Comparing and Contrasting Essential System Operations in the Linux, Windows, and FreeBSD Operating Systems

Alex Hoffer

CS 444
June 2017
Spring 2017

Contents

1 Introduction

Computer operating systems have a rich tapestry of history. When we observe the development of the modern operating systems from the ones responsible for handling giant, room-filling computers (performing what we could consider today as ho-hum computations) to the current climate of tiny processors coordinating and tackling lofty challenges in our laptops and mobile devices, it is important as computer scientists to understand which components of an operating system emerge as being essential to the proper functioning of a computer. Through this study, we can see that processes and threads, I/O and provided functionality, and memory management are three operations of an operating system that are crucial to making a computer that is intended for widespread human usage work correctly. While being able to pinpoint these three features as important operating systems topics is useful, it is only when we see how they can become reality that things become truly interesting. It is the purpose of this paper to elucidate this by explaining how Linux, Windows, and FreeBSD each implement these core functions in their operating systems. Along the way, we will be able to see where they differ in such implementations and where they run parallel to each other.

2 Processes and Threads

A *process* is an "instance of an executing [computer] program" [1]. Each process is given its own memory space to perform operations within. One central task of an operating system is to synchronize process execution, as well as to decide which process gets ran when (this is called *scheduling*). A thread is similar to a process in that it is subject to similar synchronization and scheduling constraints, though threads exist within the process space and each thread running within such a space share resources. Processes and threads are essential to an operating system because they greatly increase the amount of work being done by providing an illusion of multiple tasks being completed at once.

3 I/O and Provided Functionality

Input/Output, or I/O, operations are crucial to an operating system because a computer cannot perform particularly interesting computations without receiving data from an exterior source or writing data to an exterior source. To take a lighthearted example, try to think of a way that I could type this LaTeX file and generate a PDF from it without the operating system (at the very least) receiving data from my keyboard. Provided functionality is important to the programmers responsible for writing programs that must utilize kernel operations in some way. Without provided functionalities such as kernel data structures, for example, how would we conveniently store and access memory pages?

4 Memory Management

Memory management is a crucial aspect of any operating system. Memory management's primary functions are to provide virtual memory spaces to different processes, properly map these virtual memory spaces to their corresponding physical addresses, and allocate and de-allocate pages of virtual memory and assign such pages safely and responsibly to processes as they request them. Without the ability to manage memory, an operating system would descend quickly into chaos.

5 Conclusion

Through our research of the Linux, Windows, and FreeBSD operating systems, we have seen that there are a multitude of different ways programmers can implement processes and threads, I/O and provided functionality, and memory management. Look no further than the difference in how the parent-child process relationship is handled in Windows vs. Linux or how FreeBSD lacks the block device driver type that Linux relies heavily on. These examples make it clear that there is no one simple recipe to produce a working operating system, and that design decisions and tradeoffs are as present in kernel development as they are in any other worldly subject. But programmers who desire to write good code in all of these three operating systems need not to worry, because there are a great number of similarities between them as well. Consider, for example, how similar thread synchronization constructs are across the three systems, or how similarly the data structures within each of the three kernels resemble each other. It is only with the knowledge of the similarities and differences of how these three essential features are implemented in popular operating systems that we can confidently call ourselves programmers.

References

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