OPERATING SYSTEMS II
CS444 FALL 2017

NOVEMBER 10, 2017

OPERATING SYSTEM FEATURE COMPARISON

BLOCK AND CHARACTER IO

BY

COREY HEMPHILL

Abstract

This document examines, compares, and contrasts low-level operating system kernel block and character IO operations and implementations in Windows, FreeBSD, and Linux operating systems.

I. Compare and Contrast FreeBSD and Linux IO

Modern day operating systems generally have a number of algorithms at their disposal for scheduling IO. In FreeBSD, there are at least three different kinds of IO to schedule; character devices, such as keyboards, block devices such as storage disks, and sockets for both network and inter-process communication. By default, FreeBSD uses the ULE IO scheduler to schedule IO tasks. FreeBSD can also be tuned to run the BSD scheduler as well. The ULE scheduling algorithm was designed to handle IO scheduling on systems that have relatively heavy workloads, such as high-traffic web servers. The ULE algorithm has a constant execution time regardless of the number of threads executing at a given time. Furthermore, the ULE algorithm is comprised of multiple queues, a CPU usage estimator, slice and priority calculators, and load balancing helper algorithms. ULEs primary objective is performance, however, there are other advantages of its use. Similar to all UNIX-like operating systems, in FreeBSD, almost everything within the OS exists as a file, for which there exists a descriptor that can be used to perform IO operations. A descriptor represents an underlying object that is supported by the operating systems kernel, and in FreeBSD, that can be a file, a pipe, or a socket. The kernel keeps track of descriptors via a descriptor table. Every valid descriptor in the table contains an offset in bytes from the beginning address of the object. The FreeBSD kernel supports a couple of methods for disk encryption; one such method is the gbde kernel facility which can be utilized to perform 128-bit AES encryption on both character and block devices [1] [2] [3].

Not surprisingly, FreeBSD and Linux are fairly similar in regard to IO due to their UNIX ancestry. They do, however, use different IO schedulers by default. As mentioned above, FreeBSD utilizes the ULE scheduler, whereas Linux, by default, utilizes the Completely Fair Scheduler (CFS). CFS utilizes a binary tree process selection, as opposed to ULE which utilizes queues. The CFSs primary goal is to manage IO resources fairly, providing equal processor time to all processes, rather than performance. Both of these schedulers have valid usages and various trade-offs. Although Linux uses CFS as its default scheduler, it has a number of other schedulers available through tuning such as the Deadline, Anticipatory, and Completely Fair Queuing (CFQ) schedulers. While their IO schedulers differ, in terms of descriptors, Linux and FreeBSD are extremely similar. For all intents and purposes, they are essentially the same in that aspect. This seems to be another case of if it isnt broken, dont fix it. In regard to cryptography, Linux has the kernel crypto API which supports AES, SHA1, and several other encryption ciphers and templates. Although poorly documented, Linuxs crypto encryption functionality dwarfs that of FreeBSDs [4] [5].

```
/*
 * Handle an I/O request.
 */
static void brdd_transfer(struct brdd_device *dev, sector_t sector,
  unsigned long nsect, char *buffer, int write) {
 uint8_t *origin;
 uint8_t *target;
 unsigned long i;
 unsigned long offset = sector * logical_block_size;
 unsigned long nbytes = nsect * logical_block_size;

 /*
  * Determine whether we are performing a read or a write
  */
 if (write) {
  origin = buffer;
  target = dev->data + offset;

  printk("brdd: Writing to RAM Disk Device...\n");

  for (i = 0; i < nbytes; i += crypto_cipher_blocksize(tfm)) {
   crypto_cipher_encrypt_one(tfm, target + i, origin + i);
  }
 }
 else {
  origin = dev->data + offset;
  target = buffer;

  printk("brdd: Reading from RAM Disk Device...\n");

  for (i = 0; i < nbytes; i += crypto_cipher_blocksize(tfm)) {
   crypto_cipher_decrypt_one(tfm, target, origin + i);
  }
 }
}
```

Fig. 1. Crypto IO Example

II. Compare and Contrast Windows and Linux IO

In Windows operating systems, the IO manager provides the interface to a given systems kernel-mode drivers. In other words, the IO manager manages the communication and interaction between the systems device drivers and user applications. The communication between the IO manager and the kernels drivers is handled via IO request packets (IRPs). The Windows IO model provides what is called a layered driver model, in which these driver layers lie one above the other to create stacks. The idea is that at the bottom of the stack, you have the PCI bus driver, then the next layer up is the USB host controller driver, then the USB hub driver, then your IO device driver. The Windows IO system provides communication between the drivers in a stack by enabling the passing of the IRPs. The Windows IO system also supports both synchronous and asynchronous IO. In synchronous IO, a thread will initiate an IO operation, then it will wait until the operation completes before moving on to another job. In asynchronous IO, a thread may initiate an IO operation, bind that operation to an IO completion port, then continue working on another job until the kernel sends a signal to the waiting port letting the thread know that operation has finished. There are some pretty obvious advantages to allowing asynchronous calling of IO operations; one advantage being that it is more efficient when used for operations that are anticipated to require a relatively large amount of time to complete, allowing the thread to continue execution without wasting time waiting for completion [6] [7] [8] [9].

There are a number of notable differences between the Windows and Linux IO systems. Windows utilizes IO manager to

handle IO and passes IRPs within driver stacks for communication, whereas Linux uses CFS and descriptors for files, pipes, and sockets. These are completely different approaches to handling IO, and its arguable which is better. It all depends on which religion you subscribe to, so to speak. Linux has a number of different schedulers that a user can utilize, whereas Windows offers no alternative. This is likely due to the fact that Windows is proprietary, and Linux is open-source. Lastly, both Windows and Linux offer asynchronous calling of IO operations, as most operating system kernels are capable of handling asynchronous IO operations. Although the end result is the same, how this result is achieved behind the scenes is likely vastly different between these two kernels. This tends to be the general theme in the Windows vs Linux rivalry [4] [5].

III. Conclusion

Its refreshing to see some diversity amongst the different IO management systems within the FreeBSD, Windows, and Linux operating system kernels. After a brief examination, it doesnt seem any of these IO systems is massively superior to any other, it just seems they simply have different purposes and specializations. Its important to be aware of these options, and apply the proper approach to the given scenario. In other words, know your toolset, and use it wisely. IO management is something every computer must do, and it is arguably the most essential function of an operating systems kernel.

REFERENCES

[1] "I/o scheduling in freebsds cam subsystem," https://people.freebsd.org/~imp/bsdcan2015/iosched-v3.pdf, Netflix.com, [Online; accessed 8-November-2017].

[2] "Ule: A modern scheduler for freebsd," https://www.usenix.org/legacy/event/bsdcon03/tech/full_papers/roberson/roberson.pdf, USENIX Association, [Online; accessed 8-November-2017].

[3] "I/o in freebsd," http://www.sai.syr.edu/~chapin/cis657/IO.pdf, USENIX Association, [Online; accessed 8-November-2017].

[4] R. Love, *Linux Kernel Development*. Pearson Education International, 2010.

[5] "Linux system programming," https://www.safaribooksonline.com/library/view/linux-system-programming/0596009585/ch04s06.html, Robert Love, [Online; accessed 8-November-2017].

[6] D. Solomon and M. Russinovich, *Microsoft Windows Internals, Part 1 6th Edition*. Microsoft Press, 2012.

[7] "Overview of the windows i/o model," https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/overview-of-the-windows-i-o-model, Microsoft, [Online; accessed 8-November-2017].

[8] "Windows kernel-mode i/o manager," https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/windows-kernel-mode-i-o-manager, Microsoft, [Online; accessed 8-November-2017].

[9] "Synchronous and asynchronous io," https://msdn.microsoft.com/en-us/library/windows/desktop/aa365683(v=vs.85).aspx, Microsoft, [Online; accessed 8-November-2017].