OPERATING SYSTEMS II
CS444 FALL 2017

DECEMBER 7, 2017

OPERATING SYSTEM FEATURE COMPARISON

FINAL DRAFT

BY

COREY HEMPHILL

Abstract

This document examines, compares, and contrasts attributes of a number of low-level operating system kernel features such as processes, threads, CPU scheduling, IO management, memory management, and file systems. The following sections of this document discusses several key components of Windows and FreeBSD operating system kernels, and compares these features with those of the Linux operating system kernel. The purpose of this document is to evaluate why certain modern-day operating system features are implemented the way they are. For some of these features, similar approaches are used by many operating systems, and for others, implementations can vary widely for a number of different reasons.

CONTENTS

1 INTRODUCTION

It easy to pretend that a given operating system kernel performs all of its basic, essential operations via some kind of black magic. The reality is that, to the layman, what goes on inside of an operating system really is magic. To an engineer, however, there is no such thing as magic. In order to make intelligent technical decisions as a professional in the field, it is imperative for software engineers to have at least a fundamental understanding of the differences between kernel implementations in a number of operating systems. Here, we will examine both Windows and FreeBSD kernel processes, threads, CPU scheduling, block and character I/O, memory management, and file systems, and we will compare these to the implementations in the Linux kernel.

2 Processes, Threads, and CPU Scheduling

2.1 Compare and Contrast OS Processes

What is a process? Although this might seem like a relatively trivial question to answer, it's not really. Put simply, a process is a program in execution. However, this is not really a sufficient answer, and it requires a bit of technical knowledge to answer the question more completely. In both Windows and FreeBSD kernels, a process is a single instance of a program being executed by the CPU at a given time. A running process includes all of the necessary dependencies and resources such as included source code files, running threads, virtual address space, and system signal calls [1] [2]. A process maintains file descriptors which manage a number of vital items such as what CPU registers are being used, data stacks, tables for storing other file descriptors, and CPU runtime values. In both Windows and FreeBSD, a process is identified via a PID, or a process identifier, which is just a unique integer value. For the most part, Windows and FreeBSD processes are extremely similar with a few key differences. FreeBSD represents processes by a task struct whereas Windows represents a process via something they refer to as an executive process. Furthermore, both operating systems utilize background processes. In FreeBSD, background processes are referred to as daemons [2], whereas in Windows they are referred to as Windows Services [1].

When comparing Windows and FreeBSD with the Linux kernel, one can see that a process is very similar across the board; it is an instance of a program being executed. Processes can communicate with other processes, create child processes, create threads, etc. There are, however, some very interesting differences between Windows and Linux processes. For example, in Windows, when you stop a parent process, all of the parent's child processes will continue their execution to completion [3]. In Linux, if a parent process is stopped, all child processes are also stopped [3]. Why this is the case, I really have no idea. I've searched and searched for an answer to no avail. Obviously, there's some benefit to either of these approaches, and in Unix-like systems, there are ways in which you can obtain behavior similar to that of Windows [4]. Processes, daemons, and Windows services all maintain similar functionalities and uses. In both Linux and FreeBSD, daemons are background processes which execute imperative system features such as logging [2] [5]. Another difference that stands out between Windows and Linux is that background processes are named differently, and Windows processes also have the ability to manage files that currently live in memory, as well as print operations [1].

2.2 Compare and Contrast OS Threads

For all intents and purposes, a thread is the most basic unit of an operating system, and they are the unit for which CPU time is allocated during a process's execution. A process can create multiple threads. A thread is a high-level abstraction that provides the ability to execute more than one sequence of code for a given executing program. Threads allow for parallel processing and concurrency. FreeBSD and Windows operating systems implement both user and kernel modes; where every single thread maintains its own stack and counter when running in kernel mode. Switching to user mode resets all of these values that a thread maintains. For FreeBSD, and most Unix-like operating systems, threading follows the POSIX standard (pthreads), whereas Windows follows its own implementation standards (Win32 threads) [1] [2] [6] [7]. The following code excerpt is from the Windows threading implementation:

```
typedef struct _DEVICE_EXTENSION {
  KEVENT evKill;
  PKTHREAD thread;
};

NTSTATUS StartThread(PDEVICE_EXTENSION pdx)
{
  NTSTATUS status;
  HANDLE hthread;
  KeInitializeEvent(&pdx->evKill, NotificationEvent, FALSE);
  status = PsCreateSystemThread(&hthread, THREAD_ALL_ACCESS,
    NULL, NULL, NULL, (PKSTART_ROUTINE) ThreadProc, pdx);
  if (!NT_SUCCESS(status))
    return status;
  ObReferenceObjectByHandle(hthread, THREAD_ALL_ACCESS, NULL,
    KernelMode, (PVOID*) &pdx->thread, NULL);
  ZwClose(hthread);
  return STATUS_SUCCESS;
}

VOID StopThread(PDEVICE_EXTENSION pdx)
{
  KeSetEvent(&pdx->evKill, 0, FALSE);
  KeWaitForSingleObject(pdx->thread, Executive, KernelMode, FALSE, NULL);
  ObDereferenceObject(pdx->thread);
}

VOID ThreadProc(PDEVICE_EXTENSION pdx)
{
  KeWaitForXxx(<at least pdx->evKill>);
  PsTerminateSystemThread(STATUS_SUCCESS);
}
```

Fig. 1: Windows Thread Implementation

Both operating systems allow for threads that execute exclusively in kernel space, inaccessible to users in user mode.

In comparison to Linux, threading in FreeBSD and Linux are the same. When comparing Linux pthreads to Windows Win32 threads, there are some inherent differences, however, they are extremely similar in the way that programmers can use them; almost identical, in fact. In terms of the API and thread interface, pthreads and Win32 threads are almost exactly the same. There are some key differences in the way threads are implemented between the kernels, though. And, furthermore, what goes on “under the hood”, so to speak, is very different between these kernels. FreeBSD and Linux threads are capable of sharing resources between other executing processes and threads. Windows systems are required to have at least one thread in execution at a given time to be processed by the I/O scheduler. A process in Windows absolutely must have at least one thread in order to execute [1]. Why this is the case is hard to say. It is my opinion that, generally speaking, the Windows kernel is more strict in terms of what it allows a programmer to do with the system’s

resources. In other words, it seems that Linux provides programmers with greater power and greater responsibility. To me, this makes sense because Linux kernels are open-source, and Windows kernels are not. This seems like a silly reason, but I'm not exactly able to go dig around in the Windows kernel and modify it, etc. In Linux, I am free to do so. I think a lot of the differences between Linux and Windows threading comes down to freedom and the ability to use a given system's resources freely.

2.3 Compare and Contrast OS CPU Scheduling

The I/O scheduler is an extremely vital subsystem for any operating system. The primary function of an I/O scheduler is to manage the amount of time a given thread or resource is allowed to execute on the system CPU at a given time. It should be noted that there are a number of different I/O scheduling algorithm implementations that exist. One scheduler that FreeBSD utilizes is the ULE scheduler which retrieves threads and processes by their priority and schedules their execution on the CPU by that given priority [8]. Windows, on the other hand, implements its own I/O scheduler which determines a given tasks priority, ranks it against other tasks in the queue, and reassigns priority based on other task's rankings.

In comparison to Linux, there are a large number of differences between the I/O schedulers of Windows and FreeBSD (probably too many to cover here). The primary difference is that Linux's default I/O scheduler, CFS (completely fair scheduler), focuses on a fair sharing of resources between threads and processes [5]. FreeBSD schedules by an assigned priority, and Windows schedules based on arbitrary rankings of priority between processes and threads [1] [8]. Again, I find it difficult to speculate why this is the case.

2.4 Section Conclusion

If time is the concern, then I believe that Windows would have the faster scheduling algorithm since it simply always grabs the highest priority task first. If fairness is the concern, then Linux CFS is the most fair in terms of sharing resources. In Windows, one cannot change the I/O scheduler to behave the way they want it to. In Linux, one can. Why this matters, I don't know. I think most of it, again, comes down to freedom, and the idea behind most concepts in Windows seems to be to limit that freedom. In this case, Windows decides what tasks are more important and prioritizes giving them CPU time. That way, a programmer cannot force his program to be the highest priority. In Linux, this is quite the opposite. Freedom.

3 IO MANAGEMENT

3.1 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. ULE's 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 system's 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 [9] [10] [11].

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 CFS's 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 isn't broken, don't 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, Linux's crypto encryption functionality dwarfs that of FreeBSD's [12] [13].

```
/*
 * 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. 2: Crypto IO Example

3.2 Compare and Contrast Windows and Linux IO

In Windows operating systems, the IO manager provides the interface to a given system's kernel-mode drivers. In other words, the IO manager manages the communication and interaction between the system's device drivers and user applications. The communication between the IO manager and the kernel's 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 [14] [15] [16] [17].

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 [12] [13].

3.3 Section 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 doesn't 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 system's kernel.

4 Memory Management

4.1 Compare and Contrast FreeBSD and Linux Memory Management

In the FreeBSD virtual memory system, every process is given its own private, protected 32 or 64-bit address space—-depending on the CPU's architecture. Address spaces cannot be accessed by other processes, and are divided into three segments: text, data, and stack. When a process begins, it is mapped into the system's virtual address space. A process may begin execution even if part of the process's data is not present in the system's main memory; if not present, the system will page the necessary data into memory so that the process can access it. A process can map a file to anywhere in its address space, and can create a shared mapping of a file which can be accessed by other running processes. Shared mapping allows the system to minimize on RAM usage. When system memory becomes scarce, FreeBSD uses swapping and demand paging in an effort to continue running processes. Demand paging is where the system only retrieves pages from disk and places them in RAM when it's necessary. FreeBSD will also take unused resources from other processes if they have not been recently used and attempt to redistribute those resources to processes that need them. Furthermore, the system may begin swapping a given process's context to a secondary storage place, usually a hard disk. Memory management in FreeBSD is geared toward multi-process context switching, where there are a large number of running processes and limited resources, and fairly sharing these resources amongst processes [18] [19].

In comparison, Linux and FreeBSD are extremely similar in terms of memory management, which makes sense because they are both, at their core, UNIX based operating systems. Linux systems support both 32 and 64-bit address spaces, and they also utilize a virtual memory system that provides a protected address space to processes, as well as paging, swapping, and shared virtual memory. Again, this makes a lot of sense, and if the method isn't broken, why fix it? Since RAM is expensive and a normal hard disk isn't, memory must be managed in a clever manner by the operating system to achieve an optimal pace. There are a few small differences between Linux and FreeBSD memory management. One such difference in FreeBSD is the system will reallocate unused resources to higher priority processes, whereas Linux relies on demand paging and swapping alone [20] [5].

4.2 Compare and Contrast Windows and Linux Memory Management

In the Windows virtual memory system, every process is given its own private, protected 32 or 64-bit address space—-depending on the CPU's architecture. Processes in Windows are provided 4 GB of private virtual address space. When a process begins, it is mapped into the system's virtual address space. A process may begin execution even if part of the process's data is not present in the system's main memory; if not present, the system will page the necessary data into memory so that the process can access it. A process can map a file to anywhere in its address space, and can create a shared mapping of a file which can be accessed by other running processes. Is this all sounding familiar? That's because, conceptually, almost everything that's true for memory management in FreeBSD is also true for Windows, with very few exceptions. Windows utilizes a method called "prefetching," in which files from the hard disk are retrieved and placed into physical memory before they are actually needed by the kernel. Windows also uses a method called "lazy allocation," in which memory will not be allocated until is it absolutely necessary. When system memory becomes scarce, Windows will begin copying out 4 KB address space pieces from RAM, and move them to the hard disk into the Pagefile.sys file. Pagefiles live in the root of a given partition and the size of a pagefile can be configured by the user.

Furthermore, a pagefile can store larger pages that don't fit into physical memory. Unlike FreeBSD, Windows will not take unused resources from idle processes. Windows systems will also use swapping when RAM is too full. Memory management in Windows, similar to FreeBSD, is designed for multi-process context switching, where there are a large number of running processes and limited resources, as well as sharing its limited resources equally amongst all running processes [21] [22].

For the most part, it appears that Windows and Linux have more similarities than differences when it comes to how they both approach memory management—-which is actually not very surprising for some reason. Virtual memory management, paging, use of swap memory, shared virtual memory; it's all the same in concept. The details regarding how these concepts are implemented under the hood, so to speak, may be different, but the ideas behind the "how" and the "why" are essentially the same. Which of these operating systems is faster or more efficient in their memory management is arguable. The best answer to that question is, "it depends on what you're trying to do." [20] [5].

4.3 Section Conclusion

The underlying theme of it all seems to be that RAM is a lot more expensive than a mechanical hard drive, so system memory management has to be economical by design. Most modern operating systems' memory management is predicated on the assumption that most computers will have more hard disk space immediately available than they will RAM at any given time; this seems to be a pretty solid assumption, at least for now. The core idea behind virtual address spaces, shared virtual memory, paging, and swapping is to make data available to the processor as quickly as possible. Since RAM is very expensive, and a hard disk drive is vastly cheaper, there is usually far more hard disk to work with than there is RAM space. Therefore, memory must be managed in a clever manner by the operating system in order to make up for this disparity in the availability of these resources. These very slightly varied management schemes are far from perfect, but it is interesting to see that, for the most part, Windows, FreeBSD, and Linux are in general agreement on how to approach the management of memory resources. One day this may change, though.

5 FILESYSTEMS

5.1 Compare and Contrast FreeBSD and Linux Filesystems

The FreeBSD operating system kernel supports both UFS and ZFS filesystems. Filesystems are a fundamental component to all modern operating systems, and many operating systems vary in which file system they support by default. Unix File System (UFS), as the same implies, is a file system that is used by many UNIX-like operating systems, and FreeBSD is no exception. A UFS disk volume has a number of key features; at the start of the volume's partition, there is a reserved space for the system's boot file blocks, followed by a superblock which contains several different important file system parameters. This disk volume is comprised of a number of cylinder groups which each contain a copy of the aforementioned superblock, nodes, and data blocks. The UFS is also known as the Fast File System (FFS). Given that the FreeBSD kernel is tuned in favor of performance, it makes sense that it uses a file system designed to be fast. The ZFS file system can also be used natively in FreeBSD. ZFS is a unique file system in that it behaves as both the file system and as a system volume manager. The primary perks of ZFS include data integrity, pooled storage, and performance. Overall, it's easy to see why these two filesystems work exceptionally well within FreeBSD systems [23] [24] [25].

Remember that in Linux and FreeBSD, everything is a file, and if it's not a file, it's a directory, or a process of some kind. So, as expected, Linux and FreeBSD utilize similar filesystems since both operating systems are descendants of UNIX. FreeBSD also natively supports a number of different Linux filesystems. Linux also natively supports several different filesystems. In fact, many Linux filesystems are commonly used in many modern day operating systems. Some of the Linux filesystems include ext2, ext3, JFS, and XFS. All of these filesystems have trade-offs in terms of speed, overall efficiency, and other various performance metrics. These metrics should be greatly considered when determining which of these filesystems to use for solving a given problem [26] [27].

5.2 Compare and Contrast Windows and Linux Filesystems

The Windows operating system utilizes the New Technology File System (NTFS). The primary design goals of NTFS are compatibility, reliability, and performance. NTFS is particularly fast at performing read, write, and search operations. When formatting an NTFS volume, a number of metadata files are created which contain data pertaining to all of the files that reside on a given volume. With NTFS, everything that lives on a disk volume is considered a file, and everything that exists within a given file is called an attribute. The NTFS filesystem supports a number of security features which includes file and folder permissions, and disk encryption. Windows also supports File Allocation Table (FAT) filesystems, which are inherited from the DOS days. FAT filesystem volumes are very simple in design, and they consist of a boot sector, a block allocation table, and storage space for storing folders and files [28] [29].

Linux and Windows have a number of notable difference between them in regard to filesystems. When it comes to NTFS, Linux can read and write to NTFS, but cannot boot from NTFS. Windows does not natively support Linux filesystems, or any other filesystems really, other than FAT and NTFS. Linux, in this case is far more versatile than Windows. There are, however, ways to read and write to non-native filesystems in Windows through some kind of third-party application, but whether or not that is advised is questionable [26] [27].

5.3 Section Conclusion

When it comes to filesystems, FreeBSD and Linux share a lot in common in that they support many of the same filesystems. This is surprising to no one. Also, not surprising is that Windows does not support any of the Linux based filesystems, and Linux supports all of the Windows based filesystems. In Windows, it seems, you only get what you pay for. Linux users benefit from the fact that the project is open-source, and you get what other people needed, and implemented, at some point in time. This helps to present Linux as a more well-rounded operating system than Windows.

6 Final Conclusion

Although Windows, FreeBSD, and Linux share a lot in common in terms of their implementation of key functionalities, there are also a lot of differences in how they operate, as well as the nomenclature that describes how these functionalities are expected and/or designed to behave. The small, nuanced differences may seem insignificant, but often they have large implications when considering which to use when it comes to solving a given problem. In terms of processes, all three operating systems maintain more similarities than differences. Same with threads. However, CPU scheduling proved to be where these three operating systems differ most. Furthermore, there was notable diversity amongst the different IO management models in these three operating systems. In terms of memory management, however, there is a lack of diversity where all three operating systems utilize almost the exact same memory management scheme. Lastly, in regard to file systems, there are a large number of them that exist in the wild, and all three of the observed operating systems utilize different file systems as their defaults. It's an impossible task to describe and remember every detail of how these operating systems work.

REFERENCES

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

[2] "Processes and daemons," https://www.freebsd.org/doc/handbook/basics-processes.html, freebsd.org, [Online; accessed 22-October-2017].

[3] "Process behavior in windows and linux," https://shankaraman.wordpress.com/tag/differences-between-windows-and-linux-process/, None, [Online; accessed 8-November-2017].

[4] "Is there any unix variant on which a child process dies with its parent?" https://unix.stackexchange.com/questions/158727/is-there-any-unix-variant-o None, [Online; accessed 8-November-2017].

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

[6] "System threads," https://www-user.tu-chemnitz.de/~heha/oney_wdm/ch09e.htm, Chemnitz University of Technology, [Online; accessed 22-October-2017].

[7] "Pscreatesystemthread routine," https://msdn.microsoft.com/en-us/library/windows/hardware/ff559932(v=vs.85).aspx, Microsoft, [Online; accessed 22-October-2017].

[8] "I/o scheduling in freebsd's cam subsystem," https://people.freebsd.org/~imp/bsdcan2015/iosched-v3.pdf, Netflix.com, [Online; accessed 22-October-2017].

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

[10] "I/o scheduling in freebsd's cam subsystem," https://www.freebsd.org/doc/en/books/design-44bsd/overview-io-system.html, Netflix.com, [Online; accessed 8-November-2017].

[11] "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].

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

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

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

[15] ——, *Microsoft Windows Internals, Part 2 6th Edition*. Microsoft Press, 2012.

[16] "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].

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

[18] "Memory management," https://www.freebsd.org/doc/en/books/design-44bsd/overview-memory-management.html, FreeBSD, [Online; accessed 25-November-2017].

[19] R. N. M. W. Marshall Kirk McKusick, George V. Neville-Neil, *The Design and Implementation of the FreeBSD Operating System*. Microsoft Press, 2012.

[20] "Memory management," http://www.tldp.org/LDP/tlk/mm/memory.html, The Linux Documentation Project, [Online; accessed 25-November-2017].

[21] "Ram, virtual memory, pagefile, and memory management in windows," https://support.microsoft.com/en-us/help/2160852/ram--virtual-memory--pa Microsoft, [Online; accessed 25-November-2017].

[22] "Virtual address space," https://msdn.microsoft.com/en-us/library/windows/desktop/aa366912(v=vs.85).aspx, Microsoft, [Online; accessed 25-November-2017].

[23] "Filesystems," https://www.freebsd.org/doc/handbook/filesystems.html, FreeBSD, [Online; accessed 5-December-2017].

[24] "The z file system (zfs)," https://www.freebsd.org/doc/handbook/zfs.html, FreeBSD, [Online; accessed 5-December-2017].

[25] "Unix file system (ufs)," https://en.wikipedia.org/wiki/Unix_File_System, Wikipedia, [Online; accessed 5-December-2017].

[26] "Filesystem," http://www.linfo.org/filesystem.html, Linfo.org, [Online; accessed 5-December-2017].

[27] "General overview of the linux file system," http://tldp.org/LDP/intro-linux/html/sect_03_01.html, The Linux Documentation Project, [Online; accessed 5-December-2017].

[28] "Understanding file systems," http://www.ufsexplorer.com/und_fs.php, UFS Explorer, [Online; accessed 5-December-2017].

[29] "Ntfs basics," http://www.ntfs.com/ntfs_basics.htm, NTFS, [Online; accessed 5-December-2017].

[30] D. Solomon and M. Russinovich, *Microsoft Windows Internals, Part 2 6th Edition*. Microsoft Press, 2012.

[31] "Posix threads vs. win32 threads," https://stackoverflow.com/questions/5644912/posix-threads-vs-win32-threads, None, [Online; accessed 8-November-2017].

[32] "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].