ECE 252: Systems Programming and Concurrency Fall 2022

Lecture 3 — The File System

Jeff Zarnett *2023-01-12*

File Systems

The file system is important and very useful to programs. 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.

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., `technical-report.docx`). 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.

The UNIX approach is treating everything like a file, whether it is a device, regular file, or anything else (we'll see some examples later). This means that the functions that we use to interact with a file can be applied in several other contexts as well.

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 a structure; 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. 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. In all cases, we can look at some equivalent system calls (in the context of a trivial program) with thanks to [Sal] as the source of some examples.

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 );
```

Repeatedly opening and closing the same file is unnecessary and inefficient. If we plan to write some data into the file, then do more work, then write more data, it is probably wasted effort to close and reopen the file in between. It is okay to keep a file open when the file isn't being actively worked on, if you expect you will use that file again in the near future.

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 (but we don't have to manage this ourselves if we don't want to).

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 (and like the write pointer, we don't have to manage this ourselves usually).

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.

It turns out that creating, reading, writing, and truncating all involve the open call. The call is `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`. The modes are, according to the man7 page:

- `r` – Open text file for reading. The stream is positioned at the beginning of the file.
- `w` –Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.

- `a` – Open for appending (writing at end of file). The file is created if it does not exist. The stream is positioned at the end of the file.
- `r+` – Open for reading and writing. The stream is positioned at the beginning of the file.
- `w+` – Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.
- `a+` – Open for reading and appending (writing at end of file). The file is created if it does not exist. Output is always appended to the end of the file. POSIX is silent on what the initial read position is when using this mode. For glibc, the initial file position for reading is at the beginning of the file, but for Android/BSD/MacOS, the initial file position for reading is at the end of the file.

If we combine this 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 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.

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.

Generally – seeking in the file is only necessary if you want to "skip ahead" or go back in the file; if you read n bytes of the file, that advances the read pointer for you automatically; if you read 48 bytes and then seek 48 bytes forward, at the end of both operations your pointer is 96 bytes away from where it started.

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.

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.

```
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, 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).

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, 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.

To lock a file in Linux, the call for this is `flock()`. Think "File-Lock", not "flock of seagulls". It takes two parameters, the file descriptor and the type of lock we wish to have. But we have opened a file with `fopen()` and it returns a `FILE` pointer. To convert that to a file descriptor, there is a function `fileno()`.

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

This example locks the file exclusively. A shared lock would be `LOCK_SH`, and to unlock the parameter is `LOCK_UN`.

Reading and Writing

Writing to a file is easy enough because it works like `printf`. In fact, the function call for it is `fprintf` and the only real difference between that and `printf` is that 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`. The format specifiers are the same. Let's look at an example from [Sal]:

```
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;
}
```

The return value of this function call is the number of elements successfully read, which in this case is supposed to be two. But it's worth noting that there's no space in the read from the file, because the `%d` skips leading whitespaces (occasionally leading to hard-to-find bugs).

If reading user input, there is of course, regular `scanf`. We can, of course, read a file with `getline` as in the earlier examples.

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.

Directories

A directory is really just a symbol table that translates file names (user-readable representations) to their directory entries. A directory should support several common operations [SGG13]:

1. **Search**. We want to be able to find a file, and searching is typically not just on the file name but may include the contents of files as well, if their content is human-readable data.
2. **Add a File**. Add a file to the directory.
3. **Remove a File**. Remove a file from the directory.
4. **List a Directory**. List the files of the directory and the contents of the directory entry.
5. **Rename a File**. Change the user-friendly file name, possibly changing the file's position in the directory if it is sorted by name.
6. **Navigate the File System**. It should be possible to open subdirectories, go to parent directories, and so on.

There are some simple file systems where there are no such things as subdirectories, but they don't really require any examination. Textbooks may also bring up a structure where each user has his or her own directory but cannot have subdirectories either. Also rather uninteresting. The kind of directory we are most familiar with is 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) and from there we can navigate to any file. If we would like 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. As an example, if you want to compile something with a command like `gcc code/example.c`, the file `example.c` is in a subdirectory of the current directory called `code` and 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, to produce the absolute path of `/home/jz/ece252/code/example.c`.

OS designers have to make a choice about deletion of directories, 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 in their entirety. Also, what does it mean to delete a file or folder? In modern operating systems, the delete command sometimes does not necessarily actually delete the file or folder, but instead moves it to some deleted files directory (recycle bin, trash can, whatever you want to call it). 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 concept 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, so it is something to check on. It would be expensive, though possible, to search through the file system to find all links and remove them.

Creating a hardlink means creating a second 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. We will see later on, when we examine the file system implementation, how this actually works, but the short answer is: reference counting. 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

To protect users from one another and to maintain the confidentiality and integrity of data, files usually have some permissions associated with them, which may control access to the following operations [SGG13]:

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)

UNIX-Style Permissions. UNIX-Style permissions are commonly used still today in a lot of UNIX and UNIX-like systems. Each file has an owner and a group, and a set of permissions that can be assigned for the owner, the group, and for everyone. There are three basic permissions: read, write, and execute (run as a program). The permissions are represented using 10 bits, where a 1 indicates true and a 0 indicates false. The first bit is the directory bit and indicates if the file being examined is actually a directory. The next three bits are the read, write, and execute bits for the owner, followed by the read, write, and execute bits for the group, and finally the read, write, and execute bits for everyone.

Effective permissions are determined by the user: the owner of the file gets the owner permissions even if different permissions are assigned to the group or everyone. Precedence goes from left to right: owner takes precedence over the group; group takes precedence over the permissions for everyone.

The permissions can be shown to the screen in a human-readable format which is ten characters long. 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, and `x` to indicate execute access.

Example: permissions of `-rwxr-----` indicate that a file is not a directory; the owner can read, write, and execute; other members of the group can read it only, and everyone else has no access to the file (cannot read, write, or execute).

Permissions can also be written in octal (base 8) where r = 4, w = 2, and x = 1. To get the octal representation, start with 0, and then add the value of the permissions that are present, using zero where permissions are absent. You might then have a permission that reads `750` - meaning that the owner has read, write and execute access (0 + 4 + 2 + 1 = 7); group members have read and execute access (0 + 4 + 0 + 1 = 5); everyone else has no access to it at all.

There are some more details like what the permissions mean on directories, and some advanced topics like `setuid`, `setgid`, and "sticky bit", but we will not cover them in this course.

The obvious shortcoming of this approach is that it is very coarse-grained: there are only three groups for whom we can specify permissions. Within Linux and other similar systems there is a trend now towards using SELinux (Security Enhanced Linux) which is an Access Control List system. For the moment we'll leave off the discussion of access control lists and just proceed forward.

References

[Sal] Peter Jay Salzman. Reading And Writing Files in C. Online; accessed 4-June-2018. URL: `http://www.dirac.org/linux/programming/tutorials/files_in_C/`.

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