WRITING ASSIGNMENT 2

CS 444 Spring 2017

Author:
Brandon Dring

Professor:
Dr. Kevin MCGRATH

May 18th, 2017

Abstract

I/O and I/O processing is fundamentally what a computer was designed to do. Take an ‘X’ do some sort of processing, and give you back a ‘Y’. Here I will be talking about I/O, how it’s processed and scheduled between Windows and FreeBSD, and how Windows and FreeBSD implements devices and drivers. Comparing and contrasting them to Linux, and what is different and what is alike.

I. Windows

A. Windows I/O Processing & Devices

Windows I/O and how it processes it is based on some of the design goals of the system. Which is carried across on how it deals with block or character devices, and how it schedules them. The goals being:

- High-performance asynchronous packet-based I/O to allow for the implementation of scalable applications.[3]

- Services that allow drivers to be written in a high-level language and easily ported between different machine architectures.[3]

- Dynamic loading and unloading of device drivers so that drivers can be loaded on demand and not consume system resources when unneeded.[3]

- Support for Plug and Play, where the system locates and installs drivers for newly detected hardware, assigns them hardware resources they require, and also allows applications to discover and activate device interfaces.[3]

At the heart of the I/O system is the I/O manager, it decides when I/O requests are delivered to the respective drivers. The I/O manager and system is almost entirely packet driven.[3] Which allows an application to manage multiple I/O requests concurrently, thus allowing the ability to scale easily with more and more devices allowed on a computer. I/O requests are then transformed to I/O request packet (IRP) by the I/O manager, which is a data structure with instructions on describing the I/O request. Then the IRP is stored in memory with a pointer to the driver that it is destined for. Which allows for what was a packet to be stored long term in a queue to be ran when the driver is available. What is interesting is that the IRP is used by the driver and sent back to the I/O manager. Because, through the I/O manager and transforming the packet to an IRP, it is compatible to other devices on the system.[3] So it can either signal the I/O manager that the operation is done, or tell the I/O manager that this IRP needs to move onto another device. Inside this IRP, the I/O gives an abstraction and it treats the IRP source or destination as a file. Thus further abstracting the application interface to a driver.

Along with the IRP, since all devices on Windows require some sort of driver, there is a hardware abstraction layer (HAL). Which provides another abstraction between the actual hardware and the software of device drivers.

Separately to support runtime devices being added, inside the I/O system is the Plug-n-Play manager (PnP manager) that is responsible for loading a devices driver when detected on the system. So, a device can be added or removed while the system is running, and there is no need for a reboot.[3]

```
 typedef struct _IRP {
  PMDL MdlAddress;
  ULONG Flags;
  union {
    struct _IRP *MasterIrp;
    PVOID SystemBuffer;
  } AssociatedIrp;
  IO_STATUS_BLOCK IoStatus;
  KPROCESSOR_MODE RequestorMode;
  BOOLEAN PendingReturned;
  BOOLEAN Cancel;
  KIRQL CancelIrql;
  PDRIVER_CANCEL CancelRoutine;
  PVOID UserBuffer;
  union {
    struct {
    union {
      KDEVICE_QUEUE_ENTRY DeviceQueueEntry;
      struct {
        PVOID DriverContext[4];
      };
    };
    PETHREAD Thread;
    LIST_ENTRY ListEntry;
    } Overlay;
  } Tail;
} IRP, *PIRP;
```

An I/O Request Packet (IRP) in Windows, notice the union keyword

B. Windows Drivers

Windows has 2 main types of drivers, there is the file system driver which deals with requests to mass storage or network devices. The file system driver encompasses more general File I/O, as the name implies. Along with the device drivers and mass storage drivers, which can be broken down further such as PnP & non-PnP drivers, Windows Driver Model(WDM), and layered drivers. Which takes care of just about everything else from PnP devices like storage devices, input devices, and anything in between that is related to actual system files.

The I/O system also has a set of routines that it follows when it has an I/O request. It starts with the initialization routine which loads the driver into the operating system and creates the appropriate data structures. Then the add device routine if the device supports PnP, and allocates an object to represent the device. Dispatch routines which include CRUM operations and other capabilities for the device, which get used whenever an IRP tries to use a device driver. Then a start routine which initiates a data transfer between the device and IRP, only if the device relies on the I/O manager to queue the I/O requests. Finally, we have the interrupt service routine(ISR) which deals with device interrupts. When a device interrupts it runs as a special process level as to prevent blocking of other threads on the CPU. Among various other routines such as cancel and unload which focus more on disconnecting the device from the system.

C. *Windows I/O Scheduling*

Windows I/O is typically done in a synchronous fashion like most things done on a computer. So an I/O request causes the thread to wait on the device driver for a return status before doing the next task. But, Windows also has the ability to do asynchronous I/O. In other words, there can be a barrage of I/O requests sent and the thread will continue to execute non I/O related requests and deal with the return status whenever the device sends it back, then continue on. Which can vastly increase speed/throughput, so most kernel level I/O goes the asynchronous route. Besides the two basic scheduling types, there are a number of I/O efficiency techniques such as Fast I/O in which a thread can bypass the I/O system all together and go straight to the device stack and exchange data directly, cutting out the middle hence the name. An interesting version of I/O is called Mapped I/O in which the data on the disk or storage device is loaded into process virtual memory and therefore the process can read directly from memory instead of trying to get I/O from a device. And lastly is Scatter/Gather I/O in which a process can write from multiple buffers in virtual memory to contiguous memory in one call, instead of multiple calls. These different I/O techniques can all be used via the I/O system and therefore the manager to increase the throughput of the system based on the circumstances at the time.[3]

II. FreeBSD

A. *FreeBSD I/O Processing*

I/O in FreeBSD is based all of descriptors.[4] Which are integers that relate to a file opened up in the OS, which from there can point to a function in the descriptor table like read, write among other basic I/O operations. Then pointing to a file entry where the type of file is associated with the action and an array of pointers to functions that take generic operations into specific operations based on the type.[4] So for basic data files the file entry may point to a vnode structure, dealing with actual file I/O, or for networking or Interprocess Communication(IPC) it may point to a socket, or something else like a pipe, FIFO, etc. The perk of dealing with a descriptors for all I/O is that all I/O is treated as if it were a file, which is generally pretty simple.

```
struct request {
  struct list_head queuelist;
  union {
    struct call_single_data csd;
    u64 fifo_time;
  };
  struct request_queue *q;
  struct blk_mq_ctx *mq_ctx;
  int cpu;
  unsigned int cmd_flags;  /* op and common flags */
```

```
req_flags_t rq_flags;
int internal_tag;
unsigned long atomic_flags;
/* the following two fields are internal, NEVER access directly */
unsigned int __data_len; /* total data len */
int tag;
sector_t __sector;  /* sector cursor */
struct bio *bio;
struct bio *biotail;
.
.
.
```

Part of the request struct in FreeBSD and Linux which has since been changed

B. FreeBSD Devices

Generally the different kinds of FreeBSD devices can be broken down into three categories which are Disk Management, I/O routing and control, and Networking. As computers, devices, and networking has evolved, so has FreeBSD to keep up.

Disk management, is there to deal with the potential complexity of partitioning a disk into near infinite units, creating virtual partitions, and so on. To account for all this FreeBSD has created an abstraction layer called the GEOM layer, which deals working around all the partitions of the disk and presenting it to the filesystem as one very large disk.[4]

Then you have the I/O routing and control, which deals with multiple hardware units such as the Advanced Programmable Interrupt Controller (APIC), Peripheral Component Interconnect(PCI), amongst others. Which decides how the I/O routing is used and when.

Network devices are peculiar in that they happen asynchronously compared to the rest of the devices.[4] In their asynchronous nature, there is no acknowledgement receiving or sending data out. They also have the task of taking packet data and transforming it into something useable to the physical media below.

For the most part FreeBSD relies on character devices instead of block devices due to the fact that when a block device uses caching it makes it near impossible to figure out specific disk contents in time.[2] So instead, they use character devices due to their reliability and simplistic interface in which data transfers directly to and from a user process.

C. FreeBSD Drivers

FreeBSD has a couple different categories of drivers being character, block, and networking devices. With each device they break their drivers down into 2 sections the top half and bottom half. With the distinction of the top half taking caring of any real I/O requests made by the kernel, and servicing the requests within the top half. While the bottom half is designated for handling any interrupts that may occur.[4] With the interrupts however, they run in thread context, so they can block, sleep or do anything a thread could normally do. The bottom half is also responsible for unexpected crashes and how to recover from these failures as well.

Generally device drivers are generally straight forward in FreeBSD, as once the device is plugged in it goes through an autoconfiguration process. Which, probes the OS to install and drivers needed for the device. Then as stated as above, it partitions responsibilities between the top and bottom half of the driver. So the drivers take care of most of the work in setting up the device, and how it interacts with the system.[4]

D. FreeBSD I/O Scheduling

Typically the device drivers manage the I/O scheduling individually.[4] Where a device keeps track of requests made to it, via queues. Generally they are recorded in the top half of the driver and it is recorded in a specific data structure to the device driver and placed in the queue to be processed. When the request is completed, it is removed from the queue via an interrupt and interrupt service routine and notifies the requesting application. Generally, I/O queues are shared between halves of the driver to help synchronization of tasks. They help maintain a synchronized nature by sharing a mutex in the request queue data structure between the top half and bottom half.[4]

III. Comparing To Linux

A. I/O Management & Processing

Windows is unique because it has a individual I/O manager that helps sort and dispatch I/O requests between devices while Linux does not. But at the same time Windows, and FreeBSD are similar to Linux in the sense they are using a file system abstraction to the actual I/O. But Linux and Windows uses a virtual filesystem to help separate file system code from the rest of the kernel.[1] While FreeBSD has a whole system in place to treat I/O like actual files.

B. Devices

Throughout the operating systems there is a bundle of similarities. They're broken down into three groups being block, character and networking devices. Network devices operate through sockets, while the block and character devices operate through device nodes. Also, devices are modular in the system in which you can add them while the system is running and not have to reboot.[3] The only difference is that Windows creates a device object to be presented to the I/O manager to be recognized.[3] Where as FreeBSD and Linux map the hardware interface into a byte stream simulating the look of a file system.

C. Drivers

Drivers are relatively the same between the three main operating systems. Drivers typically consist of specific routines such as initialization, a routine to handle interrupts, and a way to disconnect or report a fatal error. However Linux and FreeBSD do share the similarity in that their drivers are divided into a top half and bottom half with specific responsibilities for each. While Windows creates a device/driver object that interacts with the I/O manager.

D. Scheduling

Windows and FreeBSD both use a request queue to determine which request to service next, which is also similar to Linux. Also, FreeBSD and Linux share how requests are structured, which is by the struct request, while Windows keeps it in an IRP. But that is about where the similarities end. Windows generally uses a synchronous scheduler, with occasionally using an asynchronous scheduler, and they rely heavily on priority numbers. Then Linux has a variety of I/O schedulers to choose from, between a no-op scheduler, deadline, and a Completely Fair Scheduler (CFQ) which is usually the default. While FreeBSD uses the C-LOOK scheduler.

```
void disksort(
  drive queue *dq,
  buffer *bp);
{
  if( active list 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;
}
```

Here is a general look at the basic C-LOOK algorithm implemented in FreeBSD, which scans the disk in a linear fashion going from the sector with the lowest number, to the sector with the highest number. When it reaches the highest currently on the queue, it goes back to the beginning. This algorithm ensures there is no wasted drive head movement when servicing requests, as it only travels in one direction. [4]

E. Functionality

Throughout the operating systems most I/O routines are run with special privileges. An ISR from an I/O device gets near instant runtime on the CPU, to finish what they were called to do as soon as possible.[4] Windows data structures are typically made with device or driver objects to interact with the I/O manager within the kernel. But, they still do have the basic data structures like linked lists, queues, and the like for most I/O related things. While FreeBSD and Linux are heavy believers of some flavor of binary tree to do sorting and searching. For all OS's, they do their encryption below the level of the file system to prevent against plaintext attacks.[3]

IV. Conclusion

For the most part, I/O features are shared between the operating systems. While FreeBSD and Linux being more closely related than they are to Windows since they are both part of the Unix family. All systems also support the modularity to connect and disconnect devices at runtime. And how devices are grouped into categories, and how they all roughly resemble a file to the actual operating system. Scheduling is really the only place where there is a divide between the two does make sense though, because I/O is the slowest part of the computer, and everyone has roughly the same approach on how to minimize the impact I/O has on general computing.

V. Citations:

[1] A. Brouwer, “The Linux kernel,” The Linux kernel: The Linux Virtual File System, 01-Feb-2003. [Online]. Available: https://www.win.tue.nl/ aeb/linux/lk/lk-8.html. [Accessed: 19-May-2017].

[2]“FreeBSD Architecture Handbook,” 9.4. Block Devices (Are Gone), 29-Oct-2016. [Online]. Available: https://www.freebsd.org/doc/en/books/arch-handbook/driverbasics-block.html. [Accessed: 19-May-2017].

[3] M. E. Russinovich, D. A. Solomon, and A. Ionescu, Windows Internals, Part 2. Microsoft Press, 2012.

[4] M. K. McKusick, G. V. Neville-Neil, R. N. M. Watson, and M. K. McKusick, The design and implementation of the FreeBSD operating system. Upper Saddle River, NJ: Addison Wesley, 2015.