

File Systems

- Result of integration of storage resources under a single hierarchy
- File
 - A collection of related information defined by its creator
 - The abstraction used by the kernel to represent and organize the system's non-volatile storage resources, including hard disks, floppy disks, CD-ROMs, and optical disks.
- Storage capacity of a system restricted to size of available virtual memory
 - May not be enough for applications involving large data
 - Virtual memory
 - * Volatile
 - * May not be good for long term storage
 - Information need not be dependent upon process
 - * /etc/passwd file may need to be modified by different processes
- Essential requirements of long-term information storage
 - Store very large amount of information
 - Information must survive termination of processes (be *persistent*)
 - Multiple processes must be able to access information concurrently
 - Store information in *files*
- File system – Part of the OS that deals with file management
 - Creating, destroying, organizing, reading, writing, modifying, moving, and controlling access to files
 - Management of resources used by files
- Basic functions of a file system
 - Present a logical or *abstract* view of files to users by hiding physical details of I/O devices
 - Facilitate the *sharing* of physical I/O devices and *optimize* their use
 - Provide PROTECTION mechanisms for data being transferred or managed by I/O devices

Files

- Most visible aspect of an OS, beside the user interface itself
- Mechanism to store and retrieve information from the disk
- Represent programs (both source and object) and data
- Data files
 - May be numeric, alphanumeric, alphabetic, or binary
 - May be free form or formatted rigidly
- Accessed by a *name*
 - In both Unix and Windows, the filename can be up to 255 characters while DOS limits the name to 8 characters with a 3 character extension (the 8.3 system)

- Unix name can consist of any characters, except / and \
- Windows filename (long filename) can be any character except \ / : * ? " < > |
 - * Windows (including W2K) creates an 8.3 version of every long filename
 - * The short filename is created by inserting ~ followed by sequential numbers after the sixth character of the filename, plus the original extension
- Created by a process and continues to exist after the process has terminated
- Information in the file defined by creator
- Naming conventions
 - Set of a fixed number of characters (letters, digits, special characters)
 - Case sensitivity
 - File type should be known to OS
 - * to avoid common problems like printing binary files
 - * to automatically recompile a program if source modified (TOPS 20)
 - File extension in DOS and Windows
 - Getting information about a file
 - * The `file(1)` command
 - * `stat(2)` system call
 - First test performed by `file(1)`
 - Returns a file's *metadata* – information about file in the system
 - File's owner, access permissions, modification time information, and size
 - File type stored as a part of file permissions – types are only those used by the kernel to distinguish between files
 - *Magic number* in Unix
 - * Second test performed by `file(1)`
 - * Identification number for the type of file
 - * The `file(1)` command identifies the type of a file using, among other tests, a test for whether the file begins with a certain *magic number*
 - * Magic number is specified in the file `/etc/magic` using four fields
 1. Offset: A number specifying the offset, in bytes, into the file of data which is to be tested
 2. Type: Type of data to be tested – byte, short (2-byte), long (4-byte), or string
 3. Value: Expected value for file type
 4. Message: Message to be printed if comparison succeeds
 - * Used by the C compiler to distinguish between source, object, and assembly file formats
 - * Developing magic numbers
 - Start with first four letters of program name (e.g., list)
 - Convert them to hex: 0x6c607374
 - Add 0x80808080 to the number
 - The resulting magic number is: 0xECE0F3F4
 - High bit is set on each byte to make the byte non-ASCII and avoid confusion between ASCII and binary files
- File structure at user level
 - Characterized by field, record, file, and database
 - Field

- * Basic data element
- * Contains a single value such as name or date
- * May be fixed length or variable length
- * Variable length fields can be indicated by special demarcation fields
- Record
 - * Collection of related fields, treated as a unit by an application
 - * Employee record
 - * May be fixed length or variable length
 - * Entire record may include a length field
- File
 - * Collection of similar records
 - * Treated as a single entity by users and applications and may be referenced by a name
 - * May be created and deleted
 - * Used to provide access control restrictions; in sophisticated systems, these restrictions may be provided at record or field level
- Database
 - * Collection of related data
 - * Explicit relationships among data elements/fields
 - * May contain one or more files
 - * May be managed by a database management system that could be independent of OS
- File structure at lower level
 - Byte sequence or stream
 - * Used in almost all modern systems, including Unix and MS-DOS
 - * Meaning on the bytes is imposed by user programs
 - * Provides maximum flexibility but minimal support
 - * Advantage to users who want to define their own semantics on files
 - Record sequence
 - * Each file of a fixed length record
 - * Card readers and line printer based records
 - * Used in CP/M with a 128-character record
 - Tree
 - * Useful for searches
 - * Used in some mainframes
- File access
 - Sequential access
 - * Bytes or records can be read only sequentially
 - * Like magnetic tape
 - Random access
 - * Bytes or records can be read out of order
 - * Access based on key rather than position
 - * Like magnetic disk
 - Distinction more apparent in older OSs
- File types

- Regular files
 - * Most common types of files
 - * May contain ASCII characters, binary data, executable program binaries, program input or output
 - * No kernel level support to structure the contents of these files
 - * Both sequential and random access are supported
- Directories
 - * Binary file containing a list of files contained in it (including other directories)
 - * May contain any kind of files, in any combination
 - * `.` and `..` refer to directory itself and its parent directory
 - * Non-empty directories can be deleted by `rm -r`
 - * Each entry made up of a file-inode pair
 - Used to associate inodes and directory locations
 - Data on disk has no knowledge of its logical location within the filesystem
- Character-special files and Block-special files
 - * Allow Unix applications to communicate with the hardware and peripherals
 - * Reside in the `/dev` directory
 - * Kernel keeps the links for device drivers (installed during configuration)
 - * Device drivers
 - Connect a logical device (`/dev/audio`) to a physical device (the speaker)
 - Present a standard communications interface
 - Take the request from the kernel and act upon it
 - * Character-special files
 - Allow the device drivers to perform their own I/O buffering
 - Used for unbuffered data transfer to and from a device
 - * Block-special devices
 - Expect the kernel to perform buffering for them
 - Used for devices that handle I/O in large chunks, known as blocks
 - Helpful to choose the best order of response to requests for devices
 - * Possible to have more than one instance of each type of device
 - Device files characterized by major and minor device number
 - Major device number
 - Tells the kernel the driver corresponding to the file
 - Used to *register* the driver with the kernel by assigning it a major number during the module's initialization
 - Minor device number
 - Tells the kernel about the specific instance of the device
 - Tape drivers may use the minor device number to select the density for writing the tape
- Hard links
 - * Additional name (alias) for a file
 - * Associates two or more filenames with the same inode
 - * Indistinguishable from the file it is linked to
 - * Share the same disk data blocks while functioning as independent directory entries
 - * May not span disk partitions as inode numbers are only unique within a given device
 - * Unix maintains a count of the number of links that point to the same file and does not release the data blocks until the last link has been deleted
 - * Created with `ln` and removed with `rm`
- Symbolic links

- * Also known as *soft* link
- * Pointer files that name another file elsewhere on the file system
- * Reference by name; distinct from the file being pointed to
- * Points to a Unix pathname, not to an actual disk location
- * May even refer to non-existent files, or form a loop
- * May contain absolute or relative path
- * Created with `ln -s`
- * Problem of using `..` in the symbolic link
- FIFO special file, or “named pipe” (ATT)
 - * Characterized by transient data
 - * Allow communications between two unrelated processes running on the same host
 - * Once data is read from a pipe, it cannot be read again
 - * Data is read in the order in which it was written to the pipe, and the system allows no deviation from that order
 - * Created with `mknod` command and removed with `rm`
- Unix domain sockets (BSD)
 - * Connections between processes for communications
 - * Part of the TCP/IP networking functionality
 - * Communication end point, tied to a particular port, to which processes may attach
 - * Socket `/dev/printer` is used to send messages to the line printer spooling daemon `lpd`
 - * Visible to other processes as directory entries but cannot be read from or written into by processes not involved in the connection
 - * Created with the `socket` system call, and removed with `rm` or `unlink` command (if not in use)
- Regular files
 - Text files (ASCII)
 - * Lines of text
 - * Lines may be terminated by carriage return
 - * File itself has an end-of-file character
 - * Useful for interprocess communication via pipes in Unix
 - Binary files
 - * Not readily readable
 - * Has internal structure depending upon the type of file (executable or archive)
 - * Executable file (`a.out` format)
 - Header: Magic number, Text size, Data size, BSS size, Symbol table size, Entry point, Flags
 - Text
 - Data
 - Relocation bits
 - Symbol table
 - * BSS or Block Started by Symbol
 - Uninitialized data for which kernel should allocate space
 - Used by an obsolete IBM assembler, BSS was an assembler pseudo-opcode that filled an area of memory with zeros
 - * Library archive: compiled but not linked modules
 - Header: Module name, Date, Owner, Protection, Size
 - Object module
- File attributes

- Data about files
- Protection, Password, Creator, Owner, Read-only flag, Hidden file flag, System file flag, Archive flag, ASCII/binary flag, Random access flag, Temporary flag, Lock flag, Record length, Key position, Key length, Creation time, Time of last access, Time of last change, Current size, Maximum size
- File operations: File treated as abstract data type
 - create.
 - * Create a new file of size zero (no data)
 - * Unix commands: `creat(2)`, `open(2)`, `mknod(8)`
 - * The attributes are set by the environment in which the file is created, e.g. `umask(1)`
 - delete.
 - * Delete an existing file
 - * Some systems may automatically delete a file that has not been used in n days
 - open.
 - * Establish a logical connection between process and file
 - * Fetch the attributes and list of disk addresses into main memory for rapid access during subsequent calls
 - * I/O devices attached instead of opened
 - * System call `open(2)`
 - close.
 - * Disconnects file from the current process
 - * file not accessible to the process after `close`
 - * I/O devices detached instead of closed
 - read.
 - * Transfer the logical record starting at current position in file to memory starting at `buf`
 - * `buf` known as input buffer
 - * Older systems have a `read_seq` to achieve the same effect and `read_direct` for random access files
 - write.
 - * Transfer the memory starting at `buf` to logical record starting at current position in file
 - * If current position is the end of file, file size may increase
 - * If current position is in the middle of file, some records may be overwritten and lost
 - * Older systems have a `write_seq` to achieve the same effect and `write_direct` for random access files
 - append.
 - * Restrictive form of `write`
 - seek.
 - * Primarily used for random access files
 - * Repositions the pointer to a specific place in file
 - * Unix system call `lseek(2)`
 - get_attributes.
 - * Get information about the file
 - * In Unix, system calls `stat(2)`, `fstat(2)`, `lstat(2)`
 - Device file resides on
 - File serial number, i-node number
 - File mode
 - Number of hard links to the file
 - uid/gid of owner

- Device identifier (for special files)
- Size of file
- Last time of access, modification, status change
- Preferred block size of file system
- Actual number of blocks allocated
- `set_attributes`.
 - * Set the attributes of a file, e.g., protection, access time
 - * Unix system calls: `utimes(2)`, `chmod(2)`
- `rename`.
 - * Change the name of a file
- Memory-mapped files
 - Map a file into the address space of a running process
 - First introduced in MULTICS
 - Given a file name and virtual address, map the file name to the virtual address
 - System internal tables are changed to make the actual file serve as the backing store in virtual memory for the mapped pages
 - Works best in a system with segmentation
 - * Each file can be mapped to its own segment
 - Unix system calls `mmap(2)` and `munmap(2)`
 - * `mmap` establishes a mapping between the process's address space at an address `pa` for `len` bytes to the memory object represented by `fd` at `off` for `len` bytes
 - * `munmap` removes the mappings for pages in the specified address range
 - * `mprotect(2)` sets/changes the protection of memory mapping
 - Problems with memory-mapped files
 - * System cannot tell the size of the file when it is ready to unmap
 - File (or data segment) size in Unix is increased by system calls `brk(2)`, `sbrk(2)`
 - System page size can be determined by `getpagesize(2)`
 - * What if a memory-mapped file is opened for conventional reading by another process
 - * File may be larger than a segment, or entire virtual address space

Directories

- Provided by the file system to keep track of files
- Hierarchical directory systems
 - Directory contains a number of entries – one for each file
 - Directory may keep the attributes of a file within itself, like a table, or may keep them elsewhere and access them through a pointer
 - In opening a file, the OS puts all the attributes in main memory for subsequent usage
 - Single directory for all the users
 - * Used in most primitive systems
 - * Can cause conflicts and confusion
 - * May not be appropriate for any large multiuser system
 - One directory per user

- * Eliminates name confusion across users
- * May not be satisfactory if users have many files
- Hierarchical directories
 - * Tree-like structure
 - * Root directory sits at the top of the tree; all directories spring out of the root
 - * Allows logical grouping of files
 - * Every process can have its own working directory to avoid affecting other processes
 - * Directories . and . .
 - * Used in most of the modern systems
- Information in directory entry
 - * File name
 - Symbolic file name
 - Only information kept in human readable form
 - * File type
 - Needed for those systems that support different file types
 - * Location
 - Pointer to the device and location of file on that device
 - * Size
 - Current size of the file
 - May also include the maximum possible size for the file
 - * Current position
 - Pointer to the current read/write position in the file
 - * Protection
 - Access control information
 - * Usage count
 - Number of processes currently using the file
 - * Time, date, and process identification
 - May be kept for creation, last modification, and last use
 - Useful for protection and usage monitoring
- Directory entry may use from 16 to over 1000 bytes for each file
- Size of directory may itself become very large
- Directory can be brought piecemeal into memory as needed
- Data structures for a directory
 - * Linked list
 - Requires linear search to find an entry
 - Simple to program but time consuming in execution
 - To create a new file, must search entire directory to avoid name duplication
 - To delete a file, search the directory and release the space allocated to it
 - Entry can be marked as unused or attached to a list of free entries
 - * Sorted list
 - Allows binary search and decrease average search time
 - Search algorithm is more complicated to program
 - May complicate creation and deletion as large amount of directory information may be moved to keep list sorted
 - * Hash table
 - Greatly improves directory search time
 - Insertion and deletion straightforward

- Problem because of fixed size of hash tables
- * Balanced tree
 - Fast search for a file
 - May require some effort for maintenance
- Path names
 - Convention to specify a file in the tree-like hierarchy of directories
 - Hierarchy starts at the directory /, known as the root directory
 - Made up of a list of directories crossed to reach the file, followed by the file name itself
 - Absolute path name
 - * Path from root directory to the file
 - * Always start at the root directory and is unique
 - * /usr/bin/X11/xdvi
 - Relative path name
 - * Used in conjunction with the concept of “current working directory”
 - * Specified relative to the current directory
 - * More convenient than the absolute form and achieves the same effect
- Unix allows arbitrary depth for the file tree
 - The name of each directory must be less than 256 characters
 - No pathname can be longer than 1023 characters
 - Files with pathname longer than 1023 characters can be accessed by `cd`ing to an intermediate directory
- Directory operations
 - `create_directory`.
 - * Create a directory and put entries for `.` and `..` in there
 - * `mkdir` command in Unix and MS-DOS
 - * `mkdir(2)` system call in Unix
 - `int mkdir(path, mode)`
 - Creates a directory with the name `path`
 - Mode mask of the new directory is initialized from `mode`
 - Set-GID bit of `mode` is ignored and is inherited from the parent directory
 - Directory owner-id is the process’s effective user-id
 - Group-id is the GID of the directory in which the new directory is created
 - `delete_directory`.
 - * Delete an existing directory
 - * Only empty directory can be deleted
 - * Directory containing just `.` and `..` is considered empty
 - * `rmdir` command in Unix and MS-DOS
 - In Unix, it is forbidden to remove the file `..`
 - * `rmdir(2)` system call in Unix
 - `int rmdir(path)`
 - Can remove a directory only if its link count is zero and no process has the directory open
 - If one or more processes have the directory open when the last link is removed, the `.` and `..` entries, if present, are removed before `rmdir()` returns and no new entries may be created in the directory, but the directory is not removed until all references to the directory have been closed

- `rmdir()` updates the time fields of the parent directory
- `open_directory.`
 - * Open a directory to read/write
 - * Used by many applications, e.g. `ls`
 - * C library function `opendir(3)`
 - `DIR *opendir (dirname)`
`char *dirname`
 - Open the directory named by `dirname` and associate a `directory` stream with it
 - Returns a pointer to identify the directory stream in subsequent operations
 - If directory cannot be accessed, a `null` pointer is returned
- `close_directory.`
 - * Close the directory that was opened for reading
 - * C library function `closedir(3)`
 - `int closedir (dirp)`
`DIR *dirp`
 - Closes the named directory stream and frees the structure associated with the `DIR` pointer
- `read_directory.`
 - * Returns the next entry in an open directory
 - * C library function `readdir(3)`
 - Returns a pointer to the next directory entry
 - Returns `null` upon reaching the end of the directory or detecting an invalid `seekdir(3)` operation
- `rename.`
 - * Operates the same way as renaming a file
- `link.`
 - * Allows creation of aliases (links) for the files/directories
 - * Same file can appear in multiple directories
 - * User command `ln(1)`
 - Creates hard or symbolic links to files
 - A file may have any number of links
 - Links do not affect other attributes of a file
 - Hard links
 - Can only be made to existing files
 - Cannot be made across file systems (disk partitions, mounted file systems)
 - To remove a file, all hard links to it must be removed
 - Symbolic links
 - Points to another named file
 - Can span file systems and point to directories
 - Removing the original file does not affect or alter the symbolic link itself
 - `cd` to a symbolic link puts you in the pointed-to location within the file system; changing to the parent of the symbolic link puts you in the parent of the original directory
 - Problem can be solved in `C-shell` by `pushd` and `popd`
 - * System call `link(2)`
 - Used to make a hard link to a file
 - Increments the link count of the file by one
 - * System call `symlink(2)`
 - Used to make a symbolic link to a file
- `unlink.`

- * Remove a directory entry
- * If the file being removed is present in one directory, it is removed from the file system
- * If the file being removed is present in multiple directories, only the path name specified is removed; others remain
- * User commands `rm(1)` and `rmdir(1)`
- * System call `unlink(2)`
 - Removes the directory entry
 - Decrements link count of the file referred to by that entry
 - If the entry is the last link to the file, and no process has the file open, all resources associated with the file are reclaimed
 - If the file is open in any process, actual resource reclamation is delayed until it is closed, even though the directory entry has disappeared

File system implementation

- File system manages files, allocating files space, administering free space, controlling access to files, and retrieving data for users
- Processes interact with the file system using a set of system calls
- File data is accessed using a buffering mechanism that regulates data flow between the kernel and secondary storage devices
 - Buffering mechanism interacts with the block I/O device drivers to initiate data transfer to and from kernel
 - Device drivers are kernel modules that control the operation of peripheral devices
 - It is the device driver that makes the kernel treat a device as a block-special device (or random access storage device)
 - * Yes, the Unix kernel can allow a tape drive to be treated as a random access storage device
 - File system also interacts directly with the character-special devices
- Implementing files
 - Support of primitives for manipulating files and directories
 - Make an association between disk blocks and files
 - Contiguous allocation
 - * Simplest allocation technique
 - * Simple to implement; files can be accessed by knowing the first block of the file on the disk
 - * Improves performance as the entire file can be read in one operation
 - * Problem 1 – File size may not be known in advance
 - * Problem 2 – Disk fragmentation; can be partially solved by compaction
 - Linked list allocation
 - * Files kept as a linked list of disk blocks
 - * First word of each block points to the next block
 - * Only the address of the first block appears in the directory entry
 - * No disk fragmentation
 - * Random access is extremely slow
 - * Data in a block is not a power of 2 (to accommodate the link to next block)
 - Linked list allocation using an index

Unix model of ownership

- Processes and files are owned by someone
- Files have both a user owner and a group owner, with the two ownerships decoupled
- Allows file protections and permissions to be organized as per local needs
- File ownership
 - Owner of a file has its primary control and can be superseded only by root
 - Every file has a primary owner and one or more group owners
 - Primary owner can modify the access privileges on the file for everyone (except root) by using the `chmod` command
 - Groups are defined in `/etc/group`
 - Owner of the file decides for group privileges to allow shared access among members of the group
 - The user owner of a file need not be a member of the group that owns it
 - Ownership of file can be changed by `chown/chgrp` commands
 - Owner and group are tracked by `uid` and `gid`, which are mapped to user names and group names by using the files `/etc/passwd` and `/etc/group`, respectively
- Ownership of new file
 - User ownership with the user who creates it
 - Ownership may be changed by owner of the file, or root
 - Group ownership dependent on BSD or SYS V
 - Group ownership in BSD
 - * Same as the group ownership of the directory in which the file is created
 - * Default on DEC Ultrix
 - Group ownership in SYS V
 - * Current group of the user who creates the file
 - * Default on most Unix systems
 - The default attributes can be changed by setting the SGID bit (set group-id bit) on the directory

Deconstructing the filesystem

- File tree
 - Composed of chunks called filesystems
 - * Filesystem consists of one directory and its subdirectories
 - * Attached to the file tree with the `mount` command
 - `mount` maps a directory within the tree at mount point
 - Previous contents of mount point are not accessible as long as a filesystem is mounted there
 - Preferable to declare mount points as empty directories
 - Example


```
mount /dev/sd0a /users
```

 installs the filesystem on device `/dev/sd0a` at the mount point `/users`
 - * Detached from a file tree with the `umount` command
 - A busy filesystem cannot be detached
 - Filesystem is busy if it contains open files, has a process entered into it using the `cd` command, or contains executables that are running
 - In some flavors (OSF/1 and BSDI), busy filesystems can be detached by using `umount -f` command

- * `lsdf` program
 - List of open files
 - Catalogs open file descriptors by process and filenames

File access in UNIX

- Controlled by a set of nine permission bits
- Three sets of three bits each (`rxw`) corresponding to user, group, and other
- An additional three bits affect the operation of execution of a program
- The twelve mode bits are stored together with four bits of file type information in one 16-bit word
- File type bits
 - Set at the time of file creation
 - Cannot be changed
- Mode bits
 - Control three types of file access – read, write, and execute
 - Meaning for file and directory

Access	File	Directory
read	View contents	Search contents (using <code>ls</code>)
write	Change contents	Change contents (add or delete files)
execute	Run the executable	Allowed to get into the directory

- Can be changed by the file owner or root using `chmod(1)`
- The `setuid` and `setgid` bits
 - Bits with octal values 4000 and 2000
 - Allow programs to access files and processes that may be otherwise off limits
 - On SunOS, `setgid` bit on a directory controls the group ownership of the newly created files within the directory
- Sticky bit
 - Bit with octal value 1000
 - Not important in current systems
 - If the sticky bit is set on a directory, you cannot delete or rename a files unless you are the owner of the directory, file, or executable
 - Makes directories like `/tmp` somewhat private
- Summary of the three bits

Code	Name	Meaning
t	Sticky bit	Keep executable in memory after exit
s	SUID	Set process user-id on execution
s	SGID	Set process group id on execution
l	File locking	Set mandatory locking in reads/writes

Under SunOS, turning on `sgid` access on a file and removing execute permission results in mandatory file locking

- If the `setuid` or sticky bit is set but the corresponding execute permission bit is not set, the `s` or `t` in the filename listing appear in the uppercase

- Assigning default permissions
 - Set by the `umask` command
 - Complement of the permissions allowed by default

Inodes

- Index node
- Structure to keep information about each file, including its attributes and location
- Contain about 40 separate pieces of information useful to the kernel
 - UID and GID
 - File type
 - File creation, access, and modification time
 - Inode modification time
 - Number of links to the file
 - Size of the file
 - Disk addresses, specifying or leading to the actual disk locations for disk blocks that make up the file
- Inode table
 - Created at the time of creation of filesystem (disk partition)
 - Always in the same position on the filesystem
 - Inode table size determines the maximum number of files, including directories, special files, and links, that can be stored into the filesystem
 - Typically, one inode for every 2 to 8 K of file storage
- Opening a file
 - Process refers to the file by name
 - Kernel parses the filename one component at a time, checking that the process has permission to search the directories in the path
 - Kernel eventually retrieves the inode for the file
 - Upon creation of a new file, kernel assigns it an unused inode
 - Inodes stored in the file system but kernel reads them into an in-core inode table when manipulating files

File system structure

- Four components
 1. Boot block
 - Occupies the beginning of the file system, typically the first sector
 - May contain the bootstrap code
 - Only one boot block needed for initialization but every filesystem has a possibly empty boot block
 2. Super block
 - Describes the state of the file system
 - * Size of the file system

- * Maximum number of files it can store
- * Location of the free space

3. Inode list

- List of inodes following the superblock
- Kernel references inodes by indexing into the inode list
- Root inode
 - * Inode by which the directory structure of the file system is accessible after execution of the `mount` system call

4. Data blocks

- Start at the end of the inode list
- Contain file data and administrative data
- An allocated data block can belong to one and only one file in the file system

Distributed file systems

- Allows file systems to be distributed on different physical machines while still accessible from any one of those machines
- Advantages include easier backups and management
- Exemplified by NFS (network file system, developed by Sun) and RFS (developed by AT&T) on UNIX and NTFS on Windows NT
- Must provide features for performance, security, and fault tolerance
- May provide support for cross-OS semantics
 - Almost impossible for a Unix system to understand other systems' protection mechanism
 - DOS has no protection system to speak of
 - VAX/VMS and Windows NT have access control lists which cannot be mapped easily onto every Unix file system
 - Other issues include mapping of user IDs, file names, record formats, and so on
- RFS
 - Supplied as a part of SYS V
 - Designed to provide Unix file system semantics across a network
 - Non-Unix files systems cannot be easily integrated into RFS
 - Called a *stateful*¹ system because it maintains the state of the open files at the file server where files reside
 - Efficient but can cause problems when a remote file server crashes causing a loss of state
- NFS
 - Stateless; no state is maintained for open files at the remote server
 - Less efficient as remote file server may have to reopen a file for each network I/O transaction
 - * Can be improved by caching
 - Recovery from a crash on remote server is transparent to client since the remote server has no state to lose
 - * Some Unix semantics are impossible to support if remote server does not save state
 - * Set of Unix semantics supported is relatively primitive enough that most non-Unix file systems can be accessed under NFS

¹A state is a unique configuration of information in a program or machine. In information processing, a state is a complete set of properties transmitted by an object to an observer via one or more channels. A protocol that relies upon state is *stateful*; one that does not rely upon state is *stateless*.

Windows NT file system

- Windows NT provides the option of different filesystems
 1. Pre-FAT
 - DOS 1.0 based on CP/M
 - No directories
 - Maximum of 1024 files in each filesystem
 - Filesystem reference was by drive letters (A:, B:, ...)
 2. FAT
 - File allocation table; evolved from DOS
 - Allows compatibility with previous Windows/DOS OSs
 - Filesystem considered to be a set of sectors
 - Each filesystem has two FATs – original and backup
 - Allocation table tracks allocation of sectors
 - * If file requires only one sector, relative FAT entry for that sector would be noted as end of file
 - * If file takes up more than one sector, initial FAT entry for the file points to next sector of file
 - Each file is a chain of entries in FAT referencing next sector
 - Original FAT references (FAT12) were 12-bit; allowing for up to 4K sectors (typical sector size 512 bytes) – maximum file size 2MB; file names limited to 8.3 convention
 - DOS 2.1 introduced the concept of directory, suing source code form SCO Xenix, and 16-bit FAT references – maximum file size 32MB
 - DOS 3.31 expanded some 16-bit features, allowing allocation of multiple sectors into a single block allocation unit
 - * Single sector per FAT entry was replaced by a *cluster* of sectors, increasing the filesystem size limit
 - * Each cluster could have a maximum of 64 sectors
 - * Raised the maximum filesize of FAT16 to 2GB
 3. NTFS – New Technology File System
 - Default file system on newer Windows machines
 - Intended for high-end applications
 - * Client/server applications such as file servers, compute servers, and database servers
 - * Resource-intensive engineering and scientific applications
 - * Network applications for large corporate systems
 - Provides a modern namespace and supports large-sized files and volumes, by modifying FAT
 - Borrows heavily from High Performance File System (HPFS) of OS/2, and VMS from DEC
 - Has a more comprehensive security mechanism compared to UNIX filesystem permission bits
 - NTFS files and directories have access control lists (ACLs) to control access at the user level (available in Solaris; `getfacl`, `setfacl`, and other ACL commands)
 - NTFS also associates specific rights with users and groups; the rights match actions that users may wish to perform such as backing up files and directories
 - * This enables an admin to distribute permissions as needed, without giving away the entire system
 - Key features
 - * Recovery from system crashes and disk failures
 - Transaction processing model for changes to file system
 - Each significant change is treated as an atomic action – either performed completely or not performed at all
 - Each transaction that was in the process at the time of a failure is subsequently backed out or brought to completion

- Redundant storage for critical file system data
- * Security
 - Windows object model to enforce security
 - Open file implemented as a file object with a security descriptor to define its security attributes
 - Security descriptor persisted as an attribute of each file on disk
- * Large disk and large files, compared to FAT
- * Multiple data streams
 - Makes a distinction between file data stream and file metadata, treating them as two separate objects
- * Journaling
 - Keeps a log of all changes made to files on the volumes
 - Programs may read the journal to identify which files have changed
- * Compression and encryption

Active Directory in Windows 2000

- Directory service used to store information about the network resources across a domain
 - Implementation closely resembles Netscape Directory Server
- Supported on NTFS-mounted file systems (not on FAT or FAT32)
- Hierarchical framework of *objects*
 - Objects can be resources (printers), services (email), and users (user accounts and groups)
- Employs a namespace closely resembling the internet's DNS (Domain Name System) namespace
- Directory service provider locations are stored in a DNS server and can be located by clients and other services using Lightweight Directory Access Protocol (LDAP) queries
- Service-provider location updates can be applied automatically with Dynamic DNS
- Updates to DNS can also be done manually independent of the Active Directory
- Hierarchy of *Organizational Units* and other objects within domains; also hierarchy of domains
 - Domain tree
 - * Hierarchy of domains constitutes a contiguous namespace
 - * Example: `hoare.cs.umsl.edu` and `laplace.cs.umsl.edu` are linked to `cs.umsl.edu`
 - Domain forest
 - * Domains do not constitute a contiguous namespace
 - * Example: `cs.umsl.edu` and `cnn.com`
 - * Active Directory can create a *virtual root* to cope with such a scenario
- Global catalog
 - Part of domain tree and domain forest hierarchy
 - Keeps track of all the objects in multiple domains without storing every attribute
 - Indexes the limited number of attributes to facilitate fast searches, avoiding the search in entire domain
- Security issues
 - Security and administration privileges can be assigned to users in a highly granular fashion within a domain (domain tree)

- The process is known as *delegation* (Microsoft term)
- Rights can be inherited
 - * A person with certain privileges in domain `cs.umsl.edu` will get the same privileges on all machines in the domain tree, for example on `hoare.cs.umsl.edu`
- Trust between domains
 - * In Windows NT, trust between domains is managed manually and is one of the problematic issues for multiple domains
 - * Active directory automatically establishes bidirectional trust relationships between domain and supports transitive trust relationships
- Integration with Internet
 - Active directory namespace is almost identical to Internet namespace, using a `host.com` structure
 - Windows 2000 can use DNS as service locator for the directory
 - Exception to the rule
 - * Internet permits the existence of duplicate names within a domain, for example, `laplace.cs.umsl.edu` and `laplace.phys.umsl.edu`
 - * W2K would not permit the reuse of name `laplace` within a domain because an account named `sanjiv@umsl.edu` can be assigned to only one of these names