OS - Lab 1

Thomas Magnusson

11 September 2018

1 What are the advantages and disadvantages of using the same system call interface for manipulating both files and devices?

An interface is defined to abstract details that the programmer calling it need not comprehend. Using a singular system call interface for two portions of the computer sounds like it might be useful. Creating a common programmatic vocabulary helps the programmer using the interface understand the underlying behavior more quickly. That being said, if the interface is not carefully designed, and if the contract that the interface purports is flawed, a singular interface is extremely dangerous. Breaking up the interface into smaller, more specialized ones allows for programmers to more clearly and explicitly understand the context and nuances of their calls, because they are explicitly dealing with files. "Write" makes much more sense in the context of a file rather than a device... although one might construe that to mean to send data over to the device itself. The fact that the behavior is ambiguous means the interface is dangerous.

Splitting the interface up might be problematic because it requires the interface's creator to write more code, perhaps duplicating portions of similar but slightly different behavior. "write" is a good example: in the case of files, it probably means to consign data to a specified file; for a device, it might mean consign data to the device so that it can process the data. A music streaming application might "write" data to the headphone jack, although the more fitting terminology might be "stream." If the interface were to be broken up into "stream" and "write," the creator might have to duplicate lots of code in order to do slightly similar things.

The authors could employ good object oriented practices, and create abstractions over their interfaces to create "skeleton" interfaces that would deal with each context appropriately. That might call an internal library that might have a more unified architecture, fit for "writing" to any place rather than just a file or a device.

2 Would it be possible for the user to develop a new command interpreter using the system call interface provide by the operating system? How?

I think it would be possible to develop a new command interpreter using the system call interface provided by the operating system as long as the operating system had the appropriate system call interfaces available. The only special permissions perhaps needed would be the ability to kill and load processes, which the kernel might grant to an authenticated command interpreter. It might be a little more difficult for a third party to access the kernel's kill and load system calls given an operating system with robust security settings, but entirely possible if access were given. File system access would also have to be a security consideration. As long as the program has the appropriately bestowed credentials, and access to the kernel, it would be a matter of using the keyboard interrupts to generate input and output, and arbitrary code execution to do whatever the command interpreter would need to do. Interfacing with devices and their drivers would require knowledge of those kinds of interrupts. The same goes for file I/O.

It might be easier to create a command interpreter using existing system call APIs in languages that have built-in support for system call interrupts. Most modern languages, such as Java, have robust APIs for obtaining access to System.out and System.in, as well as file IO. Though this is an abstraction over the raw system call interface, it would probably make developing a whole lot smoother and faster.