

Final Project: Processes, I/O and Memory Management ¹

Behnam Saeedi
CS444: Operating systems II

Spring 2017

Abstract

This document is the final project for CS444 Operating systems II class for the Spring term of 2017. The first operating system with the commercial name of Windows was released in 1990 at version 3.0. Followed by that in 1991 the first version of Linux operating system was released at version 0.01-0.1. Two years after that in 1993 the first distribution of FreeBSD was released to the public. Each one of these operating systems have a specific target audience and excel at performing certain operations. Many generations of these Operating systems have been released and they have gone through many changes since the first distributions of these kernels. They provide series of supports and services to the users and some are capable of performing certain tasks better than others. There are 3 topics presented in this document. These topics are Processes, I/O and Memory Management. This document will cover these three topics in 3 Operating systems. These operating systems are Linux, Windows and FreeBSD. After the material with regards to each one of the three topics a comparison is provided in order to demonstrate some of the differences between these kernels and compare them based on performance and approach. The first subsection of each kernel describes the implementation of processes, threads and CPU schedulers that are used in Linux, Windows and FreeBSD kernels. The second subsection of each kernel analyzes them based on the data structures, schedulers, algorithms, cryptography and I/O scheduling. The third and last subsection of each kernel section covers some of the details about how the memory management features implemented, what are some of their features and how do they stack up against Linux's implementation of those features. These information has been gathered by looking into a wide variety of resources which are listed in the references section of this document.

CONTENTS

1	Introduction	4
2	Linux	4
2.1	Process	4
2.1.1	Process	4
2.1.2	Thread	6
2.1.3	CPU Scheduling	7
2.2	I/O	8
2.2.1	Data Structure	8
2.2.2	Algorithms	8
2.2.3	I/O scheduling	8
2.2.4	Cryptography	9
2.3	Memory Management	9
2.3.1	Memory Management	9
2.3.2	Services	9
2.3.3	Pages	10
2.3.4	Features	10
3	Windows	11
3.1	Process	11
3.1.1	Process	11
3.1.2	Thread	12
3.1.3	Fiber	12
3.1.4	CPU Scheduling	12
3.2	I/O	13
3.2.1	Data Structure	13
3.2.2	I/O scheduling	13

3.2.3	Cryptography	14
3.3	Memory Management	14
3.3.1	Memory Management	14
3.3.2	Size	15
3.3.3	Service	15
3.3.4	Page	15
3.3.5	Feature	15
3.3.6	Comparing to Linux	16
4	FreeBSD	16
4.1	Process	16
4.1.1	Process	16
4.1.2	Thread	16
4.1.3	CPU Scheduling	16
4.2	I/O	16
4.2.1	Data Structure	17
4.2.2	I/O scheduling	17
4.2.3	Cryptography	17
4.2.4	Kernel System Queue Synopsis	17
4.3	Memory Management	19
4.3.1	Memory Management	19
4.3.2	Pages	19
4.3.3	Services and Features	20
4.3.4	FreeBSD and Linux Comparison	20
5	conclusion	20
	References	21

1 INTRODUCTION

An operating system has a set of responsibilities in order to allow users to be able to use their hardware. Among these requirements are important elements such as concurrent operations and processes, Input and output of file and memory management, interrupt handling and filesystem. Each one of these requirements are important for the operating system's performance. Linux, Windows and FreeBSD provide different solutions in order to satisfy these requirements. Each operating system provides its solution based on their target user and performance needs of kernel. Following sections will cover Linux Processes, I/O and Memory management in order to develop a control for comparison. Next section covers the same concepts from a windows perspective. It will cover the design and implementation of each item and provides a comparison of their approach with Linux. Finally, This document looks into the Processes, I/O and memory management of the FreeBSD operating system. Similar to windows section, a comparison will be provided between Linux and FreeBSD.

2 LINUX

The first Operating system that this document covers is the Linux Operating system. Linux 0.01-0.1 was developed in 1991. It lacked many features that we take for granted for our modern operating systems, include a graphical user interface. in the past 26 years, Linux has come a long way and the newest version of Linux is 16.04 LTS. This section covers the Processes, I/O and memory management features of Linux operating system.

2.1 Process

2.1.1 Process

In Linux processes are instances of running programs. In Linux the first process that is created with ID 1 is the init process. The processes could be viewed by two commands:

```
$ top
# and
$ ps
```

These two commands show a list of processes that are happening on the operating system. figures 1,2. top is a live view and ps is a static view of current processes. The init process could be found and viewed by the following command:

```
$ ps fax | grep [i]init
  1 ?          Ss      0:06 /sbin/init splash
```

Each process in Linux could be in one of the following states:

- **Running:** Task is being processed or is about to be assigned to CPU for process.
- **Waiting:** interruptible and un-interruptible process waiting for resources to be assigned.
- **Stopped:** The process has been stopped by a signal. The process is in a halt.
- **Zombie :** This is a terminated process that is still consuming resources.

In Linux each process has an identifier and a task vectors. in order to see this feature better, Linux provides the "pstree" command:

```

top - 13:11:08 up 5 days, 2:44, 4 users, load average: 1.75, 1.50, 1.71
Tasks: 313 total, 1 running, 310 sleeping, 2 stopped, 0 zombie
%Cpu(s): 2.8 us, 1.3 sy, 0.0 ni, 95.6 id, 0.2 wa, 0.0 hi, 0.1 si, 0.0 st
KiB Mem : 8051244 total, 634660 free, 4039868 used, 3376716 buff/cache
KiB Swap: 8264700 total, 8117508 free, 147192 used. 2602332 avail Mem

```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
2576	behnams+	20	0	501160	11464	7952	S	5.0	0.1	17:16.25	pulseaudio
5755	behnams+	20	0	1165668	322480	122760	S	4.3	4.0	5:50.73	chrome
16118	behnams+	20	0	1789264	332632	136556	S	4.3	4.1	29:31.75	chrome
5045	behnams+	20	0	988592	220724	77028	S	1.0	2.7	0:56.50	chrome
9364	behnams+	20	0	1100184	310564	140128	S	0.7	3.9	0:43.19	chrome
9895	behnams+	20	0	1015216	244816	73172	S	0.7	3.0	0:30.50	chrome
12102	root	20	0	0	0	0	S	0.7	0.0	0:00.60	kworker/u16:0
16249	behnams+	20	0	989772	178816	62168	S	0.7	2.2	0:50.72	chrome
779	root	20	0	597500	16220	10804	S	0.3	0.2	0:34.16	NetworkManager
1025	root	20	0	697624	116704	102948	S	0.3	1.4	60:56.44	Xorg
1051	root	20	0	0	0	0	S	0.3	0.0	0:23.28	rtss_usb_ms_1
3975	behnams+	20	0	28268	3936	2308	S	0.3	0.0	0:58.69	screen
4620	behnams+	20	0	2699060	447744	89736	S	0.3	5.6	4:00.11	chrome
5101	behnams+	20	0	910272	142564	67328	S	0.3	1.8	0:28.73	chrome
12554	behnams+	20	0	42060	4024	3152	R	0.3	0.0	0:00.06	top
16251	behnams+	20	0	1060540	194164	56960	S	0.3	2.4	0:39.24	chrome
16299	behnams+	20	0	1170436	365940	70648	S	0.3	4.5	4:35.79	chrome
1	root	20	0	119820	5468	3696	S	0.0	0.1	0:06.83	systemd
2	root	20	0	0	0	0	S	0.0	0.0	0:00.22	kthreadd
3	root	20	0	0	0	0	S	0.0	0.0	0:38.59	ksoftirqd/0
5	root	0	-20	0	0	0	S	0.0	0.0	0:00.00	kworker/0:0H
7	root	20	0	0	0	0	S	0.0	0.0	3:00.84	rcu_sched
8	root	20	0	0	0	0	S	0.0	0.0	0:00.00	rcu_bh

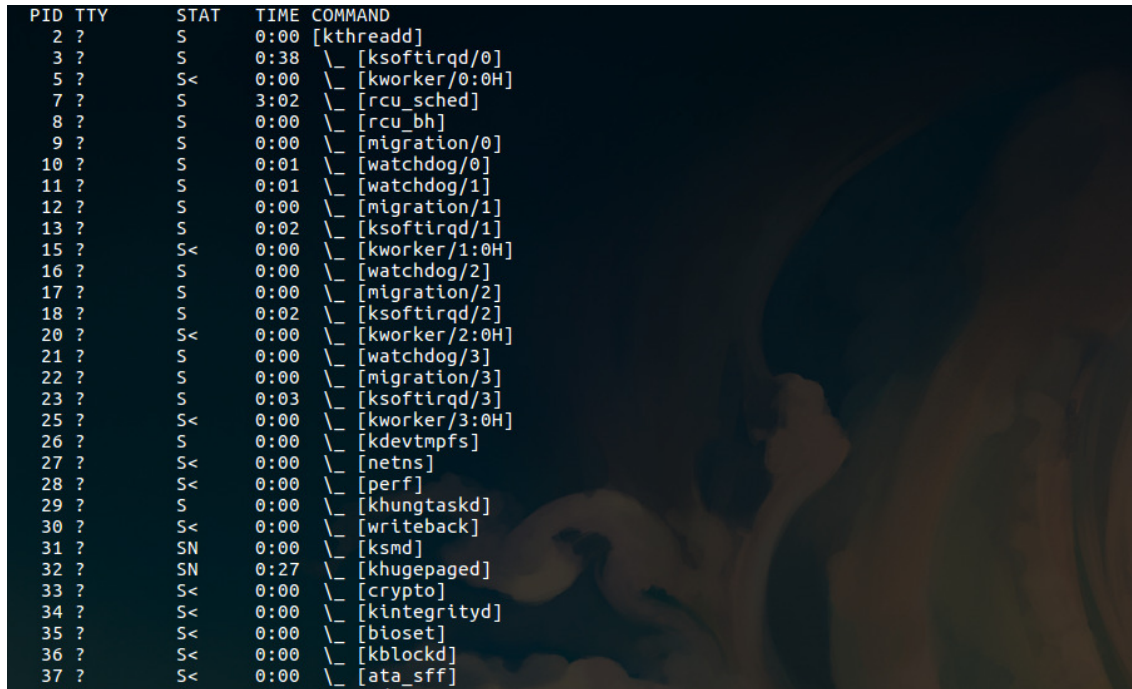
Figure 1. The view of "top" command in Linux shell

```

$ pstree
systemdModemManager{gdbus}
    {gmain}
        NetworkManagerdhclient
            dnsmasq
                {gdbus}
                {gmain}
Xvfb{llvmpipe-0}
    {llvmpipe-1}
    {llvmpipe-2}
    {llvmpipe-3}
accounts-daemon{gdbus}
    {gmain}
acpid
agetty
at-spi-bus-laundbus-daemon
    {dconf worker}
    {gdbus}
    {gmain}
at-spi2-registr{gdbus}
    {gmain}
avahi-daemonavahi-daemon
bluetoothd
#
#
#

```

Task vector is an array of pointers in a data structure sense that points to ever task struct data structure in the



PID	TTY	STAT	TIME	COMMAND
2	?	S	0:00	[kthreadd]
3	?	S	0:38	_ [ksoftirqd/0]
5	?	S<	0:00	_ [kworker/0:0H]
7	?	S	3:02	_ [rcu_sched]
8	?	S	0:00	_ [rcu_bh]
9	?	S	0:00	_ [migration/0]
10	?	S	0:01	_ [watchdog/0]
11	?	S	0:01	_ [watchdog/1]
12	?	S	0:00	_ [migration/1]
13	?	S	0:02	_ [ksoftirqd/1]
15	?	S<	0:00	_ [kworker/1:0H]
16	?	S	0:00	_ [watchdog/2]
17	?	S	0:00	_ [migration/2]
18	?	S	0:02	_ [ksoftirqd/2]
20	?	S<	0:00	_ [kworker/2:0H]
21	?	S	0:00	_ [watchdog/3]
22	?	S	0:00	_ [migration/3]
23	?	S	0:03	_ [ksoftirqd/3]
25	?	S<	0:00	_ [kworker/3:0H]
26	?	S	0:00	_ [kdevtmpfs]
27	?	S<	0:00	_ [netns]
28	?	S<	0:00	_ [perf]
29	?	S	0:00	_ [khungtaskd]
30	?	S<	0:00	_ [writeback]
31	?	SN	0:00	_ [ksmd]
32	?	SN	0:27	_ [khugepaged]
33	?	S<	0:00	_ [crypto]
34	?	S<	0:00	_ [kintegrityd]
35	?	S<	0:00	_ [bioaset]
36	?	S<	0:00	_ [kblockd]
37	?	S<	0:00	_ [ata_sff]

Figure 2. The view of "ps fax" command in shell

system. Furthermore, Unix introduced a system known as IPC or Inter-Process Communication. This allows different processes to communicate with one another.

2.1.2 Thread

From the kernel's perspective there is no distinction between threads and processes. Threads are series of processes that get access to the same memory space. From a CPU and scheduling perspective each thread gets its own time slot. We can trace and see each of these threads that are related to a process. Running the following code reveals an interesting fact about the Linux processes:

```
$ Htop
# or
$ top -H
# or
$ ps -T -p 1
```

init actually does not launch any threads meaning that none of its children and the init itself would have access to the memory space of the child processes. Using the following command we can see the threads of each process:

```
$ ps -T -p $(ps efax | grep [m]y-app | awk '{print $1}')
# replace my-app with name of a program such as mysqld -> [m]ysqld
$ ps -T -p $(ps efax | grep [m]ysqld | awk '{print $1}')
```

PID	SPID	TTY	TIME	CMD
8302	8302	?	00:00:00	mysqld
8302	8307	?	00:00:00	mysqld

```

8302  8308  ?          00:00:03 mysqld
8302  8309  ?          00:00:04 mysqld
8302  8310  ?          00:00:03 mysqld
8302  8311  ?          00:00:03 mysqld
8302  8312  ?          00:00:03 mysqld
8302  8313  ?          00:00:03 mysqld
8302  8314  ?          00:00:03 mysqld
8302  8315  ?          00:00:03 mysqld
8302  8316  ?          00:00:03 mysqld
8302  8317  ?          00:00:03 mysqld
8302  8318  ?          00:00:03 mysqld
8302  8320  ?          00:00:03 mysqld
8302  8321  ?          00:00:05 mysqld
8302  8322  ?          00:00:00 mysqld
8302  8323  ?          00:00:04 mysqld
8302  8324  ?          00:00:00 mysqld
8302  8325  ?          00:00:00 mysqld
8302  8326  ?          00:00:00 mysqld
8302  8327  ?          00:00:00 mysqld
8302  8328  ?          00:00:00 mysqld
8302  8329  ?          00:00:00 mysqld
8302  8330  ?          00:00:00 mysqld
8302  8331  ?          00:00:00 mysqld
8302  8332  ?          00:00:00 mysqld
8302  8333  ?          00:00:00 mysqld
8302  8336  ?          00:00:00 mysqld

```

All of these threads use the same address space as the mysqld process that was displayed in ps. In implementation, thread is actually implemented as a process. We can imagine process as a memory space and threads as units of execution of the program. many programs have a single threads for the process. Kernel to has a 1-1 ratio between both of them, it treats threads as full processes that share the same memory space. Even we try to spawn a new thread we use the same command as spawning a new process and pass different arguments to indicate if we want our new thread to be independent process or not.

2.1.3 CPU Scheduling

Linux uses complex data structures in order to become very efficient in binding processes to the CPU. Each processor gets its own run queue and selects processes from that specific run queue. This creates an issue where one processor ends up idling while there are processes on other processor's queue. In order to avoid this, the queues get rebalanced periodically. Each process gets a bitmask in order to identify which processor it can be processed on. This mask could be reassigned but re-balancing will not override affinity. The processor follows the following algorithm: [1][page 6]

- Highest priority queue with runnable processes is selected
- First process on that queue is selected
- Quantum size is calculated
- Runs until its time runs out
- its put on the expire queue
- the process repeats

As of 2007-04-13 a new patch put the CFS or completely fair scheduler in Linux kernel. [2]. As this memo mentions, a complete rewrite of the Linux task scheduler goes into place after this patch and replaces the O(1) scheduler that was used before. This scheduler follows the following algorithm: [3]

- The left most node of the scheduling tree is sent for execution.
- If completed, it is removed from the system and scheduling tree.
- If reached maximum execution time, it is stopped (voluntarily or via interrupt) and reinserted into the scheduling tree and a new execution time is calculated
- repeat

2.2 I/O

In this subsection we are going to have a closer look on the I/O aspect of three common operating systems. These Operating systems are Linux, Windows and FreeBSD. In the first section we will cover the Linux in order to hold it a control to compare the other two operating system s to it. In this section we will talk about data structures, I/O scheduling, Cryptography and implementations.

2.2.1 Data Structure

According to Linux documentation the Linux Kernel provides various versions of double linked list specifically purposed for I/O. Some examples of these data structures are doubly linked list, B+ tree, priority heap and many many more. [4]. Everything that requires a data structure in Linux is dependent on these few models. The thing to consider is that the Linux actually does not create a data based structure. Instead it provides the pointers needed in order to link them. The developers can create their own structs and include this pointer structure into their data containers. This has an interesting and useful side effect. Due to this method of implementation user has the capability to generate a linked list where the type of the data elements does not need to match. This means a link containing and integer could be followed by a link that contains character.

2.2.2 Algorithms

There are many different scheduling algorithms. These algorithms include:

- Random scheduling (RSS)
- First In, First Out (FIFO), also known as First Come First Served (FCFS)
- Last In, First Out (LIFO)
- Shortest seek first, also known as Shortest Seek / Service Time First (SSTF)
- Elevator algorithm, also known as SCAN (including its variants, C-SCAN, LOOK, and C-LOOK)
- N-Step-SCAN SCAN of N records at a time
- FSCAN, N-Step-SCAN where N equals queue size at start of the SCAN cycle
- Completely Fair Queuing (CFQ) on Linux
- Anticipatory scheduling
- Noop scheduler
- Deadline scheduler
- mClock scheduler [5]
- Budget Fair Queuing (BFQ) scheduler [6]
- Kyber [7]

2.2.3 I/O scheduling

Checking the Linux kernel files we can see that the algorithms which Linux supports are NOOP, deadline, and CFQ. In previous assignment we implemented the SSTF version for Linux. (it is good to note that there are 3rd party implementations of that available).

2.2.4 Cryptography

Linux provides one of the most extensive cryptographic APIs available. It supports a wide variety of algorithms available for memory. There is an extensive list of cypher algorithms available: [8]

- Symmetric ciphers
- AEAD ciphers
- Message digest, including keyed message digest
- Random number generation
- User space interface

These algorithms are used in order to create the following templates available to the users: [9]

- aes: or The Advanced Encryption Standard is a symmetric block cypher
- ecb(aes): or Electronic Codebook and block mode of operations
- cmac(aes): or Cipher-based Message Authentication Code
- ccm(aes): or Counter with CBC-MAC
- rfc4106(gcm(aes)): Galois/Counter Mode (GCM) in Advanced Encryption Standard
- sha1: or Secure Hash Algorithm 1 which is crypto-hash function
- hmac(sha1): or Keyed-hash message authentication code (HMAC)

As mentioned, the Linux Kernel provides both synchronous and asynchronous operations depending on the users need.

2.3 Memory Management

2.3.1 Memory Management

Due to size limitations on the amount of memory available to a kernel it is extremely important to have a subsystem dedicated to organize and manage distribution and handling of memory. This task falls under Kernel's responsibilities under Kernel utilities known as Memory management tools. This section will be covering the basics of Linux Kernel's memory management tools in depth. Furthermore, it will provide a brief description of Pages, Services, Features and capabilities of Linux Kernel's memory management.

2.3.2 Services

2.3.2.1 VM: Linux provides the memory to processes at the same time through a concept known as virtual memory. This means every process has access to all of the memory at the same time giving it the illusion of a much larger memory than the actual physical RAM.[10]

2.3.2.2 Address Space Protection: Linux Kernel Guarantees that each memory gets its own virtual address space. Each virtual address space is completely independent of others and is achieved through offsets in memory.[10]

2.3.2.3 Memory Mapping: In Linux, files could be mapped directly to a process's virtual address space. This process is known as memory mapping. This feature is one of the requirements for the memory management and is present in Linux, Windows and FreeBSD.[10]

2.3.3 Pages

Since the size of the virtual memory is significantly larger than the physical memory there is a need for a reliable way to address the memory. The process of assigning the virtual memory to the physical memory is known as paging. Pages are 4KB units of contiguous memory on the RAM and the basic unit of memory known by both Kernel and the CPU. This is regardless of the fact that both Kernel and CPU do have access to individual bits written on each page.[11]. The operating system could fall into two dangerous scenarios: Data Collision and unused data. These two problems are the two extremes of the problems that can occur. memory collision happens when one physical memory is addressed by more than one virtual memory. The second issue occurs when a physical memory is not addressed by any virtual memory of running processes and is not being used efficiently.[10]

2.3.3.1 Efficiency and Demand Paging: One way of preserving efficiency of the program is to avoid allocation of virtual memory while the process is not running. This is known as Demand Paging. Linux uses this method to take care of file load and maps it to the memory.[10]

2.3.4 Features

Linux kernel provides a set of features that allow users to fully take advantage of the provided memory.

2.3.4.1 Cache: As discussed in the class, in order to have the programs run more efficiently, concept of caching is introduced. Task of this feature is to keep a copy of the data for future use with a faster access time than the storage drive. This process is known as caching and it can take place on several different levels.

- Buffer Cache: Block device driver's data buffers
- Page Cache: Memory map speed boost (allowing faster file access)
- Swap Cache: file data that is going to be kicked off of the memory. (dirty pages only)
- Hardware Cache: Translation, look-aside buffers [10]. CPU will not check the entire memory every time it needs access to data.

2.3.4.2 Swapping: What if a process requires a memory and there are no more available memory to be allocated for the process? In this case the Kernel has the important task of opening up more memory. In order to deal with this the kernel will select a process that is unlikely to be used soon again and places it on a different storage (drives). Then the recently emptied memory could be used for memory allocation.[10]. There are several algorithms available for kernel to make the important decision of which process needs to be swapped. The current algorithm is LRU or Least Recently Used. Kernel looks for each process and if that process has not been used for a long time, it gets swapped out from the memory. Lets assume that kernel makes a bad decision in swapping a process and every time it swaps a process, that process gets requested. This will lead to an issue known as thrashing. In this scenario, the performance suffers significantly since the entire CPU's execution time gets limited to the speed of the storage device.

2.3.4.3 Threads: IEEE Computer Society has series of standards that are specific to compatibility across multiple operating systems. These standards are known as The Portable Operating System Interface or POSIX. One set of libraries provided by POSIX are known as the thread libraries for c and C++. These libraries operate based on the concept of concurrent processing. In Linux, this concept allows multiple programs to be run in a way that gives the illusion of multiple programs running at the same time to the user. It is important to consider that each CPU package still handles each process one at the time per physical core. In Linux threads can be create by a parent process. This concept is further described in the Processes portion of the writing assignments.[12].

2.3.4.4 Shared Virtual Memory: As covered earlier, process has the capability of creating threads. This however, poses an issue. The parent process needs to be able to monitor the data processed by each thread.

In order to accommodate with this need, threads in Linux share the same address space with other threads of the same parent and the parent itself.[10]. Linux requires each process and thread that shares memory to have its physical page frame to be stored in a page table entry stored at all sharing processes and threads.[10]

2.3.4.5 Access Control: Based on what was covered on Shared Virtual Memory (2.3.4.4), The page table needs to contain the access control information for each page. As a side effect, now the process can also see who else has access to a specific memory space. This information could be found in the following lookup table (refbitfields) this table was found at source [10].

Table 1
Meaning of bit fields and OS control on access.[10]

Field	Meaning
V	Valid, if set this PTE is valid,
FOE	“Faulton Execute”, Whenever an attempt to execute instructions in this page occurs, the processor reports a page fault and passes control to the operating system,
FOW	“Faulton Write”, as above but page fault on an attempt to write to this page,
FOR	“Faulton Read”, as above but page fault on an attempt to read from this page,ASM
ASM	Address Space Match. This is used when the operating system wishes to clear only some of the entries from the Translation Buffer,
KRE	Code running in kernel mode can read this page,
URE	Code running in user mode can read this page,
GH	Granularityhint used when mapping an entire block with a single Translation Buffer entry rather than many,
KWE	Code running in kernel mode can write to this page,
UWE	Code running in user mode can write to this page,

2.3.4.6 Paging tables: In Linux the paging happens through three page tables. These tables are Level 1, Level2 and Level 3. This is achieved by having each level providing an index number containing the frame for the following level similarly to what is described in Intel architecture but with only 3 levels.

3 WINDOWS

Next covered operating system is Windows. Windows provides a different approach towards these requirements. Their approach is more developer oriented and they provide elements in away that gives the developer and the user more control over the operating system. This section covers the implementation of these requirements and compares them to Linux operating system.

3.1 Process

3.1.1 Process

In windows the processes are explained very similarly to Linux. A process is an executing program as described by MSDN. [13]. Windows also creates a new terminology called “Job Object”. Job object allows multiple processes to be managed as a single unit. These object could have features assigned by OS such as namable, securable, etc.[13].

Similarly to Linux a process gets its own independent virtual address space. Each process contains one or more independently executing threads. Threads can create threads, processes and fibers. (Fibers are explained in

Fibers subsection). It appears that in Windows, Process is a container that contains all of the threads. In other words, Process describes the address space which its threads will be using. Processes could be viewed and managed using several available tools.

3.1.1.1 Command: Tasklist: Windows provides few tools in order to analyze the processes. The first tool was the tasklist. It provides a list of all of the tasks and processes that are running on the processor. Later, a Graphical user interface was developed for this command with the same name.

3.1.1.2 Close Program: The second tool that was introduced was called Close Program. This gave the user the ability of controlling the programs that were running at the time. This however did not last long since by introduction of multi processing it needed to change for more versatile tools.

3.1.1.3 Task Manager: The third tool is a graphical user interface designed to display the processes that are running on the system very similarly to the top command in Linux. This GUI is the Windows Task Manager.

3.1.2 Thread

Threads are also similar to Linux threads. The MSDN describes the threads as the basic unit which the kernel allocates time. A thread is part of a process that actually executed the programmed code. [13]. This functionality of a process still is fairly similar to the functionality of a thread in Linux with the difference that processes do not necessarily have to have threads in Linux where in windows each process must at least have one or more threads in order to execute code. From a code perspective, Windows has a completely different sets of syntax for creation and destruction of threads and processes.

In order to view threads in windows there aren't any tools provided that are shipped with the kernel for the computer user. However, there are third party programs that display the threads of each process. **Process Explorer** is a third-party tool that gives the user the ability to monitor different threads in windows processes.

3.1.3 Fiber

The MSDN document [13] introduces a concept called "fiber". Fiber is a unite of execution that is created by a thread and needs to be manually scheduled by the threads to be executed. [13]. Fibers share the thread context of their parent thread.

3.1.4 CPU Scheduling

Windows system scheduler controls the multitasking process by deciding which processing thread will receive the next time slice. This process takes advantage of scheduling priorities.

3.1.4.1 Priorities: There are 6 possible processing priority classes for each process.

- Idle
- Bellow normal
- Normal
- High
- Real-time

Each priority class itself has 7 priority levels for each thread:

- Idle

- Lowest
- Below normal
- Normal
- Above normal
- Highest
- Time critical

The priority classes could be managed by the user in Task Manager. By default the priority of a process is normal. The priority levels on the other hand are assigned by the developer to each thread or is inherited from the parent process. This concept is described by the nice command in Linux. [14]. In windows the value of this priority is a number between 1 to 31 where 31 is the highest priority and 1 is Idle.

3.1.4.2 Context Switching: Similarly to Linux the scheduler maintains a queue for ready tasks and priorities. Unlike Linux, the queue contains threads and their priorities only. The context switching happens in 5 steps:

- Save the context of the current thread
- Place the current thread at the end of the priority queue
- Find the next highest priority queue with ready threads
- remove the new thread, load its context and execute
- repeat

[15].

3.1.4.3 Algorithm: According to last sub section, windows uses a prioritized Round-Robin technique with multi-level feedback for priority handling. This technique has been the algorithm used for scheduling since NT. However, Vista did introduce some new heuristics in order to set priorities of certain processes. This scheduler is referred to as "Multi-level feedback queue" which still is a Round-Robin scheduler. Windows has also introduced a new concept to scheduling. This concept is referred to as User-mode Scheduling or UMS.[14]. This allows processes to schedule their own threads and we can see that in the fiber concept discussed earlier.

3.2 I/O

The next element which we are going to look into is the Windows, We are going to look into data structures used in Windows I/O, I/O scheduling, Windows Cryptography and finally a brief comparison of Windows performance stacked next to Linux.

3.2.1 Data Structure

Windows has a radically different approach to data structures than Linux. Windows has a specific data structure in place known as IRP or Input/Output Request Packets. [16]. According to windows documentation, Unlike Linux, this data structure contains two elements. a pointer to a `DEVICE_OBJECT` and a pointer to an IRP. The IRQs are large data files containing all of the necessary driver critical information. This seems to be an inefficient approach to data structure design.

3.2.2 I/O scheduling

Windows scheduler is a very complex setup that could be thoroughly covered in a full document of its own. In the source found, it is clear that windows uses a variation of multilevel feedback queue. [17]. As covered in

earlier assignment, it works based on setting a specific value for the priority of each task described through a set of classes.[18]. This value could be set through task manager. A similar concept to this exists in Linux which is known as niceness of a program. A program can decide how much of CPU time it would like to have. The nicer a program the lower the scheduling priority becomes. The idea behind this concept is based on the fact that true concurrency does not exist. The CPU creates the illusion of concurrency by switching the process that is running and making sure that all of the processes get access to resources.

3.2.3 Cryptography

Windows has a service known as Cryptographic Service Providers or CSP. The task of this contains a vast library of cryptographic standard and algorithms. [19]. According to this documentation minimum requirement for the CSP assets are DLLs containing implementation the functions in CryptoSPI (a system program interface). The algorithms used are consistent of:

- **Digital signature algorithm:** In this algorithm, every type specifies only one key exchange algorithm. Then using the application's specification, provide type is selected.
- **Key BLOB formats:** Similar to last element with the exception that the tool creates a digital signature by selecting the appropriate type.
- **Digital signature format:** The Type is determined as the format of the key BLOB is used to export keys from the CSP and to import keys into a CSP.
- **Session key derivation scheme:** A session key is generated by hash which leads to selection of the type.
- **Key length:** Public and private keys are used.
- **Default modes:** Block encryption padding.

Some of the implementations include:

- PROV_RSA_FULL
- PROV_RSA_AES
- PROV_RSA_SIG
- PROV_RSA_SCHANNEL
- PROV_DSS
- PROV_DSS_DH
- PROV_DH_SCHANNEL
- PROV_FORTEZZA
- PROV_MS_EXCHANGE
- PROV_SSL

It is not as simple to compare the cryptography methods that are available. However we can definitely see some familiar types that are present in windows version such as AES and SSL.

3.3 Memory Management

3.3.1 Memory Management

Windows provides its own series of services and features for memory management. In this section we will cover some of these services and features.

3.3.2 Size

In windows the 32-bit processes take advantage of 2 GB of memory by default. This size could be increased to 3/4 GB on 32/64-bit Windows distributions. Furthermore 64-bit processes can take advantage of up to 8182 GB of virtual memory space.

3.3.3 Service

The Windows kernel provides the following services to the user for memory management:

- Address Mapping
- Paging
- Memory Mapped Files
- Copy on Write Memory
- Physical Memory Allocation and Use.

Furthermore, On working set (Set of pages physically present), there are 5 key parts:

- 1) Working Set Manager
- 2) Process / Stack Swapper
- 3) Modified Page writer
- 4) Mapped Page Writer
- 5) Zero Page thread: This thread is always running and has the task of setting the memory to zero. This happens on a hardware level and is very fast.

3.3.4 Page

The Windows memory locking feature is much more fine grain than both Linux and FreeBSD even on memory subsystem level. This is in comparison to Linux and FreeBSD where they have a Kernel level lock. Windows provides 2 page sizes: Large (2 MB) and Small (4 KB). Furthermore, These pages can classify under 4 states:

- Free: The memory is free to be used.
- Reserved: Memory is requested but it is not yet being used. In other words, that memory is private.
- Committed: This memory falls under Private and valid mapping.
- Sharable: This memory is similar to Committed with the only difference being that this memory is not private.

3.3.5 Feature

In Windows, Process acts as a container which holds threads. This is explained further in the Processes section of writing assignments. To recap. These processes fall under a process group and each at least have to have one thread. These processes could create heap, stack, or a combination of any number of them if the developer chooses to. This provides a unique opportunity for the process to switch its stack on the fly as the program is running. Finally windows has a unique feature called Fibers. Fibers are user level scheduled abstract processes that the developer needs to take care of its scheduling. From the kernel's perspective all of fibers within a process are a single thread.

3.3.6 Comparing to Linux

There are several major differences between how Linux and Windows handle the memory. Windows has a completely different construct for the processes. These differences are in Windows' advantage since they have managed to come up with a fine tune setup in order to improve the kernel's performance. Linux has a rather more simplistic but effective approach towards these problems. Windows provides more options and a more developer oriented approach towards these issues.

4 FREEBSD

The final operating systems that this document covers is the FreeBSD kernel. FreeBSD has always been far behind the Linux operating system. The developers of the FreeBSD usually spend more time on refining and implementing different features of the kernel. FreeBSD is very similar to Linux, however, some modules and requirements are implemented much more elegantly and efficiently than Linux operating system. Furthermore, this section will compare FreeBSD operating system to the Linux.

4.1 Process

4.1.1 Process

According to the documentation of FreeBSD, the processes are designed very similarly to Linux processes. The commands used for monitoring them are too same as Linux.[20]

4.1.2 Thread

The documentation for the FreeBSD addresses 1:1 threading and 1:n Threading and describes the basics of the two threading methods. [21] Furthermore, it explains the libraries which implement threading in FreeBSD. It appears many of the features described in documentation are similar to Linux's implementation.

4.1.3 CPU Scheduling

FreeBSD Kernel has two available schedulers. The ULE scheduler first introduced in FreeBSD 5.0 and The 4.4BSD scheduler.

4.1.3.1 4.4 BSD scheduler: This scheduler was originally designed for Linux kernel in order to address interactivity issues. This scheduler has a single run-queue and is variation of Round-Robin. This queue is in fact a double linked list and is shared across all the CPUs. The reason to that is to avoid complex load balance algorithms.[22][Page 2].

4.1.3.2 ULE Scheduler: ULE scheduler is based on the BSD scheduler and improves on some of its features in order to improve efficiency specially with single processor systems. This scheduler is disabled by default in favor of the traditional BSD scheduler. Furthermore, it is fair by default but it could be set to favor specific processes.

4.2 I/O

In this section we are going to dive into the FreeBSD kernel and have a look at the data structures, I/O scheduling, cryptography and implementations of them. Then we are going to compare those to Linux kernel and its features.

4.2.1 Data Structure

In the section below titled, "Kernel System Queue Synopsis" a synopsis of the system queue library is provided. [23]. This is identical to Linux approach to data structures. This means FreeBSD too provides only a pointer for the developer to include in their own container providing a flexible linked list.

4.2.2 I/O scheduling

FreeBSD Kernel has two available schedulers. The ULE scheduler first introduced in FreeBSD 5.0 and The 4.4BSD scheduler.

- **4.4BSD scheduler:** this scheduler was originally designed for Linux kernel in order to address interactivity issues. This scheduler has a single run-queue and is variation of Round-Robin. This queue is in fact a double linked list and is shared across all the CPUs. The reason to that is to avoid complex load balance algorithms.[22][Page 2].
- **ULE scheduler:** ULE scheduler is based on the BSD scheduler and improves on some of its features in order to improve efficiency specially with single processor systems. This scheduler is disabled by default in favor of the traditional BSD scheduler. Furthermore, it is fair by default but it could be set to favor specific processes.

4.2.3 Cryptography

The FreeBSD takes advantage of Linux OpenCrypto, crypto and cryptodev libraries. They have been working constantly to significantly reduce the overhead involved with using this API. [24]. Another supported encryption method is the OpenSSL cryptographic based on libcrypto.[25].

4.2.4 Kernel System Queue Synopsis

The following code could be found at FreeBSD library manual. [23]. This tool is capable of providing kernel and user-level to hardware crypto device for hash handling purposes.

```
#include <sys/queue.h>
SLIST_CLASS_ENTRY(CLASSTYPE);
SLIST_CLASS_HEAD(HEADNAME, CLASSTYPE);
SLIST_EMPTY(SLIST_HEAD *head);
SLIST_ENTRY(TYPE);
SLIST_FIRST(SLIST_HEAD *head);
SLIST_FOREACH(TYPE *var, SLIST_HEAD *head, SLIST_ENTRY NAME);
SLIST_FOREACH_FROM(TYPE *var, SLIST_HEAD *head, SLIST_ENTRY NAME);
SLIST_FOREACH_FROM_SAFE(TYPE *var, SLIST_HEAD *head, SLIST_ENTRY NAME,
    TYPE *temp_var);
SLIST_FOREACH_SAFE(TYPE *var, SLIST_HEAD *head, SLIST_ENTRY NAME,
    TYPE *temp_var);
SLIST_HEAD(HEADNAME, TYPE);
SLIST_HEAD_INITIALIZER(SLIST_HEAD head);
SLIST_INIT(SLIST_HEAD *head);
SLIST_INSERT_AFTER(TYPE *listelm, TYPE *elm, SLIST_ENTRY NAME);
SLIST_INSERT_HEAD(SLIST_HEAD *head, TYPE *elm, SLIST_ENTRY NAME);
SLIST_NEXT(TYPE *elm, SLIST_ENTRY NAME);
SLIST_REMOVE(SLIST_HEAD *head, TYPE *elm, TYPE, SLIST_ENTRY NAME);
```

```

SLIST_REMOVE_AFTER(TYPE *elm, SLIST_ENTRY NAME);
SLIST_REMOVE_HEAD(SLIST_HEAD *head, SLIST_ENTRY NAME);
SLIST_SWAP(SLIST_HEAD *head1, SLIST_HEAD *head2, SLIST_ENTRY NAME);
STAILQ_CLASS_ENTRY(CLASSTYPE);
STAILQ_CLASS_HEAD(HEADNAME, CLASSTYPE);
STAILQ_CONCAT(STAILQ_HEAD *head1, STAILQ_HEAD *head2);
STAILQ_EMPTY(STAILQ_HEAD *head);
STAILQ_ENTRY(TYPE);
STAILQ_FIRST(STAILQ_HEAD *head);
STAILQ_FOREACH(TYPE *var, STAILQ_HEAD *head, STAILQ_ENTRY NAME);
STAILQ_FOREACH_FROM(TYPE *var, STAILQ_HEAD *head, STAILQ_ENTRY NAME);
STAILQ_FOREACH_FROM_SAFE(TYPE *var, STAILQ_HEAD *head, STAILQ_ENTRY NAME,
    TYPE *temp_var);
STAILQ_FOREACH_SAFE(TYPE *var, STAILQ_HEAD *head, STAILQ_ENTRY NAME,
    TYPE *temp_var);
STAILQ_HEAD(HEADNAME, TYPE);
STAILQ_HEAD_INITIALIZER(STAILQ_HEAD head);
STAILQ_INIT(STAILQ_HEAD *head);
STAILQ_INSERT_AFTER(STAILQ_HEAD *head, TYPE *listelm, TYPE *elm,
    STAILQ_ENTRY NAME);
STAILQ_INSERT_HEAD(STAILQ_HEAD *head, TYPE *elm, STAILQ_ENTRY NAME);
STAILQ_INSERT_TAIL(STAILQ_HEAD *head, TYPE *elm, STAILQ_ENTRY NAME);
STAILQ_LAST(STAILQ_HEAD *head, TYPE *elm, STAILQ_ENTRY NAME);
STAILQ_NEXT(TYPE *elm, STAILQ_ENTRY NAME);
STAILQ_REMOVE(STAILQ_HEAD *head, TYPE *elm, TYPE, STAILQ_ENTRY NAME);
STAILQ_REMOVE_AFTER(STAILQ_HEAD *head, TYPE *elm, STAILQ_ENTRY NAME);
STAILQ_REMOVE_HEAD(STAILQ_HEAD *head, STAILQ_ENTRY NAME);
STAILQ_SWAP(STAILQ_HEAD *head1, STAILQ_HEAD *head2, STAILQ_ENTRY NAME);
LIST_CLASS_ENTRY(CLASSTYPE);
LIST_CLASS_HEAD(HEADNAME, CLASSTYPE);
LIST_EMPTY(LIST_HEAD *head);
LIST_ENTRY(TYPE);
LIST_FIRST(LIST_HEAD *head);
LIST_FOREACH(TYPE *var, LIST_HEAD *head, LIST_ENTRY NAME);
LIST_FOREACH_FROM(TYPE *var, LIST_HEAD *head, LIST_ENTRY NAME);
LIST_FOREACH_FROM_SAFE(TYPE *var, LIST_HEAD *head, LIST_ENTRY NAME,
    TYPE *temp_var);
LIST_FOREACH_SAFE(TYPE *var, LIST_HEAD *head, LIST_ENTRY NAME,
    TYPE *temp_var);
LIST_HEAD(HEADNAME, TYPE);
LIST_HEAD_INITIALIZER(LIST_HEAD head);
LIST_INIT(LIST_HEAD *head);
LIST_INSERT_AFTER(TYPE *listelm, TYPE *elm, LIST_ENTRY NAME);
LIST_INSERT_BEFORE(TYPE *listelm, TYPE *elm, LIST_ENTRY NAME);
LIST_INSERT_HEAD(LIST_HEAD *head, TYPE *elm, LIST_ENTRY NAME);
LIST_NEXT(TYPE *elm, LIST_ENTRY NAME);
LIST_PREV(TYPE *elm, LIST_HEAD *head, TYPE, LIST_ENTRY NAME);
LIST_REMOVE(TYPE *elm, LIST_ENTRY NAME);
LIST_SWAP(LIST_HEAD *head1, LIST_HEAD *head2, TYPE, LIST_ENTRY NAME);
TAILQ_CLASS_ENTRY(CLASSTYPE);
TAILQ_CLASS_HEAD(HEADNAME, CLASSTYPE);
TAILQ_CONCAT(TAILQ_HEAD *head1, TAILQ_HEAD *head2, TAILQ_ENTRY NAME);
TAILQ_EMPTY(TAILQ_HEAD *head);
TAILQ_ENTRY(TYPE);

```

```

TAILQ_FIRST(TAILQ_HEAD *head);
TAILQ_FOREACH(TYPE *var, TAILQ_HEAD *head, TAILQ_ENTRY NAME);
TAILQ_FOREACH_FROM(TYPE *var, TAILQ_HEAD *head, TAILQ_ENTRY NAME);
TAILQ_FOREACH_FROM_SAFE(TYPE *var, TAILQ_HEAD *head, TAILQ_ENTRY NAME,
    TYPE *temp_var);
TAILQ_FOREACH_REVERSE(TYPE *var, TAILQ_HEAD *head, HEADNAME,
    TAILQ_ENTRY NAME);
TAILQ_FOREACH_REVERSE_FROM(TYPE *var, TAILQ_HEAD *head, HEADNAME,
    TAILQ_ENTRY NAME);
TAILQ_FOREACH_REVERSE_FROM_SAFE(TYPE *var, TAILQ_HEAD *head, HEADNAME,
    TAILQ_ENTRY NAME, TYPE *temp_var);
TAILQ_FOREACH_REVERSE_SAFE(TYPE *var, TAILQ_HEAD *head, HEADNAME,
    TAILQ_ENTRY NAME, TYPE *temp_var);
TAILQ_FOREACH_SAFE(TYPE *var, TAILQ_HEAD *head, TAILQ_ENTRY NAME,
    TYPE *temp_var);
TAILQ_HEAD(HEADNAME, TYPE);
TAILQ_HEAD_INITIALIZER(TAILQ_HEAD head);
TAILQ_INIT(TAILQ_HEAD *head);
TAILQ_INSERT_AFTER(TAILQ_HEAD *head, TYPE *listelm, TYPE *elm,
    TAILQ_ENTRY NAME);
TAILQ_INSERT_BEFORE(TYPE *listelm, TYPE *elm, TAILQ_ENTRY NAME);
TAILQ_INSERT_HEAD(TAILQ_HEAD *head, TYPE *elm, TAILQ_ENTRY NAME);
TAILQ_INSERT_TAIL(TAILQ_HEAD *head, TYPE *elm, TAILQ_ENTRY NAME);
TAILQ_LAST(TAILQ_HEAD *head, HEADNAME);
TAILQ_NEXT(TYPE *elm, TAILQ_ENTRY NAME);
TAILQ_PREV(TYPE *elm, HEADNAME, TAILQ_ENTRY NAME);
TAILQ_REMOVE(TAILQ_HEAD *head, TYPE *elm, TAILQ_ENTRY NAME);
TAILQ_SWAP(TAILQ_HEAD *head1, TAILQ_HEAD *head2, TYPE, TAILQ_ENTRY NAME);

```

4.3 Memory Management

4.3.1 Memory Management

The FreeBSD memory management is very similar to Linux. In fact many of the provided features and services are based on Linux libraries and some are improved on. In general Linux and FreeBSD's differences are mostly in licensing. FreeBSD's developer team is much slower.

4.3.2 Pages

In FreeBSD, similarly to Linux and Windows, memory is managed through a page by page basis. [26]. This means similarly to Windows and Linux the smallest unit of memory that could be accessed is 1 page or 4 KB of contiguous chunk of memory. The memory itself is accessible down to a single bit, but the handling of memory happens at a page level.

4.3.2.1 States of Memory: According to FreeBSD manual, in FreeBSD, memory could be in one of the following states at any given time:

- Wired
- Active
- Inactive
- Cache

- Free

The memory in all states except Wired, are stored in a doubly linked list. Wired pages are not stored on any queue. /cite4. FreeBSD provides more involved paging queue for pages that classify as Cache and Free states with comparison to Linux.

4.3.2.2 Swapping: FreeBSD swaps out entire idling processes. This is unlike Linux where only least recently used gets swapped out. It takes advantage of a construct known as Global-LRU across all of user pages. Another used construct for FreeBSD for swapping is page-coloring optimization. This optimization achieves a more accurate method of swap selection by incorporating the processor's cached pages. (See Hardware Cache at 2.3.4.1).[27].

4.3.3 Services and Features

FreeBSD provides a very similar constructs and services to Linux. In this subsection we can look into few differences that FreeBSD and Linux have.

4.3.3.1 Generic VM Objects: FreeBSD provide an idea known as generic virtual memory objects. Virtual memory objects could be unbacked, backed with swap, physical device or file storage. This results in Unification of the buffer cache (2.3.4.1) in FreeBSD. [26]. FreeBSD dynamically tunes page queues in order to balance the pages in for maintaining a reasonable breakdown of clean and dirty pages. [26].

4.3.4 FreeBSD and Linux Comparison

As mentioned earlier, FreeBSD and Linux have many similarities on abstract level features. However, FreeBSD usually improves on the Linux implementations. These improvements come at a cost of FreeBSD having a much slower development.

5 CONCLUSION

This document covered information with regards to Processes, I/O and memory Management of Linux, Windows and FreeBSD. Each one of these operating systems have their own capabilities and provide a different perspective for solving similar problems. Some patterns have emerged from comparing these three operating systems and looking into their implementation and performance. Linux Operating systems, has a rather direct approach in designs. In Process, I/O and Memory management it is clear that windows comes up with simple, effective and elegant solutions for many problems. FreeBSD on the other hand, takes the same concepts from Linux and improves them. FreeBSD usually uses very similar constructs to Linux but implemented more efficiently and more effectively. Finally, windows introduces complex concepts and expands the initial basic kernel service requirements. Windows emphasizes accessibility of certain capabilities for the developers over the simplicity. In the end, it is developers, users and consumer that decides what services they need and which operating system is best fit for their needs. In conclusion, This document covered 3 of the main features of Linux, Windows and FreeBSD operating systems. This document barely scratches the surface on the amount of information that could be found on these topics. It is recommended to look into the references section of this document for more information.

REFERENCES

- [1] "cs.columbia.edu," <https://www.cs.columbia.edu/~smb/classes/s06-4118/l13.pdf>, accessed: 2017-05-01.
- [2] "[patch] modular scheduler core and completely fair scheduler [cfs]," <https://lwn.net/Articles/230501/>, accessed: 2017-05-01.
- [3] "Inside the linux 2.6 completely fair scheduler," <https://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/>, accessed: 2017-05-01.
- [4] "Data structures in the linux kernel," <https://0xax.gitbooks.io/linux-insides/content/DataStructures/>, accessed: 2017-05-19.
- [5] "Vmware academic program," <https://labs.vmware.com/academic/publications/mclock>, accessed: 2017-05-19.
- [6] "Budget fair queueing (bfq) storage-i/o scheduler," http://algo.ing.unimo.it/people/paolo/disk_sched/, accessed: 2017-05-19.
- [7] "blk-mq: Kyber multiqueue i/o scheduler," <https://lwn.net/Articles/720071/>, accessed: 2017-05-19.
- [8] "Cipher algorithm types," <https://www.kernel.org/doc/html/v4.10/crypto/architecture.html#cipher-algorithm-types>, accessed: 2017-05-19.
- [9] "Ciphers and templates," <https://www.kernel.org/doc/html/v4.10/crypto/architecture.html#cipher-algorithm-types>, accessed: 2017-05-19.
- [10] "2-memory management," 2-2-<http://www.tldp.org/LDP/tlk/mm/memory.html>, 2-Accessed: 2017-06-06.
- [11] "2-paging and swapping," 2-2-<http://www.linux-tutorial.info/modules.php?name=MContent&obj=page&pageid=89>, 2-Accessed: 2017-06-06.
- [12] "2-posix thread (pthread) libraries," 2-2-<http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html>, 2-Accessed: 2017-06-06.
- [13] "Processes and threads," [https://msdn.microsoft.com/en-us/library/windows/desktop/ms684841\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms684841(v=vs.85).aspx), accessed: 2017-05-01.
- [14] "Scheduling," [https://msdn.microsoft.com/en-us/library/windows/desktop/ms685096\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms685096(v=vs.85).aspx), accessed: 2017-05-01.
- [15] "Context switches," [https://msdn.microsoft.com/en-us/library/windows/desktop/ms682105\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms682105(v=vs.85).aspx), accessed: 2017-05-01.
- [16] "I/o request packets," <https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/i-o-request-packets>, accessed: 2017-05-19.
- [17] "The microsoft press store," <https://www.microsoftpressstore.com/articles/article.aspx?p=2201309&seqNum=3>, accessed: 2017-05-19.
- [18] "Scheduling," [https://msdn.microsoft.com/en-us/library/windows/desktop/ms685096\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/ms685096(v=vs.85).aspx), accessed: 2017-05-19.
- [19] "Cryptographic service providers," [https://msdn.microsoft.com/en-us/library/windows/desktop/aa380245\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/aa380245(v=vs.85).aspx), accessed: 2017-05-19.
- [20] "Processes and daemons," https://www.freebsd.org/doc/en_US.ISO8859-1/books/handbook/basics-processes.html, accessed: 2017-05-01.
- [21] "A look inside," https://www.freebsd.org/doc/en_US.ISO8859-1/articles/linux-emulation/inside.html, accessed: 2017-05-01.
- [22] "Bfs by con kolivas," https://vellvisher.github.io/papers_reports/doc/BFS_FreeBSD.pdf, accessed: 2017-05-01.
- [23] "Freebsd library functions manual," <https://www.freebsd.org/cgi/man.cgi?query=queue&sektion=3>, accessed: 2017-05-19.
- [24] "Hardware crypto support status," <https://www.freebsd.org/news/status/report-2002-11-2002-12.html>, accessed: 2017-05-19.
- [25] "Freebsd man pages," [https://www.freebsd.org/cgi/man.cgi?crypto\(3\)](https://www.freebsd.org/cgi/man.cgi?crypto(3)), accessed: 2017-05-19.
- [26] "2-management of physical memory," 2-2-<https://www.freebsd.org/doc/en/books/arch-handbook/vm.html>, 2-Accessed: 2017-06-07.
- [27] "2-a comparison of the memory management sub-systems in freebsd and linux," 2-2-file:///C:/Users/Behnam/Downloads/Documents/aa5e870fdb525abf7ed5ca40f58cbb007a3.pdf, 2-Accessed: 2017-06-07.