CS 444 - Spring 2016 - Writing Assignment 2

Cody Malick

malickc@oregonstate.edu

Abstract

Understanding how an operating system handles input and output to devices and how they are scheduled are important topics for an aspiring kernel developer. In this report, we will cover how Linux, Windows, and FreeBSD handle these important tasks in their respective kernels, and contrast them.

CONTENTS

I. Introduction

Every computer, whether it's a mobile phone or a digital watch, needs to be able to read and write to storage. For general desktop computers, they have to be able to communicate with a vast variety of devices from hard drives to MicroSD cards. In this paper, we will discuss how Linux, Windows, and FreeBSD handle input and output, how they make it efficient, and contrast them to each other.

II. Linux

A. Virtual Filesystem

At the top layer of our input-output system, we have the virtual filesystem, otherwise known as the VFS. The virtual filesystem allows user-space programs to use standard unix systems calls such `read()` and `write()` regardless of storage medium. This is setup is only possible because Linux adds a layer of abstraction around the low-level filesystem. Although we take such functionality for granted, it is a very important system to have. [1]

The VFS provides an abstraction of the filesystem, or multiple filesystems through a few specific interfaces. Specifically, the VFS provides the following abstractions:

Files: an ordered string of bytes.

Dentries: Files that contain directory information. For example, '\this\is\a\path' has four dentries on the path to that directory. This is how the VFS contains directories as files.

Inodes: Otherwise known as an 'index node,' these files contain metadata about other files such as permissions, size, owner, etc.

Superblock: A file containing all the relevant information about a filesystem as a whole. This is a metadata file for an entire FS.

Each of these interfaces have sizable data structures, along with a corresponding operations struct that complements them, containing function pointers to give the VFS the great functionality it has.[1]

These interfaces are critical to how we use Linux every day. Without them, we would have to make manual calls to the different filesystems that managed different devices, in the protocols they require. Sounds like a mess!

B. Block IO Layer

The Block IO layer is where the dirty work actually gets done. This layer manages how data is actual written to and retreived from the storage device. Any device that contains storage is considerd a block device to the kernel. Usually, these block devices contain segments called *sectors*, which are the smallest size that data can be stored in. The standard size is usually 512 bytes, but it's possible for these sizes to vary.

After a sector, we have the *block*. The block is the smallest addressable unit by the kernel. This is an abstraction imposed by the filesystem. The block can be the same size as a single sector, but not smaller, and a block must also contain a multiple of the sector size. If the sector size is 512 bytes, then the block cannot be 800 bytes, it must be 1024, or 2048 bytes. It cannot contain anything but whole sectors.

The actual reading and writing is handled by buffers and buffer heads. These buffers contain blocks that they are mapped to while they are pending to read or write from any given block. Buffer heads are the kernels way of having a descriptor

pointing at these buffers. A buffer head contains all the information needed for the kernel to control and manipulate these buffers. [1]

```
struct buffer_head {
        unsigned long b_state; /* buffer state flags */
        struct buffer_head *b_this_page; /* list of pages buffers */
        struct page *b_page; /* associated page */
        sector_t b_blocknr; /* starting block number */
        size_t b_size; /* size of mapping */
        char *b_data; /* pointer to data within the page */
        struct block_device *b_bdev; /* associated block device */
        bh_end_io_t *b_end_io; /* I/O completion */
        void *b_private; /* reserved for b_end_io */
        struct list_head b_assoc_buffers; /* associated mappings */
        struct address_space *b_assoc_map; /* associated address space */
        atomic_t b_count; /* use count */
};
```

As you can see from the above code, the `buffer_head` struct contains all the information the kernel needs. It has `b_state` to track the state of the buffer, `buffer_head` to track the page's buffers, a pointer the the page associated with this buffer, and more. With this struct, the kernel can have direct control over what information is handled in the Block IO layer.

C. Scheduling

Along with actually getting IO done, the kernel must decide what order these operations need to be done in. The kernel also needs to reduce the total number of requests by efficiently merging requests that can be merged. This is handled by the IO scheduler. There are many different approaches to different schedulers. It is a very fascinating area for that reason.

For example the noop (no operations) scheduler is the most basic one out there. All it does is maintain a request queue that is a first-in first-out data structure. It's only other responsibility is merging requests that are adjacent to other requests. This scheduler is the default scheduler for solid state drives and any sort of flash memory. The reason for this being that solid state devices have no moving parts, no head constantly seeking for different sectors. All read and write locations are self-sorting, and instant to read and write from. This scheduler performs absolutely poorly, however, when confronted with a spinning drive.

A spinning drive requires a scheduler to group requests by location. This is needed because if we simply serviced requests in a FIFO manner, then the head would spend enormous amounts of time seeking for sectors, and slow down [2]the OS tremendously. For this reason, the default scheduler for disk devices on Linux is the CFQ scheduler.

The Completely Fair Queuing IO Scheduler has the following functionality: it maintains a seperate qeuue for each process, it services these requests in a round robin fashion, and performs merging of adjacent requests. This works very well for a number of reasons. The first of which, is each process gets the IO it needs done in a fairly prompt manner, but also prevents any one process from being starved out of IO by another read or write heavy operation. Another is that if the system is

a multiple user sytem, which almost every computer is these days, then each user will get their IO serviced in a prompt manner. [1]

D. Interrupts

Interrupts are how a piece of hardware communicates with the cpu, letting it know that it has some data ready for it. An interrupt must have an interrupt handler associated with it in order to have the communication handled properly. These interrupt handlers are contained in a devices driver. This is pretty universal across operating systems. What we are going to look at specifically is the structure of interrupts that Linux handles. [1]

E. Interrupt Structure

Interrupts are split into two parts in Linux: top halves, and bottom halves. These names are quite misleading as top halves and bottom halves are not halves at all. The top half is usually closer to a fifth or even an eighth. The reason for this is that interrupt handlers have to be very fast! Interrupts, as they are aptly named, stop everything the cpu is doing in order to communicate with hardware. When this interrupt is called, all the code in the top half of the interrupt has to be handled extremely quickly. Because of this, only time sensitive information can be stored in the top half, and all information in the top half is handled by the interrupt handler.

The bottom half, on the other hand, can contain any extra information needed to process the request from the hardware. Because this information is not deemed time critical, it is queued up with the other requests that the CPU needs to handle and is put off until its time to be processed has come. [1]

This is a basic overview of how Linux handles IO. Next, we'll examine how Windows and FreeBSD handle these same tasks, and compare them to how Linux operates.

III. Windows

Windows does things quite a bit different from Linux, as we would expect. Because it is not originally a Unix based or Unix inspired operating system, we get to see a very different approach to solving similiar problems.

A. IO Manager

Windows has a module called the IO Manager that is the core of all input and output requests. This system is packet based, meaning ever request comes through in the form of an IRP (IO request packet). The IO manager handles IO by taking a given IRP in memory, and handing it off to the appropriate driver, then disposing of the packet. Because of this, a lot of the IO done in windows is actually handled directly by the device drivers. In addition to handling IRPs, the IO manager provides an interface that allows simplification of driver tasks. For example, common tasks such as one driver calling functions in another driver. It also maintains IO buffers for the drivers. The IO manager gives a simple interfaces for these, reducing the complexity of implementing drivers in Windows.

In general execution of IO in Windows is done in the following pattern. First a request is made by an application, read for example, and is handed to the IO manager. The IO manager creates an IRQ which it then hands off to a driver for processing. [2]

B. Types of IO

Windows has three classifications of IO: buffered IO, direct IO, and Neither IO. Buffered IO is IO dumped into a buffer for a specific device for write operations, and pulled from that devices buffer for read requests. Direct IO is when the IO manager takes a devices buffer and locks it into memory. It then handles the IO request, and releases control of that memory to the driver. The driver then pulls the data from the memory buffer. Doing this, the driver can have direct access to memory when it needs it. Lastly is Neither IO. As the name implies, this type of IO is when the method of IO is left entirely to the driver to handle. The IO manager simply hands off a request to the driver to handle.[2]

This is a very interesting classification of IO that neither FreeBSD nor Linux have. FreeBSD and Windows simply treat all devices as files, and the drivers simply handle the input from the user as if it was writing to a file, and the device communicates via interrupts to the CPU.

C. Drivers

Drivers are important in every OS, but in Windows they have a very specific structure. Drivers have a few responsibilities besides handling interrupts as they do in Linux. They have the responsibility to start IO requests to the hardware, handle plug-and-play behavior, cancel IO routines, unload routines, and more. Drivers are much more of entire modules instead of specifically interrupt handlers as they are in Linux. [2]

This view of drivers makes the handling of IO much more of the drivers responsibility than it does in Linux. In Linux, the driver simply handles the interrupt and hands other information off to the OS. In Windows, the driver is more robust and has more responsibility on how things are communicated.

D. Scheduling

The Windows IO scheduler handles IO requeusts by priority. It is a fairly straightfoward system. There are five levels of priority, critical, high, normal, low, and very low. Very low is usually restricted to background processes and content indexing. Doing things this way allows for a very interactive environment and high priority to any user initiated actions. Per the `Windows Internals Part 2` book I'm reading, low and high priority are not used. Normal is reserved for regular application IO, while cricial is reserved for the memory manager exclusively. These different priorities are sorted into individual queues, and are executed in decending order from high priority to low priority.

This scheduler shares the idea of multiple queues with the Linux scheduler, but otherwise it's a completely different beast. The scheduler isn't fair at all, and priority is king.[2]

Now that's we've seen the basic structure of Windows IO, we'll take a look at how FreeBSD handles these same ideas.

IV. FreeBSD

FreeBSD is very similar to Linux in how it handles many of these processes. As both borrowed ideas from a purely unix operating system, we will see that things such as the VFS handle things similiarly.

A. Basic IO

The IO in FreeBSD has a few structs that are similiar to the Linux IO structs. Something interesting about the FreeBSD versus the Linux is that FreeBSD makes explicit statements about how a file's existence is defined. For example, a file in

FreeBSD exists until no references or descriptors are open in the OS. Linux may have similiar functions, but the rules for these are not explicitly defined. Here are the basic objects that make up the FreeBSD basic IO system: [3]

Files: A Linear array of bytes with at least one name. A file exists until all its names are explicitly deleted explicitly and no process holds a descriptor of that file. All IO devices are treated as files.

Directory Entry: A file that contains information about itself and other files and directory entries.

Pipes: Pipes are also a linear array of bytes, but are used exclusively for one direction data transfer through the use of an IO stream. If you need to read and write from a file at the same time, two pipes will need to be open.

Socket: An object that is used for interprocess communication, it exists only as long as some process holds the descriptor for it.

This setup is almost identical to how Linux handles things, but with one big difference: other files sytems mounted inside of the virtual file system are treated simply as directories, not as whole units like Linux's superblocks. This simplifies the handling of the entire filesystem structure. A root directory of a filesystem is set so that the OS knows where everything starts.

B. Scheduling

The FreeBSD scheduler uses a simplistic algorithm called `disksort()`. Disksort is almost incredibly straight forward: [4]

```
void disksort(drive qeuue *dq, buffer *bp)
{
  if(active list is empty) {
    place the buffer at the front of the active list;
    return;
  }
  if(request lies before the first active request) {
    locate the beginning of the next-pass list;
    sort bp into the next-pass list;
  } else {
    sort bp into the active list;
  }
}
```

The above pseudo code explains the basic idea. It is essentially a basic elevator that does merging and sorting as needed. It maintains two lists, current pass and next pass, and if a request requires a change in direction, add it to the next pass list. It is much more straightforward than the linux CFQ scheduler, though arguably less efficient. [3]

C. Interrupts

Interrupts are handled almost identically to Linux's interrupt setup. Each interrupt must be registered via a device driver. FreeBSD has two types of interrupts: hardware interrupts and software interrupts. These are the equivalent of Linux's top

and bottom halves. If a piece of information is time critical, it is put in a hardware interrupt, otherwise they are handed to a software interrupt. [3]

V. Conclusion

Linux, Windows, and FreeBSD all have their own way of doing things. Although Linux and FreeBSD share a lot of the same ideas, they are unique. Windows is another beast entirely with its modules and driver structures. Understanding how all these systems work are important as a developer working in the each of their respective kernels. I am looking forward to diving into this topic more.

VI. Appendix A - Linux Structs

All following structs have been pulled from the classroom text. [1]

for future code samples

VII. Appendix B - Windows Structs

for future code samples

VIII. Appendix C - FreeBSD Structs

for future code samples

sectionBibliography

References

[1] R. Love, *Linux Kernel Development*, 3rd ed. Developer's Library, 2010.

[2] M. R. D. Solomon and A. Ionescu, *Windows Internals Part 2*, 6th ed. Microsoft Press, 2012.

[3] FreeBSD. (2016, jan) 2.6. io system. [Online]. Available: https://www.freebsd.org/doc/enUS.ISO8859-1/books/design-44bsd/overview-io-system.html

[4] K. McCusick and G. V. N. Neil, *Design and Implementation of the FreeBSD Operating System*, 2nd ed. Pearson Education, 2015.