CS444: OPERATING SYSTEMS II

I/O and Cryptography

Author:
Mark BEREZA

Instructor:
D. Kevin MCGRATH

November 15, 2017
Fall Term

Contents

1 Introduction

Since modern computers are ridden with various I/O devices, it is critical that any modern operating system provides a way to interface with these devices by reading data from them or writing data to them. In particular, it common to create an common interface between user applications and the drivers for the I/O devices in question so that programs can access data from I/O devices without needing to know anything about how reading or writing is handled for that particular device. Furthermore, the kernel may provide developers with various built-in data structures and algorithms to aid in faciliation of these services. Finally, many modern operating systems have begun to support cryptographic services for I/O operations to allow users to secure their data by performing encryption before writing and performing a corresponding decryption before reading. This paper will analyze how FreeBSD and Windows implement these various services and compare them to their corresponding Linux implementations and attempt to justify any differences.

2 I/O Abstraction

Devices on hardware I/O busses are referenced in the Linux kernel by their I/O ports, which simply serve as numerical IDs. The Linux kernel provides auxiliary functions to simplify accessing I/O ports for reading and writing [1]. The kernel keeps track of the mapping between specific I/O devices and I/O ports through structures called resources [1]. These structures contain a range of I/O port addresses and are organized into a tree-like structure to facilitate devices that are split into subsections. Like most things in Linux, I/O devices are represented as files (called device files) that usually live in /dev/. The advantage here is that the same system calls used to interact with regular files (read and write) can also be used to access I/O devices. In order to make these generic system calls provide the specific functionality in the device, the Linux kernel employs a Virtual File System (VFS). The VFS serves as the middleman between user file operations and the device drivers; it converts system calls into the appropriate device functions [1].

FreeBSD, being also UNIX-based, is similar in many ways. FreeBSD also does not distinguish much between regular files and I/O device files (called special files in FreeBSD) at the user level. In fact, all I/O devices are abstracted to function as a simple stream of bytes, also known as an I/O stream. I/O streams are referenced using descriptors and these descriptors are also used to refer to everything from files, pipes, sockets, cryptographic hardware, shared memory, and more [2]. Like Linux, these special files can be accessed using the read/write system calls used to access conventional files. Unlike Linux, where the files point directly to I/O devices, all files in FreeBSD (both regular and special) point first to vnodes, which is part of FreeBSD's virtualaziation of its file system. Vnodes are used to describe various file system implementations in a generic way. That being said, Linux has its own version of vnodes known as generic inodes so there differences here are minimal.

Windows utilizes a service known as the I/O manager to serve as the glue between user applications, kernel services, and device drivers. This manager defines the model for I/O requests that are delivered to device drivers in a Windows system [3]. The specific model used by the manager is an object called an I/O request packet, or IRP, which completely describes the I/O request [3]. The I/O manager constructs these packets for each I/O operation and passes them to device drivers which then perform the requested I/O operations. The drivers then pass back the IRPs to the I/O manager, which will dispose of the them if it determines that the request has been completed. Moreover, the I/O manager exports common functionality to drivers, such as access to other drivers, I/O request buffers, timeouts, and filesystem data reporting to simplify driver development. This relationship between drivers and the kernel is quite different than Linux's, and is likely a result of Windows outsourcing driver development to device manufacturers using the Windows Driver Model (WDM), whereas the Linux kernel is distributed with the drivers included. This difference itself is a clear result of Windows being a closed-source microkernel, while Linux employs an open-source monolithic kernel.

Windows threads perform their I/O operations on virtual files in a fashinon very similar to how Linux and FreeBSD treat I/O devices as files. This similarity is likely a consequence of the developers of all three systems seeking to employ abstraction to simplify things for users and developers. That is, because applications either do not want to worry about or simply cannot know ahead of time the features of I/O hardware devices, a common abstract model for data reading/writing is instead provided, and that model is the file. That being said, Windows does not take the "devices are files" analogy as far as Linux does. For example, Windows does not map devices to a path in the filesystem.

3 Types of I/O Devices

Linux organizes its I/O devices intro three categories:

1) Block devices are devices in which input/output operations can operate on random locations and operations are performed on integral numbers of constant-sized chunks known as blocks. Examples of I/O devices that would categorized as block devices as RAM, hard drives, SSDs, flash memory, floppy disks, etc. Linux performs block I/O by first passing the read/write operation to its virtual file system, which invokes the corresponding VFS function with a file descriptor and file offset. The VFS function first tries to access the data in question from the disk cache. If the disk must be accessed, the kernel determines the physical location on the device to be read/written. The kernel the uses the generic block layer to issue the necessary read/write operations to fulfill the I/O request. Because of virtual memory, the same request may access non-adjacent chunks of physical memory and thus require multiple operations. The operations are sent to the I/O scheduler, which adds, sorts, delays, merges, and dispatches the requests based on some algorithm to attempt to reduce seeking, increase throughput, prevent starvation, minimize latency, ensure fairness or some combination of these desired qualities. Finally, the requests are passed to the device drivers which actually perform the operation(s).
2) Character devices are devies which only allow the system to read/write sequentially. Linux in many ways treats character devices as streams of bytes. Examples include keyboards, mice, and monitors. Character I/O is often easier than block I/O because buffering strategies and disk caches are not needed.
3) Network devices, seperated likely because they do not map intuitively to the file abstraction, are outside the scope of this paper.

FreeBSD, on the other hand, organizes its I/O devices in these categories:

1) Filesystem sead/write data to and from kernel buffers and are used to access various filesystems, as the name would imply.
2) Character devices in FreeBSD are virtually identical to their Linux counterparts, a result of their Unix-y heritage. However, while character devices in Linux map fairly well to byte streams, character devices in FreeBSD are used to access disks and other organized data, as well. This is known as a unstructured or raw interface to a block-oriented device. I/O operations made to a disk through this interface do not go through the filesystem or the page cache.

FreeBSD used to have block devices, but these were eliminated because few applications needed them and the character interface improved throughput. This difference from Linux is likely because Linux must support vastly more I/O devices and can run on significantly more systems and thus the flexibility provided by having interface for both block and character devices likely outweighs the simplicity afforded to FreeBSD by removing the former.

3) Socket devices in FreeBSD are essentially Linux's network devices and are also outside the scope of this paper.

Windows organizes its devices not based on how data is read/written, but rather based on what layer the device driver lives in:

1) Function devices are the fundamental devices and each driver must implement a function driver at a minimum. It provides the operational interface for a device, including reading/writing.
2) Bus devices represent a logical or physical bus [3], such as PCMCIA, PCI, USB, etc. The bus driver will inform the Plug and Play manager of devices attached to its bus.
3) Filter devices are essentially helper devices that augment the functionality of function and bus devices in various ways. An example of such a device would be a keylogger.

This difference in organization is likely due to Windows' microkernel approach which moves a lot of the complexity away from the core of the system and into these third-party created drivers. By creating a model that organizes different devices (and their drivers) based on where they fit in the stack, Windows can simply 'plug in' these complex drivers in their core system instead of baking in generalized functionality.

4 CRYPTOGRAPHY

All three of these operating system also implement some interface for utilizing hardware-assisted cryptography, which has become more common in modern processors. This allows users to add encryption/decryption to their standard I/O operations to secure transmitted information from prying eyes. The Linux kernel has a built-in cryptography API, which provides calls for symmetric ciphers, AEAD ciphers, message digest, random number generation, and a user-space interface. Similarly, FreeBSD has ported the OpenBSD Cryptographic Framework (OCF), which is also an in-kernel API to cryptographic resources. This framework supports just about every common block cipher. The reason FreeBSD choose to port OpenBSD's framework instead of implementing its own like Linux is likely because FreeBSD has less support than the Linux kernel and because FreeBSD's shared heritage with OpenBSD made leveraging the existing framework a lot more attractive than writing a new one from scratch. Windows also has its own cryptography API, like Linux, but it is not baked into the kernel. Instead, this API instead calls upon various Cryptography Service Providers (CSPs), one of which comes included with the operating system, to allow user applications to encrypt their data [4]. Each CSP has its own implementation of the cryptography API layer, with some having stronger encryption algorithms than others. This difference is a result of Windows' closed-source nature, forcing it to move the cryptography layer away from the core of the system to grant users flexibility in what cryptography service they use without granting access to the system source code. Additonally, moving this complexity strengthens Windows' identity as a microkernel.

References

[1] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, 3rd ed. O'Reilly, 2005.
[2] M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, *The Design and Implementation of the FreeBSD Operating System*, 2nd ed. Addison-Wesley Professional, 2014.
[3] M. Russinovich, D. A. Solomon, and A. Ionescu, *Windwos Internals Part 2*, 6th ed. Redmond, Washington: Microsoft Press, 2012.
[4] "The cryptography api, or how to keep a secret," https://msdn.microsoft.com/en-us/library/ms867086.aspx, (Accessed on 11/10/2017).