FILE-SYSTEM IMPLEMENTATION

Overview

- **File systems** provide efficient and convenient access to the disk by allowing data to be stored, located, and retrieved easily
- A **file control block (FCB)** (an **inode** in UNIX file systems) contains information about the file, including ownership, permissions, and location of the file contents

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

Figure 12.2 A typical file-control block.

- UNIX uses the **UNIX file system (UFS)**, Windows supports disk file-system formats of **FAT (File Allocation Table) and NTFS (New Technology File System).** Linux file system is known as the **extended file system**
- On disk, the file system may contain information about how to boot an OS stored there, the total number of blocks, the number and location of free blocks, the directory structure, and individual files.

- A **boot control block** (per volume) can contain information needed by the system to boot an OS from that volume.
- If the disk does not contain an OS, this block can be empty. It is typically the first block of a volume.
- In UFS, it is called the **boot block**. In NTFS, it is the **partition boot sector**.
- A **volume control block** (per volume) contains volume (or partition) details, such as the number of blocks in the partition, the size of the blocks, a free-block count and free-block pointers, and a free-FCB count and FCB pointers.
- In UFS, this is called a **superblock**. In NTFS, it is stored in the **master file table**.
- A directory structure (per file system) is used to organize the files. In UFS, this includes file names and associated inode numbers. In NTFS, it is stored in the master file table.
- The data are loaded at mount time, updated during file-system operations, and discarded at dismount.
- Several types of structures may be included.
    - An in-memory **mount table** contains information about each mounted volume.
    - An in-memory directory-structure cache holds the directory information of recently accessed directories.

- The **system-wide open-file table** contains a copy of the FCB of each open file, as well as other information.
- The **per-process open-file table** contains a pointer to the appropriate entry in the system-wide open-file table, as well as other information.
- Buffers hold file-system blocks when they are being read from disk or written to disk.

- To create a new file, an application program calls the logical file system. It allocates a new FCB
- The system then reads the appropriate directory into memory, updates it with the new file name and FCB, and writes it back to the disk.
- Now that a file has been created, it can be used for I/O. First it must be opened. The open() call passes a file name to the logical file system.
- The open() system call first searches the system-wide open-file table to see if the file is already in use by another process. If it is, a per-process open-file table entry is created pointing to the existing system-wide open-file table.
- If the file is not already open, the directory structure is searched for the given file name.
- Once the file is found, the FCB is copied into a system-wide open-file table in memory.

- Next, an entry is made in the per-process open-file table, with a pointer to the entry in the system-wide open-file table
- The open() call returns a pointer to the appropriate entry in the per-process file-system table. All file operations are then performed via this pointer.
- The name given to the entry varies. UNIX systems refer to it as a **file descriptor**; Windows refers to it as a **file handle**.
- When a process closes the file, the per-process table entry is removed, and the system-wide entry's open count is decremented.
- When all users that have opened the file close it, any updated metadata is copied back to the disk-based directory structure, and the system-wide open-file table entry is removed

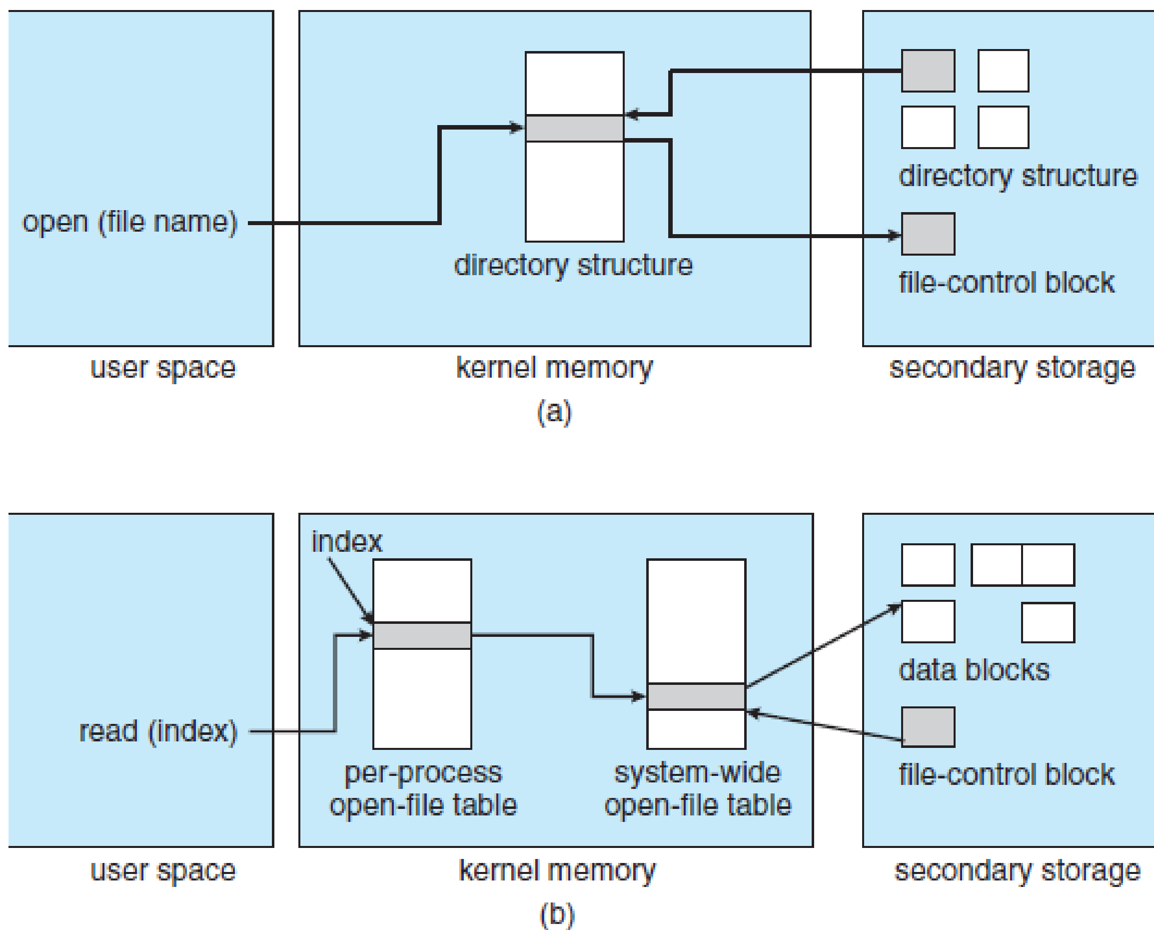

Figure 12.3 In-memory file-system structures. (a) File open. (b) File read.

Partitions and Mounting

- A disk can be sliced into multiple partitions
- Each partition can be either "raw," containing no file system, or "cooked," containing a file system.
- **Raw disk** is used where no file system is appropriate. Eg: UNIX swap space can use a raw partition since it uses its own format on disk and does not use a file system. Likewise, some databases use raw disk and format the data to suit their needs
- Boot information can be stored in a separate partition. Again, it has its own format, because at boot time the

system does not have the file-system code loaded and therefore cannot interpret the file-system format.

- Rather, boot information is usually a sequential series of blocks, loaded as an image into memory. Execution of the image starts at a predefined location, such as the first byte. This **boot loader** in turn knows enough about the file-system structure to be able to find and load the kernel and start it executing.
- It can contain more than the instructions for how to boot a specific OS. For instance, many systems can be **dual-booted**, allowing us to install multiple OS on a single system.
- How does the system know which one to boot? A boot loader that understands multiple file systems and multiple OS can occupy the boot space.
- Once loaded, it can boot one of the OS available on the disk. The disk can have multiple partitions, each containing a different type of file system and a different OS.
- The **root partition**, which contains the OS kernel and sometimes other system files, is mounted at boot time.
- Other volumes can be automatically mounted at boot or manually mounted later, depending on the OS. As part of a successful mount operation, the OS verifies that the device contains a valid file system.
- It does so by asking the device driver to read the device directory and verifying that the directory has the expected

format. If the format is invalid, the partition must have its consistency checked and possibly corrected, either with or without user intervention.

- Finally, the OS notes in its in-memory mount table that a file system is mounted, along with the type of the file system. The details of this function depend on the OS
- Microsoft Windows–based systems mount each volume in a separate name space, denoted by a letter and a colon. Eg: F:
- On UNIX, file systems can be mounted at any directory. Mounting is implemented by setting a flag in the in-memory copy of the inode for that directory. The flag indicates that the directory is a mount point.

Virtual File Systems (VFS)

- Modern OS must concurrently support multiple types of file systems
- Most OS, including UNIX, use object-oriented techniques to simplify, organize, and modularize the implementation.
- The use of these methods allows very dissimilar file-system types to be implemented within the same structure
- Users can access files contained within multiple file systems on the local disk or even on file systems available across the network.
- Data structures and procedures are used to isolate the basic system call functionality from the implementation details.

Thus, the **file-system implementation consists of three major layers**

- The first layer is the file-system interface. The second layer is called the **virtual file system (VFS)** layer. Third layer will be the actual file systems

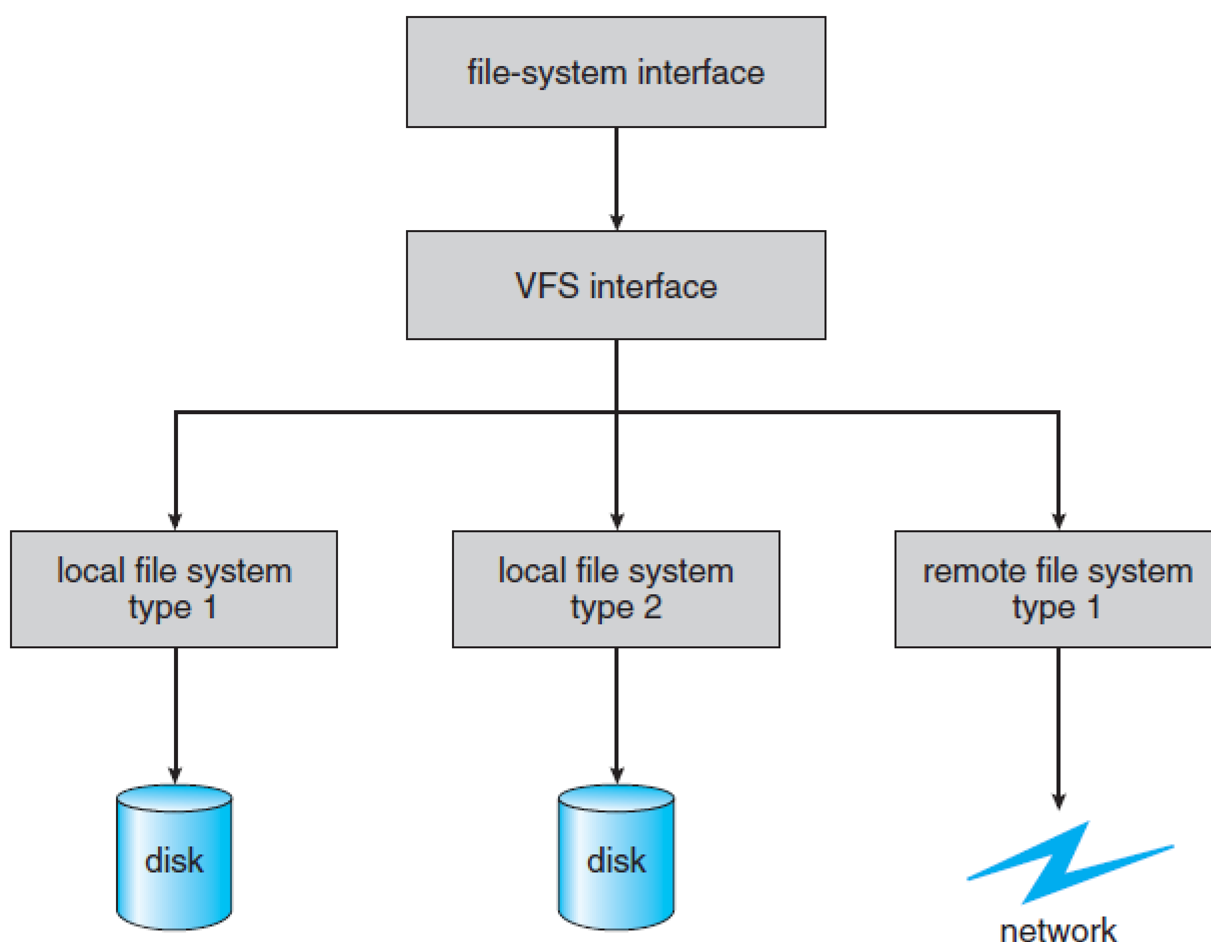

Figure 12.4 Schematic view of a virtual file system.

- The VFS layer serves two important functions:

  1. It separates file-system-generic operations from their implementation by defining a clean VFS interface. Several implementations for the VFS interface may coexist on the

same machine, allowing transparent access to different types of file systems mounted locally.

2. It provides a mechanism for uniquely representing a file throughout a network. The VFS is based on a file-representation structure, called a **vnode** that contains a numerical designator for a network-wide unique file. This network-wide uniqueness is required for support of network file systems.

- Thus, the VFS distinguishes local files from remote ones, and local files are further distinguished according to their file-system types.
- Consider the VFS architecture in Linux. The four main object types defined by the Linux VFS are:
  - The **inode object**, which represents an individual file
  - The **file object**, which represents an open file
  - The **superblock object**, which represents an entire file system
  - The **dentry object**, which represents an individual directory entry
- For each of these four object types, the VFS defines a set of operations that may be implemented.
- Every object of one of these types contains a pointer to a function table. The function table lists the addresses of the actual functions that implement the defined operations for that particular object

DIRECTORY IMPLEMENTATION

- The selection of directory-allocation and directory-management algorithms significantly affects the efficiency, performance, and reliability of the file system
- The two major approaches are using linear list and hash tables

Linear List

- The simplest method of implementing a directory is to use a linear list of file names with pointers to the data blocks.
- This method is simple to program but time-consuming to execute.
- To create a new file, we must first search the directory to be sure that no existing file has the same name. Then, we add a new entry at the end of the directory.
- To delete a file, we search the directory for the named file and then release the space allocated to it.
- To reuse the directory entry, we can do one of several things.
    - We can mark the entry as unused (by assigning it a special name, such as an all-blank name, or by including a used–unused bit in each entry)
    - We can attach it to a list of free directory entries.
    - Copy the last entry in the directory into the freed location and to decrease the length of the directory.

- A linked list can be used to decrease the time required to delete a file.
- The real **disadvantage** of a linear list of directory entries is that finding a file requires a linear search
- Directory information is used frequently, and users will notice if access to it is slow.
- Many OS implement a software cache to store the most recently used directory information. A cache hit avoids the need to constantly reread the information from disk.
- A sorted list allows a binary search and decreases the average search time. However, the requirement that the list be kept sorted may complicate creating and deleting files, since we may have to move substantial amounts of directory information to maintain a sorted directory.
- A more sophisticated tree data structure, such as a balanced tree, might help here. An advantage of the sorted list is that a sorted directory listing can be produced without a separate sort step.

Hash Table

- Here, a linear list stores the directory entries, but a hash data structure is also used.
- The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list.
- Therefore, it can greatly decrease the directory search time.

- Insertion and deletion are also fairly straightforward, although some provision must be made for collisions—situations in which two file names hash to the same location.
- The major **difficulties** with a hash table are its generally fixed size and the dependence of the hash function on that size.
- Alternatively, we can use a chained-overflow hash table. Each hash entry can be a linked list instead of an individual value, and we can resolve collisions by adding the new entry to the linked list.
- Lookups may be somewhat slowed, because searching for a name might require stepping through a linked list of colliding table entries. Still, this method is likely to be much faster than a linear search through the entire directory.