FINAL PAPER

TOPICS ON OPERATING SYSTEMS

December 4, 2016

Steven Silvers
Oregon State University
CS 444 Operating Systems II

Contents

I. Introduction

This document takes a look at various topics of operating systems and how they are implemented in Windows, Linux and FreeBSD. Covered topics include CPU scheduling, I/O, Provided Functionality, Filesystems, Interrupts and Synchronization. Many similarities were found in FreeBSD and Linux as both are based on UNIX, with most differences found being between Windows and the two UNIX based systems.

II. Threads, Processing and CPU Scheduling

Process implementation in Windows, FreeBSD and Linux differ in some ways but are similar in many others. An Example of this is that Windows, FreeBSD and Linux all store processes in memory as structs, such as the task_struct in Linux[1] or the Process Environment Block(PEB) structure in Windows[2]. Another similarity between Windows, FreeBSD and Linux is that new processes inherit security permissions of their parent process[3]. What this means is that if a parent process doesn't have permission to view a particular file, any process created by that parent also cannot view that particular file.

A major way that FreeBSD and Linux differ from Windows is how they create new processes. In Linux and FreeBSD new processes can only be created when an existing process makes a call to the system call fork() which creates a duplicate process called a child process, which can then be overwritten with the new process to be ran using the system call exec() in Linux or execve() in FreeBSD[4]. In the Windows operating system the fork() and exec() system calls are combined into a single system call named CreateProcess()[3]. While this simplifies creating new processes in Windows, should the situation arise where you wish to only call fork() without the exec() you can do this in Linux and FreeBSD but not in Windows.

Threads are used fairly similarly in Windows, FreeBSD and Linux. A thread is what the operating system assigns processing time to, and any one process could have one or multiple threads related to it. Windows, FreeBSD and Linux all have the capability of running threads in two different modes, user mode and kernel mode[4]. When a thread is executing application code it will operate in user mode, which is a protection mode with fewer privileges than kernel mode. When a thread makes a request for services from the operating system it will operate in kernel mode. The purpose of having the two separate modes is to protect the system from possible damage when running an application. There is no point in having your system exposed to outside applications when it doesn't need to be, so it is best to keep it protected.

A. *Scheduling*

Scheduling is a key operation of all operating systems. Scheduling is basically the operating system's way of deciding which thread should get processing time and most operating systems are fairly similar in how they do this but still vary slightly. The action of changing between what thread is currently being ran is called context switching. Context switching allows the operating system to give an illusion of multiprogramming, it can appear to run multiple applications at the same time even though only one or two threads might have processing time at any given moment[4]. Context switching does require some overhead, which creates a lower bound on just how small of time slices the operating system can give to threads. If the time slices were smaller than the context switching overhead, threads wouldn't actually have any time to run process code and the system would not work.

The Windows operating system uses a priority based scheduling system. In this type of scheduling system each thread is given a priority level from zero to thirty-one where zero is least priority and thirty-one is highest priority. The system treats all threads on the same priory level as equals, assigning time slices using the round robin method to the highest priority level with threads available to run[5]. If a thread at a higher priority becomes available to run, the system will stop giving time slices to whatever level it is currently on and then start assigning time slices to that higher level thread. For example, if the system is currently doing round robin assignment on threads with priority twenty-two and then a thread with priority twenty-eight becomes unblocked the system will stop giving time slices to the twenty-two level threads and move up to the twenty-eighth level. Round Robin is a scheduling algorithm that assigns time slices equally in a circular fashion with no priority. While the Round Robin algorithm doesn't consider priority, the Windows system only uses the algorithm against the threads on the highest available priority level.

FreeBSD schedules time slices in a slightly different way than Windows. While FreeBSD's default scheduler also uses a priority system like Windows, FreeBSD is biased towards interactive programs like text editors and other similar applications[4]. If a thread uses its entire time slice without exiting have their priority level lowered. This system allows new processes and interactive processes maintain high priority while long running process get pushed down the priority line.

Scheduling in modern versions of Linux typically will use the Completely Fair Scheduler(CFS) which has been in use since Linux 2.6.23[6]. The idea behind the Completely Fair Scheduler is that all processes should get equal time using the processor. It maintains this balance between the processes by detecting if any processes are not getting fair time on the processor, and then corrects this to bring the system back into balance. The CFS manages processes using a red-black tree, which is a self balancing tree that can operate on any node in the tree in O(log n) time giving us efficient process insertion and deletion[6].

Scheduling practices vary less from operating system to operating system and more from the different goals of the operating system. For example, a Linux system and a Windows system both configured to be general use operating systems will schedule tasks more similarly than say two Linux systems where one is general use and the other is a mainframe. Two different operating systems both focused on general use would need to be much more focused on multiprogramming because users typically want to use more than one application at a time, whereas the mainframe system would be more focused on completing tasks quickly and wouldn't really care to appear to be running more than one application at a time.

III. I/O and Provided Functionality

A. *Provided Functionality in Linux*

The Linux operating system provides a lot of useful built-in functionality including popular data structures, common algorithms and various security and cryptographic features[1]. Data structures are seen commonly throughout the Linux kernel and are defined in header files in the include directory. Things like sockets, network devices, files and PCI buses are all represented and described by data structures built in to the Linux kernel [7]. Without these data structures in place the kernel would not be able to interact with the various devices properly. Linked Lists in the Linux kernel are implemented differently from the traditional linked list where the data field is part of the node structure. In the kernel version, there is a data type called "list_head" defined in the types.h file[1]. This data type is used to embed the list into the data, as opposed to the method mentioned before where the data is embedded into the list. This method combined with how the C language can be manipulated makes the kernel's implementation of linked lists completely type safe.

The next major category in Linux provided functionality is the collection of built-in algorithms. One of the collections of algorithms provided by Linux are the elevator algorithms. The elevator algorithms are used to handle request servicing by deciding what order requests should be serviced in a way that reduces overall service time[1]. Almost all elevator algorithms perform some of the same tasks, such as merging and sorting as well as a way to handle aging of requests to prevent any one request from sitting around and not being serviced[1]. Other common provided Linux algorithms include red black trees, binary searches and a variety of sorting methods. Red black trees are a type of binary search tree that is self-balancing and are commonly used in searches as well as virtual memory management[8]. Red black trees are similar to AVL trees, but in red black trees lookup speed as been sacrificed to improve insert and delete worst case performance[8]. Linux also provides its own version of the popular binary search. In order to function it requires the array that it is acting upon to already be sorted in ascending order. The Linux binary search is most commonly used for interrupt handling within the kernel.

The Linux kernel has a large amount of provided functionality in regards to cryptography, including the Linux Crypto API introduced in version 2.5.45[9]. In newer versions of Linux this API has been expanded to include block ciphers and hash functions. This API allows users to add encryption to their own custom devices within the Linux kernel. For example, in the third kernel project for this class we are building our own RAM disk driver and using the Linux crypto API to add encryption functionality to the driver. Since everything in Linux is represented as a file, it is possible to encrypt your entire file system. This makes your file system impossible to read without the encryption keys, which could be stored on a small flash drive to keep you system secure[10].

B. I/O

I/O is one of the most fundamental pieces of functionality in all operating systems. Without I/O things like the keyboard, mouse, monitor wouldn't exist and would be near impossible for a user to interact with a computer at all. Block I/O is used when the data coming through the I/O stream can be moved as blocks using a large buffer to hold the data. Block I/O is most commonly used reading disk drives and other memory storage systems. Character I/O is when each piece of data comes through character by character and is not lumped together like in the block I/O. An example of character I/O would be a keyboard or a terminal. Currently my text editor is waiting on input from my keyboard and as soon as I press a key, that character is sent to the text editor which then displays the new character back to me on the terminal.

Drivers are the software layer of peripheral devices that enable the I/O device to communicate with the kernel properly. Drivers are typically developed by the third parties who also developed the I/O device in question. Windows operating system handles device drivers very well, because Windows is a closed operating system it keeps drivers separated from the Windows kernel and does not include any pre-installed drivers in the kernel. On the other hand, Linux is open source and treats drivers as modules that can be loaded and unloaded into the kernel dynamically[11]. Windows provides a binary interface so that drivers can interact with the kernel, keeping a healthy level of separation. This separation allows for very well executed plug and play functionality for I/O devices with the Windows operating system. Due to the open source nature of Linux, there isn't great need for the layer of separation between drivers and the kernel like in Windows, since the source code for Linux is readily available to everyone it makes sense to go ahead and integrate drivers as modules within the kernel. The biggest similarity between Linux and Windows in regards to drivers is that they both use event-driven APIs, meaning the driver code is only called in response to something happening[11].

Linux has many schedulers built in and can change between them at runtime. These include the NOOP scheduler, LOOK, C-LOOK and the Completely Fair Queuing which is typically the default I/O scheduler for Linux[1]. FreeBSD uses C-LOOK as its default scheduling algorithm which is designed to maximize throughput[4]. Windows uses their own priority based system where the task scheduler gives each I/O task a priority level and then executes tasks based on priority. Linux has the edge here, having the ability to implement multiple schedulers makes Linux better suited to adopt well to its needed roll based on the expected I/O load and type. For example, if you plan to use Linux to run your server, you could use an I/O scheduler that focuses on block I/O and not worry about character I/O from things like keyboards and monitors, since those will rarely be used. If you want to run Linux as a user desktop, you could then again select a scheduler that is better suited to those kinds of tasks. In the Windows operating system you would have to completely change what version of Windows you are running to get the same kinds of benefits.

FreeBSD and Linux have the most similarities when it comes to device I/O, this is mostly due to both being based on UNIX and both being open source. This allows both operating systems to be highly customizable in regards to how the user would like them to run, whereas standard Windows I/O is optimized to be a user desktop workspace that works well with third party drivers and devices.

IV. Filesystems

A. Filesystem Overview

A file system is the way that a particular operating system uses methods and data structures to manage and keep track of files on the disk[12]. There are numerous file systems available for use, from Windows' New Technology File System(NTFS) to the popular ext3 file system that was built on ext2, keeping most of the file system the same but added journaling.

Journaling is a file system feature that is most helpful when the computer crashes or has a power failure. File systems that make use of journaling track desired but not yet committed changes to the file system in a data structure, typically a circular log[13]. This data structure of uncommitted changes is referred to as the journal, and when the system crash or power failure occurs, the data stored in the journal is used to help bring the system back online quicker than if it did not have a journal, and with a lower likelihood that the data will be corrupt. It is for this reason and almost this reason alone that ext3 has become the default file system in Linux, because it is a journaled file system on top of the already popular and feature packed ext2[12]. There are a few options when using a journaled file system, for example you could opt to use a journaled file system that only tracks metadata which would improve the overall performance of the file system but would be more at risk to memory corruption during a crash than using a journal that tracks stored data as well as metadata.

B. *File Systems in Linux*

The Linux operating system supports a wide variety of file systems that all have pros and cons based on what you need out of your system. As Linux is based on UNIX, everything in Linux is considered a file unless it is a process. This means that apart from processes, everything in Linux must be managed by a file system in one way or another. FreeBSD is also based on UNIX meaning there will be many similarities in regards to file system between Linux and FreeBSD. FreeBSD typically uses the traditional UNIX File System(UFS) or the Fast File System(FFS)[14].

In a traditional file system, such as ext3, the file system is not aware of the structure of the disks mounted to the kernel. This means that for every disk you had mounted, you would also need one file system mounted to manage that disk. There is however one file system that can manage multiple disks all by itself, and that is the Zettabyte File System or ZFS developed by Sun Microsystems as part of Solaris, but is now owned by Oracle.

While ZFS is not a Linux file system, it is used in Solaris OS, it is worth mentioning because ZFS is unique in that it combines the volume manager with the file system, which allows ZFS as a file system to see the structure of the disks[15]. This means that ZFS can manage multiple file systems that can work across multiple disks, instead of the usual one file system to one disk system. The biggest advantage of using ZFS is that since it is aware of the physical disks, existing file systems can be grown simply by adding another disk to the memory pool[15].

ZFS is packed with many other features that set it apart from typical file systems, such as the fact that ZFS is a 128-bit file system. This means that theoretically the upper bound on the size of a single file is 16 exbibytes which is 2^{64} bytes[15]. ZFS also checksums the entire file tree to ensure data integrity. Each block of memory is checksumed, and the checksum value is then saved with the pointer to the block, not at the actual block itself. This continues on until even the root itself is checksumed creating a Merkle Tree or hash tree of the entire file system, so that the entire pool of memory can be verified. Since the checksum value of a block is stored in its parent block as a pointer, that block can still be verified if good or corrupt regardless of what someone tries to do to modify it[15].

C. *File Systems in Windows*

Early versions of Windows used the File Allocation Table(FAT) family of file systems, which are widely popular still today for their simplicity and performance[14]. FAT is supported by almost all operating systems including mobile devices. Because almost everything supports FAT, it is commonly used nowadays in flash drives and other portable storage, making it easy to transfer files between a Windows machine and a Mac using portable media.

With the introduction of Windows NT came NTFS, which is the current Windows standard for file systems[14]. NTFS uses a master file table to store every file as a file descriptor. This master file table holds all metadata related to a file, such as name, size or allocation[14]. NTFS Was designed to be POSIX compliant, one of the ways NTFS does this is that it supports case sensitive file naming[16]. Another difference between NTFS and FAT is that NTFS offers metadata journaling.

D. *Comparison*

File systems in Linux and Windows ultimately serve the same purpose, helping you the user find what you need saved on your computer, the way they do this has some differences. Linux and Windows have completely structure for how their file systems are implemented. Windows assigns a letter to every drive such as C: or D: and each drive then has its own file system[16]. In Linux all drives are mounted in a single tree as subdirectories with '/' being the top of the tree. While NTFS in Windows supports case sensitive file names, by default file names in Windows are not case sensitive meaning that the files ”school” and ”School” both could not be in the same folder as they would be viewed as having the same name. As for Linux and all UNIX based file systems, file names are case sensitive so the files ”school” and ”School” could exist in the same directory.

A key difference between Windows and Linux file systems is whether or not the operating system will allow you to modify a file’s information while it is open in an application. In Windows when a file is opened, say you’re editing a word document in Microsoft Word, you cannot modify the file’s name or location from the file explorer as long as that file is open in Word. Linux on the other hand will allow you to modify the files location, name or other related data regardless if it is open in an application or not.

The implementation of file systems in Linux and Windows can be quite different, however they still have many of the same features. File systems are one of the single most important aspects of an operating system functionality and are still simple enough for the average computer user to be able to understand the basics and use.

V. Interrupts and Synchronization

A. Interrupts and Synchronization in Linux

Interrupts in the Linux operating system are split into two main categories, top halves and bottom halves. Top halves deal with hardware events that are timing critical, meaning that a top half interrupt cannot sleep or block. The main function of the top half is to make sure that the hardware is ready to handle the incoming interrupt request.

Anything that does not fall into the top half category is called a bottom half. Bottom halves are not as time sensitive as top halves, so some bottom halves can sleep, block and can also be interrupted. There are three main types of bottom halves currently used in the Linux kernel, and those are softirqs, tasklets and work queues[1]. Softirqs are rarely used, as they cannot be dynamically allocated or destroyed and there is a maximum of 32 softirqs in existence within the kernel. Because of these limitations, most interrupt drivers are written with tasklets. Tasklets are built on and processed by softirqs based on their priority level. The key advantage of using tasklets is that they can be dynamically allocated and destroyed[1]. Work queues are completely different from softirqs and tasklets in that while the first two run in interrupt context, work queues run in process context. This means that they run like a normal process, with the ability to sleep and block. This allows work queues to perform tasks such as memory allocation and I/O.

Synchronization is necessary in every modern operating system, which includes Linux. Synchronization tools are needed whenever processes might execute concurrently to prevent the creation of race conditions. To attempt to prevent race conditions in Linux, a concept called critical sections was implemented. A critical section is a segment of code that is protected from concurrent multiple access[17]. Whenever a thread is operating within a critical section, all other threads are excluded from that critical section. The problem this can raise is that of deadlock. If their are two protected resources, and two different threads each are currently accessing one of resources but need to use both resources to finish executing their process, neither thread will be able to complete their process as they will permanently lock each other out of their respective resources.

One way that synchronization is implemented in the Linux kernel is called a spinlock. A spinlock is a busy-wait method of mutual exclusion. If the lock is available, the thread takes the lock and runs what it needs to run and then releases the lock. If the lock is unavailable the thread busy-waits on the lock until it becomes available again[17]. Other synchronization methods in the Linux kernel include atomic functions, reader/writer locks, mutexes and the Big kernel Lock.

B. Interrupts and Synchronization in Windows

At a high level, the Windows operating system handles interrupts in a fairly similar way to Linux. Only when you get into the finer details do you start to notice a difference. Instead of top halves and bottom halves, Windows uses what it calls ”traps” which are mechanisms in the kernel that capture a thread that is currently executing when an interrupt occurs. Control is then transferred to a trap handler, which is responsible for running a function that is specific to whatever interrupt or exception that occurred[18]. The kernel distinguishes interrupts from exceptions in that interrupts are asynchronous events that have nothing to do with what the CPU was currently running, such as I/O. Exceptions are synchronous and are usually caused by whatever the CPU was running at the time.

The Windows operating system sets interrupt priority with a system called Interrupt Request Level (IRQL). This is represented in the kernel on x86 systems as number 0 through 31. The higher the number, the higher the priority of the interrupt. A level 0 means to run it at regular thread execution, 1 and 2 are software level interrupts and 3 through 31 are all various types of hardware interrupts[18]. A higher level interrupt can always interrupt a lower level interrupt, causing the processor to save what was already done and then call the trap dispatcher for the higher level interrupt. This is different from Linux, where there are many interrupts that cannot be interrupted and disable other interrupts from happening.

Synchronization within the Windows operating system follows many of the same principles that Linux does, just with slightly different implementation. Windows also makes use of the idea of mutual exclusion to prevent race conditions from happening. Windows uses different methods of mutual exclusion based on the priority level of the IRQL. When dealing with high-level IRQLs Windows makes use of Spinlocks, as described in the Linux section. Windows mostly uses what it calls a Queued Spinlock, where processors that want the lock are held in a queue and instead of waiting on the lock itself they wait on a pre-processor flag[18]. This reduces that amount of traffic on the bus, as well as creates a FIFO order of which processor gets the lock, instead of giving it to a random processor. For low level IRQLs, Windows has what it calls kernel dispatcher objects that can be used in the Windows API. Examples of these objects are semaphores, mutexes events, and waitable timer[18].

The implementation of interrupts in Linux and Windows follow the same high level algorithm of interrupt happened, stop what we are doing, handle interrupt, resume what we are doing. Only when you begin to look at the specific implementation of how interrupts are handled do you start to see differences in the two operating systems. The same can be said for synchronization, both operating systems are trying to prevent race conditions from occurring. There are slight differences in implementation, but ultimately the concepts are the same.

VI. Conclusion

After reviewing these operating system concepts and seeing how they are implemented in three different operating systems, differences between them become more noticeable as you get closer to the hardware level. From a high level view, these three operating systems ultimately operate very similarly, I/O manages hardware devices and filesystems allow users to organize and find their files easily.

References

[1] K. McGrath, *Operating Systems*. Top Hat, 2016.
[2] "Microsoft developer resources: Peb structure." https://msdn.microsoft.com/en-us/library/windows/desktop/aa813706(v=vs.85).aspx. Accessed: 10-17-2016.
[3] "Microsoft developer resources: Createprocess function." https://msdn.microsoft.com/en-us/library/windows/desktop/ms682425(v=vs.85).aspx. Accessed: 10-17-2016.
[4] G. V. N.-N. Marshall Kirk McMusick, *Design and Implementation of the FreeBSD Operating System*. Addison-Wesley, 2nd ed., 2015.
[5] "Microsoft developer resources: Createprocess scheduler." https://msdn.microsoft.com/en-us/library/windows/desktop/ms685100(v=vs.85).aspx. Accessed: 10-17-2016.
[6] "Inside the linux 2.6 completely fair scheduler." http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler/. Accessed: 10-17-2016.
[7] D. A. Rusling, *The Linux Kernel*. 0.8-3 ed., 1999.
[8] R. Landley, "Red-black trees (rbtree) in linux." https://www.kernel.org/doc/Documentation/rbtree.txt. Accessed: 11-11-2016.
[9] "Kernel crypto api interface specification." http://www.chronox.de/crypto-API/crypto/intro.html. Accessed: 11-11-2016.
[10] "Linux and cryptography." http://www.antipope.org/charlie/old/linux/shopper/167.crypto.html. Accessed: 11-11-2016.
[11] "Linux vs. windows device driver model: architecture, apis and build environment comparison." http://xmodulo.com/linux-vs-windows-device-driver-model.html. Accessed: 11-11-2016.
[12] S. S.-A. W. Lars Wirzenius, Joanna Oja, "Linux system administrators guide." http://www.tldp.org/LDP/sag/html/filesystems.html. Accessed: 11-26-2016.
[13] T. Jones, "Anatomy of linux journaling file systems." http://www.ibm.com/developerworks/library/l-journaling-filesystems/. Accessed: 11-26-2016.
[14] U. Explorer, "Understanding file systems." http://www.ufsexplorer.com/und_fs.php. Accessed: 11-26-2016.
[15] T. F. D. Project, "Freebsd handbook, chapter 19." https://www.freebsd.org/doc/handbook/index.html. Accessed: 11-26-2016.
[16] M. Support, "Overview of fat, hpfs, and ntfs file systems." https://support.microsoft.com/en-us/kb/100108. Accessed: 11-26-2016.
[17] T. Jones, "Anatomy of linux synchronization methods." http://www.ibm.com/developerworks/library/l-linux-synchronization/. Accessed: 12-3-2016.
[18] D. S. M. Russinovich, *Windows Internals*. Microsoft Press, 6th ed., 2012.