Operating System Concepts
Ch. 10: File System Interface

Silberschatz, Galvin & Gagne

Universiteit Leiden
The Netherlands

File System Interface

- It is clear we want to store data, in fact *lots* of it.
- We are given a disk which can hold several terabytes of data.
  - In fact a terabyte-long byte string.
- How do we organize/structure this storage?
  - On the disk, using a specific format (next chapter)
  - Common interfaces for the users.
    - What choices can we make here?

Filing Cabinet

Ikea ERIK

Directory Organization

- Filing cabinet.
- Drawers ↔ Disks / volumes
  - Folders ↔ Directories
    - Sheets of paper ↔ Files

File

- A file is a contiguous logical address space.
- Can be seen as a sequence of
  - Records
  - Characters (historically: there have been systems with 5-bit or 6-bit characters)
  - Bytes (these days very common)
- The kernel provides this logical address space, the contents are up to user space.
  - Different file formats: binary, textual, XML, etc.
  - Different contents: source code, text, executable code, formatted documents, etc.
  - These different formats are all up to the users, the kernel does not care about this.

File Metadata

- Attributes are commonly associated with the actual data (the content) of a file.
  - "Data about the data": *metadata*.
- Think of:
  - File name,
  - file size,
  - owner,
  - access permissions,
  - creation, modification, access time,
- But also more file-system specific data:
  - Unique identifier within the file system
  - Where the "actual data" is stored on the disk (see later).

File System

- The file's contents and its metadata need to be stored somehow.
- This can be done in different formats, or as we say using different *file systems*.
  - Well-known file systems: FAT, NTFS, HFS, ext4, xfs, zfs.
  - These are general-purpose file systems. Some systems also implement special-purpose file systems, examples: tmpfs (fast in memory file system), procfs (file system to obtain information about running processes), sysfs.
- The file system also encodes the directory structure.
  - File attributes are sometimes stored as part of the directory structure.
- Attributes of the file system itself are stored in a superblock at a specific location within the file system.

Common File Operations

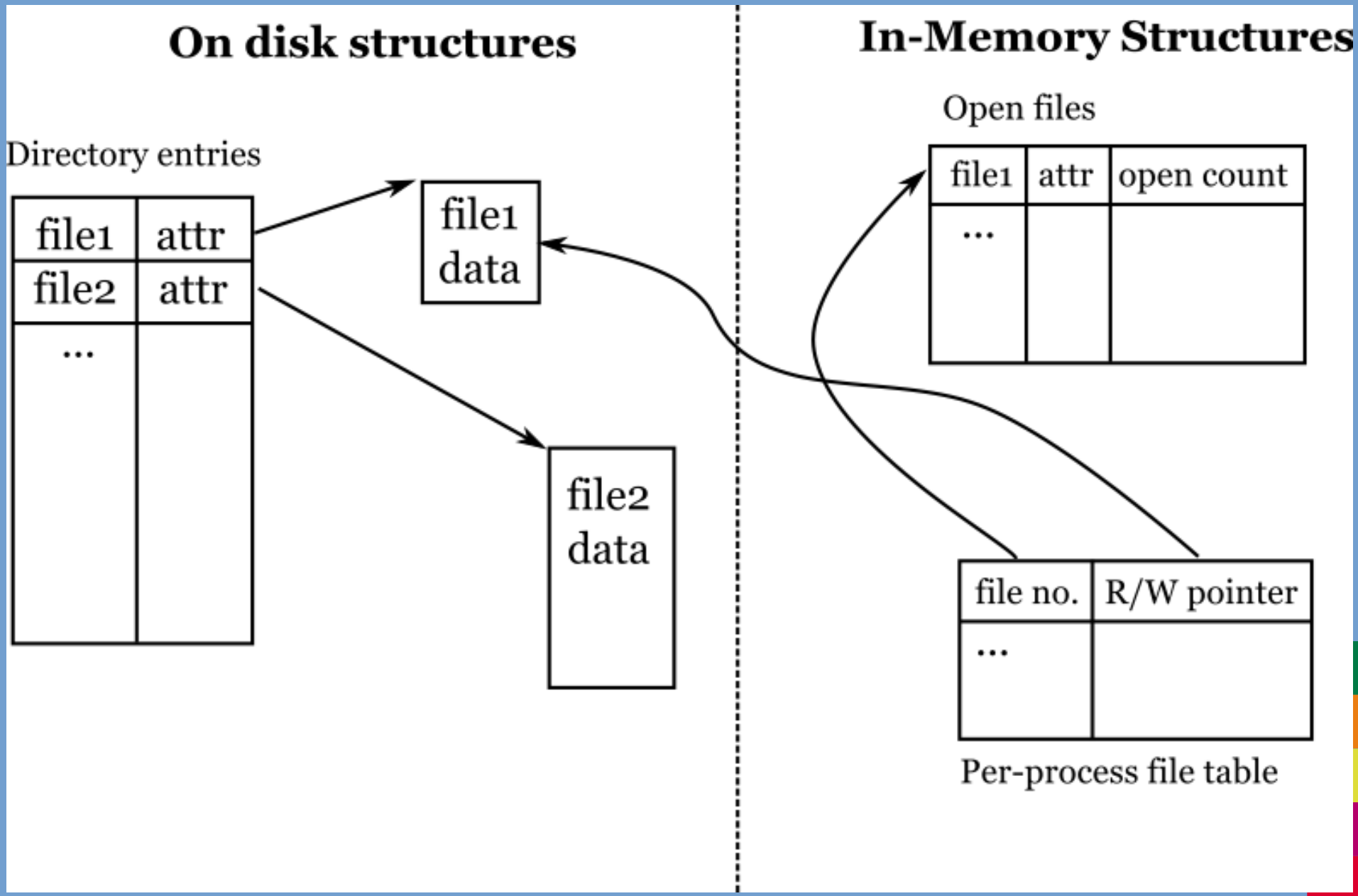
- A file can be seen as an abstract data type on which different operations can be performed:
  - *read data* (usually from the point where the read/write pointer is located)
  - *write data* (usually from the point where the read/write pointer is located)
  - *seek* (move the read/write pointer to a specific position)
  - *create*
  - *remove*
  - *truncate* (remove file contents after the read/write pointer)

Common File Operations (2)

- Before one can operate on a file, it needs to be opened:

  open(filename)

- In fact, opening a file is an operation on the directory structure.
  - The directory structure is searched for the given filename.
  - If successful, the contents of the directory entries & attributes are copied to an in-memory structure of the OS.
  - A *global table of open files* is maintained, this includes *file open count*.
  - Per process a *open file table* is also maintained, this includes the *read/write pointer*.
- When done, the file needs to be closed.
  - Changes to attributes in in-memory structure are written to disk if not already done; in-memory structure is released.


On disk structures
In-Memory Structures
Directory entries
file1
attr
file2
attr
...
file1
data
file2
data
Open files
file1
attr
open count
...
file no.
R/W pointer
...
Per-process file table

File Locking

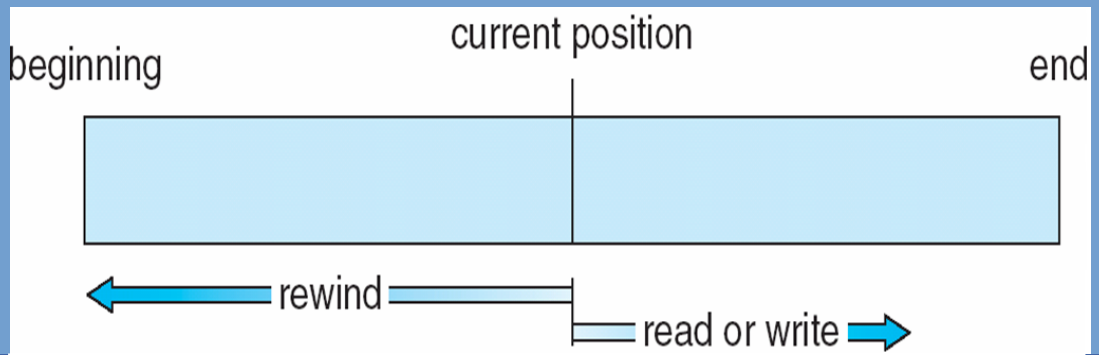
- Another file operating is locking. This is useful when more than one process needs to access a file at the same time.
  - What will happen if two processes start to manipulate the same file at the same time?
- Types of locks:
  - *Shared lock* – all processes that acquire the lock may manipulate the file.
  - *Exclusive lock* – only a single process is granted exclusive access.
- Some systems implement *mandatory* locking and some *advisory* locking.
  - **Mandatory** – the OS kernel enforces the state of the locks, access to a file that has been exclusively locked will be denied.
  - **Advisory** – not enforced by kernel, processes need to request state of the locks and decide what to do.

File content structure

- The structure of the content of the file is usually up to the users.
  - Binary files in certain structure, line-based files, comma separated value fields, etc.
  - Notable exception: files that the OS kernel needs to read, such as program files with executable code (Linux: “ELF format”).
- File extensions are suffixes of filenames often used to indicate the format of the file's content.
  - “.png” suffix -> file in PNG image format
  - Some systems always deduce file type on extension, others don't care about the extension at all (such as Linux systems).
  - The latter may use *file sniffing* which detects file type by looking for specific signature in the file.

Accessing Files

- The most common way to access files is to access these as sequential-access files.
  - Kind of inspired by tape drives used in old systems ...
- There's a read/write pointer that is moved by subsequent read/write calls
  - `read(file_handle, n_bytes, buffer)`
    read the next n_bytes from the file
  - `write(file_handle, n_bytes, buffer)`
    write the next n_bytes from buffer to file



Source: Silberschatz et al., Operating System Concepts, 9th Edition

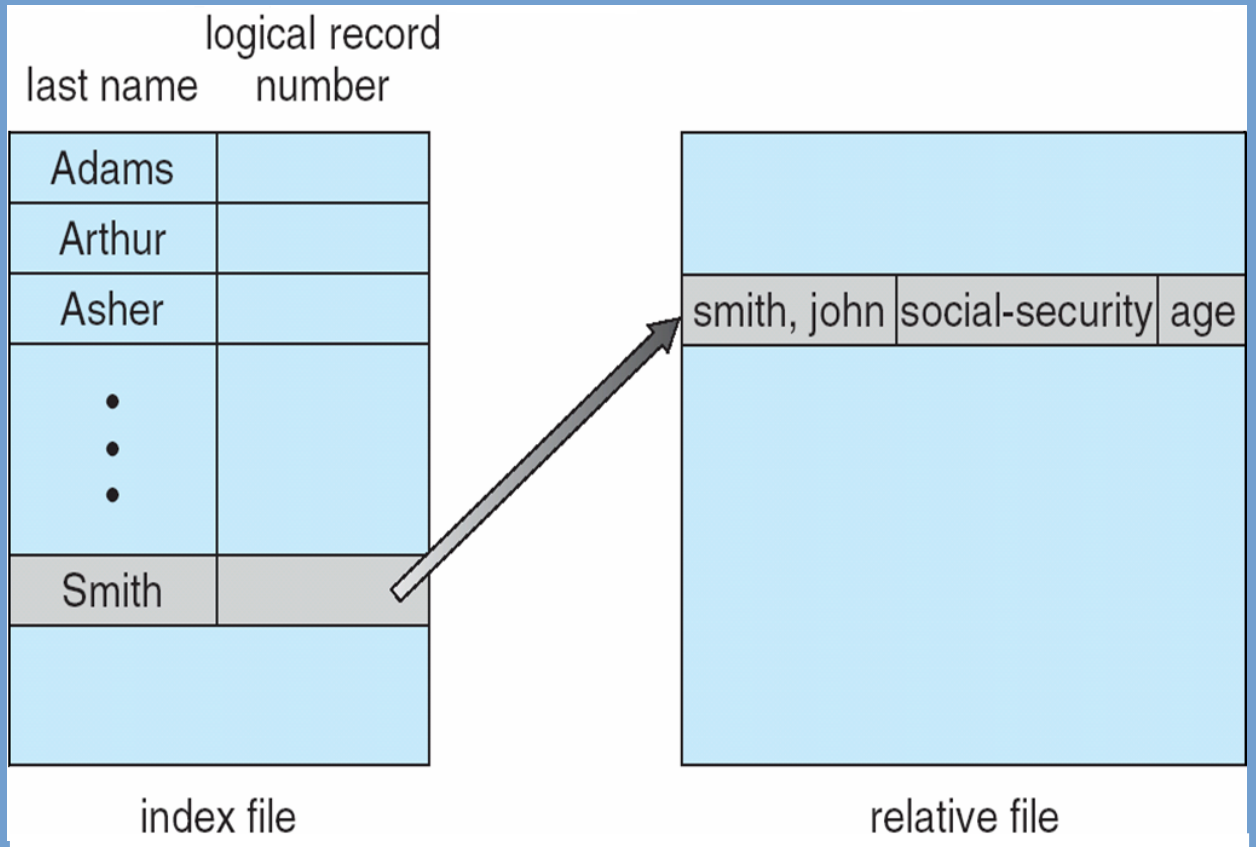
Accessing Files (2)

- In direct-access mode, files are structured as sequences of fixed-length records.
- In every call, the number of the record to access is specified.
  - `read(n)` - read record n
- The record numbers correspond to logical addresses in the contiguous address space of the file. The OS decides in what order to store the records on disk (so this can be in a different order).
- Given read, write and seek calls, you can implement direct-access on top of the sequential-access method.

Accessing Files (3)

- Nothing restricts the OS to also implement and provide other means of accessing files.
- For example, some older systems supported the creation and maintenance of indexes on files (think database indexes).
  - Textbook mentions IBM indexed sequential-access method (ISAM), which was fully implemented in the OS.
- VMS implemented relative files (direct-access method) and index files within the operating system.

Accessing Files (4)

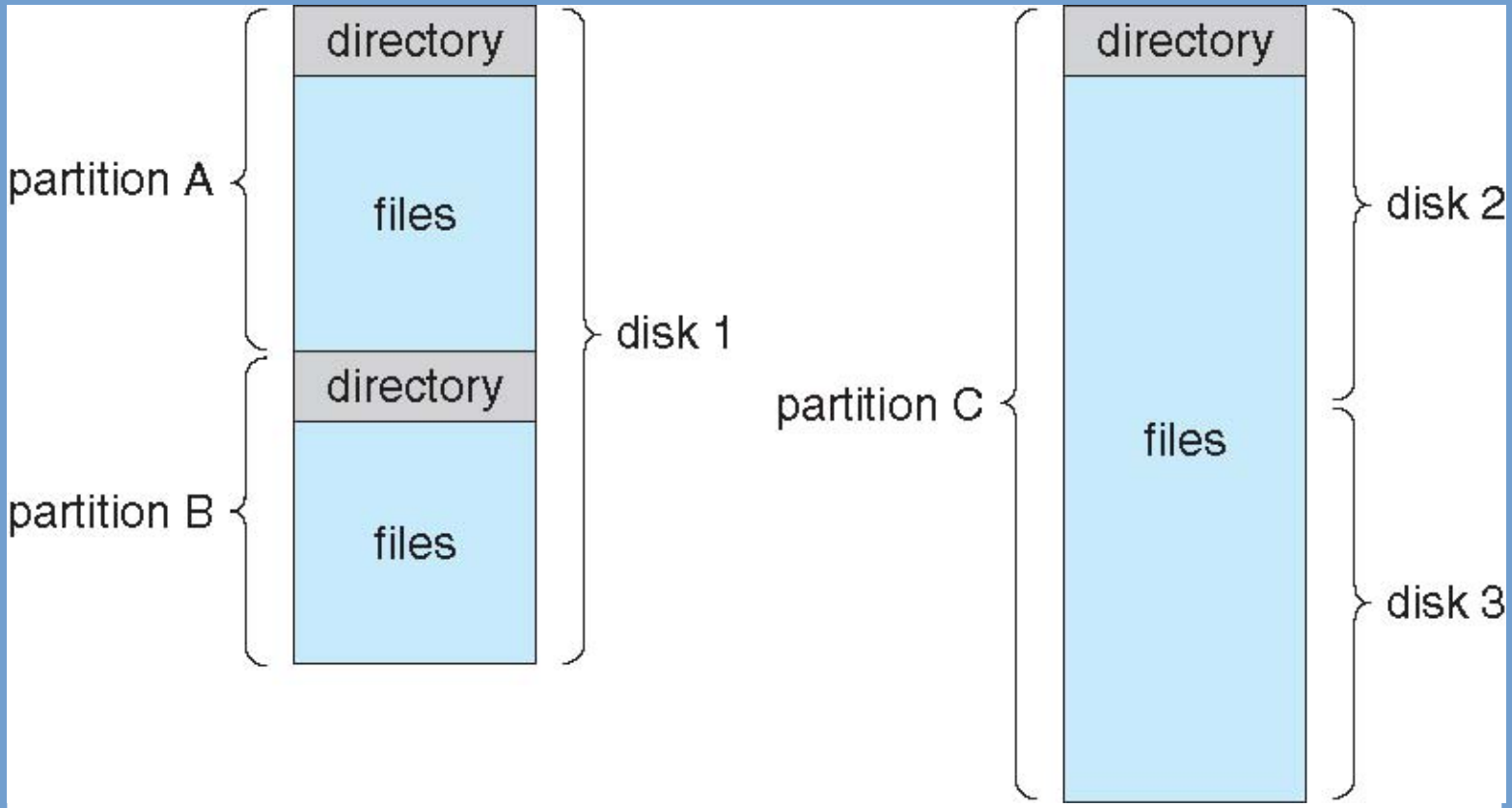


Source: Silberschatz et al., Operating System Concepts, 9th Edition

Putting File Systems on Disks

- Next: how / where are file systems stored?
- A file system needs to be stored in a contiguous area on the disk.
- To create these areas, a disk is subdivided into *partitions*.
  - Partitions present on a disk are recorded into a *partition table* located at the start of a disk.
  - We refer to a partition populated with a file system as a *volume*.
- We can also bundle multiple disks to form a single disk, such that a file system can span multiple disks.
  - Often done using RAID technology, see the chapter on mass storage systems.

Disk Partitions



Source: Silberschatz et al., Operating System Concepts, 9th Edition

Disk Partitions (2)

- Disk partitions are not very flexible.
  - Say we want to enlarge the file system in partition A in the example of the previous slide.
  - We need a contiguous area, so we would have to move partition B. Expensive operation!
- One solution is a logical volume manager, such as Linux LVM.
  - Disks are physical volumes, which are placed in a volume group.
  - Within a volume group, logical volumes can be created. Logical volumes may span multiple disks and do not have to be contiguously stored on the physical volumes.
  - The volume manager is responsible for maintaining the mapping from logical to physical.
  - The logical volume is always contiguous and as such suitable to store a file system on.

Directories

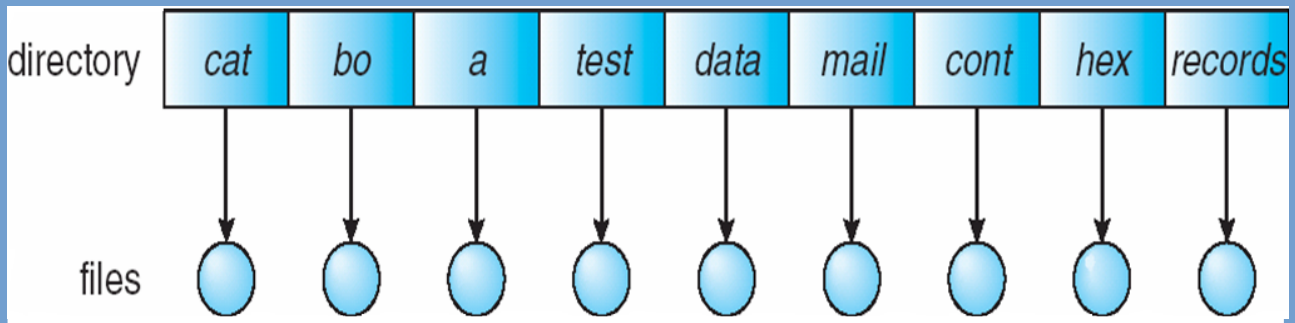
- Files are usually organized into directories.
  - A directory needs to record what files are contained in that directory. Sometimes, file attributes are also stored in the directory.
  - Needless to say, directories also need to be stored within the file system on disk.
- Typical directory operations:
  - Create / remove directories (mkdir, rmdir).
  - List files in a directory (opendir, readdir, closedir).
  - Create / delete file (and associated directory entry).
  - Search for a file (for open file system call).

Directories (2)

- How to organized directories? This can be done in several ways.
- Considerations:
  - *Grouping*: ability to group files (as opposed to putting everything in a single directory).
  - *Efficiency*: ability to quickly locate a file in a directory.
  - *Naming*: allow a filename to not be unique within the file system, for instance a filename only has to be unique within the enclosing directory. This allows two users, with different directories, to give the same name to different files. Can be seen as consideration of convenience.
- We'll now see an evolution of directory structures, from simple to more complex tree structures.

Single-Level Directory

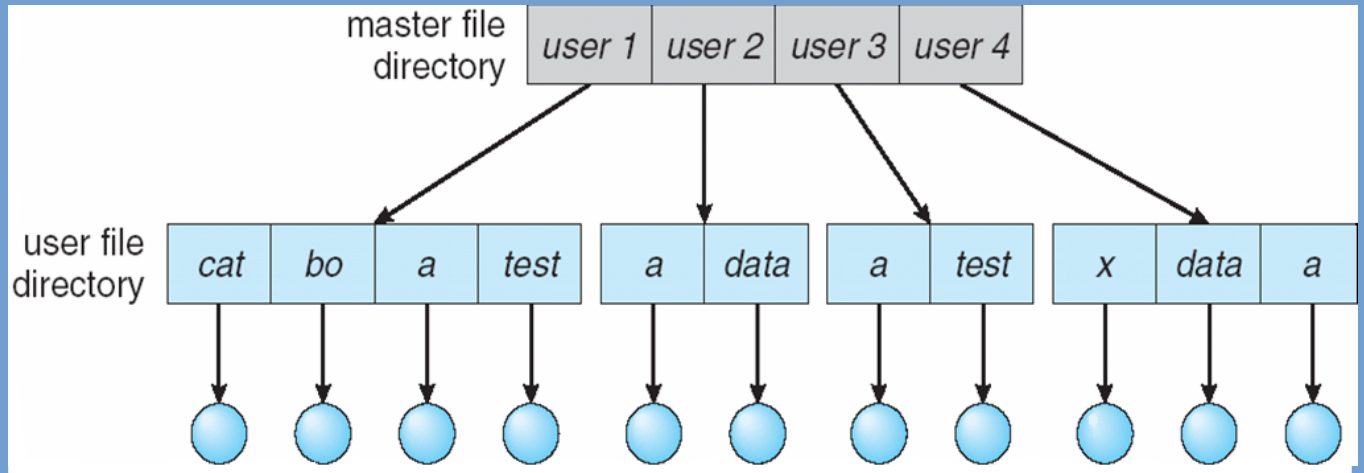
- Early model, a single directory that was used by all users.
- Has grouping problem (it does not allow groups) and naming problem (a name must be unique in the file system).



Source: Silberschatz et al., Operating System Concepts, 9th Edition

Two-Level Directory

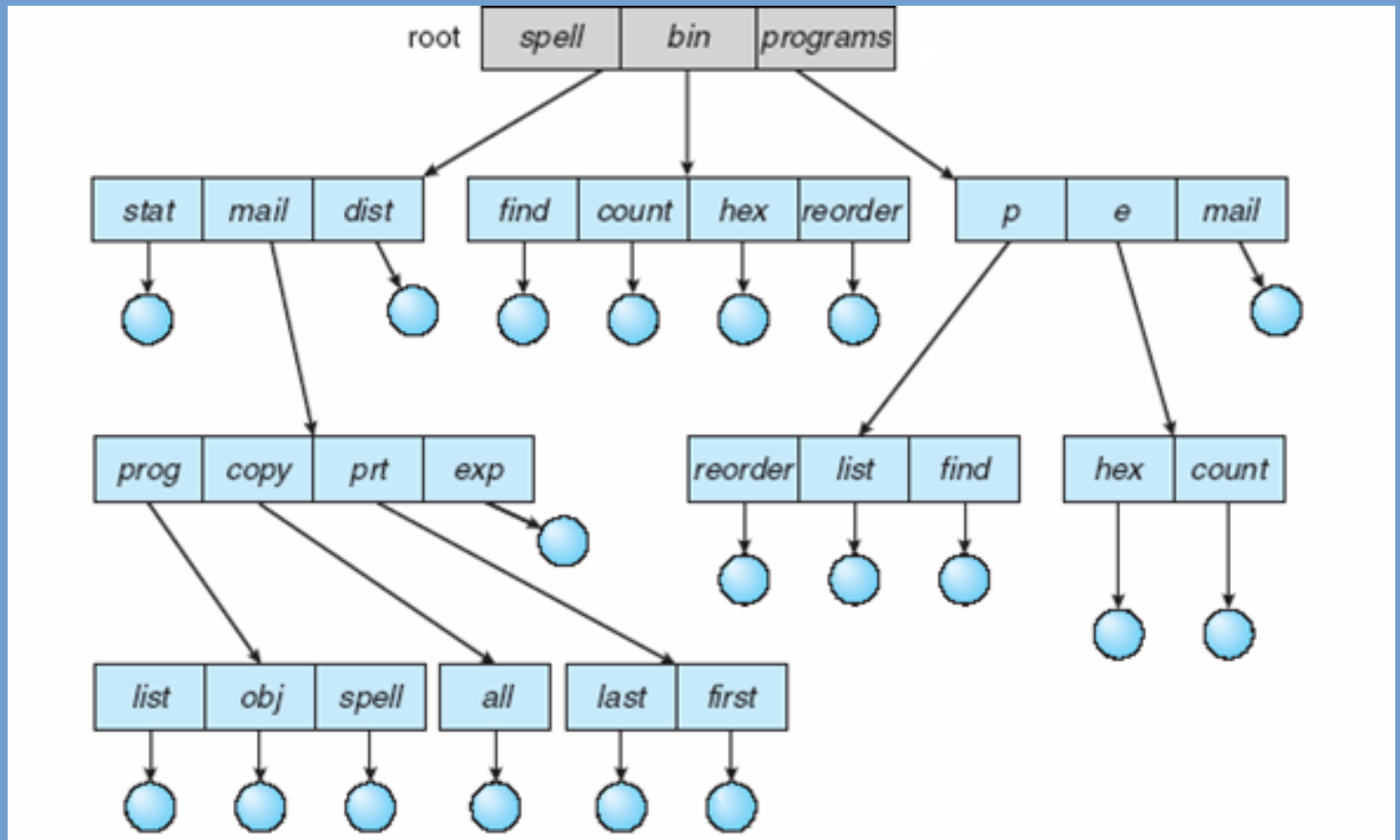
- Fix the naming problem for individual users by giving every user a separate directory.
- Still no grouping capability.
- We now need to identify files by path name, first mention the user, then the file name.



Source: Silberschatz et al., Operating System Concepts, 9th Edition

Tree-Structured Directories

- Allow for subdirectories, fixing the grouping problem.
- More efficient searching, as directories are smaller.
  - (Searching in directories is often a linear scan).



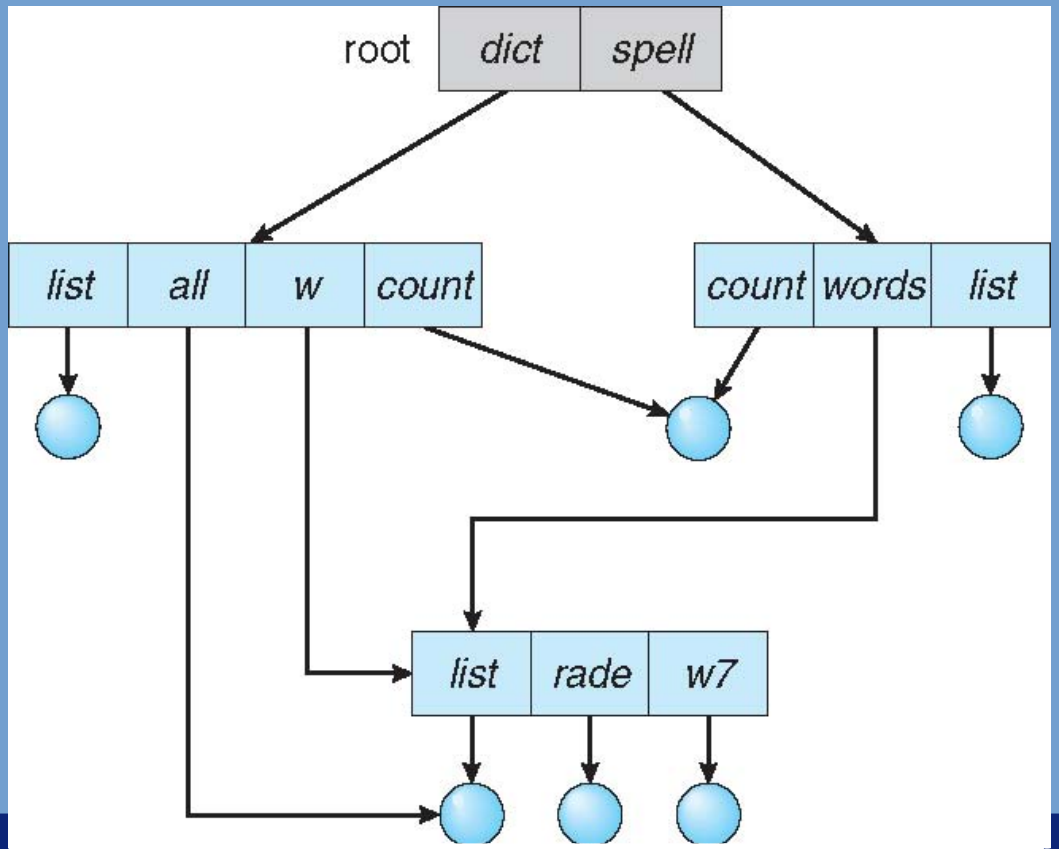
Source: Silberschatz et al., Operating System Concepts, 9th Edition

Tree-Structured Directories (2)

- File paths are now rather important, a path indicated a path through the directory tree.
  - We start at the root of the tree, often designated "/".
  - We add directory components until we have reached the target directory.
  - Finally, we add the name of the file in that directory.
- A file name that starts at "/" is an *absolute* path.
- Otherwise, the file name is a *relative path*, relative to the *current working directory* (a process property).
  - ".." is used to go "up" a directory.
  - Assume we are in `/spell/mail` we may write `../../programs/p/list`.

Acyclic-Graph Directories

- Acyclic-Graph Directories allow files and directories to be shared.
  - This means there can be more references to the same file or directory.
  - Put differently: a file or directory can have multiple names.



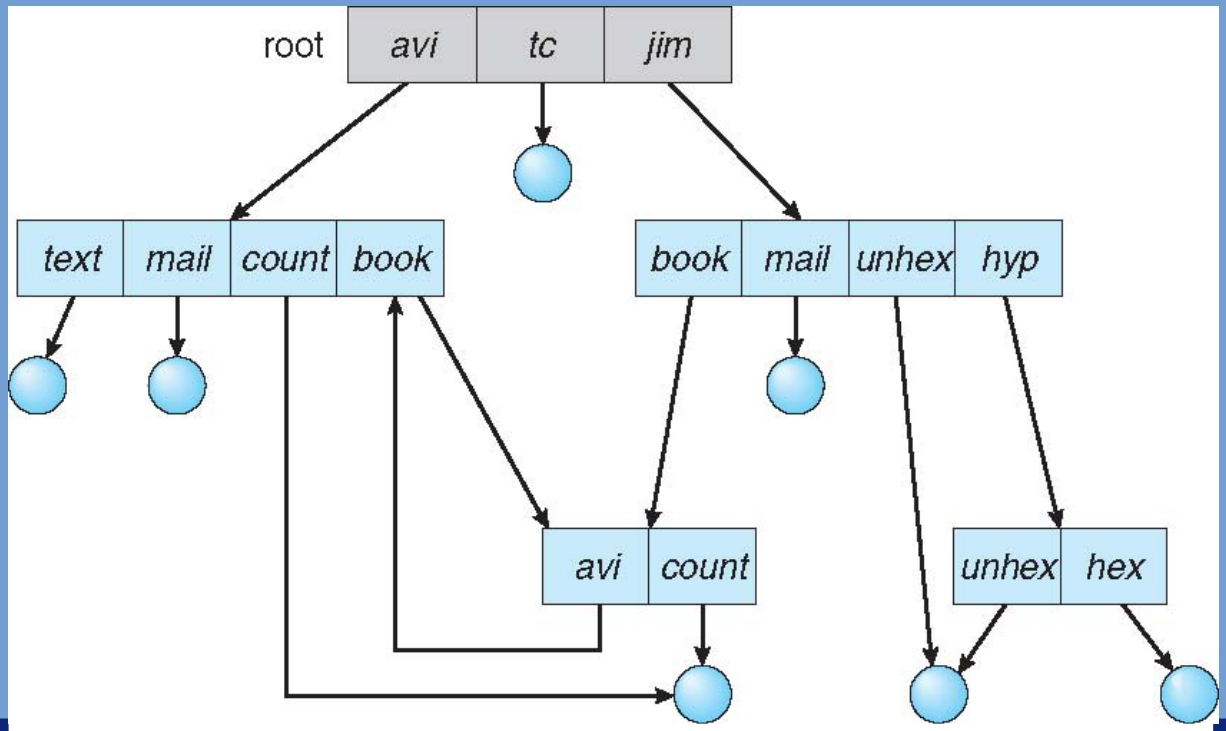
Source: Silberschatz et al., Operating System Concepts, 9th Edition

Link types

- Symbolic link
  - New directory entry type.
  - Contains file name (symbolic) of link target.
  - Following this pointer: “resolving the link”.
  - If link target is deleted, the symbolic link will be broken. It is not deleted nor “repaired”.
- Hard link
  - Duplicated directory entry.
  - Directory entries point at “inodes”, which store the actual file attributes.
  - In the “inode” a reference count is kept. A file is only deleted if the reference count reaches zero.

General Graph Directory

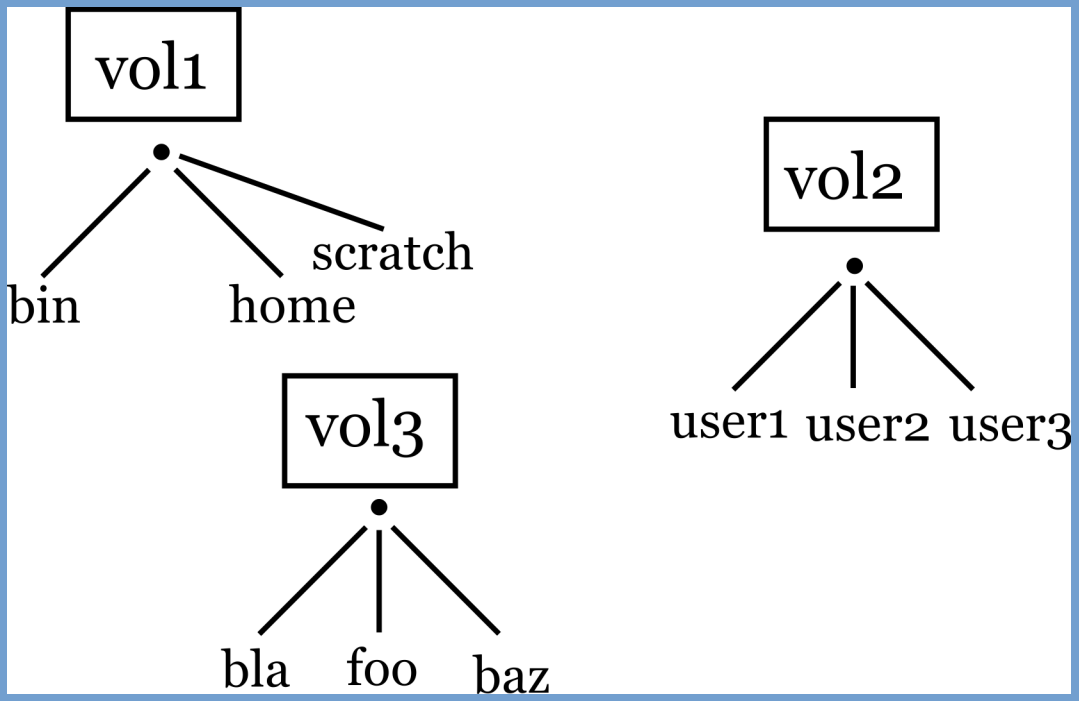
- If we allow for links (back) to a directory higher in the hierarchy, cycles will come into play.
  - Note: this problem does not exist with links to files.
- The directory structure is now a general graph.
- Either disallow cycles, or make file system tools resilient to it.



Source: Silberschatz et al., Operating System Concepts, 9th Edition

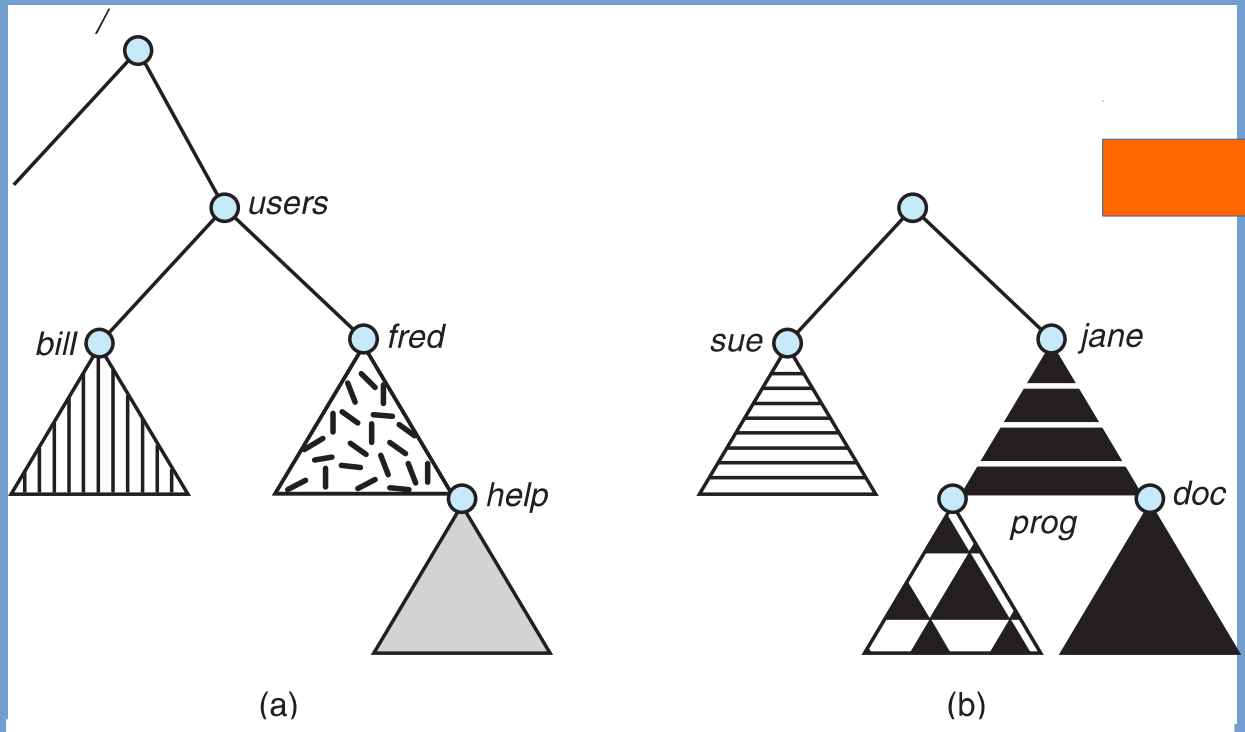
File System Mounting

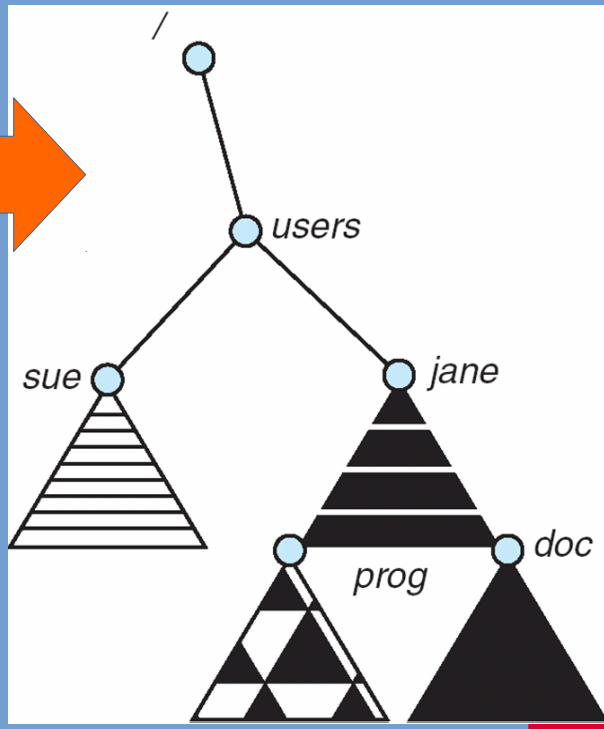
- Assume we have three volumes, how do we access the file systems?
- DOS/Windows solution: drive letters!



File System Mounting (2)

- UNIX solution: file system *mounting*. We must mount a file system at a *mount point* (typically a placeholder directory) before it can be accessed.
- In the example the file system on the right (b) is mounted at /users. Now this will shadow the content of /users on the left (not always what you want).

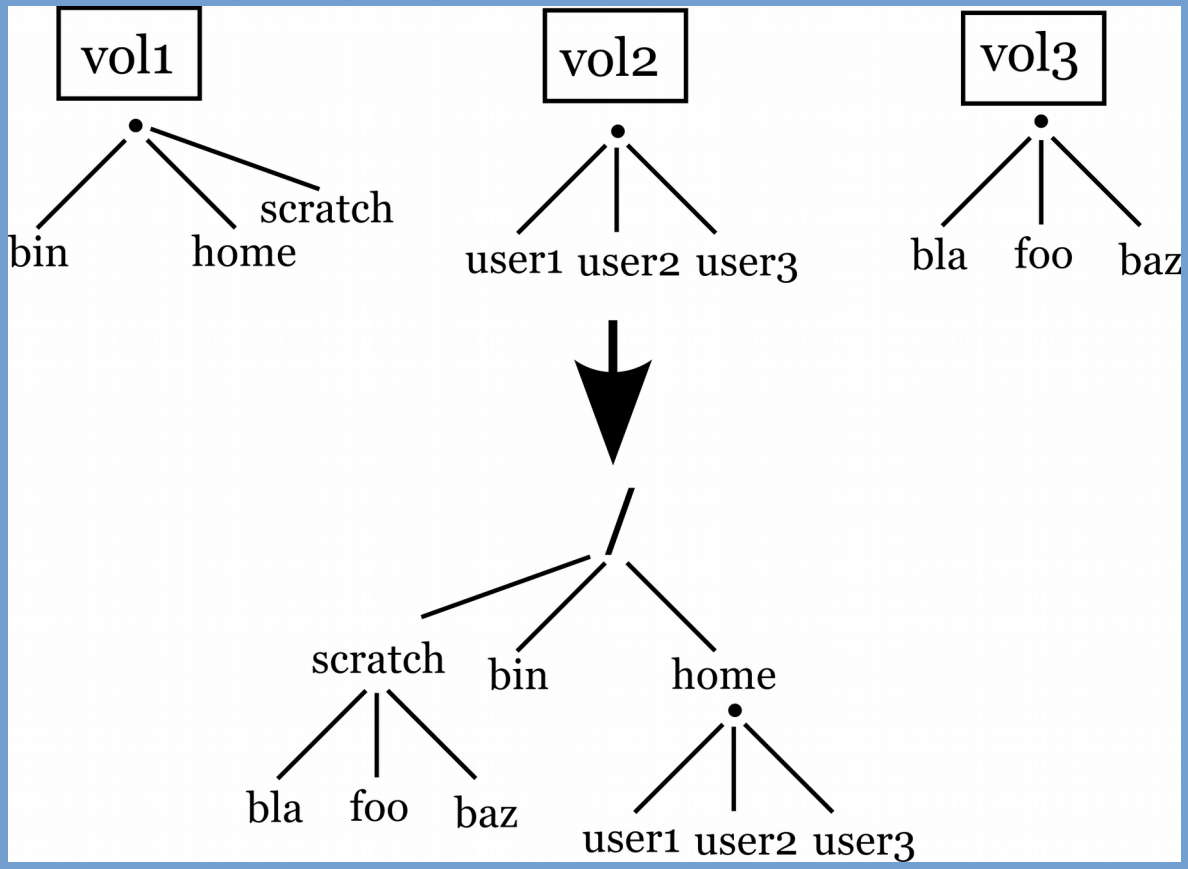




Source: Silberschatz et al., Operating System Concepts, 9th Edition

File System Mounting

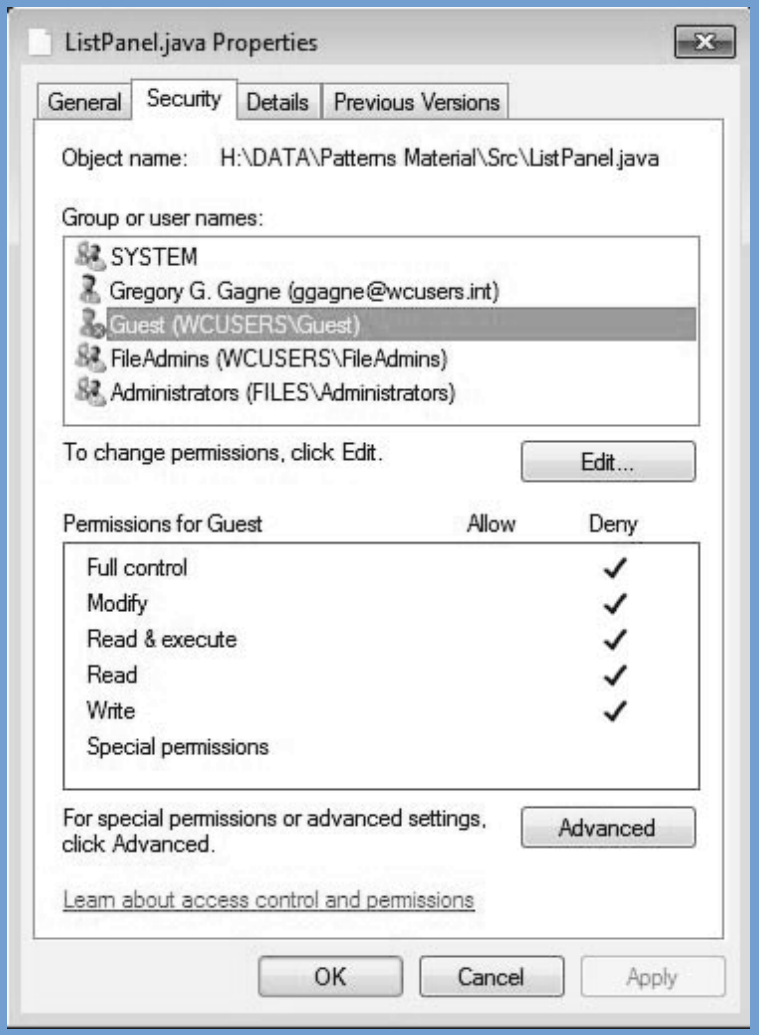
- Back to our concrete example:



Protecting Files

- On multi-user systems, we want to protect our files/directories from unauthorized access.
- To this end, a file has a designated owner and sometimes also a designated group. Numeric user ID and group ID are stored in the file's attributes.
- Examples of permissions:
  - On UNIX systems, permissions are set for owner, group, and “all others”. For each of these the allowed bits are set: read, write, execute (abbreviated rwx).
  - Modern UNIX-like implementations also support more elaborate access control lists (ACLs).
  - Windows systems distinguish some more permissions such as modify and full control and have extensive support for ACLs.

Example Windows Access Control



Source: Silberschatz et al., Operating System Concepts, 9th Edition

Accessing Files on Remote Systems

- Several schemes are used to access files on remote systems.
  - Manual and explicit downloads from servers: FTP and HTTP.
  - Cloud storage systems (DropBox, Google Drive).
  - Client-server network file systems, such as UNIX NFS and Windows CIFS.
    - NFS allows a remote file system to be mounted at a local mount point. After that it can be accessed like a local directory.
    - Important: user IDs are stored in the file attributes. This implies that the user and group IDs must be the same on clients and the server.
    - To accomplish this, distributed naming services such as LDAP, ActiveDirectory, Novell NDS (and in the past NIS) are used.

End of Chapter 10.