Operating Systems II Writing 2

9th May 2018

Prepared by
Corwin Perren

1 Introduction

For every operating system a way is needed to abstract complex physical hardware into easier to process logical items. All of these pieces of hardware perform some kind of I/O operation, whether it be memory, a graphics card, networking card, hard drive, or USB device. Abstraction layers are needed so that different providers for different hardware can conform to a standard such that developers don't need to learn a new API for every individual piece of hardware. This means that a keyboard made my Logitech will show up the same as a keyboard made by Microsoft. All the developer has to think about it interacting with a generic keyboard device and the driver abstracts away any of the hardware differences. In the following sections we'll take a look at how Windows, Linux, and FreeBSD handle their I/O as well as similarities and differences between the different operating systems.

2 Individual OS IO Info

2.1 Windows

In Windows the most common data structure for interacting with drivers and hardware devices are IRPs or I/O request packets. These IRPs are created by the I/O manager built into windows that takes in a request made by a call to a DLL, such as request to send a network packet. The IRP stores useful header information about the I/O operation such as whether the request is synchronous or not, as well as the interface to the low level driver. Once this IRP is made, it exists until access to the resource is no longer needed. Due to the complexity of some I/O devices, it's possible for the IRPs to point to another device driver rather than a hardware device. For example, if using a pen drive that is encrypted, the first would be the abstracted IRP that sees the resource as a file, but that IRP would point to the cryptography driver which would transparently provide block level encryption / decryption from a driver level. For options when it comes to encryption, the most common and most used ones are available such as RSA, AES. This largely is the case as hardware on CPUs is leveraged to do faster crypto calculations than if that processing were done without the extra hardware.

2.2 Linux & FreeBSD

Linux and FreeBSD both run kernels that share historical roots in Unix, meaning that the core of how I/O is handled is the same minus different naming conventions in the raw code itself. The most interesting feature of I/O on Unix based kernels is that everything is treated like a "file", or more accurately, accessed via file descriptors. This means that you can access hardware under an absolute path on the filesystem, but instead of reading and writing to a place on disk, the bytes read and written are actually massaged through kernel drivers to actually interact with the hardware. Reading and writing on Unix kernels in done in either structured (block) or unstructured (character) device modes. For block read/writes, an entire "block" (4kb for example) is sent or received at any given time. Character mode is just that, reading or writing a single character at a time. When it comes to cryptography, all the most common ones are available via the kernel such as RSA or AES. This largely is the case as hardware on CPUs is leveraged to do faster crypto calculations than if that processing were done without the extra hardware.

3 IO Comparisons

3.1 Similarities

Due to the intense similarities between Linux and Unix, I'll only be comparing Unix-based vs. Windows here. The biggest similarity between the two operating systems is that there is abstraction from the hardware. In older computers, there

was very little abstraction which made writing code, or an operating system for that matter very difficult. Everything had to be custom written to fit the exact configuration of the computer as it was on a user's table. By putting a layer of abstraction above this hardware, a single operating system or single source file could be easily ported from one computer (or even variant in the case of Linux vs Unix) to another, at the expense of processor time to manage a driver subsystem. Without this layer, computers and operating systems as we know them would be very different than what they actually are today.

3.2 Differences

Beyond the simple fact that abstraction exists, there's very little that's similar between the way Windows handles IO vs Unix-based systems. Unix treats everything like a file, to the point that you can directly read/write to memory using /dev/mem if you have elevated privileges. Trying to do something like that on Windows would be incredibly difficult as Windows tries to hide that driver manager layer and doesn't provide easy access to hardware directly. One thing this means is that writing drivers for Linux is significantly easier and less time consuming than on Windows, as there is less overhead management of the drivers. Also, the open system source code also helps.