Chapter 12

File Systems

We are not required to have file systems in our operating system, but it is a very usable abstraction, and it often plays a central part of many operating systems, especially UNIX-like operating systems. Before we start the process of supporting multiple processes and system calls we might want to consider implementing a simple file system.

Why a File System?

How do we specify what programs to run in our OS? Which is the first program to run? How do programs output data or read input?

In UNIX-like systems, with their almost-everything-is-a-file convention, these problems are solved by the file system. (It might also be interesting to read a bit about the Plan 9 project, which takes this idea one step further.)

A Simple Read-Only File System

The simplest file system might be what we already have - one file, existing only in RAM, loaded by GRUB before the kernel starts. When the kernel and operating system grows this is probably too limiting.

A file system that is slightly more advanced than just the bits of one file is a file with metadata. The metadata can describe the type of the file, the size of the file and so on. A utility program can be created that runs at build time, adding this metadata to a file. This way, a "file system in a file" can be constructed by concatenating several files with metadata into one large file. The result of this technique is a read-only file system that resides in memory (once GRUB has loaded the file).

The program constructing the file system can traverse a directory on the host system and add all subdirectories and files as part of the target file system. Each object in the file system (directory or file) can consist of a header and a body, where the body of a file is the actual file and the body of a directory is a list of entries - names and "addresses" of other files and directories.

Each object in this file system will become contiguous, so they will be easy to read from memory for the kernel. All objects will also have a fixed size (except for the last one, which can grow), therefore it is difficult to add new files or modify existing ones.

Inodes and Writable File Systems

When the need for a writable file system arises, then it is a good idea to look into the concept of an *inode*. See the section "Further Reading" for recommended reading.

A Virtual File System

What abstraction should be used for reading and writing to devices such as the screen and the keyboard?

A virtual file system (VFS) creates an abstraction on top of the concrete file systems. A VFS mainly supplies the path system and file hierarchy, it delegates operations on files to the underlying file systems. The original paper on VFS is succinct and well worth a read. See the section "Further Reading" for a reference.

With a VFS we could mount a special file system on the path `/dev`. This file system would handle all devices such as keyboards and the console. However, one could also take the traditional UNIX approach, with major/minor device numbers and `mknod` to create special files for devices. Which approach you think is the most appropriate is up to you, there is no right or wrong when building abstraction layers (although some abstractions turn out way more useful than others).

Further Reading

- The ideas behind the Plan 9 operating systems is worth taking a look at: http://plan9.bell-labs.com/plan9/index.html
- Wikipedia's page on inodes: http://en.wikipedia.org/wiki/Inode and the inode pointer structure: http://en.wikipedia.org/wiki/Inode_pointer_structure.
- The original paper on the concept of vnodes and a virtual file system is quite interesting: http://www.arl.wustl.edu/~fredk/Courses/cs523/fall01/Papers/kleiman86vnodes.pdf
- Poul-Henning Kamp discusses the idea of a special file system for `/dev` in http://static.usenix.org/publications/library/proceedings/bsdcon02/full_papers/kamp/kamp_html/index.html