Lecture 12 - File systems

Gidon Rosalki

2025-06-22

Notice: If you find any mistakes, please open an issue at `https://github.com/robomarvin1501/notes_operating_systems`

This content will not appear in the exam this year!

1 Files

A file is a named persistent sequential data storage. Sequential indicates that the data within the file is ordered, and may have some structure. Persistent indicates that it lasts between restarts, and named means that it can be found based off its name.

1.1 Names

Names are external to the operating system, and exist in many contexts:

- Access hsared memory segments (shmget, shmat)
- Domain name service to find web hosts
- Well known ports for certain services

One needs to remember the name to access the file, or possibly find it in a list, though this can take a long time. A modern alternative is to search by content.

1.2 Persistence

Files are stored on non-volatile devices, such as disks and other magnetic media, SSDs (non-volatile memory), and so data is retained when computer turned off. Data also survives the process which created the data. Typically one process creates the file, and others read the data later. This is as opposed to memory which is created and dies with the process.

1.3 Structure and meaning

In unix, data is a sequence of byutes. It can represent text, pictres, video, sound, program source code, program binary, numerical data, and so on. In IBM mainframes, data is a sequence of records for databases. Finally, in windows NTFS, it is a set of attribute value pairs, e.g. the file's icon with an indication of the internet source. It always has an unnamed data stream wit contents.

1.3.1 Filename extensions

By **convention**, the filename is divided into 2 parts, the name, and the extension. However, extension-less files are also common in Linux. Applications such as a C compiler might insist on extensions.

1.4 OS connection

There needs to be a separation of concerns between the OS and the application. Operating system:

- Provide infrastructure and support
- Store and retrieve the data

- Keep metadata about the data
- Don't limit the application

Application:

- Do interpretation and processing
- Create and use the data
- Decide on data format (how things are represented)
- Know what the data means

The file system has an API, which abstracts files, directories and naming, and sharing and protection. To implement a file system, the OS needs to be responsible for storing the data, space allocation methods and management, and provide reliability and performance.

The FS operations are **system calls**. The OS implementation uses privileged instructions to activate storage devices, and devices report completion using interrupts.

There are a few popular file systems. Linux generally uses ext4, the latest revision of the extended file system. The previous version, ext3 could only support disks up to 16TB, which have since become more common, along with even larger disks. We will mainly focus on ext3, but there is a lot of equivalence between it and ext4. Windows just uses NTFS, and finally USB sticks tend to use FAT32 or exFAT. FAT32 can only support a limited size of files, so everyone is moving over to exFAT (you are recommended to always reformat new USB sticks to exFAT, if they were originally FAT32).

1.4.1 Basic file system operations

Have a nice big list of basic file system operations:

- Create a file
- Delete a file
- Open a file
- Close a file
- Write to a file
- Read from a file
- Seek to somewhere in a file
- Truncate a file
- Set attributes of a file
- Get attributes of a file
- Rename a file

1.4.2 File attributes

File specific info is maintained by the OS, called the metadata. This varies a lot across different OSs. There are two parts to the metadata, the user facing, and the internal.
User facing: This contains things like the name (used to identify the file, human readable), the type (needed for systems that support different types), the size, the owner user identification (for protection and security), access rights (who can read, write, execute), and the time of creation, modification, and access. This last field is used extensively by GNU make. The internal OS use parts include the identifier, a unique tag (number) that identifies the file within the file system, and the location, which is a pointer to the file location on the device.

2 Files namespace

The motivation behind naming the file is to re-access the data that was stored previously, such that the user need not remember the block, sector, and so on. This way a process can create a file, and give it a name, and the human can later on find it in another process, and read it.

Naming conventions are OS dependent. Usually names as long as 255 characters are allowed, and digits and special characters (such as space) are sometimes allowed. MS-DOS and Windows are not case sensitive (for they are fools), where the UNIX family is. The naming administration is done separately to the data.

2.1 Organisation

Files are organised in a hierarchy of directories, which aim to obtain:

1. Efficiency - locating a file quickly, we want to avoid a single huge list with all the files
2. Separation - convenient to users. This way two users can have the same name for different files, and the same file can have several different names
3. Grouping: Logical grouping of files based off contents, properties, or usage. E.g. all java programs, all games, and the files related to OS exercise 4, and so on.

2.1.1 Directories

Directories map names to file objects. They are one level of the naming hierarchy. This data is stored in files, so directories are in fact files themselves. A bit in the file metadata object (inode) indicates if it is a directory, and the OS interprets the format of the directory file data. One should ideally use an efficient data structure, to enable speed in searching / other file operations.

We can achieve this by having the namespace have a tree structure, of arbitrary height. Nodes are directories and files, and internal nodes are directories. Directories contain files, and sub-directories. Names are relative to a directory, and files are identified by their path from the root.

File systems have a root directory, and processes have a working directory. This is typically the home directory of their user. To access a file, the process needs to specify the path in which the file is located. Path names are either absolute or relative:

- Absolute: path of file from the root directory
- Relative: path from the current working directory

Path components are separated by / (unix) or
(windows, like fools). Most OSes have two special entries in each directory: "." for current directory ".." for parent.

We can also have acyclic links between directories, where we have files located under two different directories simultaneously (remember, directories are also files). This can be done through hard links and soft links.

- Hard links: We can only perform these for files, not directories, to avoid recursive loops. They may also only be created across the same file system. They point at files (not names), and so are move safe. They are also reference counted, and so are delete safe
- Soft links (AKA symbolic links): These are links between any file and directory, and are neither move, nor delete safe. Should you delete a file, the soft link will still exist.

2.1.2 Protection

File owner / creator should be able to control what can be done, by whom. There are 6 types of access, Read, Write, Execute, Append, (Dir) Delete, (Dir) List. Note, creating and deleting files are operations on the **directory**.

We have some categories of users to consider when it comes to protection and access rights. The individual user, who establishes a user id on log in, which might be just local on the computer, or could be through interaction with a network service, and groups, to which users belong, such as wheel, or docker.

To express access rights, we have two methods:

- Minimal (*nix): Just a single user, group, and the rest, which is compact enough to fit in just a few bytes. However, this is not very expressive.
- Detailed (Windows): A per file list that tells exactly who can do what to that file, which is useful in its expressiveness, but is in no way compact

Linux access rights dictate 3 modes of access: read, write, and execute. There are 3 classes of users, the owner, the group, and the others. For each file or directory, we define desired access rights as 9 bits. Each group of 3 indicates RWX for each of the groups.

3 Layout and Access

When accessing a file, we maintain our current location in an open file, and access is relative to this location. This location can be changed based off `rewind` or `seek`.

How we layout the information on the disk can cause problems. It we simply store data contiguously, this can lead to fragmentation, where there is enough memory to store a file, but it is not contiguous. There are a couple of possible solutions:

- Compaction: Move around data in disk, to unify holes and create large enough holes to accommodate future files. This may improve performance thanks to serial access, however this is very costly, and there is not necessarily enough space to move around data like this.
- Divide files into fixed size blocks (usually 4KB), and all operations are made on blocks. This is similar to paging in memory management.

When using a block system, the user views a file as contiguous data, the API views it as a sequence of bytes, of arbitrary size, the FS views it as a collection of 4KB blocks, and the disk views it as individual sectors (512 bytes). Accesses are not necessarily aligned with blocks, and blocks may span multiple sectors.

Note, these notes are incomplete in comparison to the lecture. I ran out of energy, especially since the content is extra (though fascinating).