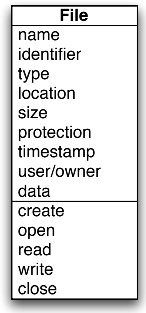
File Systems: Interface

- As mentioned previously, operating systems offer an *abstraction* to user data in the form of *files*
- A file is a *logical storage unit* — they do not appear as such on storage devices, but whatever is on these devices is *presented to us* by the OS as files
- In addition to the data that they hold, files also contain attributes of their own, and have certain operations associated with them — thus, in UML, a file may be (partially) modeled as shown on the right



File Management Issues

- While it is technically possible to operate on any file at any time, it is not generally practical; instead, many file operations must be "book-ended" by *open* and *close* system calls
- The *open* call allows the operating system to track the set of files that are actively being used by processes
- File access and sharing by multiple processes can also be an issue, so an operating system may provide a variety of *locks* that help coordinate and protect files among these processes

File Types

- Data comes in many forms and formats (documents, images, audio, video, executables, source code…), so we attach a notion of *type* to a file
- By far the most common typing mechanism is really a "type hint" — the *filename extension*
- Other approaches include *magic numbers* in Unix, *type and creator codes* in the original Mac OS, and *MIME* on the Internet and BeOS
- Mac OS X has introduced *uniform type identifiers* (UTIs) — good potential, but yet to be proven

File Structure

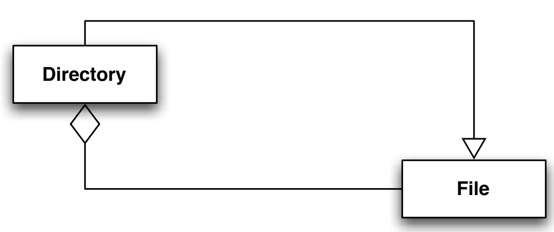
- An OS may also choose to support known *file structures* — predefined ways for what is in a file and how it is organized
- Generally, user processes manage this; only a handful of file structures need to be truly known by the OS:
  - Executables, libraries, and other files containing code
  - Text vs. binary: text implies some conventions, such as newlines and character mappings (ASCII, Unicode)
- Original Mac OS used separate *data* and *resource forks*

Access Methods

- Two primary approaches have evolved for accessing the information in a file:
  - *Sequential access* views a file as a *linear stream*, to be accessed from beginning to end
  - *Direct access*, a.k.a. *random* or *relative access*, assumes that files consist of fixed-length *logical records*, and allows immediate movement to any record in the file
- Other methods (e.g., *indexed access*; Java's "stream zoo") are composites of sequential/direct access

Directory Structure

- Collections of files are typically gathered into *directories* — in design-pattern terms, directories and files may be viewed as forming a *composite* pattern:



- Directories typically hold many of the attributes associated with a file; internally, they also hold a reference to the file's data on a device

Types of Directories

- *Single-* or *two-level directories* are just that — they do not allow arbitrarily deep directory structures
- *Tree-structured directories* allow directories within directories, potentially of unlimited depth
  - The top of the tree is typically called the *root*
  - Unix presents a single tree, regardless of the underlying number of devices; Windows presents multiple trees, each rooted at a device (thus it can be viewed as having an "extended" two-level structure)

- *Acyclic-graph directories* allow multiple directories to refer to a single file — specifics vary by OS
  - Windows uses a special *.LNK* file (a "shortcut") that encodes assorted information about a file
  - Unix has 2 techniques: *symbolic links* use only a file's path, while a *hard link* is an independent directory entry that points to the same underlying data
  - Mac OS X also supports an *alias* file that encodes additional data for finding the target file in case it is moved or renamed
- When file reference cycles are allowed, we have a *general graph directory*

File-System Mounting

- Note again that files and directories are *logical* structures — they are meant to abstract out the concrete reality of storage devices and media
- *Mounting* is the act of "connecting" different storage devices to the logical directory structure
- Unix (including Linux, Mac OS X) *subsume* the devices into a single logical directory; Windows uses a separate drive letter, so devices are not path-transparent
- Mounting on a non-empty directory requires a design decision: prohibit or obscure?

File Sharing

- Sharing across users — traditional approach is to assign an *owner* and *group* to a file; owners can do anything, while group members can perform a subset
- Sharing across computers (on a network) — two main paradigms; may be anonymous or authenticated
  - Manual transfer: *ftp* (file transfer), *rcp* (remote copy), *scp* (secure copy), *http*
  - Remote mount: *NFS*, *smb* (a.k.a. CIFS), *afp* (a.k.a. AppleShare)
- Failure modes: remote file access is subject to more possible errors than local devices (network partition, server crash), so error-handling must be more robust

Consistency Semantics

- *Consistency semantics* refers to how concurrent file modifications by multiple users should behave; in the general case, shared file access can be viewed as a critical-section problem, but is generally not solved in that way due to performance reasons
- In Unix, a file is a mutually-exclusive resource; writes are serialized (possibly causing processes to wait), and changes are visible right away to other processes
- The *Andrew file system* (AFS) allows for multiple writes, invisible to other processes until a file is closed

File Protection

- File systems also provide some form of *protection* — the prevention of "improper access" to files
- General approach is to define the *operations* to be controlled (read, write, execute, etc.), then specify which users can perform which operations — this ranges from traditional Unix owner/group/all permissions to variable-sized lists of user/operation rules, called *access control lists* or ACLs
- The brave new world of malware adds the need to protect a user from some of his or her *own* files!