File-System Internals

Practice Exercises

15.1 Explain how the VFS layer allows an operating system to support multiple types of file systems easily.

Answer:
VFS introduces a layer of indirection in the file system implementation. In many ways, it is similar to object-oriented programming techniques. System calls can be made generically (independent of file system type). Each file system type provides its function calls and data structures to the VFS layer. A system call is translated into the proper specific functions for the target file system at the VFS layer. The calling program has no file-system-specific code, and the upper levels of the system call structures likewise are file system-independent. The translation at the VFS layer turns these generic calls into file-system-specific operations.

15.2 Why have more than one file system type on a given system?

Answer:
File systems can be designed and implemented with specific uses in mind, and optimized for those uses. Consider a virtual memory file system vs. a secondary storage file system. The memory-based one need not concern itself with fragmentation, or persisting data structures in the face of power loss. There are also special-purpose file systems like the procfs file system, designed to give the convenient file system interface to system aspects like the process name space and process resource use.

15.3 On a Unix or Linux system that implements the procfs file system, determine how to use the procfs interface to explore the process name space. What aspects of processes can be viewed via this interface? How would the same information be gathered on a system lacking the procfs file system?

Answer:

On systems containing the procfs psuedo-filesystem, details vary but generally the file system is mounted at /proc, and exploring it with file system commands can reveal the following:

- Each process is represented by its processID, so counting them reveals the number of processes in the system.
- Within each directory under the processID, details of the process state such as its current working directory, command line used to start the process, priority information, memory use information, lock information, open file information, and so on.
- Some procfs also provide interfaces to other kernel structures such as DMA structures, device lists, file system lists and so on.

See the `proc(5)` manual page for details on a given system.

Without procfs, to provide the same information, separate system calls per information type is used, or the ability to open the kernel memory space through `/dev/kmem` or `/dev/sys` is provided. Then programs using these interfaces need to be written to extract the data and present it in human-understandable form. See http://osxbook.com/book/bonus/ancient/procfs for a nice exploration of using procfs vs. not having it available.

15.4 Why do some systems integrate mounted file systems into the root file system naming structure, while others use a separate naming method for mounted file systems?

Answer: As with many aspects of operating system design, choices can be arbitrary or based on some small implementation detail and then exist long after any justify reason. Generally regarding file system mounting, integration with the root file system naming has proven to be more flexible and useful and separate mount point naming and therefore prevails on most file systems.

15.5 Given a remote file access facility such as `ftp`, why were remote file systems like NFS created?

Answer: Users of computer systems value ease-of-use in most cases. The more general purpose, and more widely used and operating system is, the more seamless its operation should be. In the case of remote file access, it is easier for users to use an familiar facility (such as a file system interface) rather than separate commands. And because file systems are tried and true, well integrated, and full featured, existing tools, scripts, and use cases can apply to remote file systems just as local file systems by using a file system interface for remote file access.