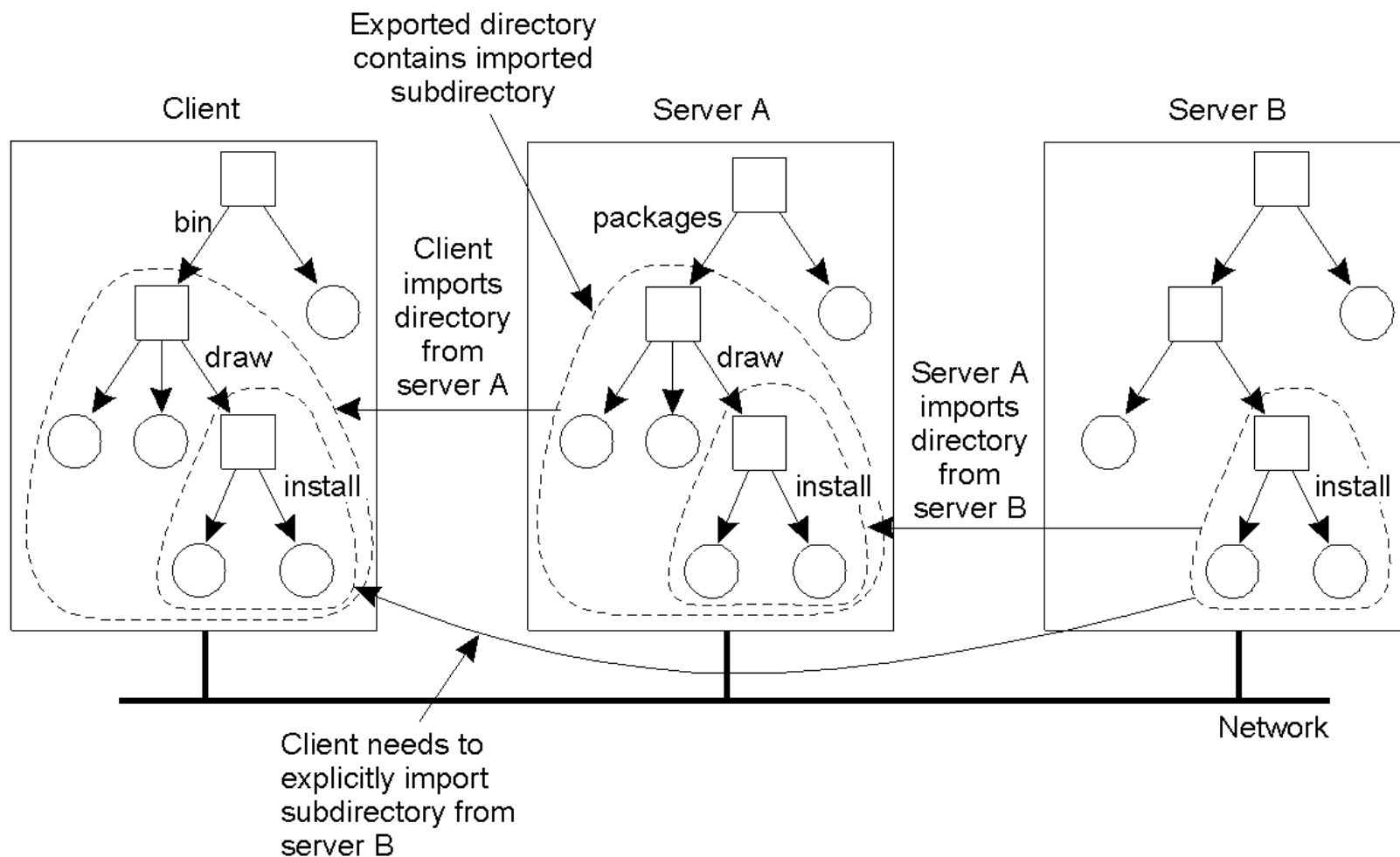
- Hard-Mounted:  More commonly used
  - A user-level process accessing a file is suspended until the request can be completed
  - If server fails, user-level processes are suspended until the server restarts (requests are retried)
  - Programs are unable to recover gracefully when an NFS server is unavailable for a significant period
- Soft-Mounted:
  - NFS client module returns a failure indication to user-level processes after a small number of retries

# Mounting Same Directory by Multiple Clients



- Users do not share namespace
- Example: `/remote/mbox` at A is the same as `/work/mbox` at B
- Sol: standardization

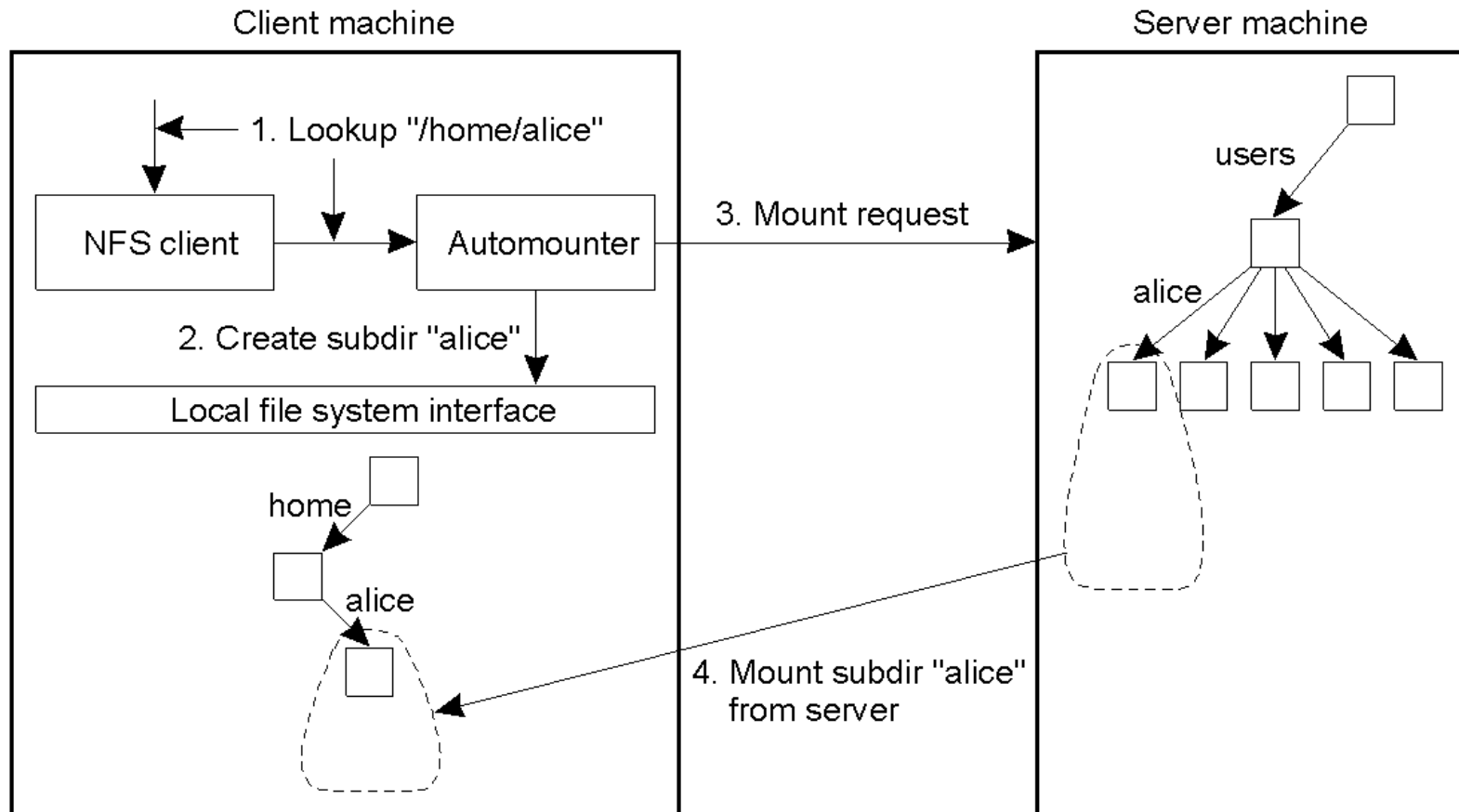
# Mounting nested directories from multiple servers in NFS



# Pathname Translation

- Unix use a step-by-step process to translate multi-part file pathnames to i-node
- Directory name cannot be translated at server
  - File name may cross a mounting point (directories holding multiple mounted file systems)
- Translation process at the client:
  - Pathnames are parsed
  - Translation is done iteratively
  - Each part of a name that refers to a remote-mounted directory is translated to a file handle using a separate lookup request to the remote server

# Automounting

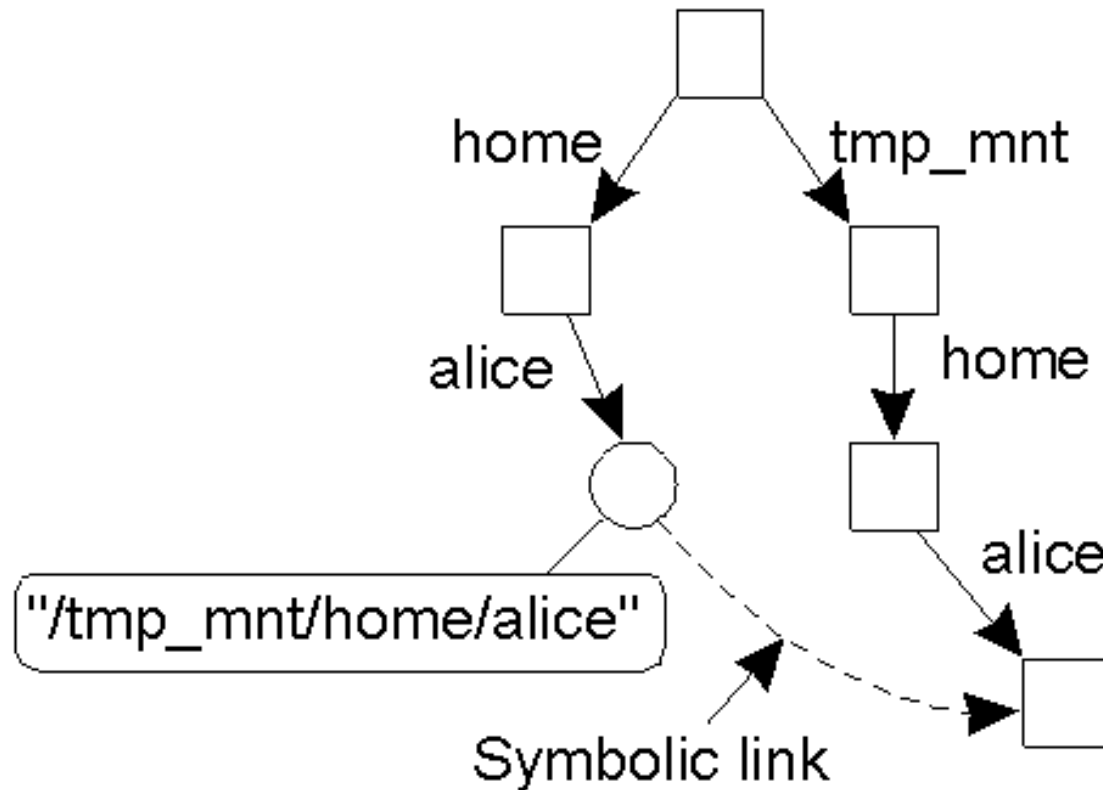


Mount a remote directory dynamically whenever an 'empty' mount point is referenced by a client

# Automounting Procedure

- The automounter maintains a table of mount points (pathnames) with a reference to one or more NFS servers
- NFS client module attempting to resolve a pathname:
  - NFS client passes to the local automounter a `lookup()` request that locates the required file system in its table and sends a 'probe' request to each server listed
  - The file system on the first server to respond is then mounted
  - The mounted file system is linked to the mount point using a symbolic link, so that accesses to it will not result in further requests to the automounter


# Using Symbolic Links with Automounting



Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms. Prentice-Hall, Inc. 2002



# Server Caching in NFS

- Caching in server and client is essential to achieve performance
- Disk caching as in non-networked file systems
- Read operations: simple, no problems introduced
- Write operations: consistency problems
- Write- through caching:
  - Store updated data in cache and written on disk before sending reply to client
  - Relatively inefficient if frequent write operations occur
- Write is persistent when commit operation is performed:
  - Stored only in cache memory  More commonly used
  - Write back to disk only when commit operation for file received

# Client Caching

- read , write, getattr, lookup and readdir operations are cached.
- Potential inconsistency: the data cached in client may not be identical to the same data stored on the server.
- Timestamp-based scheme used in polling server about freshness of a data object:
  - $T_c$ : time cache entry was last validated .
  - $Tm_{client/server}$  : time when block was last modified at the server as recorded by client/ server .
  - $t$ : freshness interval.
  - Freshness condition: at time  $T$   
 $[(T - T_c) < t] \vee [Tm_{client} = Tm_{server}]$  (validity condition)
    - if  $(T - T_c) < t$  (can be determined without server access), then entry presumed to be valid .
    - if not  $(T - T_c) < t$ , then  $Tm_{server}$  needs to be obtained by a getattr call.
    - if  $Tm_{client} = Tm_{server}$  , then entry presumed valid; update its  $T_c$  to current time, else obtain data from server and update  $Tm_{client}$

# Write Operation of Cached Data

- When a cached page is modified it is marked as ‘dirty’ and is scheduled to be flushed to the server asynchronously
- Modified pages are flushed when the file is closed or a sync occurs at the client
- Bio-daemons: perform read-ahead and delayed-write operations.
  - Notified after each read request, and it requests the transfer of the following file block from the server to the client cache
  - In the case of writing, the bio-daemon sends a block to the server whenever a block has been filled by a client operation
  - Directory blocks are sent whenever a modification has occurred
- Bio-daemon processes improve performance:
  - Client module does not block waiting for reads to return or writes to commit at the server

# Outline

- Introduction
- Distributed File System Requirements
- File Service Architecture
- **Case Studies:**
  - Sun Network File System (NFS)
  - **Andrew File System (AFS)**
  - Google File System (GFS)

# Andrew File System (AFS)

- Developed at Carnegie Mellon University (CMU) as a joint work with IBM
- Compatible with NFS
- AFS servers hold ‘local’ UNIX files
- Local Files are referenced by NFS-style file handles (rather than i-node numbers)
- Differences between AFS and NFS:
  - Most important design goal of AFS is scalability
  - Key strategy for scalability: caching of whole files at client nodes

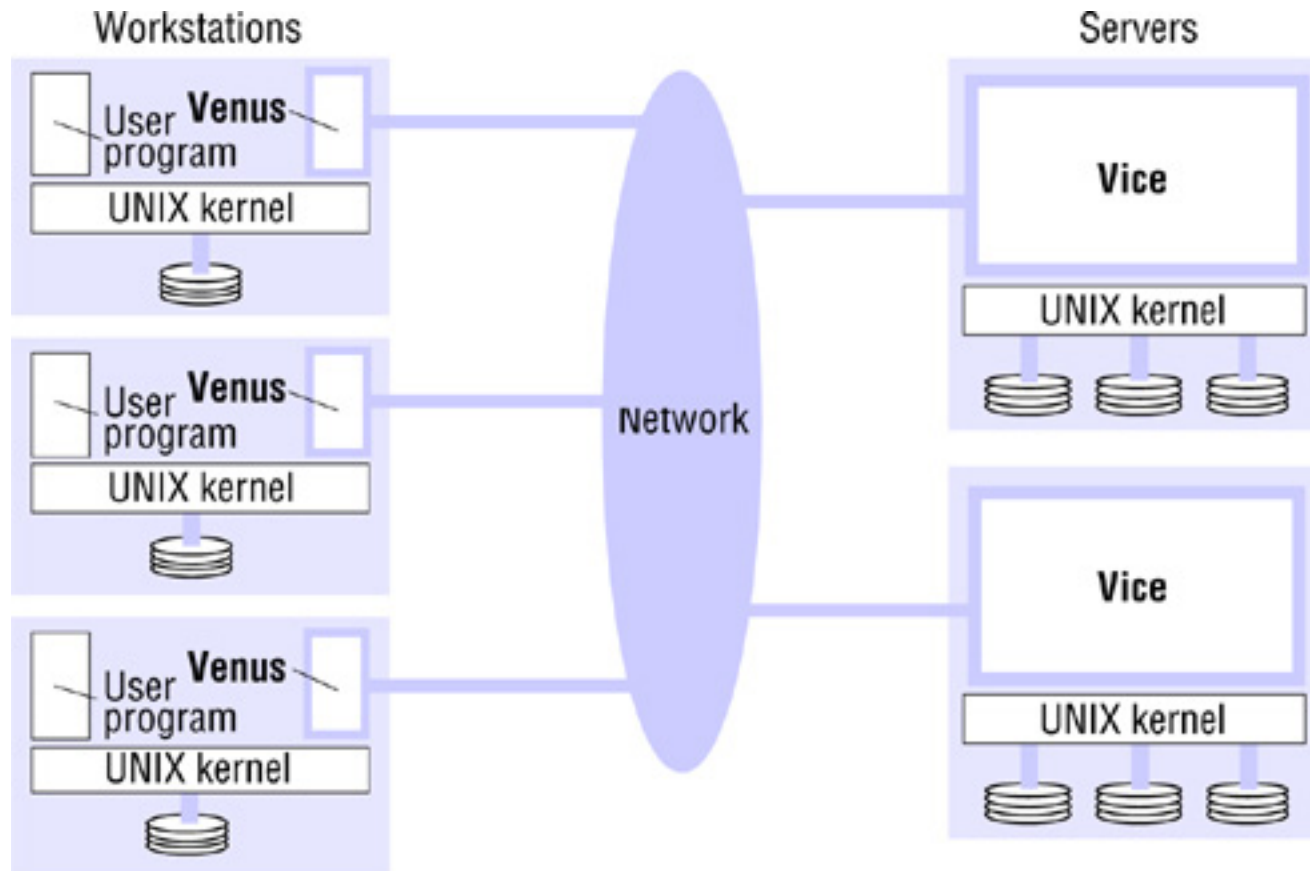
# AFS Design Characteristics

- Whole-file serving: The entire contents of directories and files are transmitted to client computers by AFS servers
  - AFS-3: files larger than 64 kbytes are transferred in 64-kbyte chunks
- Whole-file caching: file/chunk transferred to a client is cached on local disk
  - Most recent files are cached (even after the user close them)
  - The cache is permanent (survive reboots)
  - Local copies are used whenever possible

# Observed Performance of AFS

- Locally cached copies of shared files that are infrequently updated and files accessed by only a single user remain valid for long periods of time
- Disk space is allocated for the cache (e.g., 100MB)
- Assumptions based on UNIX workloads:
  - Files are small, less than 10 KB
  - Read are more common than write operations (x 6)
  - Sequential access is more common
  - Most files are read and written by only one user. Shared files are updated by one user
  - Files are referenced in bursts. Recently referenced files have high probability of being referenced again

# AFS Architecture

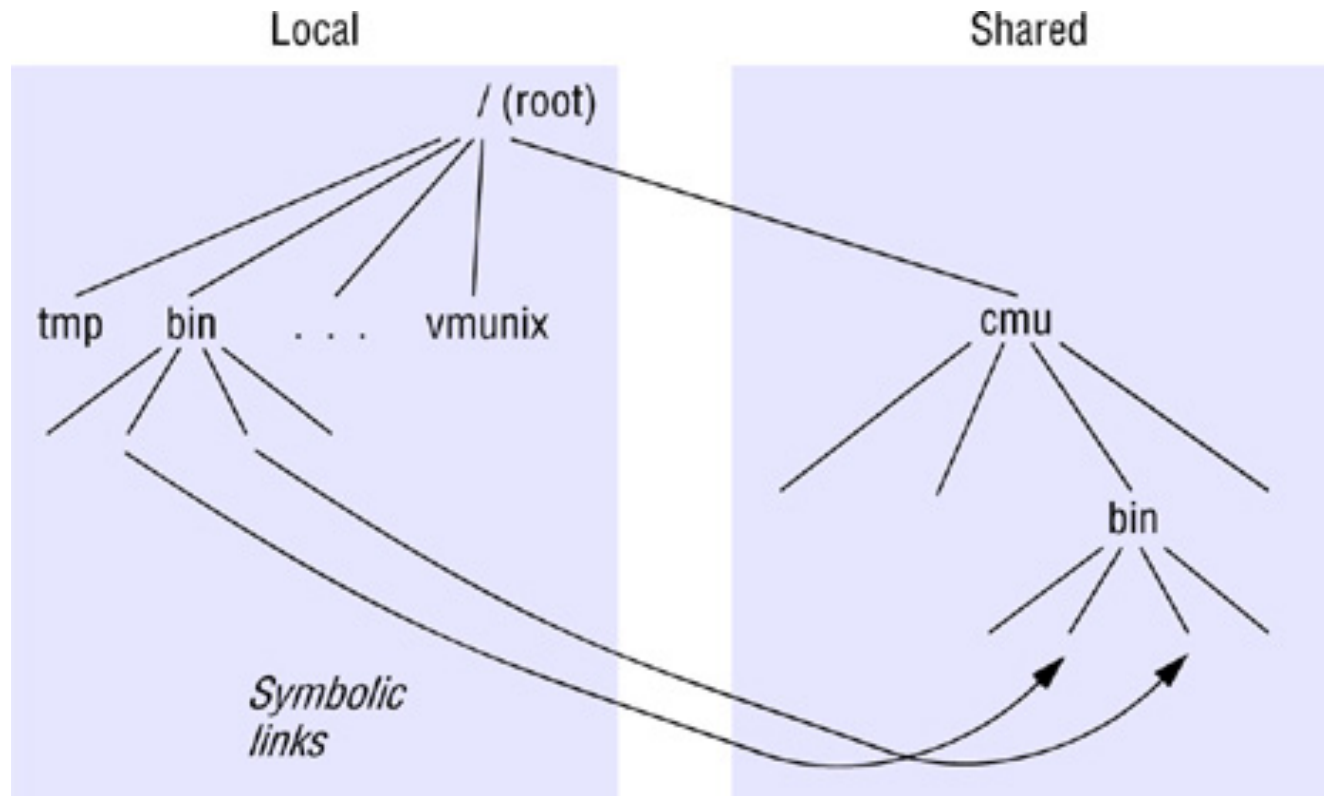


**Note: Vice and Venus are Unix processes**



# File Name Space in AFS

Location transparency ??



**System calls referring to shared name space are intercepted and passed to Venus**

# Callback Mechanism

- **Callback promise:** a token issued by the Vice server that is the custodian of the file, guaranteeing that it will notify the Venus process when any other client modifies the file
- States of callback promise: valid, cancelled
- Callback promise is assigned the “cancelled” state when a file is changed at a workstation, and the vice issues a callback
- Callback promises in the cancelled state are fetched from the Vice server
- Callback promises of cached files are checked after a restart or recovery from a failure (timestamps are compared)
- Callback tokens are renewed when opening a file that was not accessed for a time interval greater than  $T$

# Implementation of File System Calls in AFS

<i>User process</i>	<i>UNIX kernel</i>	<i>Venus</i>	<i>Net</i>	<i>Vice</i>
<i>open(FileName, mode)</i>	<p>If <i>FileName</i> refers to a file in shared file space, pass the request to Venus.</p> <p>Open the local file and return the file descriptor to the application.</p>	<p>Check list of files in local cache. If not present or there is no valid <i>callback promise</i>, send a request for the file to the Vice server that is custodian of the volume containing the file.</p> <p>Place the copy of the file in the local file system, enter its local name in the local cache list and return the local name to UNIX.</p>		<p>Transfer a copy of the file and a <i>callback promise</i> to the workstation. Log the callback promise.</p>
<i>read(FileDescriptor, Buffer, length)</i>	Perform a normal UNIX read operation on the local copy.			
<i>write(FileDescriptor, Buffer, length)</i>	Perform a normal UNIX write operation on the local copy.			
<i>close(FileDescriptor)</i>	Close the local copy and notify Venus that the file has been closed.	<p>If the local copy has been changed, send a copy to the Vice server that is the custodian of the file.</p>		<p>Replace the file contents and send a <i>callback</i> to all other clients holding <i>callback promises</i> on the file.</p>

**Thank You**