Writing Assignment Two

Brennan Giles

CS 444 Oregon State University

Fall 2018

Abstract

Computers at their core are created with the idea that for every input a user provides, the device will give an output. I/O in operating systems is the portal that allows users to fully utilize the benefits of their machine and control how they accomplish tasks, and without I/O there is no reason to use a computer at all. In this paper, we will investigate how I/O works on the worlds most popular Operating Systems and how each handles I/O to the users desires.

1 WINDOWS I/O

Windows is very robust in the way it handles I/O, and offers many features designed to streamline I/O for the user. Windows allows something called asynchronous I/O, which means that instead of waiting for an I/O access to complete it allows other processes to continue until the I/O requires attention again. For example, reading a disk during an I/O operation can take several milliseconds. During this time, the processor could have performed millions of instruction processes, which with synchronous I/O would all go to waste. Asynchronous I/O works by asking job threads to let the kernel know every time it asks for a read or write request. If the threads request is deemed large enough, then the kernel allows the thread to process other jobs until the I/O operation is complete. Upon completion the job is interrupted and the thread resumes its previous task. For extremely quick I/O operations, the kernel will not allow it to perform asynchronous processing and instead have it wait. [1]

Windows has something called the I/O manager, which presents a consistent interface to all kernel-mode drivers. All I/O requests to drivers are sent as I/O request packets or IRPs [1]. Another key feature of windows I/O is that its operations are layered, meaning that the I/O manager exports I/O system services and user-mod protected subsystems call to carry out I/O operations on behalf of their applications and/or end users. The I/O manager intercepts these calls, sets up one or more IRPs, and routes them through possibly layered drivers to physical devices [1]. The I/O manager doesnt stop there; it can also define a set of standard routines that drivers can support. This way, all drivers follow a similar implementation model despite the differences between the peripherals they apply to. Lastly, Drivers are object-based. Drivers, their devices, and system hardware are represented as objects. This allows the I/O manager to then export kernel-mode support routines that drivers can call to get work done by manipulating those objects [1]. The I/O manager also has the ability to handle and respond to power requests from various peripherals.

Lastly, the Windows I/O system uses priorities to establish which I/O operations should be done the soonest. It has five categories: Critical, high, normal, low, and very low. It only uses Critical, normal, and very low, with the other two reserved for some potential new system down the road. Critical handles low memory situations where room in RAM needs to immediately be made (so that data is shoved onto a different memory device), normal is used for normal application I/O, and very low is used for scheduled tasks, background processes, and hard drive functions such as defragmenting. The task scheduler only sets I/O priority for tasks that have the default priority as very low. [2]

In summary, Windows uses an I/O manager to oversee all I/O operations, choose when to use asynchronous mode, and outsource processes.

2 LINUX / FREEBSD I/O

Since Linux and Free BSD are both Unix based Operating systems, I will explain how both handle I/O at the same time since they are extremely similar. Each use something called Block devices, which provide the main interface to all disk devices in a system. They also allow random access and interpretation of memory such as CDs, hard drives, etc. The main advantage of using block devices is that it allows access to random locations on the device [3]. A Block represents the unit with which the kernel performs I/O, and when a block is read in to memory it is stored in a buffer. A list of

requests for each block device is stored and scheduled by a unidirectional elevator algorithm, and each request is not removed from that scheduled list until the input AND output is fully complete. [4]

The kernel itself does no preprocessing of I/O requests and for the most part hands the request off to the asked for device and doesnt deal with it beyond that. This means that the kernel cannot back up and investigate the request any further after it is handed off. The only exception to this are the special tty struct structures, which provide buffering and flow control on the data stream from the terminal device and feeds that data to a line discipline[4]. A line discipline is an interpreter for the information from the terminal device, and decides which processs data should be remembered or not.

Overall, the simplicity of the Unix handling of I/O gives it a huge advantage and very streamlined approach at I/O handling. The only problem is that sometimes it can be too streamlined and lack safety nets for data.

3 COMPARISON

FreeBSD and Linux are almost exactly the same in how they handle I/O. Both use Blocks that allow access randomly or sequentially, and both utilize an I/O stream to receive input and interpret it. Both also use the three descriptors of standard input, output, and error to handle processes as well. Both systems are extremely lightweight and efficient, which results in tailor made and exceptional user experiences when it comes to interaction. Its one major weakness is that while it does allow for asynchronous processing thorugh threading, it struggles with large amounts of concurrent I/O. Each thread requires a stack, consuming large amounts of memory and the threads that can be used on any process are often limited.

Windows on the other hand doesnt rely on the cyclic nature of Unix data structures, and instead has a central overlord, the I/O manager that handles the majority of Input and output operations. The I/O manager provides an efficient and effective method of handling I/O streams, while simultaneously providing better safety nets for data being transferred. Furthermore, Windows uses a priority system that enables the effective use of I/O to clear up room on the RAM in cases of low memory or other special circumstances. This results in a more stable environment, especially under memory intensive situations.

In conclusion, All operating systems referenced in this paper have exceptionally well thought out systems of handling I/O that seek to provide the ultimate bridge between user and machine.

REFERENCES

[1] Windows Hardware Dev Center. Overview of the Windows I/O model June 15 2017.
https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/overview-of-the-windows-i-o-model

[2] Mark Russinovich, Alex Ionescu, David Solomon Understanding the Windows I/O System September 15 2012.
https://www.microsoftpressstore.com/articles/article.aspx?p=2201309seqNum=3

[3] Sarath Pillai Linux System IO monitoring February 01 2013.
https://www.slashroot.in/linux-system-io-monitoring

[4] Raghu Udiyar Memory Management in Linux February 13, 2012
https://www.slideshare.net/raghusiddarth/memory-management-in-linux-11551521?next$_slideshow = 1$