Lecture 3 — The File System

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

April 15, 2019

The file system is important and highly visible.

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.

Files can contain programs (`word.exe`) and/or data (`technical-report.doc`).

Skiena - The Algorithm Design Manual.pdf

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:

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

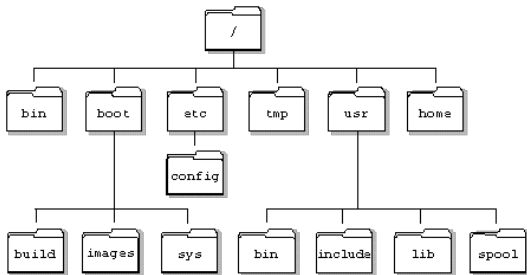Files are maintained in a structure.

Image Credit: QNX

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.

It makes some sense to consider a file to be a structure.

A file has some data (fields, metadata).

There are defined functions, 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.

We already saw in an earlier example how to open and close a file:

```
FILE* f = fopen( argv[1], "r" );
if ( f == NULL ) {
    printf("Unable to open file! %s is invalid name?\n", argv[1] );
    return -1;
}
readfile( f );
fclose( f );
```

It turns out that creating, reading, writing, and truncating are all done with the open call.

```
FILE * fopen( const char* filename, const char* mode )
```

In addition to the filename as the first parameter, the function is called with the mode as the second parameter.

In the last example, the mode we provided was a string literal of r.

- r – Open the file for reading only.
- w – Open the file for writing. If a file with the name exists, it's overwritten.
- a – Open the file, writing data at the end of the file.
- r+ – Open for read and update (the file must exist).
- w+ – Create a file and open it for update; if the file exists it's overwritten.
- a+ – Open a file for update, with new output at the end of the file.

If we combine an option with a b, such as `rb` then we are opening the file as a binary file.

Also, as of the C 2011 standard, there is a new add-on `x` which can be used to make any write operation fail if the file exists.
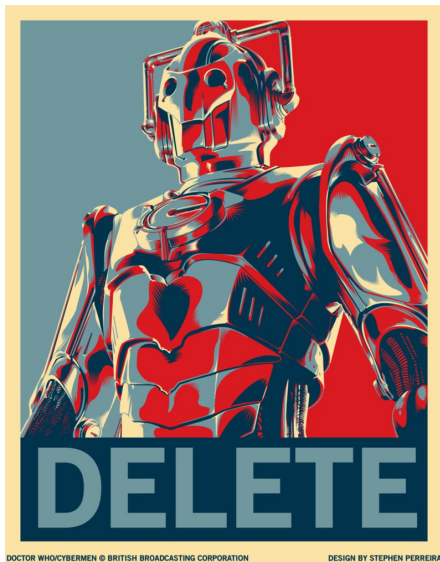
Repositioning is also sometimes called a seek operation.

In C this is done with the `fseek()` call.

This adjusts the pointer for reading or writing.

This should be done with caution though, because you can go to an arbitrary location, even the middle of a two (or more) byte character.

And we can't seek when a file is opened for append.

In C a file is deleted with the remove() function. This simple program deletes whatever file is provided as the second argument. In reality, we would more likely delete a temporary file that the program has used for some purpose, at the end of execution.

```c
int main( int argc, char** argv ) {
  if (argc != 2) {
    return −1;
  }

  remove( argv[1] );

  return 0;
}
```

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...

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).

Some operating systems support file locks.

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 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.

To lock a file in Linux, the call for this is `flock()`.

```
FILE* f = fopen("myfile.txt", "r");
int file_number = fileno( f );
int result = flock(file_number, LOCK_EX);
```

This example locks the file exclusively.

A shared lock would be LOCK_SH, and to unlock the parameter is LOCK_UN.

Writing to a file is easy enough because it works like `printf`.

In fact, the function call for it is `fprintf`.

The first argument to this is the file pointer where you'd like the data to be written to:

```
void write_points_to_file( point* p, FILE f ) {
  while( p != NULL ) {
    fprintf(f, "(%d,_%d,_%d)\n", p->x, p->y, p->z);
    p = p->next;
  }
}
```

Reading from a file involves the use of fscanf which is a mirror image of fprintf.

```c
int main( int argc, char** argv ){
  FILE *fp;
  int i, isquared;

  fp = fopen("results.dat", "r");
  if (fp == NULL) {
    return -1;
  }

  while (fscanf(fp, "%d,%d\n", &i, &isquared) == 2) {
    printf("i: %d,  isquared: %d\n", i, isquared);
  }

  fclose(fp);
  return 0;
}
```

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.

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**

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 is 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.

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.

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/ece252/`) and prepend that to the given file name.

This produces the absolute path of `/home/jz/ece252/code/example.c`.

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.

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.

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".

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.

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.

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)

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

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.

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.

Example: - rwx r - - - - -

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).

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

More details: like what the permissions mean on directories,

Advanced topics like `setuid`, `setgid`, and "sticky bit".

Beyond the scope of this course.

The obvious shortcoming: very coarse grained.

Another strategy is used by SELinux and Windows NT: ACLs.

... But we will consider that beyond the scope of the course for now.