Differences and similarities in I/O and provided functionalities between the Linux, Windows, and FreeBSD Operating Systems

Alex Hoffer

CS 444
May 2017
Spring 2017

Contents

1 Windows

1.1 I/O

1.1.1 Similarities to Linux

Both operating systems have a design goal to provide what [5] calls an "abstraction of devices", meaning a universal set of software tools by which the programmer can manage I/O functions. Additionally, both operating systems implement, in their own way, a *Hardware Abstraction Layer* (HAL), which is a generalized interface that allows programs to directly manipulate device hardware [6]. Note, with the HAL as a perfect example, both operating systems' inclusion of abstraction in their design decisions. Linux and Windows both desire to be able to manage the I/O requests of a large variety of devices and file systems, since they are intended to be commercially used operating systems. While they share this design principle, the way in which they approach this abstraction is the source of a lot of differences between the two systems.

1.1.2 Differences from Linux

A key difference between Windows and Linux that explains further differences is the fact that Windows and its kernel are separated and are only combined through the use of calls to the Application Binary Interface (ABI) [2], while Linux's device drivers are located within the kernel itself and not relying on calls to the ABI [3]. Programmatically speaking, in Windows, files and devices are managed as *objects* in the object-oriented paradigm, while in Linux files and devices are managed through the use of *file descriptors*, further exemplifying the classic UNIX saying that "Everything is a file". Additionally, Windows' I/O has three device driver layers that each I/O request can be subjected to (though a request can be handled by merely one of these layers). These three layers are *filter, function,* and *bus* [5]. Consider the following block of Microsoft C code courtesy of Microsoft's Github that initializes a function driver for a Bluetooth device:

```
NTSTATUS
DriverDeviceAdd(
    IN  WDFDRIVER        _Driver,
    IN  PWDFDEVICE_INIT  _DeviceInit
    )
```

On the other hand, Linux's drivers are defined as being *block*, *character*, or *network*, and an I/O request can't be processed by all three. Since Windows' kernel doesn't have any device driver development, it instead relies on the *Windows Driver Model* (WDM) to provide device drivers. When a piece of software in Windows requests an I/O operation, the Windows kernel dispatches the request to its *I/O Manager*, which translates the request into an *I/O Request Packet*, or IRP, which is then sent to the device driver layers. The function layer consists of the predominant drivers that map programming interfaces to specific devices, the bus layer helps out device hosting bus controllers, and the filter layer offers extra IRP processing. Meanwhile, in LinuxLand, these device drivers are wildly different: the character device driver type handles simple devices that can be read one byte at a time, the block device driver type handles complex devices whose requests come in the form of blocks of data, and networking options are there for moving data to and from a network.

1.2 Provided functionality

1.2.1 Similarities to Linux

In both operating systems, their respective *Application Programming Interfaces* (APIs) are written to respond to *events*, which is when a user wants something to happen or the device wants to pass a message to the I/O manager. Windows has three provided functionalities for programmers

to allow communication between Userland and I/O drivers: *buffered I/O*, *direct I/O*, and *memory mapping* [2]. Linux also provides these same three functionalities [3]. Both operating systems also provide system-defined data structures, algorithms, and cryptographic protocols, though there is some differences in which options are provided. For example, Linux and Windows kernels both have linked lists, though Windows provides both singly and double linked lists, while Linux only provides a doubly linked circular list by default. The Windows linked lists are included in the file "Ntdef.h", while the Linux kernel's list is included in the "list.h" file. In terms of cryptography, both use things like *access control lists* (ACLs), PKI, and such to provide basic security.

1.2.2 Differences from Linux

Windows linked lists come in either doubly linked or singularly linked varieties, while Linux only offers doubly linked lists by default, which means you must use a Linux patch file to change this functionality in the Linux kernel. Naturally, there are some differences in the two operating systems' APIs for when it comes to manipulating lists. For example, Windows offers routines like *RemoveHeadList()*, whose function signature is listed below, which can perform convenient operations such as removing the first element of a list.

```
PLIST_ENTRY RemoveHeadList(
  _Inout_ PLIST_ENTRY ListHead
);
```

The same removal of a list head can only be completed in Linux using its "List.h" file by doing the following:

```
if (!list_empty(myQueue->queue))
    list_del(myQueue->queue.next);
```

As we can see, there are different routines provided in the two different systems for performing basic data structure manipulation. In addition to differences in data structure implementation between the two kernels, there is a lot of difference in how the two systems handle encryption. Some of these differences are difficult to compare since Linux is open source and Windows is proprietary. For example, while both kernels provide things like access control lists to institute a level of security, Windows also uses security measures such as the *Microsoft crypto application programming interface* [2]. Therein lies a fundamental and interesting difference between Linux and Windows that is microcosmic of a common theme expressed throughout my writing this quarter: while we can look directly at how Linux implements things like security and critize or laud them, Windows' source code is on lock down, so all we can do is look at helper routines and things of the like in order to assess how they might be used.

2 FreeBSD

2.1 I/O

2.1.1 Similarities to Linux

Since both operating systems are derived from UNIX, there are of course plenty of similarities between FreeBSD and Linux in terms of how I/O is handled and which functionalities are provided. For example, both FreeBSD and Linux specify that devices should be accessed through device-nodes, which are a special type of file [4] [3]. Another similarity is that both FreeBSD and Linux have character and network device driver types. These device driver types are essentially the same in both of these operating systems, though FreeBSD emphasizes the character device driver type, while Linux emphasizes the block device driver type (described more in the "Differences from Linux" piece of this section).

2.1.2 Differences from Linux

The principal difference between these two UNIX-derived systems is that in FreeBSD, the block device driver type is not present. This is manifested in the fact that FreeBSD has two device driver types, which are character and network. Meanwhile, Linux has three device driver types: character, block, and network. On the surface, this may seem like a superficial difference, but in fact, this fundamentally changes the way these two systems handle I/O requests. FreeBSD treats the character device driver type as its cardinal type [1]. Unlike Linux, where the character device driver type is merely used to handle the requests of simple devices where I/O can be managed one byte at a time, FreeBSD handles all devices using this one-byte-at-a-time strategy. Meanwhile, in Linuxland, the block device driver type handles devices where requests are represented by a block of memory (hence the name). Oddly enough, the FreeBSD organization's programming handbook really looks down on the block device driver type, calling this disk device type "unusable, or at least dangerously unreliable", due to the kernel's caching of its operations.

2.2 Provided functionality

2.2.1 Similarities to Linux

Both Linux and FreeBSD offer a lot of built-in functionalities for programmers to use when doing systems programming. For example, both have convenient and easy-to-use methods that allocate memory within the kernel. Here is the function signature for allocating memory in the FreeBSD kernel:

```
void* malloc(unsigned long size, struct malloc_type *type, int flags);
```

Meanwhile, in Linux, here is the simple method we use to allocate memory in the kernel: [3]

```
void * kmalloc(size_t size, int flags);
```

As we can see, these two routines to allocate memory are very similar. In fact, FreeBSD and Linux provided functionalities universally tend to be somewhat close, possibly because of their shared inheritance from UNIX. Both FreeBSD and Linux have data structures for kernel usage including lists, queues, and trees, for instance.

2.2.2 Differences from Linux

The primary difference between what these two operating systems offer in terms of provided functionalities, in my opinion, is found in how they implement cryptography, which has important ramifications for their respective securities. Specifically, FreeBSD uses the *Crypto* interface in its kernel [1], while Linux uses its own *Crypto* interface in its kernel [3]. Original, right? Regardless, Linux's Crypto interface is far more robust. Though FreeBSD's interface focuses solely on the user space, Linux's focuses on both the user space interfacec and the programming interface, and has loads of different cryptographic protocols built into its API. The breadth of Linux's different crypotgraphic protocols may be due to its popularity. Let us take the topic of built-in Cipher algorithms, for instance. FreeBSD includes the following functions that implement Cipher algorithms for security purposes: CRYPTO_AES_CBC, CRYPTO_AES_NIST_GCM_16, CRYPTO_AES_ICM, and CRYPTO_AES. Meanwhile, documenting the number of Cipher algorithms that Linux offers here would be silly: there are far, far too many. In fact, Linux offers a multitude of different cryptographic protocols by which the programmer can pretty much set up the security of his system to his fancy. It is in the vastness of Linux's provided security functionalities compared to the relative sparseness of FreeBSD's provided security functionalities where we find the most profound differences, I think.

References

[1] Nathan Boeger and Mana Tominaga. Freebsd system programming, 2001.

[2] Johnson M. Hart. *Windows System Programming (3rd Edition)*. Addison-Wesley Professional, 2004.

[3] Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, San Francisco, CA, USA, 1st edition, 2010.

[4] Marshall Kirk. McKusick, George V. Neville-Neil, and Robert N. M. Watson. *The design and implementation of the FreeBSD operating system*. Addison-Wesley/Pearson, 2015.

[5] Mark E. Russinovich, Alex Ionescu, and David A. Solomon. *Windows internals, Part 2*. Microsoft, 2012.

[6] Mark E. Russinovich, David A. Solomon, and Alex Ionescu. *Windows internals*. Microsoft Press, 2012.