File Management

CISC 3320 — CUNY Brooklyn College
Lecture Notes

File Management

In this chapter, we will:

1. Describe the file attributes (metadata) that the OS stores for each file on the device.
2. Describe the operations that we can perform on files.
3. List file types and sample file extensions for each file.
4. Explain how a file is structured and stored internally.
5. Introduce the concepts of file-system mounting and paths to files.
6. Discuss file protection and access control.

File Management

A **file** is a set of related data that are stored on secondary storage devices. Although the user or an application views a file as one solid object, the operating system might store various parts of the file (different sections of the file, or the file's metadata) at different places/addresses on the disk.

Metadata is data about data. File metadata is data about the file, which the OS uses to better control and organize files. File metadata include:

- File name (for human readability/ease.)
- File ID (usually a number, for OS use.)
- Type (e.g., binary, text, source code, executable.)
- Extension (e.g., txt, docx, html, exe.)
- Location (start address) on the disk.
- Size of the file's content in bytes.
- Permissions that users have (e.g., read, change, execute.)
- Date & time of the file's creation.
- Date & time of the file's most recent modification.
- The ID or name of the file's owener (the user who owns the file.)

File Management

A user can perform the following activities with files:

- Creating new files.
- Writing to (changing) the content of an existing file, such as by replacing some of the existing content [= "overwritting"] or appending new data to the end of the existing content.
- Reading (accessing) the content of an existing file.
- Repositioning within a file. That is, when a file is open by a program, the program can instruct the OS to put the reading or writing position at a particular place in the file. For instance, the program can issue an instruction to start reading the file at position 1000 (which is 999 bytes into the file since byte 0 is the 1st byte.)
- Deleting a file if one isn't needed anymore.
- Truncating a file, which is cutting off (discarding) some of the file's content. For example, we could discard (delete) all the content from byte 1001 and onward, and only to keep the first 1000 bytes in the file.

File Management

The table below lists frequently used file types and various exensions thereof:

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	bat, sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf, docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

Common file types and extensions. Taken from Bell, John T. "File-System Interface." University of Illinois, Chicago.

File Management

Some files contain an internal structure, which may or may not be known to the OS. For the OS to support particular file formats increases the size and complexity of the OS.

UNIX/Linux treats any file as a **sequence (array) of bytes**, with no further consideration of the internal structure. (With the exception of executable binary programs, which it must know how to load and find the first executable statement, etc.)

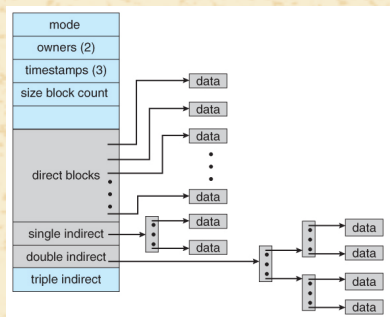
Disk files are accessed in units of physical blocks, typically 512 bytes or some power-of-two multiple thereof, like 1024 = 1K, 2048 = 2K, 4096 = 4K, and 8192 = 8K. So, if you wish to read only 100 bytes from a file, the OS will return you, for example, 4K of bytes, out of which you (or your program) will need to extract these 100 bytes.

Internally files are organized in terms of logical units, which may be as small as a single byte, or may be of a larger size corresponding to some data record or structure size.

File Management

The number of logical units which fit into one physical block determines its **packing**, and has an impact on the amount of internal fragmentation (wasted space) that occurs. As a general rule, half a physical block, on average, is wasted for each file, and the larger the block sizes are, the more space is lost to internal fragmentation.

A **directory** is a file that contains a mapping of file names to the number of the data structure named **inode** that stores the rest of the file's metadata. For example, a directory could say that a file named `helloworld.exe` is mapped to inode number 3625763. This way, when a user accesses the `helloworld.exe` file from inside the directory, the OS will know where to find this file's content on disk (since the inode contains the addresses in the disk where the file resides.)



Inode in UNIX/Linux. Taken from Bell, John T. "File-System Implementation." University of Illinois, Chicago.

File Mounting and Paths

Mounting is the action of establishing access to a directory of files that are stored on the disk. Once access is established, the OS can access the files and the directories inside the **mount point**, which is the top (`root`) directory.

The basic idea behind mounting file systems is to combine multiple file systems into one large tree structure. A **file system** is the directories and files within these directories, which form a shape of a tree.

Any files (or sub-directories) that had been stored in the mount point directory prior to mounting the new filesystem are now hidden by the mounted filesystem, and are no longer available. For this reason some systems only allow mounting onto empty directories.

Filesystems can only be mounted at the `root` directory, unless the filesystem has been previously configured to be mountable at pre-determined mount points that are other than the `root`.

File Mounting and Paths

A **path** to a file or directory is a sequence of directories separated by a `/` symbol (in UNIX/Linux/Mac) or a `\` symbol (in Windows) that tells where the files resides inside a file system.

An **absolute path** is one that starts with the root directory "`/`" (in UNIX/Linux/Mac) or "`C:\`" (in Windows). Examples: "`/home/seth/Pictures/penguin.jpg`" or "`C:\Users\Staff\Music`".

A **relative path** is one that starts with the current directory, usually with the symbols `.` (current directory) or `..` (parent directory). Examples: "`../CISC3320/diagram.jpg`" (access `diagram.jpg`, which reside in the 'sibiling' directory to the current one) or "`../../../`" (access the 'grandparent' directory ("`../../`") and stay there ("`./`")).

There exist *infinitely many* absolute and relative paths to any directory or file. Here is how: if the name of some directory inside the path is, say, `Music`, you can add infinitely many of the following 'back and forth' subsequences into the path: "`Music/../Music/ ... /../Music/`" 😁. However, each file's **shortest absolute path** is one and unique.

File Mounting and Paths

This video introduces and exemplifies file paths:

https://www.youtube.com/watch?v=ephId3mYu9o

File Mounting and Paths

This previous video also spoke about the ~ character, which stands for a user's home directory. For example, a user named Megan might have ~ standing for "`/users/megan/home/`".

Each user has a unique home directory, and, on Linux, there is a file in the OS called "`/etc/passwd`" that stores the path to each user's home directory.

A system admin can change the home directory of a user. For example, on Linux, one can do so with a terminal command such as:
`usermod --home </newhome/username> <username>`
where `</newhome/username>` is the new home directory that you are trying to set, e.g., "`/users/megan/home/Desktop`", and `<username>` is the username of the user whose home directory is changed, such as `megan`.

File Protection and Permissions

No one wishes to have people mess up with their files, either carelessly or intentionally. For this purpose, the operating system enforces a permissions system that tells who (which users) can do what (what actions) to which files.

The goal of such **access control** policies is to ensure the protection of files from the actions of other users.

Every OS has different access control policies. For example, each system defines different types of 'actions' that can be applied to files, among which are reading, writing, appending to, executing, deleting, and listing the names (and other metadata) of files.

Some systems can apply passwords, either to individual files, to specific sub-directories, or to the entire file system.

File Protection and Permissions

UNIX/Linux/Mac uses a set of 9 access control bits, in three groups of three. These correspond to `R` (read), `W` (write), and `X` (execute) permissions for each of the Owner of the file, the users in the Owner's Group, and all Other users of the device (that aren't the owner or in the owner's group.)

A string that allows all of the permissions to all three user categories looks like: `rwxrwxrwx`. The leftmost triple, `rwx`, is for the Owner, and tells that the owner can read from, write to, and execute the file. The middle triple, `rwx`, is for the users in the Owner's Group, and the rightmost triple, `rwx`, is for any Other user of the device.

When a permission is NOT granted, a dash `-` appears instead of the corresponding letter. For example, `rwxr--r--` means that the the owner can read from, write to, and execute the file, but the users in the Owner's Group and any Other user can only read the file's content, without being allowed to change the file or execute it.

File Protection and Permissions

Any questions?

File Management: Sources

Bell, John T. "File-System Interface." *University of Illinois, Chicago*, Accessed 11 July 2022. URL: https://www.cs.uic.edu/~jbell/CourseNotes/OperatingSystems/11_FileSystemInterface.html