Distributed Systems
分布式系统

Unix File Systems

Unix 文件系统

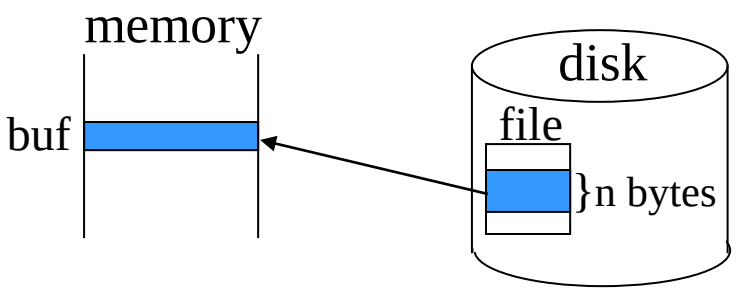
File Systems

- File system provides the most fundamental service for building any information systems.
  - file storage, management and retrieve service.

We look at two file operations:

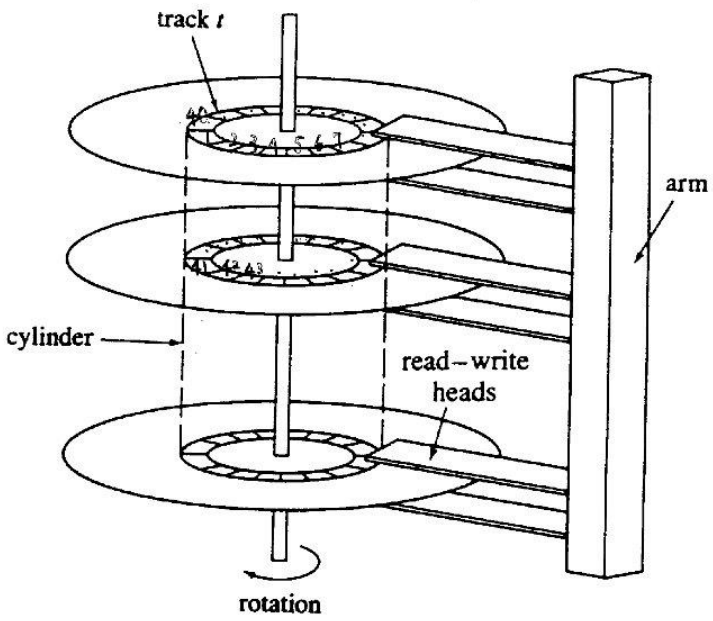
fp = **open** ("*cs4274/exam*", *r*, 0);

r = **read** (*fp*, *n*, *buf*);



Hard disk Structure

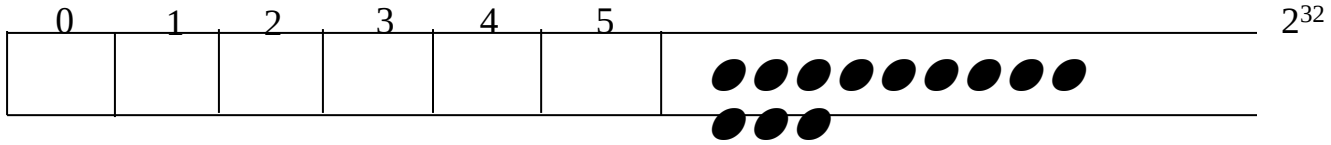
- Head, cylinder, sector
  - Data stored in the same cylinder first to reduce the head movement
- Disk operation in units of sectors



An IBM hard disk configuration:
- 10 heads
- 40 tracks-per-disk (cylinders)
- 32 sectors-per-track
- 512 bytes/sector

Data Block and Block Number

- File system uses block as a data unit for disk operation
  - Usually a block size is 1K (>= sector size)
  - Disk block is referenced by its block #



- Block # to Physical Address Translation

  sectors = **blk_num** × (*#sects-per-blk*)

  cyld_num = *sectors* ÷ (*#sects-per-cyld*)

  head_num = (*sectors* mod (*#sects-per-cyld*)) ÷ (*#sects-per-track*)

  sect_num = *sectors* mod (*#sects-per-track*)

Internal File Structure

- Each file has an i-node: file attributes and data index:

  File attributes:

  – owner, access permissions, access times, file size

  – 12 bytes

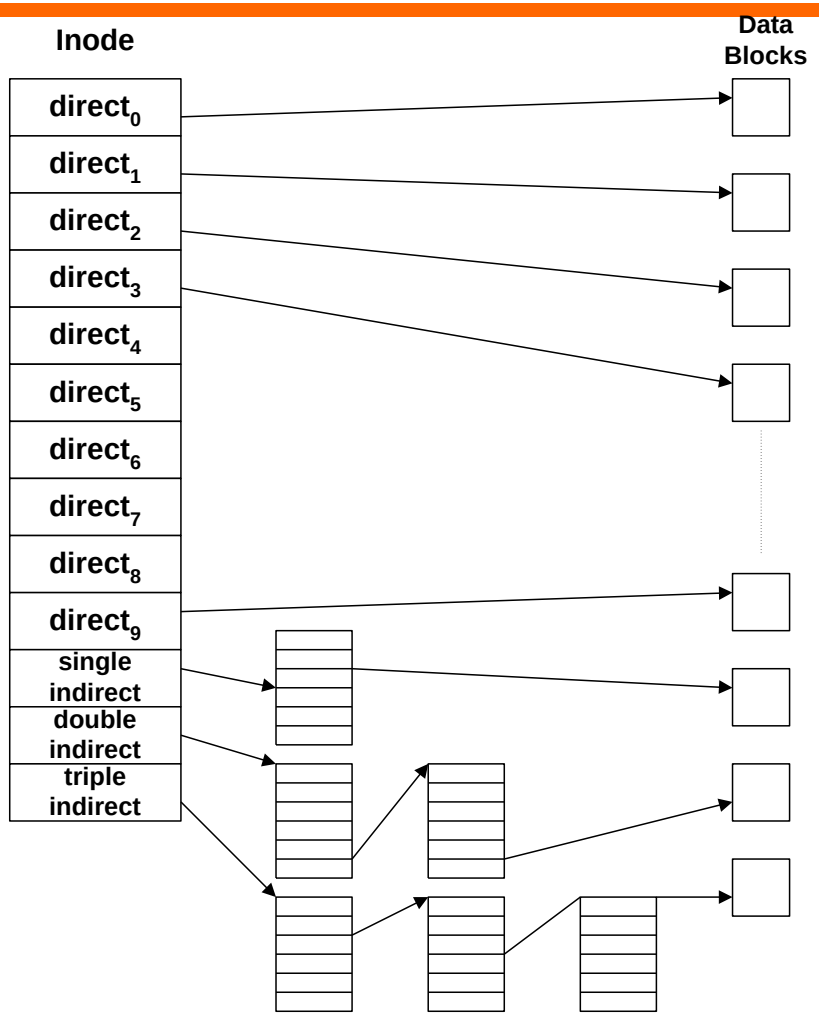
  Data index:

  – 13 indices: 10 direct indices, 1 indirect index, 1 double indirect index, 1 triple indirect index

  – 13 * 4 =52 bytes

- The size of an inode = 64 bytes.
- An inode is referenced by its inode #

Diagram of data index in inode



Unix Directory File

- Directory files are ordinary files consisting of entries mapping names to i-node #, e.g., directory file */etc*
- After BSD4.3, file names are of variable length (upto 255 bytes). Each entry has the entry-length, name & inode #.

Inode Number (2 bytes)	File Names (14 bytes)
83	.
2	..
1798	init
1276	fsck
85	clri
1268	motd
1799	mount
88	mknod

Directory structure and name resolution

- Directories are in a tree structure.
- Resolve pathname recursively to i-node number, component by component from the root (or relative) directory, e.g. /home/lec/jia/unix-fs.doc

a) Starting from inode # 3 of "/" → load inode of "/"

b) Read content of directory file "/":

1368 bin@	1478 lib/	233756 rpool/	2484 boot/	1486 mnt/
1489 sbin/	1369 dev/	154559 net/	1494 system/	2818 devices/
6 export/	2429 platform/	5 var/	27376 home/	.....

c) Map "home" → inode # 27376 → load inode 27376

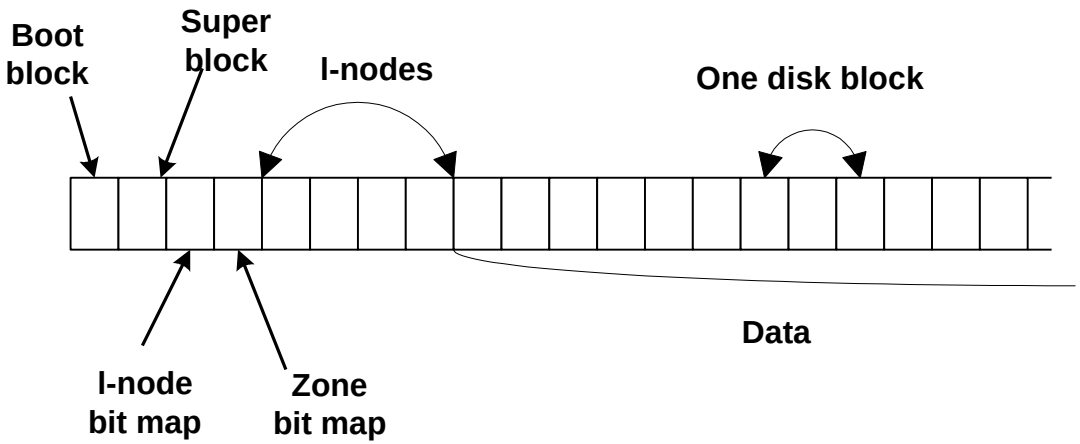
d) Read directory file "/home":

94171 bsft08/	93753 course/	231 lec/	94363 ms10/
55531 bsft09/	94125 cslab/	94239 misc/	91159 ms11/ .....

e) Map "lec" → inode # 231 → load inode 231

d) Repeat the above steps until map "unix-fs.doc" → inode #

Disk Management
(inode # → i-nodes)



Super block 超级块

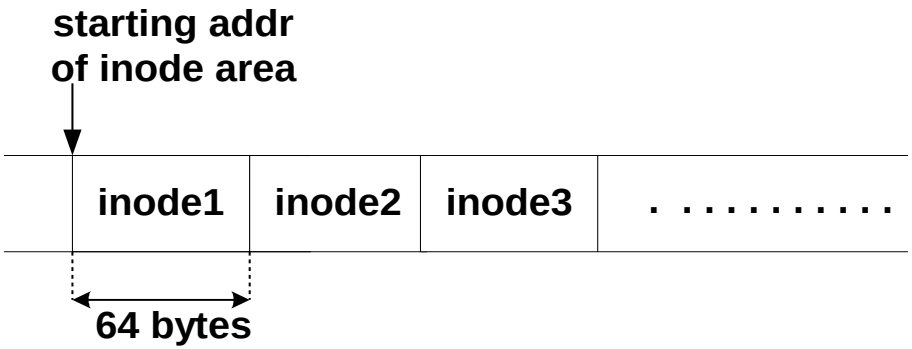
- Each file system has a super block which contains:
  - The size of the file system
  - The number of free blocks in the system
  - An array of cached free block numbers
  - The index of the next free block number
  - A pointer to the start of i-nodes area
  - The number of free i-nodes in the system
  - A list of cached free i-node numbers
  - The index of the next free i-node numbers
- The super block (usually one disk block) of the root file system must be loaded into memory when system is up.

Calculate an inode address by inode number

$blk\_num = (inode\_num - 1) \div \#inodes\text{-}per\text{-}blk + st\text{-}addr\text{-}inodes$

$byte\_offset = ((inode\_num - 1) \bmod (\#inodes\text{-}per\text{-}blk)) \times inode\text{-}size$

N.B. In UNIX, $inode\text{-}size = 64$, $\#inodes\text{-}per\text{-}blk = 16$



Summary on address translations

pathname → inode #

→ inode address (block # & offset)

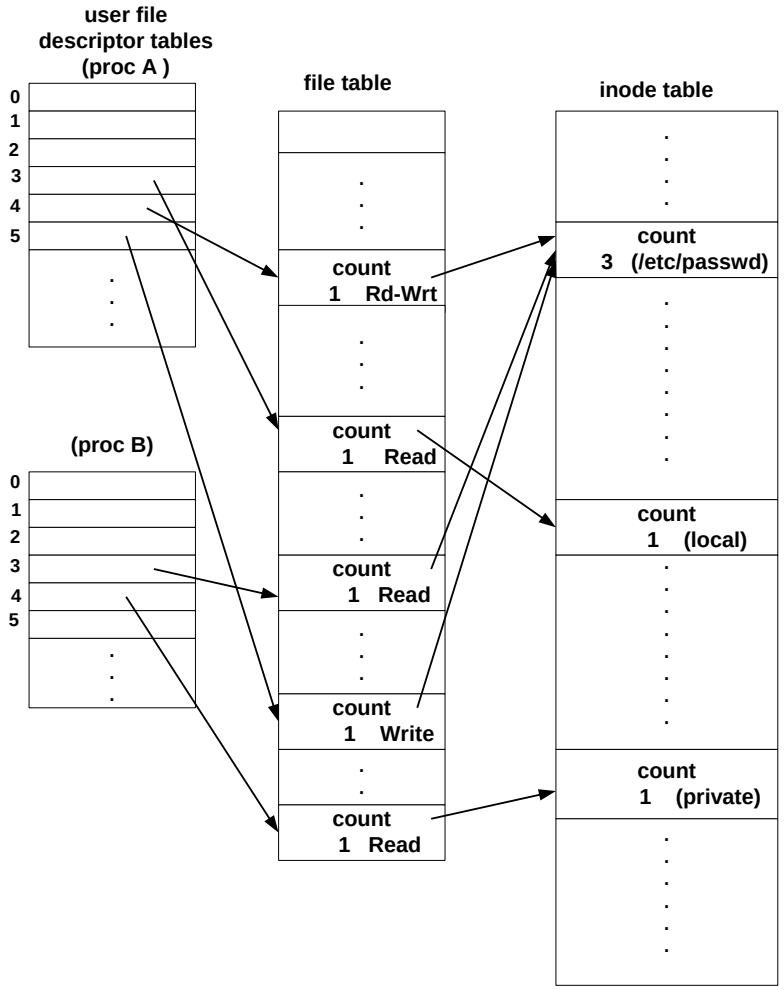
→ inode

→ data block #

→ disk address

→ data

Internal Structure for File Accesses



Pseudo-code for open operation

```
int open (path-name, op) {
    i = resolve(path-name);
    if inode(i) in inode-table then
        inode-table(i).count++;
    else
        load inode(i) into inode-table;
    create an entry in file-table, file-entry(i);
    file-entry(i).count = 1; file-entry(i).op = op; file-entry(i).offset = 0;
    file-entry(i).inode-ptr = &(inode-table(i));
    fd = get the next entry in file-descript-table, fdt;
    fdt[fd] = &(file-entry(i));
    return fd;
}
```

Pseudo-code for read operation

Steps of reading a file data:

read (*fp*, *n*, *buf*)

fp → pointer to the entry in file-table, getting *offset*

→ pointer to the entry in inode-table

→ get the block # of the data

→ convert the block # to head, cylinder, sector

→ ask disk controller to read in data and copy the *n* bytes into *buf*

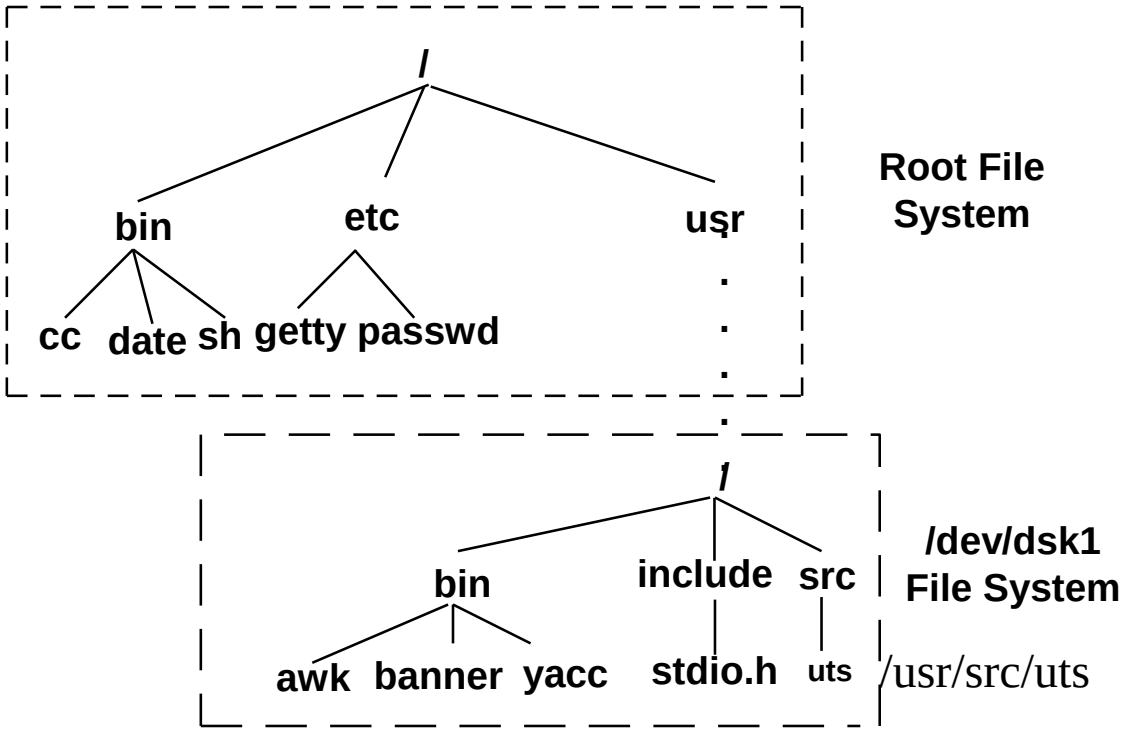
Global File Organization

- Disk Partitions
  - disk units are partitioned into sections.
  - each section is configured into a file system.
  - inode numbers and disk block numbers are valid within a file system.

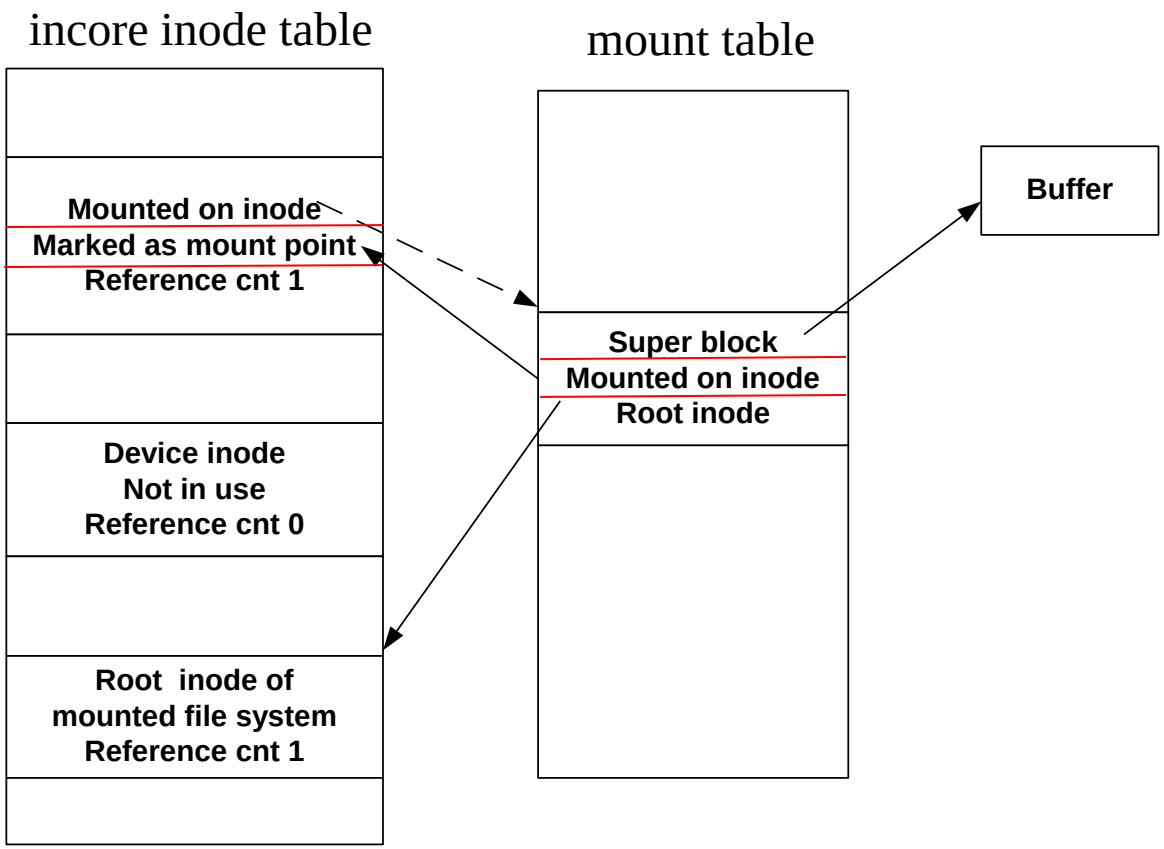
Mount/umount operations

mount(dev_name, dir_name, options); umount(dev_name)

e.g., mount("/dev/dsk1", "/usr", 0)



Mount table


incore inode table
mount table
Mounted on inode
Marked as mount point
Reference cnt 1
Device inode
Not in use
Reference cnt 0
Root inode of
mounted file system
Reference cnt 1
Super block
Mounted on inode
Root inode
Buffer

Name resolution cross mount point

- mount /dev/dsk1 /usr
- cd /usr/src/uts

- When the kernel parses the path name crossing a mount point, from the inode it knows this is a mount point.
- It finds the mount table entry, which contains the addr of the super block and a pointer to the root inode (incore).
- It then accesses the root inode of the mounted file system. The rest of the name resolution is inside the mounted fs.

Link files

Link system call is

link(sour_fname, dest_fname)

e.g. *link("usr/src/uts/sys", "/usr/include/sys")*

- File system first finds the inode # of the source file.
- It then searches for the parent directory of the dest_fname.
- It adds an entry in the parent directory of the dest file), and fills in the pair of dest_fname and the inode #.
- Notice: link cannot cross a file system.

Distributed Systems

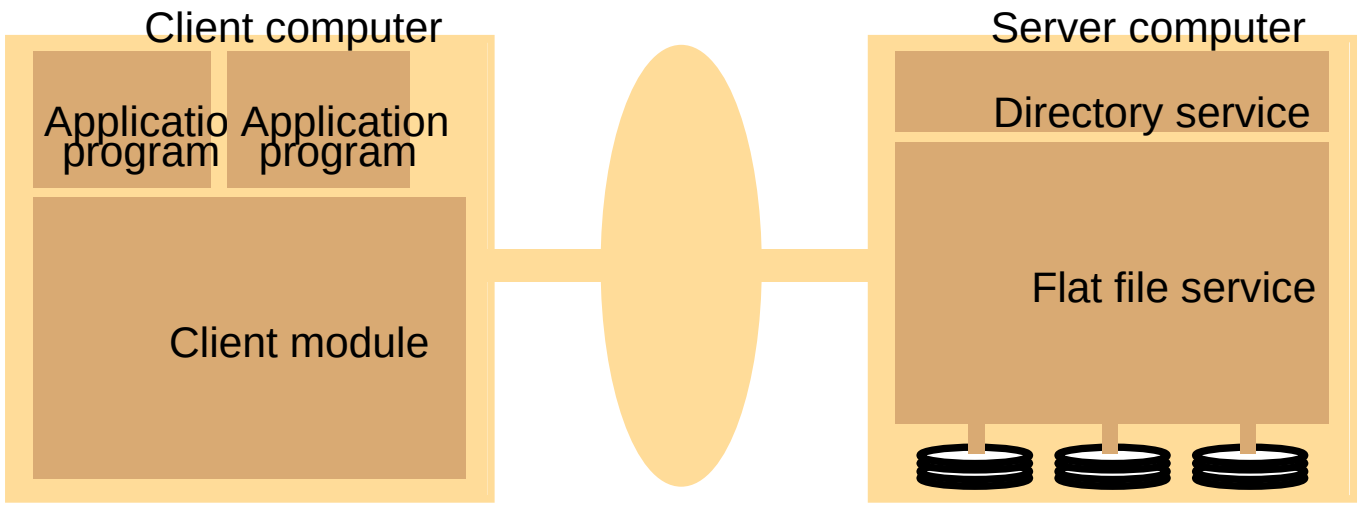
Distributed File Systems

DFS Goals

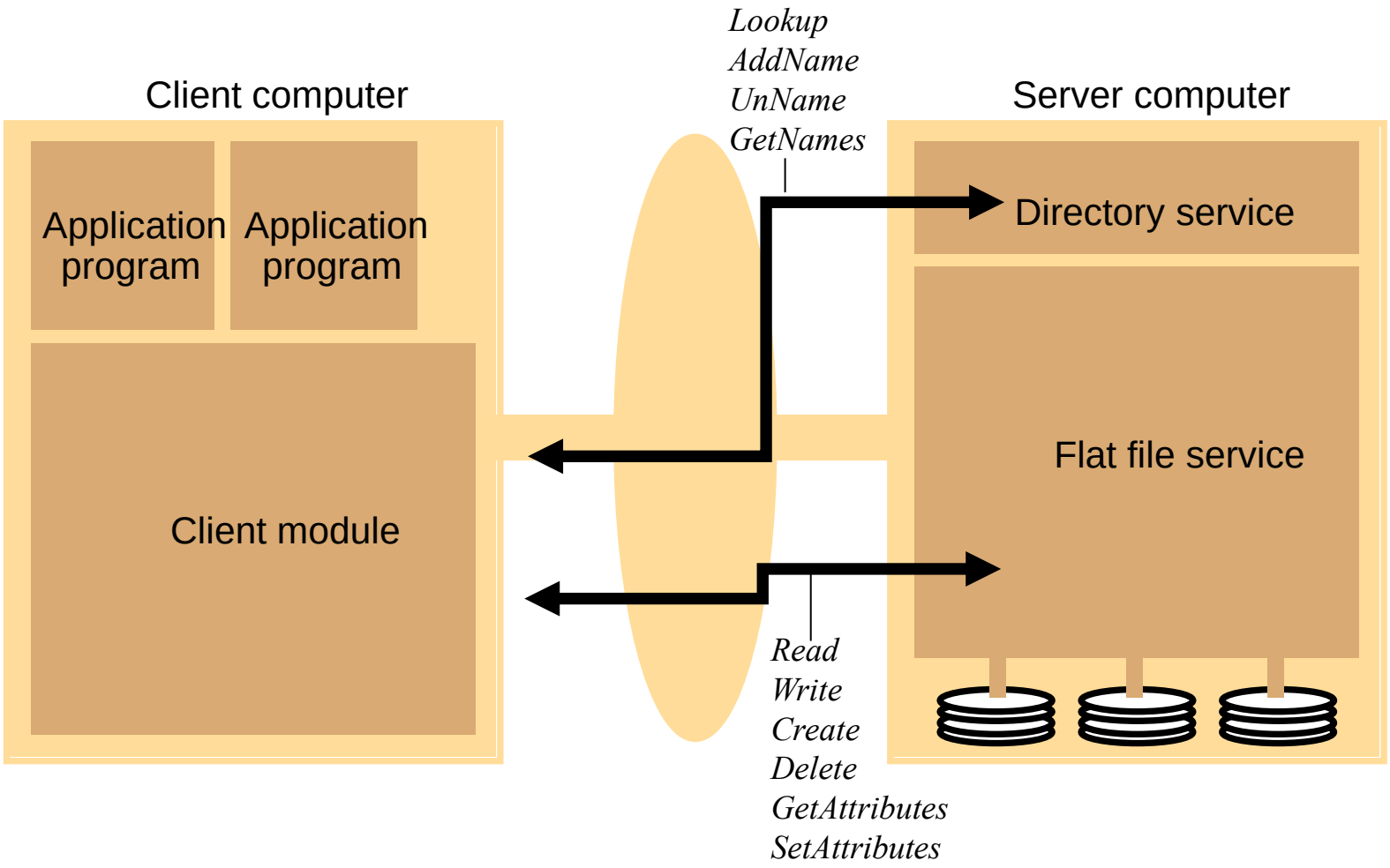
- **Location transparency**: a uniform file name space that does not contain file physical location information.
- **Concurrency transparency**: a user's file operations will not be interfered by another user.
- **Failure transparency**: when part of the system fails, the rest of the system can still provide file services.
- **Heterogeneity**: interoperable among different operating systems and communication protocols.
- **Scalability**: the system performance will not deteriorate as the system size grows.

Distributed File System Components

- Flat file service: offer a simple set of general operations to access attributes and data of files, indexed by UFIDs.
- Directory service: map text names to UFIDs.
- Client module: allow access to file & directory service under a single application programming interface.



Distributed File System Interface



Flat file service operations

read(UFID, i, n) → *Data* — throws *BadPosition*	If $1 \leq i \leq$ *Length(File)*, read a sequence of up to *n* items from a file starting at item *i* and returns it in *Data*.
write(UFID, i, Data) — throws *BadPosition*	If $1 \leq i \leq$ *Length(File)+1*, write a sequence of *Data* to a file, starting at item *i*.
create() → *UFID*	Create a new file with length 0 and return its UFID.
delete(UFID)	Remove the file from the file store.
getAttributes(UFID) → *Attr*	Return attributes of the file.
setAttributes(UFID, Attr)	Set file attributes.

Directory service operations

Lookup(Dir, Name) → *UFID* — throws *NotFound*	Locate the text name in the directory and return the relevant UFID. If *Name* is not in the directory, throw an exception.
AddName(Dir, Name, UFID) — throws *NameDuplicate*	If *Name* is not in the directory, add (*Name*, *UFID*) to the directory and update the file's attribute record. If *Name* is already in the directory, throw an exception.
UnName(Dir, Name) — throws *NotFound*	If *Name* is in the directory, the entry containing *Name* is removed from the directory. If *Name* is not in the directory, throw an exception.
GetNames(Dir, Pattern) → *NameSeq*	Return all the text names in the directory that match the regular expression *Pattern*.

Constructing a Hierarchical FS at Client Module

- A hierarchical file naming system can be constructed by the client module by using file & directory services.
- The root of the directory tree is a directory with a well-known UFID.
- A pathname resolution function is provided in the client module.

Differences between DFS and stand-alone FS

- Separation of file service and directory service:
  - The file service is made more general and the design of directory service can be more flexible.
  - It off loads directory service from file server.
- Stateless servers and emulations in the client module:
  - File server is made very light weighted, leaving most of work to the client module.
  - Server can recover from failure efficiently, with no need to restore client's states.
  - Client module emulates the traditional FS interface.
- Repeatable operations
  - File operations are idempotent, allowing the use of at-least-once RPC semantics.
  - Operation *create* is exceptional (but no side-effect to users).

Capabilities and Access Control

- Clients interacts with dir-server and file-server via RPCs. There is a security loophole, allowing unauthorized user to use other's ID to access files illegally.
- Capability (in the form of a UFID) is a digital key used to identify and access file by clients in distributed FS.
- A capability includes information of file location (groupID), FID, and access control.

48 bits	*32bits*	*32bits*	*16bits*
File group ID	File number	Encrypted permission + random number	Unencrypted permission

Capabilities and Access Control

- Directory server generates the capability of a file based on to client's ID and access permission.
- Capabilities is used for access control to the file.
- A capability must have an encrypted field to prevent it from being modified or mis-used by others. The encryption key is usually shared between the directory server and file server.

File Representation

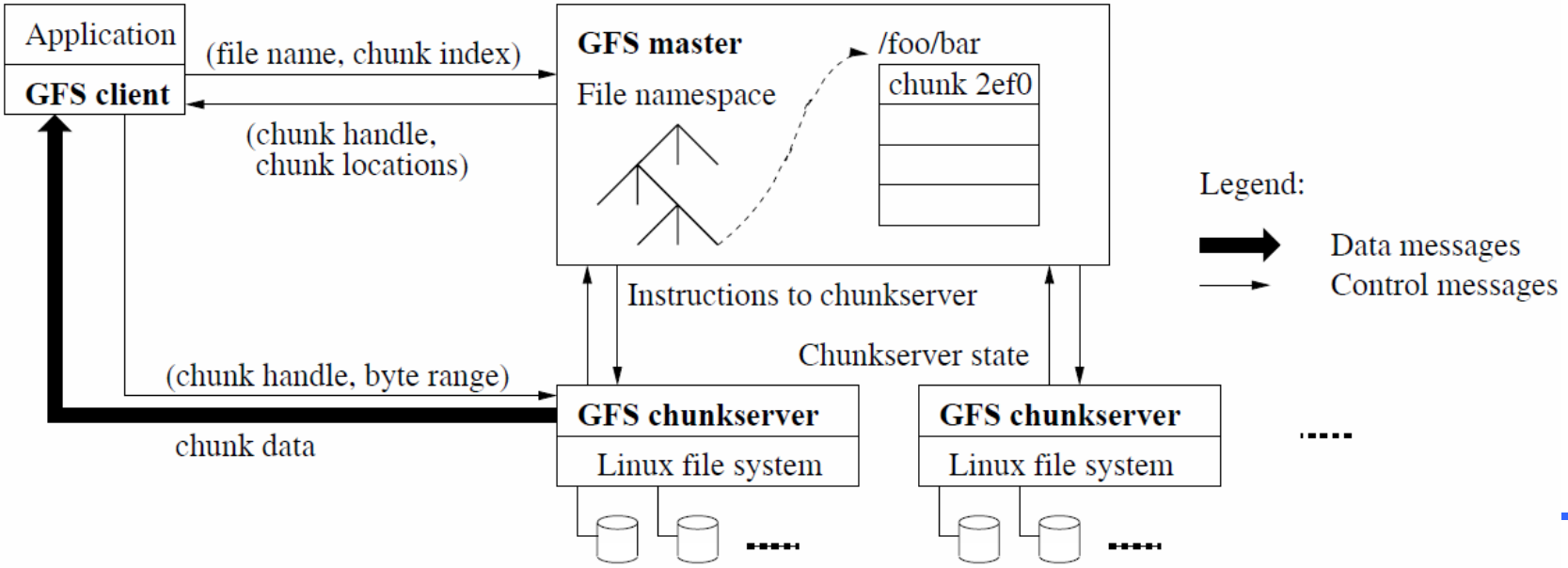
- An **index block** (similar to i-node in UNIX) is used for each file to point to data blocks (or another layer of index blocks).
- Locating an index block is done by using UFID. The UFIDs can be organized to B-tree structure for efficient search.
  - The mappings from UFIDs to the corresponding index-blocks are stored in disk, and the mappings can be too big to be in memory at once.
- Server's location (file group ID) is embedded in UFID. The client needs to know server's location before making RPC for file access. This can be done by the following RPC to any server:
  - PortID = GetServerPort(FileGroupID)

Space Leaks

- Some files whose directory links have been removed may remain in disk and never be accessed, due to the separation of directory servers from file servers.
- The problem is ignored, or prevented in most of the systems.
- A “life-time” method aims to solve the problem:
  - the **dir-server** periodically touch all files, i.e., set a life-time to all files registered with the directory-server.
  - file-server decrements the life-time of all files at some clock rate, and each access to a file will renew its lifetime.
  - The files with life-time 0 will be removed (they are not accessed).

Google File System (an example)

- GFS aims to provide fault-tolerant and highly concurrent accesses to huge files. It consists of GFS client, a single GFS master, and many GFS chunkservers.
- The client requests the master (directory server) with file-name and chunk index (translated from user's byte-offset within a file and the fixed chunk size, 64MB).
- The master replies with the chunk handle and location of the replicas.
- The client requests one of the replicas (chunkserver, i.e., data server) with chunk handle and a byte-range within the chunk for data-access.



Sun Network File System (NFS)

- The Network File System (NFS) was developed to allow machines to mount a local directory to a remote file system as if it were on a local disk. It facilitates fast and seamless sharing of files across a network.
- It is an open standard with clear and simple interfaces (an industry standard for file sharing in LANs since 1980s).
- It follows the abstract file service model defined for Distributed File Systems.

NFS server operations (RFC1813, 1995)

- *read(fh, offset, count)* → *attr, data*
- *write(fh, offset, count, data)* → *attr*
- *create(dirfh, name, attr)* → *newfh, attr*
- *remove(dirfh, name) status*
- *getattr(fh)* → *attr*
- *setattr(fh, attr)* → *attr*
- *lookup(dirfh, name)* → *fh, attr*
- *rename(dirfh, name, todirfh, toname)*
- *link(newdirfh, newname, dirfh, name)*
- *readdir(dirfh, cookie, count)* → *entries*
- *symlink(newdirfh, newname, string)* → *status*
- *readlink(fh)* → *string*
- *mkdir(dirfh, name, attr)* → *newfh, attr*
- *rmdir(dirfh, name)* → *status*
- *statfs(fh)* → *fsstats*

Modelling flat file service

Read(FileId, i, n) → *Data*

Write(FileId, i, Data)

Create() → *FileId*

Delete(FileId)

GetAttributes(FileId) → *Attr*

SetAttributes(FileId, Attr)

Modelling directory service

Lookup(Dir, Name) → *FileId*

AddName(Dir, Name, File)

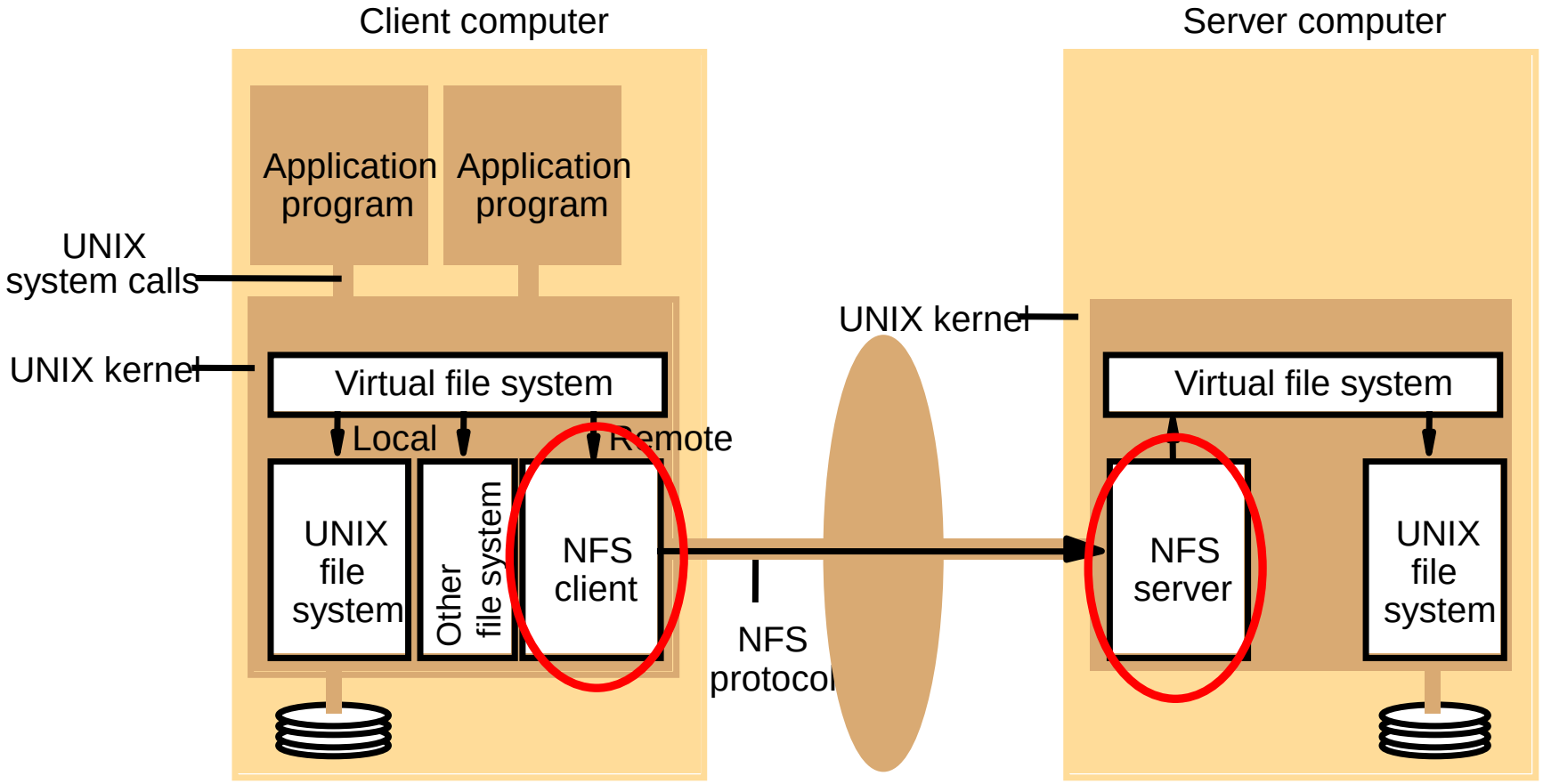
UnName(Dir, Name)

GetNames(Dir, Pattern)

→ *NameSeq*

*

NFS architecture



NFS Structure

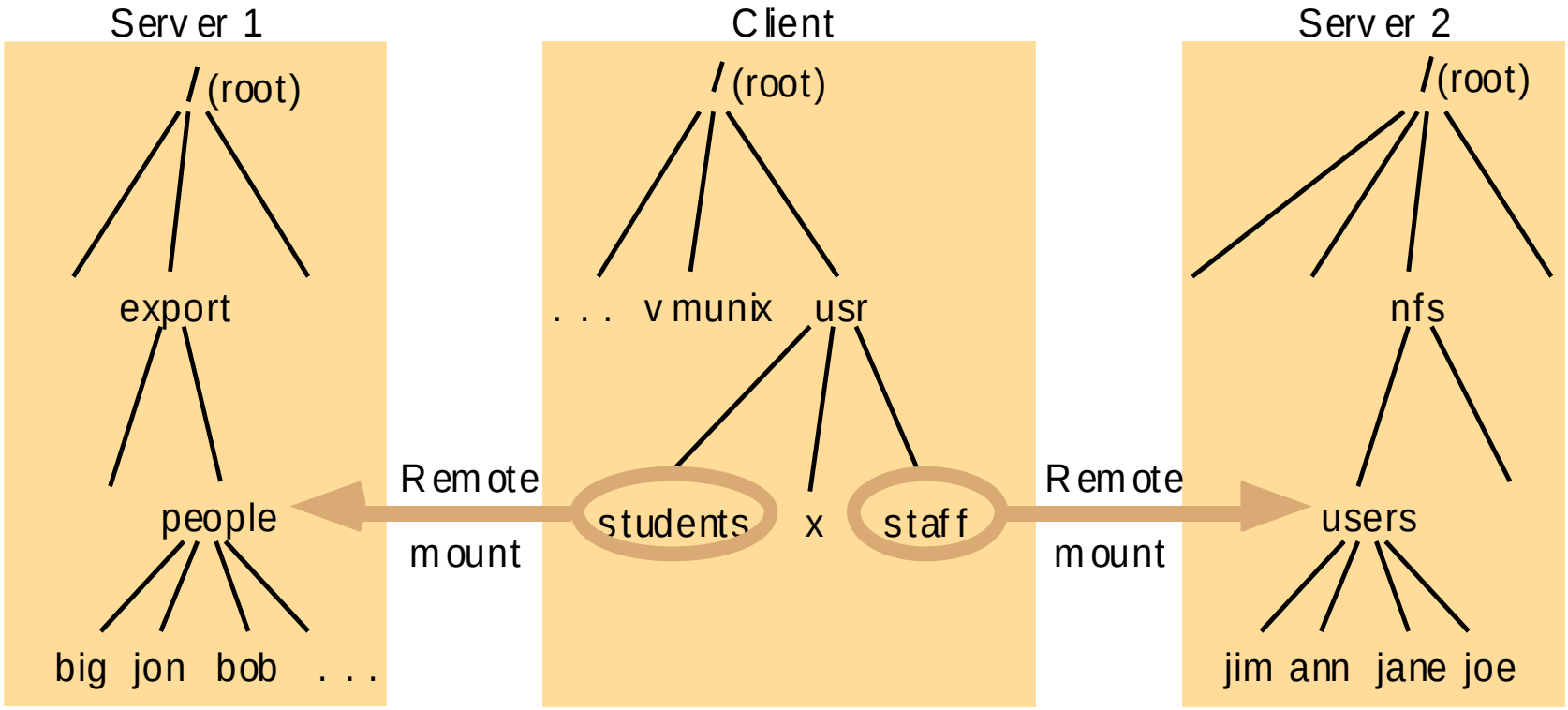
- NFS client module resides in the UNIX kernel. It emulates exactly the semantics of UNIX file system.
  - application programs can run in NFS without any change.
  - a single client module serves all user processes.
- NFS client and server modules communicate using NFS protocol and *mount* protocol (through Sun RPCs).
- NFS server is stateless. It provides reliable file operations.

NFS Virtual File System

- NFS introduces a Virtual File System (VFS) layer. It maintains an entry of *v-node* (virtual *i-node*) for each file that is currently opened or mounted.
- *v-node* is a data structure maintained in OS (similar to *incore inode-table* in unix), not stored on disk. It indicates whether a file is local or remote.
  - For a local file, it contains *i-node*.
  - For a remote file, it contains the *file-handle* as following:

FileSystem ID	i-node number	i-node generation number

Mount remote file systems to client in NFS



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

Mount Service

- Each NFS server runs a *mount process* at application level.
- Client module requests mount of a remote file system by specifying remote host name, pathname and local directory to be mounted.
- The mount process at the server communicates with the client by a mount protocol (RPCs), and returns the *file-handle* of the mounted directory to the client.
- The client keeps the file-handle (containing location of the server) in the *v-node* of the mounted directory for subsequent file accesses.

File Sharing (export for mount): by server

share [-F FSType] [-o specific_options] [-d description] [pathname]
dfshares [-F FSType] [-h] [-o specific_options] [server...]

- share: export, or make a resource available for mounting, through a type specified *FSType*, or the first file-system-type listed in /etc/dfs/fstypes file.
  - /etc/dfs/fstypes: list of DFS utilities installed on the system. For each installed DFS type, there is a line starting with file system type (e.g., “nfs” or “autofs”), followed by descriptive text.
  - /etc/dfs/dfstab: list of share commands to be executed at boot time
  - /etc/dfs/sharetab: system record of shared file systems
- dfshares: list available resources from remote or local systems
  - For each resource, the fields are: resource server access transport

Mount Remote Files: by client

mount [-F nfs] [generic_options] [-o specific_options] [-O] resource mount_point
mountall [-F FSType] [-l | -r] [file_system_table]

- The **mount** process attaches a named directory to the file system hierarchy at the pathname location mount_point.
- /etc/vfstab: list of default parameters for each file system.
  - Each entry consists of space-separated fields:
    - device to mount, device to fsck, mount point, FS type, fsck pass, mount at boot, mount options.
  - Those file systems with the mount-at-boot field “yes” are mounted automatically at boot time by **mountall**
- /etc/mnttab: a record of file systems that are currently mounted.
  - Each entry consists of TAB-separated fields:
    - special, mount_point, fstype, options, time

Automounting

automount [-t duration] [-v]

- Problem with static mounts
  - If a client has many remote resources mounted, boot-time can be excessive
- Automount: Mount and unmount in response to client demand
  - Automounter performs mount according to configuration files (/etc/auto_master, /etc/auto_home)
  - When an empty mount point is **referenced,**
    - OS sends a probe message to **each** server
    - First reply wins: then uses the mount service to mount the filesystem at the first server to respond
  - Attempt to unmount every 10 minutes
- Automounter (/usr/lib/autofs/automountd) is started at system boot-up time, and is a daemon process running at the client computer