ECE254: Operating Systems and Systems Programming Spring 2015

Lecture 32 — File System Interface

Jeff Zarnett

File Systems

The file system is an important and highly visible part of the operating system. More than just the way of storing data and programs, persistently, it also provides organization for the files through a directory structure and maintains metadata related to files.

But what is a file? The snarky UNIX answer is, "Everything is a file!", but using the word in the definition is rather bad form. As far as the computer is concerned, any data is just 1s and 0s (bytes). The file is just a logical unit to organize these. So an area of disk is designated as belonging to a file. Files can contain programs (e.g., `word.exe`) and/or data (e.g., `book-report.doc`). The content of a file is defined by its creator. The creator could be a user if he or she is using notepad or something, or it could be a program, like a compiler creating an output binary file.

Files typically have attributes, which, although they can vary, tend generally to include the following things [SGG13]:

1. **Name**: The symbolic file name, in human-readable form.
2. **Identifier**: The unique identifier, usually a number, that identifies the file inside the file system.
3. **Type**: Information about what kind of file it is.
4. **Location**: The physical location of the file, including what device (e.g., hard drive) it is on.
5. **Size**: The current, and possibly maximum, size of the file.
6. **Protection**: Access-control information, including who owns the file, who may read, write, and execute it...
7. **Time, Date, User ID**: The owner of the file, time of creation, last access, last change... any sort of data that is useful for protection, security, usage monitoring...

Files are maintained in a directory structure. The directory structure is generally quite familiar to us as the folders on the system. Directories, really, are just like files; they are information about what files are in what locations, and they too will be stored on disk.

File Operations

It makes some sense to consider a file to be something like a class in an object-oriented programming language; a file has some data (fields, metadata) and some operations. The OS provides these operations to allow users to work with and on files. Six basic operations are required for a file system to be useful, though other things like renaming and so on are nice to have [SGG13]:

1. Creating a file.
2. Writing a file.
3. Reading a file.
4. Repositioning within a file.
5. Deleting a file.

6. Truncating a file.

Let's examine each of these briefly. As you will see, there is a certain similarity between file operations and memory operations, with which we should already be familiar.

Creating a File. Like allocating memory, creating a new file has multiple essential steps: first, find a place to put the file, allocate that space and mark that file as being allocated, and finally put the file in its appropriate directory.

Writing a File. Writing a file requires the name or identifier of the file and the data to be written to the file. Using the name or identifier, the system finds that file and can then start putting data in the file. A write operation may replace the existing contents or append (write at the end) to the existing contents. A pointer will be needed to keep track of where the next write will take place, and will be updated after each write.

Reading a File. This requires the name or identifier of the file and where in memory the next block of the file should be put. A pointer will also be required to indicate where the next read will take place.

Repositioning within a File. Since a file may be read or written, but usually only one at a time, the pointer for the write location may in fact be the same pointer for the memory location. If so, we might call it a current position pointer, and this operation is just repositioning it within the file. Repositioning is also sometimes called a seek operation.

Deleting a File. Deletion works pretty much as we would expect: find the file, mark its space as free, and remove it from the directory listing. This is a "simple" deletion and it does not actually get rid of any of the data, it just makes the file system forget the existence of the file. However, it might be possible to recover the data if the space it previously occupied has not been overwritten. This is a bit like a freed pointer in C possibly still being accessible. Some systems offer a more secure deletion routine that overwrites the space the file used to occupy with zeros.

Truncating a File. If a file should be erased but its attributes maintained (e.g., all the metadata), we can truncate it: cut off all the contents. The file length is reset to zero and its data area is marked as free, but the rest of the attributes remain the same.

These six operations can be combined for most of the other things we may want to do. To copy a file, for example, create a new file, read from the old file, and write it into the new file. We may also have operations to allow a user to access or set various attributes such as the owner, security descriptors, size on disk, et cetera [SGG13].

Aside from creation and deletion, all the other operations are restricted to files that are open. When a file is opened, a program gets a reference to it, and the operating system keeps track of which files are currently open in which process. It is good behaviour for a process to close a file when it is no longer using it, but eventually when the process terminates, that will automatically close any open files (hopefully).

The implementation of the open and close file routines can be very complicated: we might open a file as read-only (modifications forbidden) etc. If the opening mode allows multiple processes to open the file simultaneously, the OS will likely maintain a per-process table and a system-wide table. Access rights and accounting information may appear in either. The per-process table contains a reference into the system-wide table. When the last process that has a file open uses the close system call, the reference count is zero and we can remove it from the table.

Some operating systems support file locks that work a lot like the mutual exclusion concepts we have examined earlier. Locks may be exclusive, or non-exclusive. When a file is locked by one process, other processes will be advised that opening failed due to someone having a lock on that file. Similarly, files in use cannot be deleted while that file is in use.

Windows, for example, uses locking and any file that is open in some program cannot be deleted. UNIX, however, does not, so UNIX-compatible programs can, if they need, lock a file, but by default this does not happen. In UNIX if a file is open in a program, another user can still delete the file and it will be removed from the directory. As long as that program remains open and retains that reference to the file, it can still operate on that file. However, once the file is no longer open in a program, its storage space will be marked as free.

File Types

Files we are familiar with often have extensions separated from the file name by a period, like `fork.txt`. The `txt` extension tells us some information about the file, i.e. that it is a text file. These things are mostly hints to the OS or user about what sort of file it is. In most operating systems, any program can open arbitrary files... that it has a `.docx` extension is only a suggestion that it should be opened by a word processing program, but nothing stops people from opening it in any other program. OSes typically allow setting a default program for the extension: e.g., always open `.docx` files with LibreOffice.

File Structure

Locating an offset inside a file can be a pain for an operating system. Disk systems operate on blocks of equal size, but the logical size of the file probably does not fit exactly into a block (or an integer multiple of blocks). So we often have multiple logical records packed into one block. As far as UNIX is concerned, any byte of a file can be accessed, but if the disk uses 4 KB blocks, the file system will need to pack bytes into and out of the physical blocks to make it work [SGG13].

Because disk space is allocated in a block, there is some internal fragmentation inherent to the system. If a file is, say, 21 KB, it will fit in 5 of the 4 KB blocks and a fifth block will be allocated that contains the last 1 KB but has 3 KB of internal fragmentation (wasted space).

Access Methods

References

[SGG13] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts (9th Edition)*. John Wiley & Sons, 2013.