A Comparison of Linux with FreeBSD and Windows

The purpose of this paper is to explore how Windows and FreeBSD implement I/O and provided functionality, processes, threads, and CPU scheduling, and memory management and compare each of them to Linux.

Mary Jacobsen
Final Paper
CS 444
Operating Systems 2
Spring 2018

The focus of this paper is to compare and contrast Linux, Windows, and FreeBSD to gain a deeper understanding of the similarities and differences between these three operating systems as well as to consider why some of these differences and similarities exist. This discussion will be comparing the processes, threads, scheduling, file I/O, provided functionality including FastIO and crypto, data structures used in the operating systems, memory management (specifically paging), and virtual memory allocation and organization. Linux will be compared with FreeBSD and Windows to glean a clear understanding of the features, performance, and inefficiencies of all three operating systems. We will see the result of the unique production of all three operating when examining certain aspects of their design and considering why the creators of the systems chose to solve problems in the ways that they inevitably did. For example, Linux and FreeBSD are being constantly developed by amateurs (in the sense that they are not paid for their contributions) while Windows has been developed by professionals in an industrial environment. Even so, all three systems have, maybe surprisingly, much in common. The data structures and features of each are quite similar with some understandable differences, so in the following sections, these systems will be compared on the certain aspects detailed above.

The next several paragraphs will discuss the execution and scheduling of processes and threads. They will compare and contrast how three different operating systems, namely Windows, Linux, and FreeBSD handle the complexity of these different problems.

First of all, they will cover processes in FreeBSD which are similar to those in Linux, as well as processes in Windows which are slightly different. For example, all three operating systems use unique identifiers to differentiate between multiple processes. The following discussion will also cover multitasking which is supported by all three operating systems mentioned above.

Processes in FreeBSD are very similar to processes in Linux. FreeBSD and Linux support multitasking. Every process has a process ID (PID) which is used for signaling and exit status. Each process needs its own process ID and its parent's process ID. In FreeBSD, a "thread is the unit of execution of a process; it requires an address space and other resources, but it can share many of those resources with other threads." [1] In Linux, a process can have multiple threads. Threads are created in Linux using the clone system call with the CLONE_THREAD flag. The newly created thread will have the same PID as the parent thread. There are thread ID's (TID) in Linux used to tell threads apart. A normal process in Linux will have just one thread with the thread ID equal to the process ID. For both FreeBSD and Linux all processes are created using the fork system call from another process making a child process except for the init process. Here is an example of creating a process in Linux using the fork system call. [5]

```
pid_t p;

p = fork();
if (p == (pid_t) -1)
        /* ERROR */
else if (p == 0)
        /* CHILD */
else
        /* PARENT */
```

However, processes in Windows differ from those in Linux. Still, all three operating systems use unique identifiers to differentiate between multiple processes. FreeBSD and Linux both support multitasking. Similarly, the scheduling of processes on the CPU is very much the same for Linux and FreeBSD in that processes are scheduled using the kernel process scheduling algorithm so the processes must share the CPU resources. Similarly, Linux and FreeBSD all use jobs to execute processes. Unlike Windows, FreeBSD and Linux use groups of processes all under one process group ID to manage terminal access and send signals to related processes. Now, Signals in FreeBSD work similarly to signals in Linux with some exceptions. Signals in FreeBSD are made to work like hardware interrupts. Similarly, scheduling in Windows has some different features to lower CPU usage when compared to scheduling in Linux.

Although, the scheduling of processes on the CPU is very much the same for Linux and FreeBSD in that processes are scheduled using the kernel process scheduling algorithm so the processes must share the CPU resources. The user can add the nice parameter to gain some control over the process priority but it still must share the CPU resources according to the scheduling algorithm. Both FreeBSD and Linux have real time schedulers which allows the processes to set their own priority. The Kernel will run the highest priority real time process while ignoring the other processes. So real time processes do not have to share CPU resources. One difference between Linux and FreeBSD is that the scheduler in Linux uses a linked list of runnable processes whereas the scheduler in FreeBSD uses a multilevel priority queue. "Threads sharing an address space and other resources are scheduled independently and in FreeBSD can all execute system calls simultaneously." [1]

In contrast, Windows defines processes differently than Linux. However, Windows uses process identifiers similar

to Linux to differentiate between processes. Windows defines a process as a container that stores everything necessary to execute a program. The process (container) holds a private virtual address space, an executable program, a table of handles to system resources, an access token, a process ID, and at least one thread. This way, everything the process holds shares the process' virtual address space including threads even though threads do have their own execution context. The threads execute programs and are held in processes in windows which is different than in Linux where both the threads and processes execute programs. Unlike Linux, Windows does not have a structured process tree. Instead, it uses jobs to control groups of processes. Jobs may have been implemented in Windows rather than a typical UNIX-style process tree because jobs have more control and are in some ways more powerful.

Similarly, Linux and FreeBSD all use jobs to execute processes. Unlike Windows, FreeBSD and Linux use groups of processes all under one process group ID to manage terminal access and send signals to related processes. Children processes inherit the process group ID of their parent. It is fairly easy to create and manipulate process groups. Sometimes, the processes in a process group is called a job. As aforementioned, process groups work in a similar way in Linux. In Linux, "every process is a member of a unique process group, identified by its process group ID. This is done because "when the process is created, it becomes a member of the process group of its parent." [5] For both Linux and FreeBSD, the process group ID is the process ID of the first member in the process group.

Now, Signals in FreeBSD work similarly to signals in Linux with some exceptions. Signals in FreeBSD are made to work like hardware interrupts. There is a set of signals that are predefined that can be delivered to a process. The user can also make a handler function that the signals for a process can be delivered to and the signal will be blocked while it is being caught by the handler. To catch the signal, the process state must be saved and a new process must be made to handle the signal. The process can also be set ignore a signal, or the process can simply use the kernel's default action for the respective signal which is most often exiting from the process. However, the signals SIGKILL and SIGSTOP cannot be caught or ignored. FreeBSD provides a stack that signals can be sent to so that the user may change the stack as needed. Signals for processes in Linux work in much the same way except for that in FreeBSD all signals have the same priority which is not true in Linux which has multiple priority levels.

Similarly, scheduling in Windows has some different features to lower CPU usage when compared to scheduling in Linux. Windows schedules threads for execution. A thread in windows holds the contents of a group of CPU registers which show the state of the processor, two stacks (one is used by the thread during kernel mode and the other is used by the thread during user mode), thread local storage, a thread ID, and sometimes their own security token. Always using the kernel scheduler to schedule threads such as Linux does is very expensive so Windows made fibers and user-mode scheduling (UMS) to lower the cost of scheduling. The user can convert a thread into a fiber by calling a Windows function. Then that fiber can create more fibers and so on. Since fibers do not rely on the Windows kernel scheduler, fibers do not execute until the user calls a SwitchToFiber function on a fiber. A fiber will run until it is done (exits) or until it calls SwitchToFiber on another fiber. So fibers allow the user to be in charge of scheduling instead of the kernel scheduler which is much less costly when switching back and forth between threads. User mode scheduling threads also help with cost when switching between threads in user mode but unlike fibers, UMS threads are visible to the kernel and can therefore use system calls and share CPU resources when needed. Only 64-bit versions of windows provide UMS threads.

Finally, the reader can see from the above analysis that Linux, FreeBSD, and Windows have similar solutions to the common problems presented by CPU scheduling of threads and processes. There are some differences seen above, although, many of these are caused by naming and convention but do not have much bearing on actual scheduling. The few major differences are found when examining Windows which has some additional features for the enhancement of performance.

The following paragraphs in this section will cover how Windows, Linux, and FreeBSD manage I/O. As well as the provided functionality that Windows, Linux, and FreeBSd offer. For example, Windows offers FastIO and FreeBSD offers crypto while Linux uses a structured tree.

In Linux, the kernel provides a subsystem called block I/O to manage block devices because performance is important for block devices and managing the I/O for block devices is more complex than managing the I/O for character devices. This is because block devices are accessed as blocks of data stored throughout memory whereas character devices are accessed as a contiguous stream of chars. Character devices have only one position to access from and block devices must be able to move around to any location. A sector is the smallest addressable unit of a block device. The size of a sector is defined in the hardware of the device. Many block devices can operate on multiple sectors at once but no block device can operate on anything smaller than a sector. A block is the smallest addressable unit in software. The block size cannot be smaller than one sector since the block device cannot access any smaller than a sector and the block size must be a multiple of a sector. Other constraints on a block are that it must be a multiple of two and can be no larger than the size of a page. A page holds one or more blocks and the information required to keep track of the data. The object responsible for representing a disk block in memory is a buffer. Each buffer represents exactly one block. Each buffer has a descriptor which is of type struct buffer

head. The buffer head contains the information necessary for the kernel to manipulate buffers. The following is the definition of the buffer head structure and is located in <linux/buffer_head.h>.

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

One way Linux efficiently performs block I/O operations is through the use of a technique called scatter-gather I/O. Block I/O operations that are in flight are represented as a list of segments by the bio structure defined in <linux/bio.h>. A segment is a part of the buffer that is contiguous in memory. Block I/O operations on a buffer can be performed from multiple places in memory because the buffer is represented in contiguous segments. This behavior is called scatter-gather I/O.

The original Linux scheduler was written by Linus and is often called the Linus Elevator. However, the Linus Elevator did not perform well enough. It would perform decently until any request was older than a particular decided time and then it would service the requests in the order they were requested which would cause more and more requests to be nearly starved and this process would continue causing poor performance. Due to this poor performance, a different system had to be developed that allowed the user to select an I/O scheduler at runtime.

In contrast, a differing technique for scheduling called the I/O manager is used by Windows. Windows uses an I/O manager to connect everything together such as applications and system components to virtual logic and physical devices. The I/O manager also lays out the organizational structure that supports device drivers. Each device driver normally manages a certain type of device. The device driver provides a layer of abstraction so that high level commands from a device can be translated into low level commands by a device driver. The I/O manager sends the device drivers commands from the devices they manage. The device drivers notify the I/O manager when they have completed commands or need to forward a command to another device driver.

There are three other parts that work with the I/O manager to make up the I/O system in Windows: the PnP manager, power manager, and the WDM WMI provider (Windows Driver Model Windows Management Instrumentation provider. The PnP manager works with the bus driver as well as the I/O manager to keep track of and take care of connections and disconnections of hardware devices and to allot hardware resources. The power manager works with the PnP manager and the I/O manager to direct power-state transitions through the system and single device drivers. The WDM WMI provider lets device drivers act as providers in an indirect way.

In windows, the I/O system is packet driven. With the exception of FastIO, I/O requests are represented by a I/O request packet (IRP). IRP's move between system components. This way, multiple I/O requests can be managed concurrently by one application thread. An IRP is a data structure container that contains the information necessary to represent an I/O request. For every I/O operation, the I/O manager makes an IRP in memory, passes a pointer to the IRP to the correct driver, and when the requested I/O operation is complete, cleans up the IRP from memory. Upon receiving an IRP, a driver performs the operation described by the IRP and then sends the IRP back to the I/O manager.

Now, FreeBSD, on the other hand, uses descriptors for I/O. In FreeBSD all I/O is handled by descriptors for user processes. System calls that specify an open file pass in a file descriptor to the kernel. The kernel indexes into a descriptor table for the current process using the file descriptor to find either a file structure or a file entry. The file entry data structure contains a pointer to an object and a file type. The special device file system handles special files by calling the correct drivers to manage I/O for them. The FreeBSD file entry points to a socket for interprocess communication (networking). The FreeBSD file entry points to a pipe for unnamed high speed local communication. To improve the performance of pipes, optimized performance was added. Because of this, pipes replaced the sockets that used to be used for local communication in earlier FreeBSD systems. The FreeBSD file entry points to a fifo for named high speed local communication. Since earlier FreeBSD systems, optimized support has been added to fifos to improve performance. The FreeBSD file entry points to a kqueue for kernel event notifications. The file entry will

point directly to the appropriate place in hardware for systems with cryptographic support in hardware.

Also, Windows has a provided functionality called FastIO which is a high speed mechanism that handles reading and writing to a file. FastIO is a way to read and write a file without having to generate an I/O request packet. FastIO works by having the I/O manager call the file system driver's FastIO procedure to determine whether or not I/O can be completed directly from the cache manager. File system drivers are able to use the cache manager to access files directly without getting an I/O request packet since the cache manager is built on top of the virtual memory subsystem.

Linux and FreeBSD, on the other hand, allow the use of the open source crypto library which implements a wide range of cryptographic algorithms. In terms of file I/O, this library is most helpful when it comes to encrypting disk partitions to prevent unwanted data access. Because of this, FreeBSD and Linux are able to offer excellent protection against file I/O from unauthorized users.

From the above analysis it can be concluded that Windows, Linux, and FreeBSD all have different, valid, solutions to the common problems presented by file I/O. Each also has unique provided functionality with windows focusing on I/O performance, and FreeBSD and Linux prioritizing exceptional data privacy and security.

In this section, there will be discussion on the problems and solutions for two of the main memory management tasks. Firstly, comparing and contrasting Linux, Windows, and FreeBSD with regards to paging. Including discussion of the definition of a page, page sizes, and page freeing. Secondly, comparing and contrasting Linux, Windows, and FreeBSD with regards to virtual memory allocation and virtual address space layout.

Physical pages are used as the main unit for memory management in the Linux kernel. The hardware that manages memory and translates virtual addresses to physical addresses is called the memory management unit (MMU). The memory management unit usually uses pages but the smallest addressable unit is a word, also known as a byte. Although, pages are the smallest unit that need to be considered in virtual memory. The memory management unit keeps page tables to organize the pages. Page sizes vary depending on the architecture they are used within. Interestingly, not all pages can be used for all tasks because of their addresses in physical memory so Linux uses zones to organize pages into groups based on their limitations due to their location in physical memory. All physical pages on the system are represented by a page struct on the kernel, and `<linux/mm_types.h>` holds the page structure definition and is as follows. [2]

```
struct page {
        unsigned long                   flags;
    atomic_t                            _count;
    atomic_t                            _mapcount;
    unsigned long                       private;
    struct address_space        *mapping;
    pgoff_t                                     index;
    struct list_head            lru;
    void                                        *virtual;
};
```

The page structure is used to describe the physical memory of the page, not the data in the page. The kernel manages all the pages in the system using the fields in the page structure such as whether the page is free and if it is not free, who owns the page. There are different 32 flags available and they store the status of the page. The field `_count` stores how many references are to the page. The page is ready for new allocation if `_count` is at negative one. The function `page_count()` takes a page structure as its only parameter and should be used in kernel code to check if a page is free. The function `page_count()` returns zero when a page is free and a nonzero number otherwise. The page's virtual address is held in the virtual field. This page structure only represents physical pages, not virtual pages.

Linux has multiple interfaces to access memory and a low level way to request memory. All the interfaces allocate memory in multiples of pages. All the interface definitions are located in `<linux/gfp.h>`. The main function for allocating memory is `struct page * alloc_pages(gfp_t, gfp_mask, unsigned int order)`. It returns a pointer to the first page of the contiguous physical pages it allocated unless there is an error in which case it returns NULL. `void * page_address(struct page *page)` returns the logical address of the page passed as the parameter. There are also more functions to allocate pages with different options as well as a group of functions designated to freeing pages. It is important that only pages that have been allocated are freed because corruption can occur if the wrong page is passed into the functions used for freeing pages. The following piece of code is an example of allocating, and then freeing, eight pages. [2]

```
unsigned long page;
```

```
page = __get_free_pages(GFP_KERNEL, 3);
if (!page) {
        /* insufficient memory: you must handle this error! */
    return -ENOMEM;
}

/* 'page' is now the address of the first of 8 contiguous pages ... */

free_pages(page, 3);

/* our pages are now freed and we should no longer access the address stored in page */
```

FreeBSD manages memory somewhat differently than Linux does. As services have been added to FreeBSD, the need for dynamic memory allocation has become greater. Just like in the Linux kernel, the FreeBSD kernel provides a generalized memory allocator that can be used in any part of the system which simplifies writing code inside the kernel. The allocation function in FreeBSD is given a parameter containing the size of the memory to be allocated just like the Linux allocation function. However, in FreeBSD, the range of sizes for memory requests has no bound but physical memory is allocated and not paged. Also, in FreeBSD, memory is freed using the free function which is given a pointer to the memory being freed but does not take in the size of the part of memory being freed.

Now, Windows is quite similar to Linux as far as paging is considered with some exceptions. The biggest differences between Linux and Windows is that Windows has some more features that aid performance for windows paging by preventing page faults wherever possible. Most notably, cluster paging prevents multiple page faults by allocating more pages where page faults have recently occurred in memory since the surrounding memory is more likely to be in need of pages. This type of predictive algorithm for page allocation can be extremely beneficial to performance.

Linux, Windows, and FreeBSD all divide the virtual address space into physical pages. Linux, Windows, and FreeBSD all have support for multiple page sizes. Linux calls its larger page sizes huge pages, Windows calls them large pages, and FreeBSD names them superpages. Support for larger pages, as opposed to the standard page size, can increase speed for large memory allocations by reducing the amount of memory stored in the TLB cache, while still managing to keep memory usage decently low for smaller memory allocations. However, attempting to allocate large pages in Windows can lead to failure if the operating system has been running for too long since the physical memory for large pages must use a significant number of contiguous, smaller, physical pages. Then, the extent of physical pages will have to start on the edge of a larger page. This is a slight drawback to contiguous large pages for all 3 operating systems, but free physical memory can become fragmented as pages are inserted and deleted from memory. Thus, having to choose from one problem or another, Windows, Linux, and FreeBSD all allow large pages.

Linux and FreeBSD organize their virtual address spaces similarly while Windows organizes its virtual address spaces differently from Linux, but they all use virtual address space to solve the same problems. In both Linux and FreeBSD, virtual address space is divided into two sections. The upper section with higher addresses is dedicated to the kernel and any data structures it requires while the lower section with lower addresses is dedicated to running user processes. However, in Windows, the virtual address space is divided into three sections. It is organized by types of memory. The first type of data is per-process private code and data. The second type is sessionwide code and data, and finally, the third type is systemwide code and data. Furthermore, in all three operating systems, each process has its own private address space, to enable address space sharing without privacy and security issues. Virtual addresses are always evaluated in the context of the process that is currently running. This way, the address cannot refer to an address defined by another process. This is one of the benefits of virtual address space. Shared processes can be abstracted away from the currently running process.

All things considered, the differences between Windows and Linux, as well as the similarities between Linux and FreeBSD, are clearly understood when considering the development and design of the three operating systems. Windows, produced by a profit and clientèle driven company, contains far more complex features for the purpose of enhancing performance. On the other hand, Linux and FreeBSD are produced by developers and hackers seeking simple and understandable solutions to problems with adequate performance that is slowly improved without overly complex features.

In summary, Windows, Linux, and FreeBSD all have their individual problems and efficiencies. Their differences are hardly surprising considering Windows has originated in commercial settings while Linux and FreeBSD have prospered through open source development. They have a lot in common and some differences. Windows, being developed by a company seeking profit and a strong client base, has undergone more thoughtful development. One could go as far as to say that some of the design decisions made for Windows prioritize better performance whereas design decisions made for Linux often elevate simplicity above performance. The result of which is that Windows

has more features but is difficult to maintain and improve from the developers' view, while Linux and FreeBSD have less features but are easier to maintain and develop.

References

[1] Marshall Kirk McCusick and George V. Neville-Neil. *Design and Implementation of the FreeBSD Operating System*. Addison-Wesley, 2015.

[2] Robert Love. *Linux Kernel Development*. Pearson Education, Inc, 2010.

[3] Mark Russinovich, David Solomon, and Alex Ionescu. *Windows Internals, Part 1*. Microsoft Press, 2012.

[4] Mark Russinovich, David Solomon, and Alex Ionescu. *Windows Internals, Part 2*. Microsoft Press, 2012

[5] Andries E. Brouwer, *https://www.win.tue.nl/ aeb/linux/lk/lk-10.html*