Lecture 32 — File System Interface

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

June 1, 2015

File System Interface

The file system is an important and highly visible part of the operating system.

It is 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.

What is a File?

But what is a file? The snarky UNIX answer is, “Everything is a file!”

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.

What is 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.

File Attributes

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

1. **Name**
2. **Identifier**
3. **Type**
4. **Location**
5. **Size**
6. **Protection**
7. **Time, Date, User ID**

Directory Structure

Files are maintained in a directory structure.

The directory structure is 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 (methods).

The OS provides these operations to allow users to work with and on files.

File Operations

Six basic operations are required:

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.

Combining the Operations

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...

Restrictions on the Operations

Aside from creation and deletion, all operations are restricted to open files.

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.

When the process terminates, that will automatically close any open files (hopefully).

Open and Close

The implementation of the open and close file routines can be 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.

File Locks

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 and UNIX File Locks

Windows uses locking; 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. 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...

A `.docx` extension is only a suggestion that it should be opened by a word processing program.

OSes typically allow setting a default program for the extension: e.g., always open `.docx` files with LibreOffice.

Access Methods

When the system wants to operate on a file, it is usually read into main memory.

We often have several options if we are dealing with a hard disk drive where we can read from or write to anywhere.

But this is not the only way.

Sequential Access

Sequential access is the simplest method.

Information in the file is processed in order, one section after another.

Compilers tend to operate in this way, for example.

Reads and writes make up most of the operations.

The operations may look something like `read_next` which advances and reads from the next record.

Sequential Access

It is possible that going backwards is allowed, either by resetting to the beginning or “reversing” by some number of records (possibly 1).

This access method stems from tape drive systems.

We could advance or reverse the tape one section at a time.

It was not THAT long ago that tape drives were used for large data backups...

Direct Access

The normal kind of access is direct access.

It is exactly what you expect: any part of a file can be accessed at any time.

There are no restrictions on the order and no need for the program to seek around to the right place.

Rather than reading the next block or advancing to the next block, we can usually read from or seek to an arbitrary block.

Direct Access

The operating system typically hides the underlying physical block information by using a relative block number.

From the perspective of the application or user, the first block of the file is numbered zero, the next one is one, and so on.

This may have no underlying relation to where a block is located on disk.

Direct Access

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.

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.

There is some internal fragmentation inherent to the system.

Directories

A directory is really just a symbol table that translates file names (user-readable representations) to their directory entries.

They typically support the operations:

1. **Search**
2. **Add a File**
3. **Remove a File**
4. **List a Directory**
5. **Rename a File**
6. **Navigate the File System**

Directories

There are simple file systems where there are no such things as subdirectories.

Textbooks may also bring up a structure where each user has his or her own directory but cannot have subdirectories either.

Tree-structured: there it a root directory, and every file in the system has a unique name when the name and path to it (from the root) are combined.

UNIX Tree Structured Directory

In UNIX the root directory is just called / (forward slash).
From there we can navigate to any file.

To run the `ls` command, we will find it in the `bin` directory as `/bin/ls`.

This is an example of an absolute path.

UNIX Tree Structured Directory

Most of the time we do not have to use the absolute path (the full file name); a relative path (the path from the current directory) will suffice.

Example: compile something with a command like `gcc code/example.c`.

The file `example.c` is in a subdirectory of the current directory called `code`.

The system will work out that we need to start from the current directory (e.g., `/home/jz/ece254/`) and prepend that to the given file name.

This produces the absolute path of `/home/jz/ece254/code/example.c`.

Deleting a Directory

What if a directory is not empty?

If it is empty, just removing the directory is enough.

If it contains some files, either the system can refuse to delete the directory until it is empty, or automatically delete the files and subdirectories.

Deletion

Also, what does it mean to delete a file or folder?

The delete command sometimes does not necessarily actually delete the file or folder, but instead moves it to some deleted files directory.

If it is deleted from there then it is really gone, but while it is in that deleted file directory it can be restored.

Sharing of Files

File systems may also support the sharing of files.

There is one copy of the file but it has more than one name.

In UNIX this is called a link and this is effectively a pointer to another file.

Links are either “hardlinks” or “symlinks”.

Symbolic Links

Symlinks, or symbolic links, are just references by file name.

So if a symbolic link is created to a file like `/Users/jz/file.txt`, the symbolic link will just be a “shortcut” to that file.

If the file is later deleted, the symbolic link is left pointing to nothing.

A future attempt to use this pointer will result in an error.

It would be expensive, though possible, to search through the file system to find all links and remove them.

Hard Links

Creating a hardlink is creating a pointer to the underlying file in the file system.

If a hardlink exists and the user deletes that file, the file still remains on disk until the last hardlink is removed.

The file structure maintains a count of how many hardlinks reference a file, and it is only really deleted if the count falls to zero.

File Permissions

Files usually have some permissions associated with them:

1. Read
2. Write
3. Execute
4. Append (write at the end of the file)
5. Delete
6. List (view the attributes of the file)

Access Control: UNIX-Style

These are used often in UNIX(-like) systems.

Each file has an owner and a group.

Permissions can be assigned for the:

- Owner
- Group
- Everyone

Access Control: UNIX-Style

There are three basic permissions:

- Read
- Write
- Execute

Permissions are represented by 10 bits:
1 indicates true and 0 indicates false.

First bit is the directory bit.

Next three are read, write, execute for the owner.
Then read, write, execute for the group.
Finally, read, write, execute for everyone.

Access Control: UNIX-Style

The permissions can be shown in a human-readable format.

The order is always the same, and so a dash (-) appears if a bit is zero (permission does not exist).

The character `d` is used to indicate a directory.

`r` to indicate read access.

`w` to indicate write access.

`x` to indicate execute access.

Access Control: UNIX-Style Example

Example: `-rwxr-----`

This means:

- not a directory;
- the owner can read, write, and execute;
- other members of the group can read it only
- everyone else has no access to the file (cannot read, write, or execute).

Access Control: UNIX-Style

Permissions can also be written in octal (base 8):

r = 4, w = 2, and x = 1.

Start with 0, and then add the value of the permissions that are present, using zero where permissions are absent.

Example: 750

Access Control: UNIX-Style

More details: like what the permissions mean on directories,

Advanced topics like `setuid`, `setgid`, and “sticky bit”.

Beyond the scope of this course.

Access Control: UNIX-Style

The obvious shortcoming: very coarse grained.

Another strategy used by SELinux and Windows NT.

Access Control: ACLs

Files are protected by Access Control Lists.

Each file can have as many security descriptors as we like.

Descriptor lists the permissions a user/group has.

Permissions are more than just `rwx`.

Access Control: ACLs

In NTFS, you can see the security descriptors for a file: right click it in Explorer, properties dialog, choose the security tab.

The descriptors have two checkboxes: allow and deny, with

Deny takes precedence over allow.

Access Control: ACLs

Permissions can start out at default deny, in which case only the users explicitly granted access may access the file.

Alternative: default permit in which case everyone has access, minus those users whose privileges are explicitly denied.

Default deny is obviously more secure.

Access Control: ACLs

Represent an access control list as a set of tuples:

```
(alice, read)
(bob, read)
(charlie, read)
(charlie, write)
(bob, execute)
```

Access Control: ACLs

ACLs have an extra complexity: inheritance.

It is supposed to be a convenience feature...
but it can often be a cause of problems.

If inheritance is enabled: Any new file crated in this directory will receive the same ACL as the directory.

The danger is that any existing file moved into the directory will retain its original ACL.

If it is supposed to get the ACL of the directory, it needs to be explicitly set.