**Chapter 10 : File-System**

**10.1 File Concept**

Computers stores information. OS provides a logical view of stored information. OS abstracts from the physical properties of its storage devices to define a logical storage unit, the file. These storage devices are nonvolatile, so the contents stay persistent between system reboots.  
Information in a file is defined by its creator. Many different types of information are stored in a file. (source or executable programs, numeric or text data, photos, music etc…)

* **Text file**: sequence of characters organized into lines.
* **Source file**: sequence of functions, each of which is further organized as declarations followed by executable statements.
* **Executable file**: series of code sections that the loader can bring into memory and execute.

**10.1.1 File Attributes**

Different OSes keep track of different file attributes, including:

* **Name** - Some systems give special significance to names, and particularly extensions ( .exe, .txt, etc. ), and some do not. Only information kept in human-readable form.
* **Identifier** - A unique tag, identifies the file within file system.
* **Type**-Text, executable, other binary, etc.
* **Location** - Pointer to a device and to location of the file on that device.
* **Size** - Current size of the file.
* **Protection** - Access-control information determines who can do reading, writing, execution, and so on.
* **Time & Data & User ID** - Data can be used useful for protection, security, and usage monitoring.

Newer file systems support extended file attributes, including character encoding of the file and security features such as a file checksum. Attributes are displayed on any OS in the file info window.

**10.1.2 File Operations**

The file is an abstract data type (ADT) that supports many common operations:

* Creating a file, writing a file, reading a file, repositioning within a file, deleting a file, truncating a file.

The OS performs these operations by:

* Creating:

1. Find space in file system for the file. (allocating space: chapter 11)
2. Entry for new file made in the directory.

* Writing:

1. System call made specifying name of the file and information to be written to the file.
2. Given name of the file, system searches directory to find file’s location.
3. System keeps a **write pointer** to the location in the file where the next write is to take place.
4. Write pointer is updated whenever a write occurs.

* Reading:

1. System call specifies name of the file and where (in memory) the next block the file should be put.
2. Directory is searched for associated entry, and the system needs to keep a read pointer to the location in the file where the next read is to take place.
3. Once read has taken place, read pointer is updated.

(Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process **current-file-position pointer**. Both read and write operations use this same pointer, saving space and reducing system complexity.)

* Repositioning:

1. Directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value.

(Repositioning within a file need not involve any actual I/O. This operation is also known as a file **seek**.)

* Deleting:

1. Search directory for named file.
2. Having found associated directory entry, release all file space, so that it can be reused by other files, and erase the directory entry.

* Truncating:

1. User may want to erase contents of a file, but keep its attributes.
2. Allows all attributes to remain unchanged, except for file length, but let’s file to be reset to length zero and its file space released.

Most OSes require that files be ***opened*** before access and ***closed*** after all access is complete. (with an open()/closed() system call) Normally the programmer must open and close files explicitly, but some rare systems open the file automatically at first access. Information about currently open files is stored in an **open file table**, containing for example:

* **File pointer** - System tracks last read/write location as the current-file-position pointer.
* **File-open count** - How many times has the current file been opened (simultaneously by different processes) and not yet closed? When this counter reaches zero the file can be removed from the table.
* **Disk location of the file**
* **Access rights** -Each process opens file in an access-mode. This information is stored on the per-process table so the OS can allow are deny subsequent I/O requests.

Some systems provide support for ***file locking.*** File locks allow one process to lock a file and prevent other processes from gaining access to it. For example:

* A ***shared lock*** is for reading only. (multiple processes)
* An ***exclusive lock***is for writing as well as reading. (one process at a time)
* An ***advisory lock*** is informational only, and not enforced. (A "Keep Out" sign, which may be ignored)
* A ***mandatory locking mechanism*** is enforced. (A truly locked door)

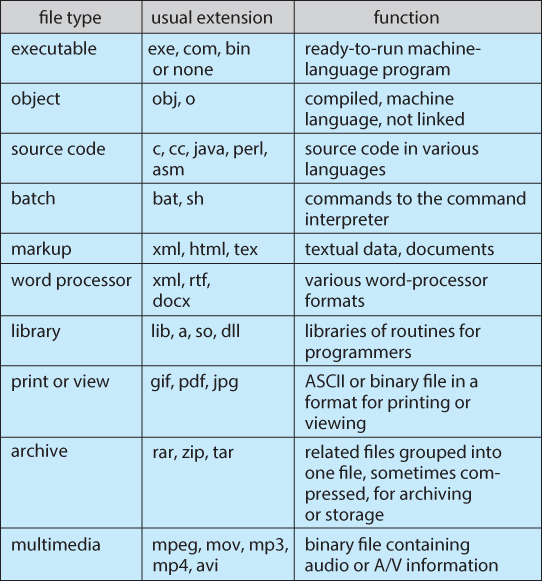
1. Once a process acquires an exclusive lock, OS will prevent any other processes from accessing the locked file.

* UNIX used advisory locks, and Windows uses mandatory locks.

**Figure 10.2 - File-locking example in Java. (see p. 461)**

**10.1.3 File Types**

Windows ( and some other systems ) use special file extensions to indicate the type of each file:

  
**Figure 10.3 - Common file types.**

* **Shell script** -- .sh extension: commands to the OS.
* Mac OS stores ‘type’ and ‘creator attribute’ for each file, according to the program that first created it with the create() system call.
* File produced by word processor has word processor’s name as its creator. When user opens that file, the word processor is invoked and the file is loaded, ready to be edited.
* UNIX stores **magic numbers** at the beginning of certain files. Indicating roughly type of the file. Not all files have a magic number. Unix doesn’t record name of creating program.

**10.1.4 File Structure**

Some files contain an internal structure, which may or may not be known to the OS. Structures of files provide expectations of the programs that read them.  
  
For the OS to support particular file formats increases the size and complexity (code to support multiple file structures) of the OS.

* UNIX treats all files as sequences of bytes, with no further consideration of the internal structure. ( With the exception of executable binary programs, which it must know how to load and find the first executable statement, etc. )

When new apps require information structured in ways not supported by the OS, severe problems may result:

* System support two files. Neither file type is appropriate. We have to circumvent or misuse the file. (text and exec binary, encrypted -> random bits -> not text, not exec -> problem!)

**10.1.5 Internal File Structure**

Disk files are accessed in units of physical blocks (typically 512 bytes or a 2^n). (Larger physical disks use larger block sizes)

Internally files are organized in units of logical records   
(fixed size: as small as a single byte, to corresponding to some data record or structure size.)

Number of logical records which fit into one physical block determines its ***packing*** (done by either user app program or OS), and has an impact on the amount of internal fragmentation (wasted space) that occurs:

* General rule: half a physical block is wasted for each file! The larger the block size, the more space is lost to internal fragmentation.

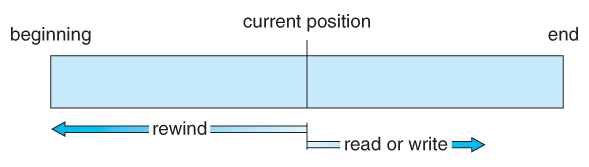
**10.2 Access Methods**

Files store information. When used, information must be accessed and read into computer memory. Information in the file can be accessed in several ways.

**10.2.1 Sequential Access**

Information processed in order, one record after the other. A sequential access file based on a tape model of a file.  
Operations generally supported:

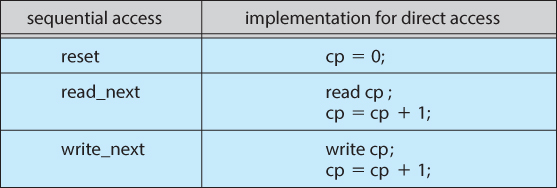
* Read next - read a record and advance the tape to the next position.
* Write next - write a record and advance the tape to the next position.
* Rewind
* Skip n records - N may be limited to positive numbers or not, or may be limited to +/- 1. (May or may not be supported)

  
**Figure 10.4 - Sequential-access file.**

**10.2.2 Direct Access ( or relative access)**

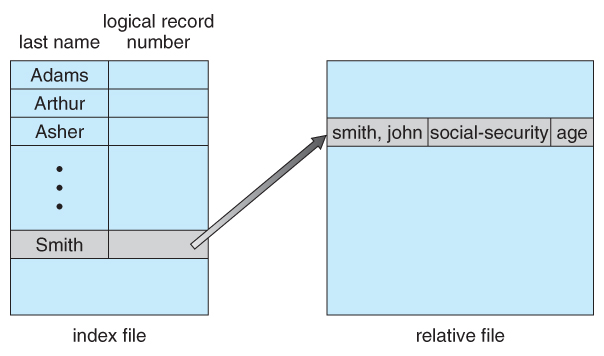
Allows programs to jump to any record and read/write from/to that **logical record**. The direct-access method is based on a disk model of a file. (disk allow random access to any file block.) Great use for immediate access to large amounts of information (databases).  
Operations supported include:

* Read n - read record number n. ( Note an argument is now required. )
* Write n - write record number n. ( Note an argument is now required. )
* Jump to record n - could be 0 or the end of file.
* Query current record - used to return back to this record later.
* Sequential access can be easily emulated using direct access. The inverse is complicated and inefficient. (position\_file(n)/jump\_record(n) and then read\_next().)
* The block number (‘n’) provided by the user to the OS: **relative block number**.  
  It’s an index relative to the beginning of the file. First one is ‘0’(or 1) and the next one ‘1’(or 2), even though absolute disk address may be 14703 for the first block and 3912 for the second.
* Use of relative block numbers -> helps OS decide where file be placed (**the allocation problem**) -> prevents user from accessing parts of the file system that aren’t part of her file.

  
**Figure 10.5 - Simulation of sequential access on a direct-access file.**

Not all OS support both sequential and direct access for files. Sometimes only one and the other not. Some systems require file be defined as sequential or direct when its created, so it can be accessed in a manner consistent with its declaration. **10.2.3 Other Access Methods**

An indexed access scheme can be easily built on top of a direct access system with the construction of an **index** for the file. Very large files may require a multi-tiered indexing scheme, i.e. indexes of indexes. (example: IBM ISAM)

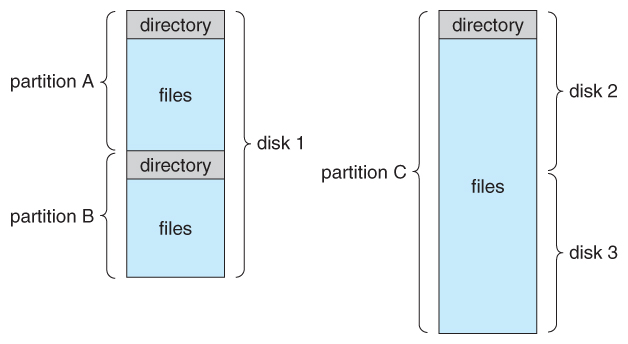
  
**Figure 10.6 - Example of index and relative files.**

**10.3 Directory Structure**

A storage device (where file are stored on) can be used in its entirety for a file system.  
Alternatively a physical disk can be broken up into multiple ***partitions, slices, or mini-disks***, each of which becomes a virtual disk and can have its own file system. (or be used for raw (unformatted) storage, swap space, etc.)   
Advantages of partitioning:

* Protection of a failing disk: RAID sets.
* Limiting size of individual file systems.

Multiple physical disks can be combined into one ***volume***, i.e. a larger virtual disk, with its own file system spanning the physical disks. (volumes can store multiple, allowing a system to boot and run more than one OS)   
 Each volume with a file system must contain information about the files in the system. Information is kept in entries in a **device directory** or **volume table of contents**. It records information -- such as name, location, size, and type -- for all files on that volume as seen in figure 10.7

  
**Figure 10.7 - A typical file-system organization.**

**10.3.1 Storage Structure**

General-purpose computer has multiple storage devices, and those devices can be sliced up into volumes that hold file systems. (system may have zero to … file systems, and file system may be of varying types.)  
(see p. 469 for Solaris file system example)

**10.3.2 Directory Overview**

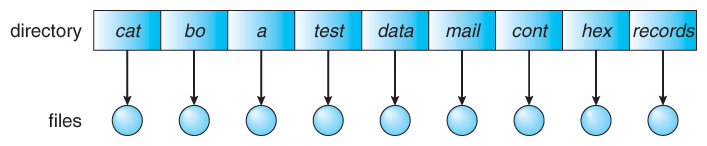
Within a file system, it’s useful to segregate files into groups and manage and act on those groups. This organization involves use of directories.  
Directory: symbol table that translates file names into directory entries.  
Directory operations to be supported include:

* Search for a file - find entry for particular file or files who match pattern.
* Create a file - add to the directory.
* Delete a file - erase from the directory.
* List a directory - possibly ordered in different ways.
* Rename a file - may change sorting order.
* Traverse the file system - access every directory and every file within a directory structure.

**10.3.3. Single-Level Directory**

Simple to implement (all files are contained in the same directory), but each file must have a unique name. Problems with large number of files are multiple users (largely due to the name restriction).

* Two users call data file test.txt -> unique-name rule violated
* One user but thousands of files -> keeping track of name creation

  **Figure 10.9 - Single-level directory.**

**10.3.4 Two-Level Directory**

Each user gets their own directory space.

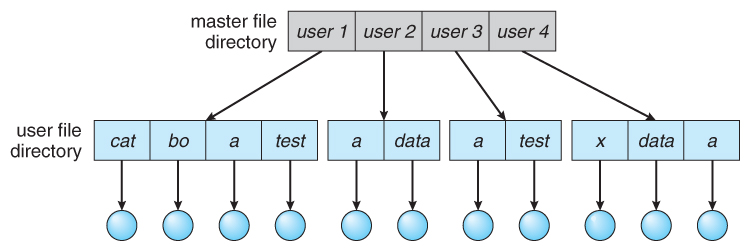
File names only need to be unique within a given user's directory.

A master file directory is used to keep track of each users directory, and must be maintained when users are added to or removed from the system.

A separate directory is generally needed for system (executable) files.

Systems may or may not allow users to access other directories besides their own

* If access to other directories is allowed, then provision (**a path name**: user name and file name) must be made to specify the directory being accessed.
* If access is denied, then special consideration must be made for users to run programs located in system directories. A ***search path*** is the list / sequence of directories in which to search for executable programs, and can be set uniquely for each user.

  
**Figure 10.10 - Two-level directory structure.**

**10.3.5 Tree-Structured Directories**

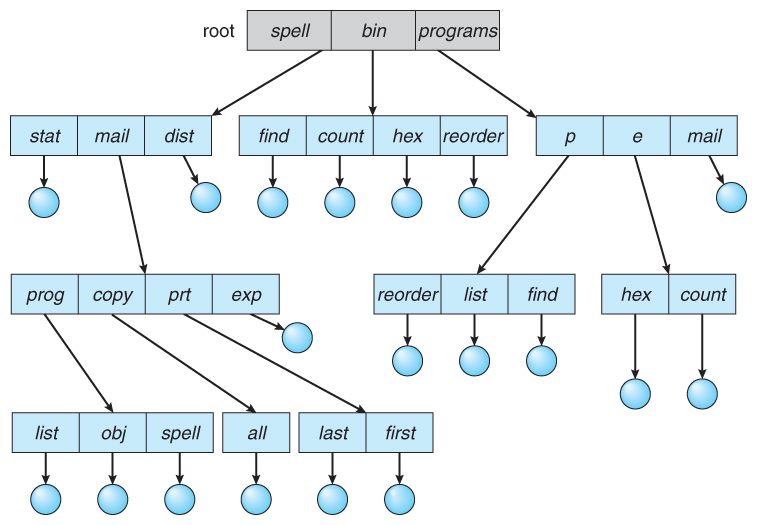
An obvious extension to the two-tiered directory structure, and the one with which we are all most familiar. A directory structure to a tree of arbitrary height.

Each user / process has the concept of a **current directory** from which all (relative) searches take place.

Files may be accessed using either **absolute path names** (relative to the root of the tree) or **relative path names** (relative to the current directory.)

Directories are stored the same as any other file in the system, except there is a bit that identifies them as directories, and they have some special structure that the OS understands.

One question for consideration (a sort of system policy) is whether or not to allow the removal of directories that are not empty - Windows requires that directories be emptied first, and UNIX provides an option for deleting entire sub-trees.

  
**Figure 10.11 - Tree-structured directory structure.**

**10.3.6 Acyclic-Graph Directories (EDIT: p. 476 wasn’t discussed properly! todo)**

When the same files need to be accessed in more than one place in the directory structure (e.g. because they are being shared by more than one user / process), it can be useful to provide an **acyclic-graph** structure.  
(Note the *directed*arcs from parent to child.) (+ shared means 1 file, no 2 copies, change to the file is immediate to the other user, new file will auto appear in the shared subdirectories)  
Acyclic-graph is more flexible than a simple tree structure, but it’s also more complex.

UNIX provides two types of ***links*** for implementing the acyclic-graph structure. (graph with no cycle)

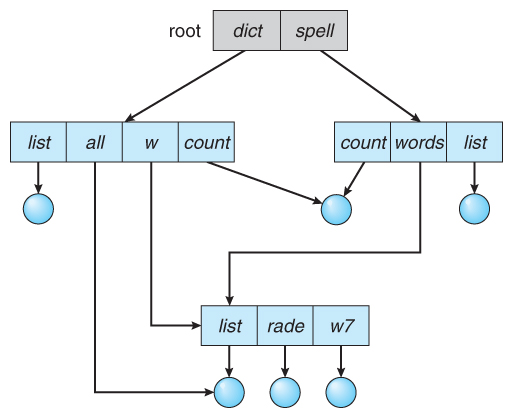
* A ***hard link*** (usually just called a link) involves multiple directory entries that both refer to the same file or subdirectory. Hard links are only valid for ordinary files in the same file system. (link to another link (indirect pointer) -> **resolve** the link by using path name to locate real file; these are ignored by OS to preserve acyclic structure)
* A ***symbolic link***, that involves a special file, containing information about where to find the linked file. Symbolic links may be used to link directories and/or files in other file systems, as well as ordinary files in the current file system.

Windows only supports symbolic links, termed***shortcuts.***

Hard links require a***reference count***, or ***link count*** for each file, keeping track of how many directory entries are currently referring to this file. Whenever one of the references is removed the link count is reduced, and when it reaches zero, the disk space can be reclaimed.

For symbolic links there is some question as to what to do with the symbolic links when the original file is moved or deleted:

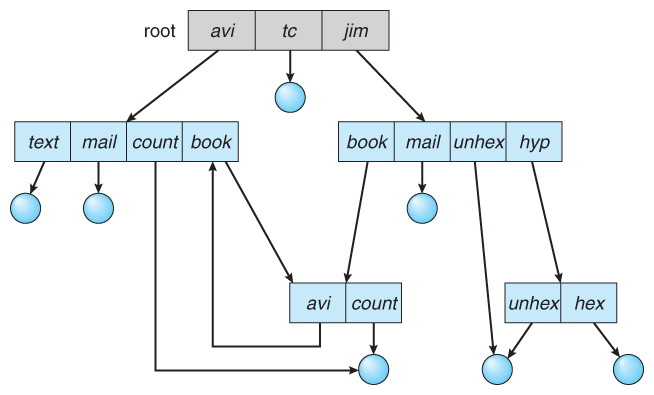
* One option is to find all the symbolic links and adjust them also.
* Another is to leave the symbolic links dangling, and discover that they are no longer valid the next time they are used.
* What if the original file is removed, and replaced with another file having the same name before the symbolic link is next used?

  
**Figure 10.12 - Acyclic-graph directory structure.**

**10.3.7 General Graph Directory**

If cycles are allowed in the graphs, then several problems can arise:

* Search algorithms can go into infinite loops.
* One solution is to not follow links in search algorithms.
* Or not to follow symbolic links, and to only allow symbolic links to refer to directories.
* Or to limit number of directories that will be accessed during a search
* When cycles exist sub-trees can become disconnected from the rest of the tree and still not have their reference counts reduced to zero. Periodic **garbage collection** is required to detect and resolve this problem.  
  Garbage collection: traversing entire file system, marking everything that can be accessed. Disconnected disk blocks that are not marked as free are added back to the file systems with made-up file names, and can usually be safely deleted. (GC is extremely time consuming)

  
**Figure 10.13 - General graph directory.**

**10.4 File-System Mounting (//TODO p. 478, p 479)**

The basic idea behind mounting file systems is to combine multiple file systems into one large tree structure. It works as follows:

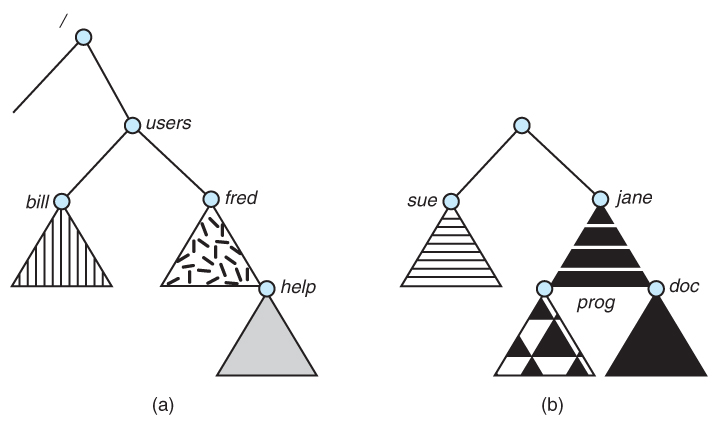
* OS is given the name of the device to mount and a **mount point**, the location within the file structure where the file system is to be attached. (some OS require type of file system)
* OS verifies that the device contains a valid file system. (asks device driver to read device directory and verifying that directory has expected format)
* OS notes in its directory structure that a file system is mounted at the specified mount point.

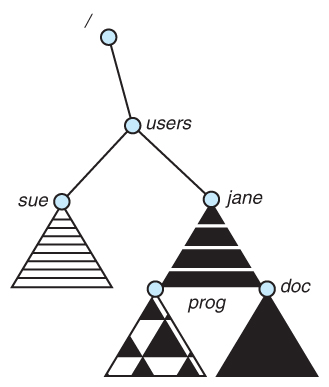
Once a file system is mounted onto a mount point, any further references to that directory actually refer to the root of the mounted file system.

Any files (or sub-directories) that had been stored in the mount 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 file systems to be mountable onto certain pre-determined mount points. (E.g. root may allow users to mount floppy file systems to /mnt or something like it.) Anyone can run the mount command to see what file systems are currently mounted.

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

  
**Figure 11.14 - File system. (a) Existing system. (b) Unmounted volume.**

  
**Figure 11.15 - Mount point.**

Examples of mounting in existing OSes:

* 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 file systems to be mounted to any directory in the file system, much like UNIX.

**10.5 File Sharing**

**10.5.1 Multiple Users**

Implementation of sharing and protection on a multi-user system, more information needs to be stored for each file:

* The **owner** (or **user**) who owns the file, and who can control its access and attributes.
* The **group** are a subset of other user IDs that may have some special access to the file.
* What access rights are afforded to the owner (**U**ser), the **G**roup, and to the rest of the world (the universe, a.k.a. **O**thers.)
* Some systems have more complicated access control, allowing or denying specific accesses to specifically named users or groups.

**10.5.2 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 and password controlled, or **anonymous (access)**, not requiring any user name or password.
* Various forms of **distributed file systems** (**DFS**) 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 file systems, generally using (anonymous file exchange) ftp as the underlying file transport mechanism. (cloud computing also used for file sharing.)

**10.5.2.1 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**.

* Server declares that a resource is available to clients and specifies which resource and which clients.
* A server can serve multiple clients, a client can use multiple servers. (depends on client-server facility)

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.
* Solution: secure authentication of the client via encrypted keys. -> hard to do, problems; compatibility server-client + security of key exchange.
* Servers may restrict remote access to read-only.
* Servers restrict which file systems 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 (used in UNIX).

**10.5.2.2 Distributed Information Systems**

**Distributed Information Systems** (**or Distributed Naming Services**) make client-server systems easier to manage by providing unified access to the information needed for remote computing.

* The **Domain Name System, DNS,**provides for a unique naming system across all of the Internet. (host-name-to-network-address translation)
* Domain names are maintained by the **Network Information System, NIS**, which unfortunately has several security issues. NIS centralizes storage of user names, host names, printer information, and the like.  
  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.

**10.5.2.3 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.

Implement recovery from failure: some kind of **state information** may be maintained on both the client and the server. If both maintain knowledge of their current activities, they can recover from a failure.  
NFS take simple approach: **stateless** DFS:

* It assumes that a client request for a file read or write would not have occurred unless the file system had ben remotely mounted and the file had been previously open. The NFS protocol carries all the information needed to locate the appropriate file and perform the requested operation.
* It does not track which clients have the exported volumes mounted, again assuming that if a request comes in, it must be legitimate.

While this stateless approach makes NFS resilient and rather easy to implement, it also makes it insecure. These issues were addressed in the industry standard NFS Version 4, in which NFS is made stateful to improve its security, performance and functionality.

**10.5.3 Consistency Semantics**

**Consistency Semantics** deals with the consistency between the views of shared files on a networked system. These semantics specify how multiple users of a system are to access a shared file simultaneously. 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.

Therefore we assume that a series of file accesses attempted by a user to the same file is always enclosed between the open() and close() operations. The series of accesses between these operations makes a **file session**.

**10.5.3.1 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 mode of sharing uses a shared location pointer, thus the advancing of the pointer by one user adjusts it for all sharing users.

The file is associated with a single physical image that is accessed as an exclusive physical resource, contention for this single image may cause delays in user processes.

**10.5.3.2 Session Semantics**

The Andrew File System, Open AFS uses the following semantics:

* Writes to an open file are not immediately visible to other users.
* When a file is closed, any changes made become 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) images at the same time. 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.

**10.5.3.3 Immutable-Shared-Files Semantics**

A unique approach is that of **immutable shared** files. 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.

**10.6 Protection**

Files must be kept safe for reliability (against accidental damage), and protection (against deliberate malicious access.) The former is usually managed with backup copies. This section discusses the latter.

**10.6.1 Types of Access**

One simple protection scheme is to remove all access to a file. However this makes the file unusable, so some sort of controlled access must be arranged.  
What we need is controlled access by limiting the types of file access that can be made.  
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 into memory and execute it.
* Append - Add new information to the end of an existing file.
* Delete - Remove a file from the system and free its space.
* List -View the name and other attributes of files on the system.

Higher-level operations, such as copy or rename, can generally be performed through combinations of the above. Protection is provided at only the lower level.

**10.6.2 Access Control**

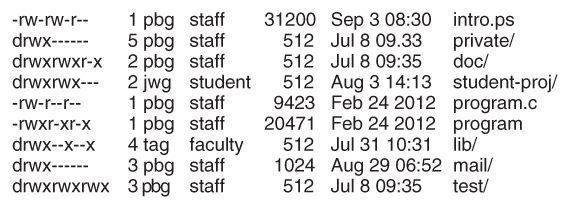
One approach is to have complicated**Access Control Lists, ACL,**which specify exactly what access is allowed or denied for specific users or groups.  
The AFS uses this system for distributed access.  
Two undesirable consequences of ACL:

* 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. )
* The directory entry now must be of variable size, resulting in more complicated space management.

These problems can be resolved by using a condensed version of the access list.

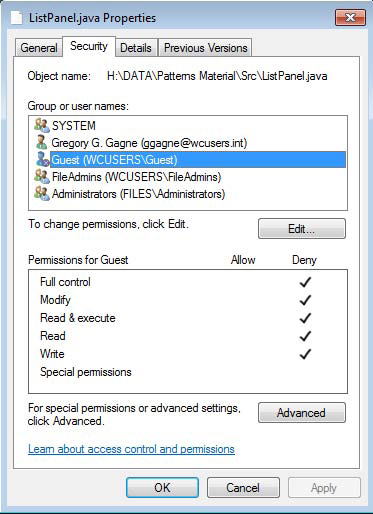
UNIX uses a set of 9 access control bits, in three groups of three. These correspond to R, W, and X permissions for each of the Owner, Group, and Others. The RWX bits control the following privileges for ordinary files and directories:

|  |  |  |
| --- | --- | --- |
| **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. |

Example on p. 487 - 488.

**Sample permissions in a UNIX system.**

Windows adjusts files access through a simple GUI:

  
**Figure 10.16 - Windows 7 access-control list management.**

**10.6.3 Other Protection Approaches and Issues**

Some systems can apply passwords, either to individual files, or to specific sub-directories, or to the entire system. There is a trade-off between the number of passwords that must be maintained (and remembered by the users) and the amount of information that is vulnerable to a lost or forgotten password.

Older systems which did not originally have multi-user file access permissions ( DOS and older versions of Mac ) must now be ***retrofitted*** if they are to share files on a network.

Access to a file requires access to all the files along its path as well. In a cyclic directory structure, users may have different access to the same file accessed through different paths.

Sometimes just the knowledge of the existence of a file of a certain name is a security (or privacy) concern. Listing contents of contents of a directory must be a protected operation. Hence the distinction between the R and X bits on UNIX directories.

**10.7 Summary (//TODO add summary)**