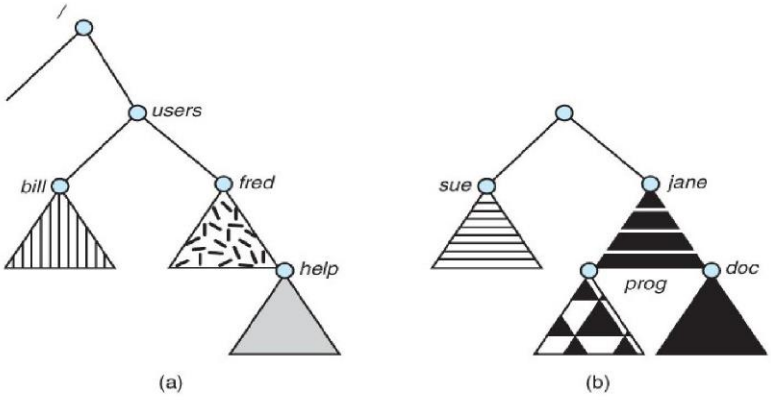
IV FILE SYSTEM MOUNTING

- The basic idea behind mounting file systems is to combine multiple file systems into one large tree structure.
- The mount command is given a file system to mount and a ***mount point*** (directory) on which to attach it.
- Once a file system is mounted onto a mount point, any further references to that directory actually refer to the root of the mounted file system.

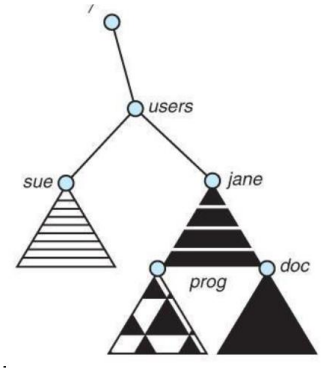
files (or sub-directories) that had stored in the amount point directory prior to mounting the new file system are now hidden by the mounted file system, and are no longer available. For this reason some systems only allow mounting onto empty directories.

File systems can only be mounted by root, unless root has previously configured certain filesystems to be mountable onto certain pre-determined mount points. (E.g. root may allow users to mount floppy filesystems to /mnt or something like it.) Anyone can run the mount command to see what file systems is currently mounted.

Filesystems may be mounted read-only, or have other restrictions imposed.



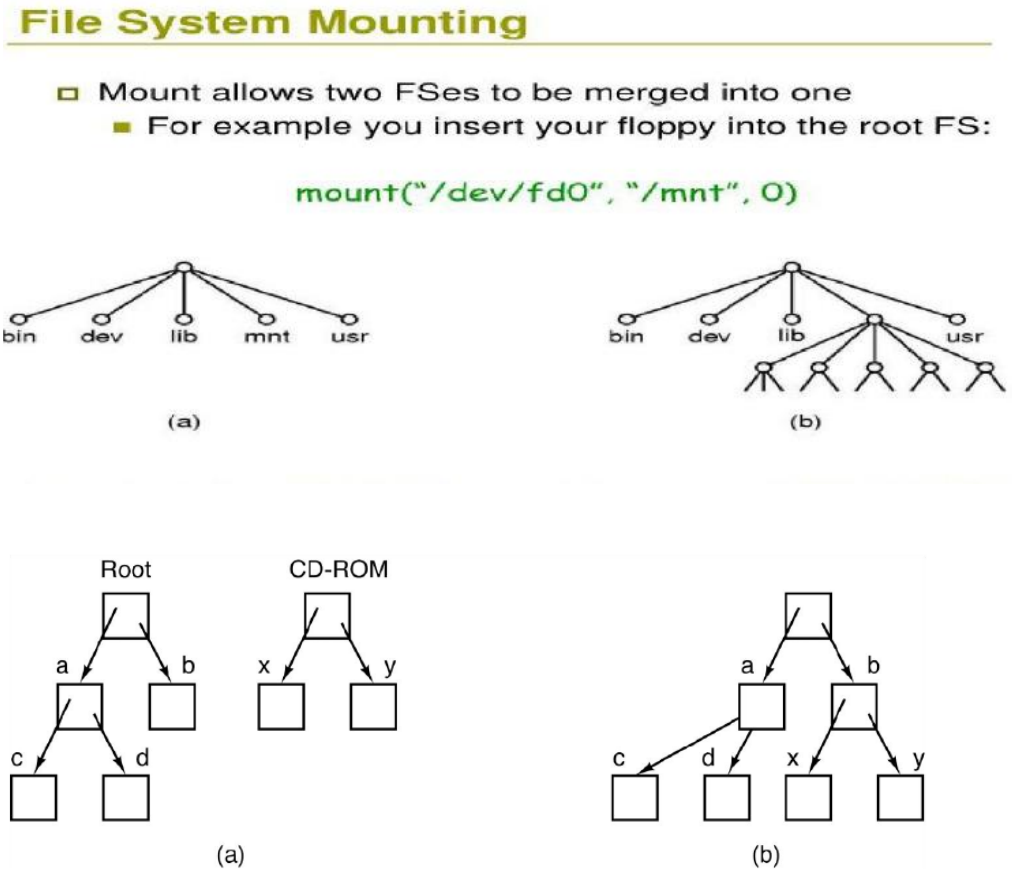
(a) Existing System **(b) Unmounted Volume**



Mount point

The traditional Windows OS runs an extended two-tier directory structure, where the first tier of the structure separates volumes by drive letters, and a tree structure is implemented below that level. Macintosh runs a similar system, where each new volume that is found is automatically mounted and added to the desktop when it is found.

More recent Windows systems allow filesystems to be mounted to any directory in the filesystem, much like UNIX.



File Sharing and Protection:

File Sharing

Multiple Users

On a multi-user system, more information needs to be stored for each file:

- The owner (user) who owns the file, and who can control its access.
- The group of other user IDs that may have some special access to the file.
- What access rights are afforded to the owner (User), the Group, and to the rest of the world (the universe, a.k.a. Others.)
- Some systems have more complicated access control, allowing or denying specific accesses to specifically named users or groups.

Remote File Systems

The advent of the Internet introduces issues for accessing files stored on remote computers

- The original method was ftp, allowing individual files to be transported across systems as needed. Ftp can be either account or password controlled, or *anonymous*, not requiring any user name or password.
- Various forms of ***distributed file systems*** allow remote file systems to be mounted onto a local directory structure, and accessed using normal file access commands. (The actual files are still transported across the network as needed, possibly using ftp as the underlying transport mechanism.)
- The WWW has made it easy once again to access files on remote systems without mounting their filesystems, generally using (anonymous) ftp as the underlying file transport mechanism.

The Client-Server Model

When one computer system remotely mounts a file system that is physically located on another system, the system which physically owns the files acts as a *server*, and the system which mounts them is the *client.*

User IDs and group IDs must be consistent across both systems for the system to work properly. (I.e. this is most applicable across multiple computers managed by the same organization, shared by a common group of users.)

The same computer can be both a client and a server. (E.g. cross-linked file systems.

) There are a number of security concerns involved in this model:

- Servers commonly restrict mount permission to certain trusted systems only. Spoofing (a computer pretending to be a different computer) is a potential security risk.
- Servers may restrict remote access to read-only.
- Servers restrict which filesystems may be remotely mounted. Generally the information within those subsystems is limited, relatively public, and protected by frequent backups.

The NFS (Network File System) is a classic example of such a system.

Distributed Information Systems

The ***Domain Name System***, *DNS,* provides for a unique naming system across all of the Internet. Domain names are maintained by the ***Network Information System, NIS***, which unfortunately has several security issues. NIS+ is a more secure version, but has not yet gained the same widespread acceptance as NIS.

Microsoft's ***Common Internet File System***, ***CIFS,*** establishes a *network login* for each user on a networked system with shared file access. Older Windows systems used *domains,* and newer systems (XP, 2000), use *active directories.* User names must match across the network for this system to be valid.

A newer approach is the ***Lightweight Directory-Access Protocol***, ***LDAP***, which provides a *secure single sign-on* for all users to access all resources on a network.

This is a secure system which is gaining in popularity, and which has the maintenance advantage of combining authorization information in one central location.

Failure Modes

When a local disk file is unavailable, the result is generally known immediately, and is generally non-recoverable. The only reasonable response is for the response to fail.

However when a remote file is unavailable, there are many possible reasons, and whether or not it is unrecoverable is not readily apparent. Hence most remote access systems allow for blocking or delayed response, in the hopes that the remote system (or the network) will come back up eventually.

Consistency Semantics

Consistency Semantics deals with the consistency between the views of shared files on a networked system. When one user changes the file, when do other users see the changes?

At first glance this appears to have all of the synchronization issues discussed in Chapter 6. Unfortunately the long delays involved in network operations prohibit the use of atomic operations as discussed in that chapter.

UNIX Semantics

The UNIX file system uses the following semantics:

- Writes to an open file are immediately visible to any other user who has the file open.
- One implementation uses a shared location pointer, which is adjusted for all sharing users. The file is associated with a single exclusive physical resource, which may delay some accesses.

Session Semantics

The Andrew File System, AFS uses the following semantics:

- Writes to an open file are not immediately visible to other users.
- When a file is closed, any change available only to users who open the file at a later time.

According to these semantics, a file can be associated with multiple (possibly different) views. Almost no constraints are imposed on scheduling accesses. No user is delayed in reading or writing their personal copy of the file.

AFS file systems may be accessible by systems around the world. Access control is maintained through (somewhat) complicated access control lists, which may grant access to the entire world (literally) or to specifically named users accessing the files from specifically named remote environments.

Immutable-Shared-Files Semantics

Under this system, when a file is declared as *shared* by its creator, it becomes immutable and the name cannot be re-used for any other resource. Hence it becomes read-only, and shared access is simple.

Protection

The processes in an operating system must be **protected** from one another's activities. To provide such **protection**, we can use various mechanisms to ensure that only processes that have gained proper authorization from the operating system can operate on the **files**, memory segments, CPU, and other resources of a system.

Goals of Protection

- Obviously to prevent malicious misuse of the system by users or programs. See chapter 15 for a more thorough coverage of this goal.
- To ensure that each shared resource is used only in accordance with system *policies*, which may be set either by system designers or by system administrators.
- To ensure that errant programs cause the minimal amount of damage possible.
- Note that protection systems only provide the *mechanisms* for enforcing policies and ensuring reliable systems. It is up to administrators and users to implement those mechanisms effectively.

Principles of Protection

The ***principle of least privilege*** dictates that programs, users, and systems be given just enough privileges to perform their tasks.

This ensures that failures do the least amount of harm and allow the least of harm to be done.

For example, if a program needs special privileges to perform a task, it is better to make it a SGID program with group ownership of "network" or "backup" or some other pseudo group, rather than SUID with root ownership. This limits the amount of damage that can occur if something goes wrong.

Typically each user is given their own privilege to modify their own files.

The root account should not be used for normal day to day activities - The System Administrator should also have an ordinary account, and reserve use of the root account for only those tasks which need the root privileges.

Domain of Protection

A computer can be viewed as a collection of *processes* and *objects* (both HW & SW).

The ***need to know principle*** states that a process should only have access to those objects it needs to accomplish its task, and furthermore only in the modes for which it needs access and only during the time frame when it needs access.

The modes available for a particular object may depend upon its type.

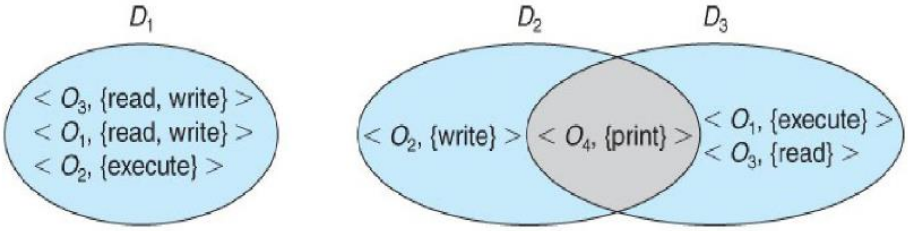
Domain Structure

A ***protection domain*** specifies the resources that a process may access.

Each domain defines a set of objects and the types of operations that may be invoked on each object. An ***access right*** is the ability to execute an operation on an object.

A domain is defined as a set of < object, { access right set } > pairs, as shown below. Note that some

domains may be disjoint while others overlap.



System with three protection domains.

The association between a process and a domain may be ***static*** **or** ***dynamic.***

If the association is static, then the need-to-know principle requires a way of changing the contents of the domain dynamically. If the association is dynamic, then there needs to be a mechanism for ***domain switching.*** Domains may be realized in different fashions - as users, or as processes, or as procedures.

E.g. if each user corresponds to a domain, then that domain defines the access of that user, and changing domains involves changing user ID. The model of protection that we have been discussing can be viewed as an ***access matrix,*** in which columns represent different system resources and rows represent different protection domains. Entries within the matrix indicate what access that domain has to that resource

object / domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Access matrix.

Domain switching can be easily supported under this model, simply by providing "switch" access to other domains:

object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Access matrix of above Figure with domains as objects.

Types of Access

The following low-level operations are often controlled

Read - View the contents of the file

Write - Change the contents of the file.

Execute - Load the file onto the CPU and follow the instructions contained therein.

Append - Add to the end of an existing file.

Delete - Remove a file from the system.

List -View the name and other attributes of files on the system.

Higher-level operations, such as copy, can generally be performed through combinations of the above.

Access Control

One approach is to have complicated ***Access Control Lists, ACL***, which specify exactly what access is allowed or denied for specific users or groups.The AFS uses this system for distributed access. Control is very finely adjustable, but may be complicated, particularly when the specific users involved are unknown. (AFS allows some wild cards, so for example all users on a certain remote system may be trusted, or a given username may be trusted when accessing from any remote system.)

UNIX uses a set of 9 access control bits, in three groups of three. These correspond to R, W, and X permissions for each of the Owner, Group, and Others. (See "man chmod" for full details.) The RWX bits control the following privileges for ordinary files and directories:

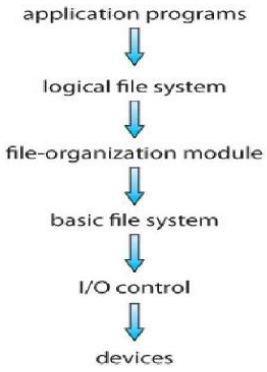
bit	Files	Directories
R	Read (view) file contents.	Read directory contents. Required to get a listing of the directory.
W	Write (change) file contents.	Change directory contents. Required to create or delete files.
X	Execute file contents as a program.	Access detailed directory information. Required to get a long listing, or to access any specific file in the directory. Note that if a user has X but not R permissions on a directory, they can still access specific files, but only if they already know the name of the file they are trying to access.

File System Structure

Hard disks have two important properties that make them suitable for secondary storage of files in file systems: (1) Blocks of data can be rewritten in place, and (2) they are direct access, allowing any block of data to be accessed with only (relatively) minor movements of the disk heads and rotational latency. Disks are usually accessed in physical blocks, rather than a byte at a time. Block sizes may range from **512 bytes to 4K or larger.**

File systems organize storage on disk drives, and can be viewed as a layered design:

- At the lowest layer are the physical devices, consisting of the magnetic media, motors & controls, and the electronics connected to them and controlling them. Modern disk put more and more of the electronic controls directly on the disk drive itself, leaving relatively little work for the disk controller card to perform.
- ***I/O Control*** consists of ***device drivers***, special software programs(often written in assembly) which communicate with the devices by reading and writing special codes directly to and from memory addresses corresponding to the controller card's registers. Each controller card (device) on a system has a different set of addresses (registers, a.k.a. ***ports***) that it listens to, and a unique set of command codes and results codes that it understands.
- The ***basic file system*** level works directly with the device drivers in terms of retrieving and storing raw blocks of data, without any consideration for what is in each block. Depending on the system, blocks may be referred to with a single block number, (e.g. block # 234234), or with head-sector-cylinder combinations.
- The ***file organization module*** knows about files and their logical blocks, and how they map to physical blocks on the disk. In addition to translating from logical to physical blocks, the file organization module also maintains the list of free blocks, and allocates free blocks to files as needed.
- The ***logical file system*** deals with all of the meta data associated with a file (UID, GID, mode, dates, etc), i.e. everything about the file except the data itself. This level manages the directory structure and the mapping of file names to ***file control blocks, FCBs***, which contain all of the meta data as well as block number information for finding the data on the disk.



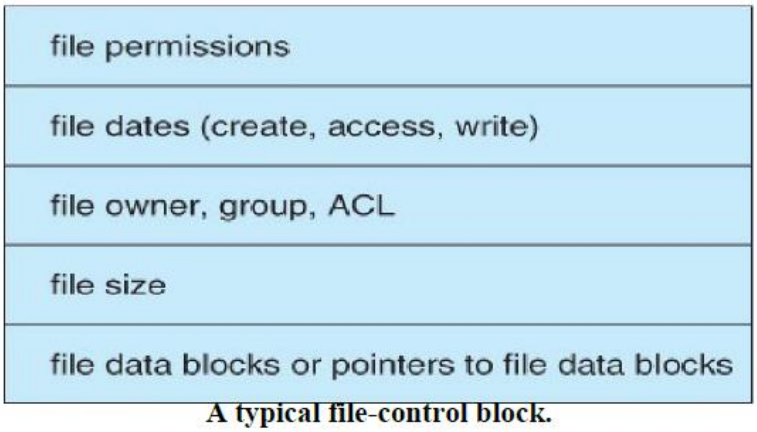
Layered file system.

File System Implementation:

Overview

File systems store several important data structures on the disk:

- A ***boot-control bloc*k**, (per volume) a.k.a. the *boot block* in UNIX or the *partition boot sector* in Windows contains information about how to boot the system off of this disk. This will generally be the first sector of the volume if there is a bootable system loaded on that volume, or the block will be left vacant otherwise.
- A ***volume control block***, (per volume) a.k.a. the *master file table* in UNIX or the *superblock* in Windows, which contains information such as the partition table, number of blocks on each file system, and pointers to free blocks and free FCB blocks.
- A directory structure (per file system), containing file names and pointers to corresponding FCBs. UNIX uses inode numbers, and NTFS uses a *master file table.*
- The ***File Control Block***, *FCB*, (per file) containing details about ownership, size, permissions, dates, etc. UNIX stores this information in inodes, and NTFS in the master file table as a relational database structure.



A typical file-control block.

There are also several key data structures stored in memory:

- An in-memory mount table.
- An in-memory directory cache of recently accessed directory information.
- *A **system-wide open file table***, containing a copy of the FCB for every currently open file in the system, as well as some other related information.
- *A **per-process open file table***, containing a pointer to the system open file table as well as some other information. (For example the current file position pointer may be either here or in the system file table, depending on the implementation and whether the file is being shared or not.)

Figure illustrates some of the interactions of file system components when files are created and/or used:

- When a new file is created, a new FCB is allocated and filled out with important information regarding the new file. The appropriate directory is modified with the new file name and FCB information.
- When a file is accessed during a program, the open ( ) system call reads in the FCB information from disk, and stores it in the system-wide open file table. An entry is added to the per-process open file table referencing the system-wide table, and an index into the per-process table is returned by the open( ) system call. UNIX refers to this index as a *file descriptor*, and Windows refers to it as a *file handle*.
- If another process already has a file open when a new request comes in for the same file, and it is sharable, then a counter in the system-wide table is incremented and the per-process table is adjusted to point to the existing entry in the system-wide table.
- When a file is closed, the per-process table entry is freed, and the counter in the system-wide table is decremented. If that counter reaches zero, then the system wide table is also freed. Any data currently stored in memory cache for this file is written out to disk if necessary.

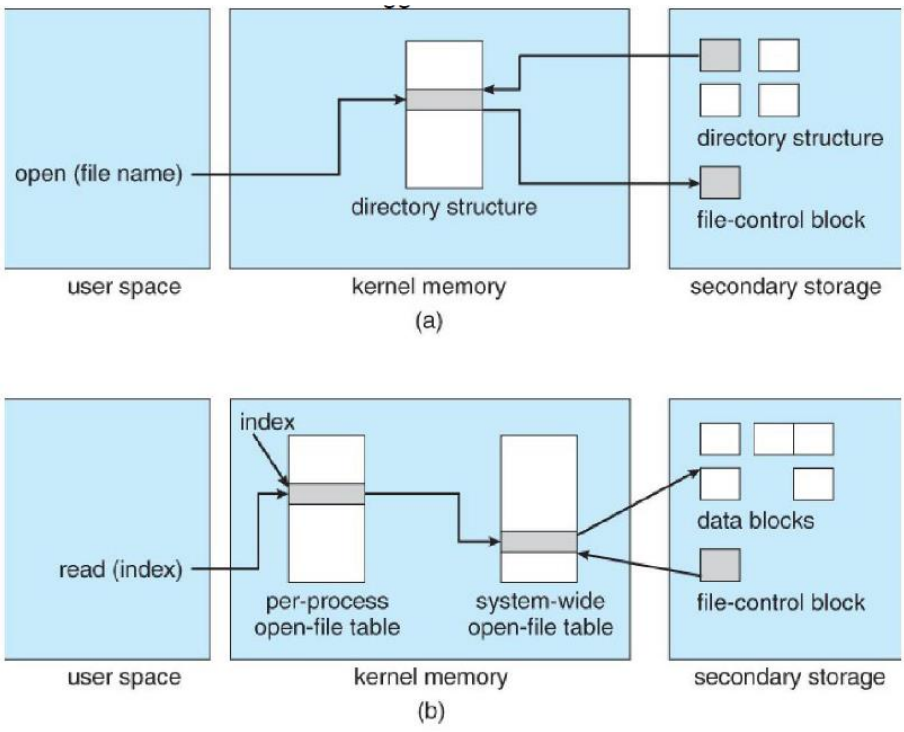


Figure 12.3 - In-memory file-system structures. (a) File open. (b) File read.

12.2.2 Partitions and Mounting

Physical disks are commonly divided into smaller units called partitions. They can also be combined into larger units, but that is most commonly done for RAID installations and is left for later chapters.

Partitions can either be used as raw devices (with no structure imposed upon them), or they can be formatted to hold a file system (i.e. populated with FCBs and initial directory structures as appropriate.) Raw partitions are generally used for swap space, and may also be used for certain programs such as databases that choose to manage their own disk storage system. Partitions containing filesystems can generally only be accessed using the file system structure by ordinary users, but can often be accessed as a raw device also by root.

The boot block is accessed as part of a raw partition, by the boot program prior to any operating system being loaded. The ***root partition*** contains the OS kernel and at least the key portions of the OS needed to complete the boot process. At boot time the root partition is mounted, and control is transferred from the boot program to the kernel found there. (Older systems required that the root partition lie completely within the first 1024 cylinders of the disk, because that was as far as the boot program could reach. Once the kernel had control, then it could access partitions beyond the 1024 cylinder boundary.)

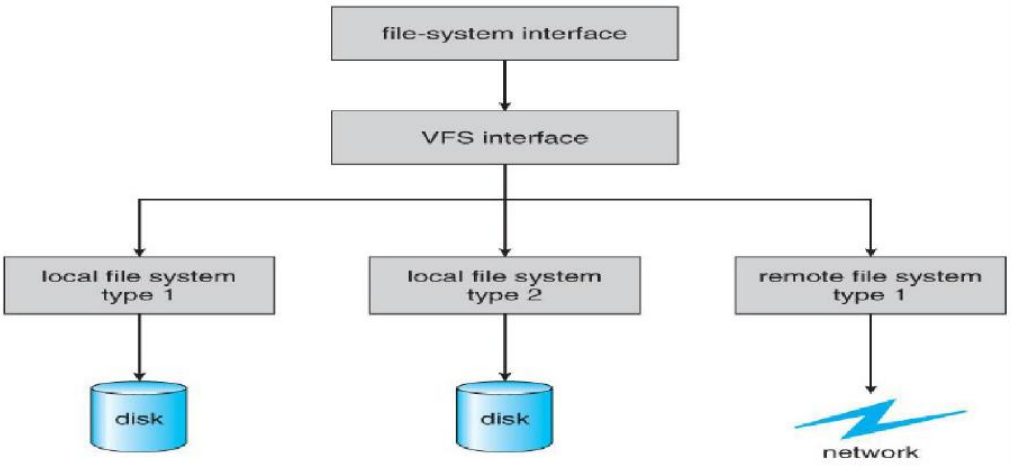
Virtual File Systems

Virtual File Systems, VFS, provide a common interface to multiple different file system types. In addition, it provides for a unique identifier (vnode) for files across the entire space, including across all file systems of different types. (UNIX inodes are unique only across a single file system, and certainly do not carry across networked file systems)

The VFS in Linux is based upon four key object types:

- The *inode* object, representing an individual file
- The *file* object, representing an open file.
- The *superblock* object, representing a file system.
- The *dentry* object, representing a directory entry.

Linux VFS provides a set of common functionalities for each file system, using function pointers accessed through a table. The same functionality is accessed through the same table position for all file system types, though the actual functions pointed to by the pointers may be file system-specific. Common operations provided include open( ), read( ), write( ), and mmap( ).



Schematic view of a virtual file system.