# File System

So far, the data/information was always linked to a process. The data is lost as soon as the process terminates.

A **file** is a way to store data that does not belong to a process.

Files are an abstraction mechanism that shields users from details how and where the information is stored.

- Files are usually created by processes.
- Files store **logical** units of information created.
- If a process creates a file it gives a name to the file.
- Files can be **concurrently** accessed from several processes. using the file name.

There are two things to consider:

- How does the file system appear?
- How is it managed?

# File Naming

- Usually the file system allows file names up $x$ characters long.
- Letters but also special characters are possible.
- Some file systems distinguish between **upper and lower cases**, others not. (MARIA, maria and Maria can specify one file or three different files.)
- File names are often like **name.tex**, meaning a name and a 3 character extension.
  - The extension specifies the **type** of the file.
  - In some file systems (like in UNIX) the file extensions are not enforced.
  - In others (like Windows) they are enforced. Here a double-click opens a file with the right application.

# File Structure

A file is usually structured as

- an unstructured sequence of bytes,
- a record sequence, or
- a tree.

**Unstructured sequence of bytes:**

- The process that reads the file gives meaning to its contents.
- This gives maximum flexibility, processes can but whatever they want into files.

**Records**

- Records have a fixed length and every record entry has some **internal structure**.
- Read and write operations access a whole record.
- **Traditionally:** a record was used to represent one punch card.

**Trees**

- Tree entries can have different lengths.
- Every entry has a key that is used for the tree order.
- Makes **searching easy**.

See Figure 4-3.

# File Types

Many operating systems have special files like

- directories (UNIX),
- *character special files* related to input/output,
- *block special files* to model disks.
- regular files that containing user information.

General files are usually

- **ASCII files** that consists of lines of text. They can be printed directly or edited with a text editor.

  **Binary files** are all files that are not ASCII files. They usually have some internal structure known to programs using them.

See Figure 4.3 for two examples for binary files.

# File Access

**Sequential access:**

- The file can only be read **in order**, no parts can be skipped.

This was O.K. for magnetic tapes. Early operating systems provided only *sequential* access.

**Random access:**

- Here the OS can access arbitrary parts of the file.
  - Every read operation gives the position to read.
  - Operation **seek** finds the position on disk, from there on the access is sequential.

# File Attributes

Every file has a **name** and, of course, the **data**.

OS associate other information with files:

- time the file was created,
- size,
- when last modified,
- who is allowed to access the file,.....

The attributes vary considerably from system to system.

See Figure 4-4 for an example.

# File Operations

Different OS provide different operations, common operations are

- **Create**: creates empty file.
- **Delete**: deletes the file using a system call.
- **Open**: it allows the system to fetch the attributes and address map into main memory.
- **Close**: Frees up internal space.
- **Read**: The caller must specify how many data is needed, and a buffer to store the data .
- **Write**: Write data into the current position. This means that file size can increase or data overwritten.
- **Append**: adds data to the end of a file.
- **Get/set attributes**.
- **Rename**.

# File System Calls

Homework: See Figure 4-5 for an example for UNIX file system calls and read Section 4.1.7.

# Directories

Directories are used to organize files by grouping them together.

- **Single-level** directory systems: one directory (usually called root) contains all files.
- **Hierarchical** directory systems: tree-like structure.

See Figure 4.6 and Figure 4.7.

When files are organized in a tree a method id needed for specifying file manes.

- **Absolute path names** consist of the path names from the root to the file. Names are divided by / or >.
- **Relative paths names** start from the working directory.

. usually refers to the working directory and .. to its parent.

# Directory Operations

There are many system calls to manage directories. Typical ones are

- **Create**: creates an empty directory.
- **Delete**: deletes an empty directory.
- **Opendir**: opens an directory, has to happen before read.
- **Closedir**: closes an dir and frees up internal table space.
- **Readdir**: returns the next entry in an open directory.
- **Rename**.
- **Link**: allows a file to appear inn more than one directory.
  - specifies an existing file name and creates a link from the existing file to the name specified in the path. (*hard links*).

    Both file names point to the same data that represents the file!
- **Unlink**: removes a directory entry. If the file is present only once, the whole file is deleted, otherwise only the specified path name is deleted.

**Symbolic links:** The file name points to a tiny file naming another file. This file is then accessed by the OS. It can be on another disk or computer.

# File System Implementation

- File are usually stored on disks.
- Most disks can be divided up into one or more partitions with independent file systems for each partition.
- Sector 0 of the disk is called *master boot record* (MBR) and is used to boot the computer.

  The end of the MBR contains the partition table giving the start and end address of every partition.
- When the computer is booted the BIOS reads in and executes the MBR.
- The MBR first locates the active partition, reads its first block (called *boot block*) and executes it.
- The program in the boot block loads the operating system contained in that partition.
- Every partition starts with a boot block, even if it does not have a bootable operating system.
- The next block is usually a *superblock* that contains all key parameters about the file system. This is read into memory when the computer is booted.

See Figure 4-9.

# Storing Files

We have to store the information which disk block goes to which file.

### I Continuous Allocation

Files are stored in consecutive blocks. See Figure 4-10.

**Advantages:**

- Easy to implement.
- One has to remember only two numbers per file.
- Very good performance, especially if the whole file is needed.

**Disadvantages:**

- It fragments the disk.
- Long list of ”holes”.
- Problems if a file size increases.

# II Linked List Allocation

Files are stored in non-consecutive blocks, usually of fixed size.

The blocks are stored as a linked list. See Figuree 4.11. The first word of each block is used as a pointer to the next block.

**Advantages:**

- Every hole on the disk can be used, no fragmentation.
- It is sufficient to store the first block for every file.
- Sequential reading is straight forward, reading an arbitrary position not.

**Disadvantages:**

- The size of the data in a block is not a power of two since the first word contains the pointer.

# Linked List Allocation Using Tables

Here the file is also stored in blocks of fixed size, but the pointers are not in the block. They are stored in a table in memory. See Figure 4-12.

**Advantages:**

- Every hole can be used, no fragmentation.
- It is sufficient to store the first block for every file.
- Sequential reading is straight forward and reading an arbitrary position got much easier.

**Disadvantages:**

- The table has to be in memory all the time. And it can be huge!

# I-Nodes

(I stands for index)

Every file has a data block specifying some attributes of the file and the list of blocks where the file is stored. See Figure 4-13.

An I-Node has a fixed size (one disk block). If the file needs more disk blocks the last entry of the I-node points to a block with more disk block addresses.

**Advantages:**

- Every hole on the disk can be used, no fragmentation.
- The I-Node is only in memory if the file is open.

# Implementing Directories

The main job of the directory system is to map the ASCI name of files onto the information needed to locate the files.

When a file is opened, the OS uses the path name to locate the directory entry. This entry provides the access information.

Sometimes, file attributes are also stored in the directory. See Figure 4.14.

Here we see fixed-size entries, one per file, containing file name, attributes, adresses,... (Windows).

In the case of I-nodes the attributes are usually stored in the node and the file entry contains only the address of the first I-node (UNIX).

# Shared Files

We saw that a file can also appear in several directories.

Then the file system is a directed and possibly cyclic graph!

**Hard linking.**

Usually the disk blocks of a file are not listed in the directory, but in an extra data structure.

Then each directory containing the file points to that structure (UNIX).

**Problem:** what happens if a file that appears in several directories is deleted?

- Some directories point to non-existing files.
- If the I-node of the deleted file is reassigned the data structure points to the wrong file!

The I-node stores the information that the file is in several directories, but there is no easy way to find them.

**One Solution:** Remove the entry in the directory but leave the I-node.

Then there can be files deleted by the owner (quota problem!).

# Symbolic linking

If a file is symbolically linked into another directory, this directory creates a new file of type *link*. That file contains the path to the file.

- A file is destroyed if the owner removes it. The remaining paths are not valid.
- Removing a symbolic link does not affect the file.

**Problem:** This causes overhead since the new paths has to be followed, which can require lots of disk accesses

# Management and Optimization

### Block size

The ideal block size depends on

- the disk (sector? track? cylinder?)
- file size distribution
- page size in paging systems
- size of the disk

There is a trade-off between space utilization and data rate. See Figure 4-21.

**Space utilization**: consider all used blocks and divide used space in these blocks by total space of these blocks.

**Data Rate**: Fraction of the time that the disk outputs data compared to the total access time.

**Note:** if blocks are huge but contain only a small amount of data the data rate is still good!

# Keeping Track of Free Blocks

**Linked list:** linked list of empty blocks. The lists can be long, but the list can be stored in empty blocks!

Only one blocks with empty blocks has to be held in main memory.

- If it is full the block is written back to the disk. See figure 4-23.
- The same is possible with bitmaps.

**Another approach**: store an approximately half-empty block in memory.

- This avoids writing the block back to main memory. (A very short list of empty blocks means that a new block has to be downloaded pretty soon,...)

**Bitmap:** here we have one bit for every block, 1 says block is in use and 0 for empty blocks.

Bitmaps require less space unless the disk is nearly full.

See Figure 4.22 for a general example of linked lists and bitmaps.