Final Paper

CS 444 Spring 2017

Author:
Brandon Dring

Professor:
Dr. Kevin McGrath

June 12th, 2017

Abstract

In this paper we will look at the two Operating Systems (Windows, FreeBSD) how they compare and contrast to the well known and loved Linux Operating System. We will specifically look at 3 different topics being, Processes & Scheduling, I/O management, and Memory Management. Examining what parts set them apart from each other in their implementation of these features, and what is similar. And what impact these changes have on the overall system.

Part I
Introduction

Most computer users of know nothing of how a computer works, they just see it as the place where they log onto Facebook, or check their email. Most people probably wouldnt even know that there are in fact 3 Operating Systems that the IT world uses to well, run the world. Between these 3 systems they daily compute almost everything on earth, and in space from videos of cats chasing lights, banking information for millions of accounts on billions of purchases, to the mars rovers rolling around a planet 34 million miles away. Here we will look not look at how they control a Mars rover, but how they manage the smallest details of ordinary OS functionality to accomplish much larger and greater tasks.

Part II
Windows

1 Processes, threads & Scheduling

1.1 Processes

Each process is represented by a EPROCESS (Executive Process) block, much like a struct, (although technically it is a process object) controlled from within the kernel. The block has values that point to attributes about it such as process ID, flags, amongst various other data structures with task related values[1]. A process in Windows is also largely just a container for threads to live in. Within the process block, there is a field for a list of the threads that relate to the parent process[1]. The EPROCESS block lives in the system address space, apart from the Process address block, which is unique to the process containing it, that is stored in the process address space. An important attribute in the EPROCESS block is the KPROCESS (Kernel Process) field, which contains information on how the kernel schedules the threads of a respective process.[1]

Creating a windows process is very ham handed and contains a total of 7 steps, along with traversing through 3 parts of the Windows OS (Windows client-side library, Executive, and subsystem process). Upon creation, it parses any flags and attributes passed in. Then it opens the image file to be executed, creates a new EPROCESS block, and instantiates the initial thread. From there, it performs post initializations of the new process and threads and loads any DLLs that need to be required as well which are yet a couple more steps[1].

```
CreateProcess( NULL, // No module name (use command line)
    argv[1],       // Command line
    NULL,          // Process handle not inheritable
    NULL,          // Thread handle not inheritable
    FALSE,         // Set handle inheritance to FALSE
    0,             // No creation flags
    NULL,          // Use parent's environment block
    NULL,          // Use parent's starting directory
    &si,           // Pointer to STARTUPINFO structure
    &pi )
```

Creating a process in Windows is a complicated process just by the number of arguements

1.2 Threads

Each process is required to have at least one thread running on it. Much like the processes have the EPROCESS block, threads have the ETHREAD block. The ETHREAD block has an attribute similar to the KPROCESS block for processes called KTHREAD; it contains information for the kernel on how to schedule the threads and synchronize them[1]. Alike to the EPROCESS block they both live in the system address space. Inside the ETHREAD block there is also a pointer to the parent EPROCESS block so that it can access pointer to any pending I/O requests that might need to be serviced. Overall the structure of the ETHREAD block is very similar to the EPROCESS block.

Creating a thread is a little more lightweight than creating a process, as the need to create an entire new EPROCESS block is gone. Creating a process only requires the call `CreateThread()` function call, and adds a thread to the EPROCESS block list of threads.

1.3 Scheduling

Windows has a preemptive priority based scheduler, of which there are 32 priority levels and the thread/process that has the highest priority level gets ran first[1]. However, Windows also uses processor affinity for particular threads, so if that processor that has been paired with the process is not available for the ready thread, it might get skipped within the run queue. Digging deeper, Windows also implements a round robin like scheduler that deals quantums, or small units of time. Each running process/thread is given an amount of time varying on system settings, other process states, and a direct manipulation of the process quantum before being switched off and allowing another thread with at least the same priority to run. Although, since windows is a preemptive scheduler, if it finds another process that is ready to run with a higher priority, the currently running process might not be able to finish its quantum do to the scheduler being preemptive, and get placed back in the run queue[1].

Generally, when threads are created they have the default priority of 8, but there are certain system based threads like security or session based activities that have a slightly elevated priority level. Having a default higher priority level ensures that Windows kernel related activities will be ahead of any user threads.

2 I/O and Provided Functionality

2.1 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[2].
- Services that allow drivers to be written in a high-level language and easily ported between different machine architectures[2].
- Dynamic loading and unloading of device drivers so that drivers can be loaded on demand and not consume system resources when unneeded[2].
- 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[2].

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 manager and system is almost entirely packet driven[2]. Which allows an application to manage multiple requests concurrently, thus allowing the ability to scale easily with more and more devices allowed on a computer. Requests are then transformed to I/O request packet (IRP) by the manager, which is a data structure with instructions on describing the 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 manager. Because, through the manager and transforming the packet to an IRP, it is generally compatible to other devices on the system. So it can either signal the manager that the operation is done, or tell the manager that this IRP needs to move onto another device[1]. 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.

2.2 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[2].

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[2]. 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.

```
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

2.3 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[2]. But, Windows also has the ability to do asynchronous I/O. In other words, there can

be a barrage of 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[2]. Besides the two basic scheduling types, there are a number of 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 Memory Management

3.1 Overview

Virtual address space is generally limited to how big the bit size is of the architecture is. On a 32-bit Windows system the size can only grow up to 3GB per process, while a 64-bit system can have over 7TB. Like most systems inside Windows, every piece of the kernel generally has a manager that over watches and controls the general going on of its department. Likewise, Windows has a memory manager that overlooks paging, swapping and translating of its Virtual memory system to and from the disk[2].

3.2 Virtual Memory

The Windows virtual memory is kept in check by the Memory manager inside the kernel. Upon boot the memory manager creates two pools divided into the Nonpaged pool, and the Paged pool. The Nonpaged pools focus on reliability, and are accessibly at any time, and there are no page faults. While the Paged pools specializes in that it can be paged into and out of the system[2]. These memory pools are located in the system address space, but are mapped into the virtual memory space of each process. Which can then access, allocate or deallocate their memory accordingly. To give the illusion of shared memory Windows implements section objects that live in the process address space and map to virtual memory/pages needed by the program. Then if a process needs a page that is already open, it simply just maps to the open page instead of pulling up its own personal page. From there the processes all just share the page, and any data that is altered in it[2].

3.3 Paging

Windows has divided its pages into two distinct and classical computer science names, being large pages which are 4MB and small pages 4KB. The memory management unit deals at the smallest level with a small page. Thus, the smallest unit of protection by the manager is at the size of a small page. Within a large page over 1000 small pages can be stored, with an advantage of large pages being that when a byte is accessed inside a large page generally the entire page is loaded into cache. Because when part of a large page is accessed, generally the whole page is going to be

accessed, as it is probably a large single file. Windows therefore leverages the hardware translation look-aside buffer for an instant reference of the bytes currently inside the buffer. But when done to small pages, each small page gets an entry in the look-aside buffer, cluttering the limited space. As large pages are made of a multitude of small pages and they are all contiguous, so a large page requests may fail due to fragmentation of the system[2].

3.4 System Calls for Memory

To further understand how a system call to the memory manager works inside the Windows kernel, one has to further break down the Windows paging scheme. Windows pages are divided into free, reserved, committed/private, or shareable[2]. With each definition coming exactly from its name. To allocate private pages there are Windows system calls such as `VirtualAlloc()`, and `VirtualAllocEx()`. The neat thing about reserving private pages, is that they come zero-initialized[2]. Windows also makes extensive use of the heap, and each processes spawned will have their own heap. But to allocate dynamic memory instead of the classic `malloc()` call, instead Windows uses `HeapAlloc()`. To write data using the Windows API they also have a special function call that takes in 5 arguments called `WriteFile()`, and conversely to read its `ReadFile()`.

```
BOOL WINAPI ReadFile(
    _In_       HANDLE      hFile,
    _Out_      LPVOID      lpBuffer,
    _In_       DWORD       nNumberOfBytesToRead,
    _Out_opt_  LPDWORD     lpNumberOfBytesRead,
    _Inout_opt_ LPOVERLAPPED lpOverlapped
  );
```

Example of a simple ReadFile call on Windows

3.5 Memory Management inside the kernel/functionality

Windows also has a feature that it has called Super-Fetch in which it assigns priority levels to a page based off of how many times it has been accessed in a specific time frame[2]. If Windows has learned that a specific page will be used frequently it will both load the page on boot to reduce loading time. And keep it around in RAM to ensure that access time to it is instantaneous and not be pushed off into secondary storage. Windows also keeps tabs on what pages are active and being used. And should a page fault occur, Windows will load the page that failed into memory, and the pages preceding and following it[2]. As seen above, Windows allocates memory differently from FreeBSD.

Part III
FreeBSD

4 Processes, threads & Scheduling

4.1 Processes

FreeBSD processes are similar to Linux processes in which they are all related from the same tree root, init[3]. FreeBSD uses a smaller struct called `proc` containing the process attributes and thread pointers in comparison Linux. But unlike Linux, and similar to Windows, threads and processes are clearly separated.

Similar to Windows processes, FreeBSD processes have at least one thread accompanied with it and runs its code[3]. And the process contains the address space available to store data, and kernel resources that it can access and make calls to, such as pipes/sockets, files and the like. Furthermore, the process also has a kernel state that it keeps track of, which can either be process structure or thread structure. The difference between the two being that the process structure keeps tabs on what needs to remain in memory at all times, and the thread structure only keeps track of what it needs while executing.

FreeBSD processes have 3 different states they can be in, NEW, NORMAL, and ZOMBIE. The new states represents that the process is in the course of initialization and allocation, while the zombie means that the process has died and it and its resources are waiting to be cleaned by the parent process. While in the normal state, a thread of a process can be in one of three states, which is: Runnable, sleeping, or stopped. Runnable indicates that it is or is being ready to execute, sleeping is waiting on an event typically like I/O, and stopped has been stopped by a signal or parent process.

```
#include <unistd.h>
fork(void);
execl(const char *path, const char *arg, ... /*, (char *)0 */);
```

Creating a new process in FreeBSD & linux is just 2 lines of code.

4.2 Threads

FreeBSD threads are separate from their processes[3]. Each thread has its own kernel stack that is up to 2 pages long, but really only one page is used, and the last page is used as a buffer as to not overflow its stack and overwriting others data structures. A new thread's kernel stack is only the top call frame is copied from the parent, speeding up the copy process from parent to child.

An interesting attribute of FreeBSD threads is that they have two modes they can operate in, either user mode or kernel mode. Typically in use, they are running in user mode which gives the thread restricted access to the system. When the thread enters kernel mode, it actually sets a bit in the CPU architecture, then continues on with its duties. Typically, when a thread enters kernel mode it is to make a system call, and after it is done it exits[3].

4.3 Scheduler

FreeBSD uses a round robin-esque scheduler(Specifically the ULE scheduler) where it allocates a slice of time for a thread to run on the CPU based on the current priority levels of the threads that are waiting to be ran. FreeBSD has a total of 256 priority levels, where 0-47 are interrupt threads that have top priority, its 48-79 which are real time threads that must be completed by a specific deadline. Then 79-120 are threads regarding the kernel but do not deal with interrupts. Past that is general purpose user threads, and idle threads (120-223 and 223-255 respectively). What is interesting about the FreeBSD scheduler is that it doesnt look for complete fairness when assigning its priority[3]. Instead, it looks to see what threads have the most resources, and have requested the most amount of CPU time, like interactive programs with I/O and raises their priority number to service I/O requests as soon as they come back. While CPU hogging threads have their priority number lowered, so if a thread uses all of its allotted quantum/time slice its priority is bumped down, as to not slow the system. And sleeping threads, or long inactive threads have their priority gradually raised, as to service them as soon as they have been signaled to return.

5 I/O and Provided Functionality

5.1 Overview

I/O in FreeBSD is based all of descriptors. 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. 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

5.2 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[3], which deals working around all the partitions of the disk and presenting it to the filesystem as one very large disk[3].

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. 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[3].

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[7]. So instead, they use character devices due to their reliability and simplistic interface in which data transfers directly to and from a user process[7].

5.3 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[3]. 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[3]. 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 auto-configuration 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[3].

5.4 Scheduling

Typically the device drivers manage the I/O scheduling individually. 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[3]. They help maintain a synchronized nature by sharing a mutex in the request queue data structure between the top half and bottom half.

6 Memory Management

6.1 Overview

In FreeBSD every process will have its own private address space, in which it is divided into 3 segments: text,data, and stack. With the text segment being read only, containing instructions of the program, the data segment has data portions of the program, and the stack holding the runtime stack of a program[3]. Within the FreeBSD kernel, memory is managed on a page by page basis, through the vm_page_t struct. As with most things inside the kernel all Virtual memory functionality is stored within structs with the v_ name preceding it.FreeBSD attempts to force pages into the L1 or L2 caches on the CPU. And monitors how often pages are used and decides whether to keep them in these small caches or kick them out[3].

6.2 Virtual Memory

Virtual memory and FreeBSD did not go hand in hand for quite some time, time implications have hindered proper releases of 4.X FreeBSD for nearly ten years. Over 40 companies and research groups had contributed just to implement shared memory spaces between processes[3]. To create a shared memory space between processes in FreeBSD 4.2, the `mmap()` function was created that allows processes to share files between address spaces. With any process having unlimited access to the file, and any changes to the file are shown throughout other processes that are using it as well. But all this was eventually scrapped to build the networking system. To create the virtual memory system it took over 10 years[3]. And in the end they settled on the design decision of storing the lion's share of the virtual memory on the disk instead of any RAM the computer has[3]. This obviously generates a lot of extra disk traffic that the hard drive must deal with, thus slowing down the system for the user. On top of all that, there was also computer architecture specific issues, and lack of support for multi-CPUs. In the end, after 10 years of attempted development in 4.4 FreeBSD the entire virtual memory system. Remedying all the ailments that had plagued prior 4.4 FreeBSD

```
#include <sys/mman.h>
     void *mmap(void *addr, size_t lengthint " prot ", int " flags ,
        int fd, off_t offset);int munmap(void *addr, size_t length);
```

The simplicity of creating a shared memory space in FreeBSD / Linux

6.3 Paging

Paging in FreeBSD can happen in a multitude of different way depending on the system load. If there is a heavy system load and lack of memory space the entire contents of the page will be

swapped to secondary storage (i.e Hard drive). While if there is a modest system load, the paging system will implement a LRU-2 (Last Recently Used 2) system. Where the pages are broken up into two tiers between use frequency. And when the bottom tier becomes crowded, the last recently used page is thrown out. For the longest time FreeBSD pages were all one size, 4KB, it wasnt until FreeBSD 7.2[3] that they had introduced Superpages which were 2MB in size.

6.4 System Calls for Memory

System calls are relatively simple in FreeBSD, when a user makes a system call on a file like `read()` or `write()`, the FreeBSD Kernel switches into kernel mode and will copy the data into a buffer inside the kernel[3]. Thus transferring the data from the user space into kernel space, and then switch back into user mode. Alternatively if there is a large quantity to transfer on a `read()` or `write()` call the kernel will instead remap the address space into the kernel instead of copying all the data over.

```
#include <unistd.h>
read(int fd, void *buf, size_t nbytes);
```

The simplicity of a simple read file call in FreeBSD & Linux

6.5 Memory Management inside the kernel/functionality

Inside each process address space it has the stack space, but the stack size is based off of the bit size architecture of the operating system. Since the stack size is limited to create dynamic memory various system calls are implemented. To allocate memory inside the kernel FreeBSD uses the C traditional `malloc()` and `free()`. To create very large persistent dynamic memory allocations `zalloc()` and `zfree()` system calls are made.

Part IV
Comparison to Linux

7 Processes, threads & Scheduling

7.1 Processes:

Windows processes are very heavy weight compared to how relatively small Linux processes are. Windows processes can be either a normal process or a protected process with special rights. To create them is a rather extraneous process. For Linux, it is just about 2 calls just fork/clone and exec, but with Windows creating a process behind the scenes is about a 7 step process[1]. Windows processes are also just essentially an object that contains its threads. Like most Windows features, every vital task to the kernel is contained within an object, even how it manages processes. Between Windows and Linux, they both have a separation of concerns in that they have processes that run in kernel space, and user space.

FreeBSD processes are overall very similar to Linux, processes are all instantiated from a single root with PID #1, and all processes and threads are kept in a similar struct. However in FreeBSD they are kept in a `proc` struct, as opposed to a `task_struct`. Which is slightly smaller. They are all connected as well by a linked list, from a pointer within the `proc` struct itself.

7.2 Threads

Threads are distinct in Windows and seen as different entities from their parent processes[1]. While in Linux, threads share the same address space and resources as their parent process and are scheduled and treated as the exact same as processes. Threads are also an object like entity in Windows and contained within the Process object inside the kernel as well.

Similar to Windows in FreeBSD, threads are definitely separate from processes. Linux, is unique in the sense that threads and processes are seen as the same. Linux and FreeBSD although do share the division of labor between kernel space and user space thread abilities.

```
CreateThread(
NULL,                   // default security attributes
0,                      // use default stack size
MyThreadFunction,       // thread function name
pData,          // argument to thread function
0,                      // use default creation flags
&dwThreadId); // returns the thread identifier
```

```
pthread_t Mythread;
   pthread_create(&Mythread,NULL,fnc,args);
```

Once again showing that system calls in Linux/FreeBSD are vastly simpler to Windows

7.3 Scheduling

Windows scheduling is uniquely different because of the processor affinity that a task can have, therefore altering the potential queue that Windows uses. Windows also is different in the way the prioritize the threads. Windows bases its scheduler off of 32 different levels, and Linux with a two factor niceness value, and soft real-time priority[1][4]. Linux nowadays uses the CFQ scheduler, in which all threads attempt to have a complete and equal amount of CPU time. And should a process/thread fall out of being relatively equal to the other threads, they get moved up or down the queue respectively to be back to being equal[4].

There is another unique and very helpful feature in the Linux scheduler such that tasks can be grouped together and treated fairly within that group. For example, a web server might spin off a bunch of threads to handle incoming connections. All the new threads are put into a group where they share virtual runtime[5].

FreeBSD and the Linux scheduler are similar but have a slight, yet important difference between them. Both have a round robin approach in they give essentially loop back in a circle giving all tasks a time certain amount of CPU time. In Linux, the current scheduler is looking to keep an equal amount of time split between all the processes/thread, no matter what kind of task they are doing. But the FreeBSD scheduler is favored to more I/O bounded and interactive programs[3].

Conversely FreeBSD tries not to allocate a lot of CPU time to CPU bounded programs as to not bog down the system. Along with different priority levels between Linux and FreeBSD.

8 I/O and Provided Functionality

8.1 I/O Management

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[6]. While FreeBSD has a whole system in place to treat I/O like actual files.

8.2 Devices

Throughout the operating systems there is a bundle of similarities. Theyre 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. The only difference is that Windows creates a device object to be presented to the I/O manager to be recognized. Where as FreeBSD and Linux map the hardware interface into a byte stream simulating the look of a file system.

8.3 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[3][4]. While Windows creates a device/driver object that interacts with the I/O manager[1].

8.4 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[2]. 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 a variation of 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]

8.5 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[2][3], to finish what they were called to do as soon as possible. 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 OSs, they do their encryption below the level of the file system to prevent against plaintext attacks.

9 Memory Management

9.1 Virtual Memory

Here is where it gets interesting, there are some notable differences between the operating systems in how they implement virtual memory. Firstly, Windows of course has a manager that overwatches its VM department. And Windows has section objects within the process address space that maps page requests to a singular page somewhere in memory[2]. And Windows Virtual memory is divided at boot into paged and non-paged pools, in which the only difference is how stable and persistent that pages in each of the pools are. And FreeBSD had a rough rough rough time getting started with their virtual memory system in that it 10 years to develop, and was eventually scrapped. Although in the end they do have a system call to map memory to multiple processes to share data. Which is very similar to Linux.

9.2 Paging

Overall each of the operating systems use pages somewhat similar. But what is different is that while FreeBSD and Windows both have zones (Groupings of pages). Linux is unique in that it has 28 possible zones that a page can have[8]. Which is ginormous compared to Windows and FreeBSD. Pages across all OSs also implement 2 different page sizes that are named very oddly. And are roughly 1000x the size of their small page counterparts. And page swapping to secondary

storage occurs roughly the same way as well depending on system load, and what page was recently used.

```
struct zone {
        spinlock_t              lock;
        unsigned long           free_pages;
        unsigned long           pages_min;
        unsigned long           pages_low;
        unsigned long           pages_high;
        unsigned long           protection[MAX_NR_ZONES];
        spinlock_t              lru_lock;
        struct list_head        active_list;
        struct list_head        inactive_list;
        unsigned long           nr_scan_active;
        unsigned long           nr_scan_inactive;
        unsigned long           nr_active;
        unsigned long           nr_inactive;
        int                     all_unreclaimable;
        unsigned long           pages_scanned;
        int                     temp_priority;
        int                     prev_priority;
        struct free_area        free_area[MAX_ORDER];
        wait_queue_head_t       *wait_table;
        unsigned long           wait_table_size;
        unsigned long           wait_table_bits;
        struct per_cpu_pageset  pageset[NR_CPUS];
        struct pglist_data      *zone_pgdat;
        struct page             *zone_mem_map;
        unsigned long           zone_start_pfn;
        char                    *name;
        unsigned long           spanned_pages;
        unsigned long           present_pages;
};
```

A look at how extensive Linux page zones can be.

9.3 System calls for memory

As one could guess the system calls between FreeBSD and Linux are pretty similar. It is just a simple `read()` or `write()` system call to read or write data from kernel space to and from user space. And to allocate memory it is just a simple `malloc()` and `free()`. Windows is where it gets fancy and there are certain uniquely named system calls to read and write file data, along with special ways in allocating page types.

```
LPVOID WINAPI HeapAlloc(
    _In_ HANDLE hHeap,
    _In_ DWORD dwFlags,
    _In_ SIZE_T dwBytes
  );
```

```
#include <stdlib.h>

void *malloc(size_t size);
```

Windows Memory allocation system call vs Linux & FreeBSD simplicity

9.4 Memory management inside the kernel & functionality

Windows was ahead of the curve when it comes to memory management functionality. Windows had its SuperFetch ability in that it predicted what pages were going to be learned, and it keeps it in main memory for faster access[2]. Linux followed suit and implemented a `readahead()` function to their systems, and to my knowledge FreeBSD has no such system.

Part V
Conclusion:

There is are 2 general themes throughout all of these different topics of each Operating Systems. One, being that Windows encapsulates everything inside of an object, typically a manager. And that manager object points to smaller objects that handle smaller and smaller features[1][2]. This is largely due to the Windows OS is written in C++, C#, and C[10], with the first 2 languages being very object orientated. So it is only natural that the kernel features also resemble a object that manager smaller sub features within their department. Opposed to Linux and FreeBSD which are written solely in C, which doesnt have objects, only structs.

Secondly, FreeBSD and Linux are generally very similar in how they implement any feature, but there is typically a small difference here or there. And when a small difference is compounded here or there at the size of an Operating System, you get the right to call it a separate operating system.

There isnt particularly one Operating System that should rule them all, each OS comes with its pros and cons. As such, one should use the best OS to fit their needs. For instance, if the user isnt incredibly tech savvy, and wants compatibility for all their applications, they should choose Windows. If they are tech savvy and want general versatility, and relative compatibility for all computer needs, theyd go with Linux. If they are tech savvy, want extremely secure, and at times fastest computing power, then they would go with FreeBSD[9].

With that being said, throughout all these Operating Systems, the thousands of developers that contribute to any single feature of the OS, largely do it independently of how the other Operating Systems do it. But in the end, for the layman user, there is no noticeable difference between the three. Throughout all the diversity, a user can still go onto the internet, manage multiple processes, write files / store data, and manipulate memory at large scales without ever noticing the complexity, time and effort that goes into creating such versatile and stable systems.

Part VI
Citation:

[1] M. E. Russinovich, D. A. Solomon, and A. Ionescu, Windows Internals, Part 1. Microsoft Press, 2012.

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

[3] 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.

[4] M. Jones, Inside the Linux 2.6 Completely Fair Scheduler, IBM - United States, 15-Dec-2009. [Online]. Available: https://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/. [Accessed: 02-May-2017].

[5] M. Mitchell, J. Oldham, and A. Samuel, Advanced Linux Programming, 1st ed. Indianapolis , IN: New Riders Publishing, 2001

[6] 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].

[7]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].

[8] Love, Robert M. "Linux Kernel Development." Zones. Sams Publishing, 12 Jan. 2005. Web. 07 June 2017.

[9] Lehey, Greg. "Comparing BSD and Linux." 4. Comparing BSD and Linux. N.p., 04 Feb. 2016. Web. 12 June 2017.

[10] Waite, Ryan. "N/A." What Programming Language Is Windows Written In? N/A, 9 Jan. 2009. Web. 12 June 2017.