Final Writing Assignment

Isaac Stallcup

CS 444

Spring 2017

1 INTRODUCTION

Windows, FreeBSD and Linux are three of the most popular operating systems available to users today. Windows suffers from the problem of being proprietary software, and thus its innards are not on display for all to see, but Linux and FreeBSD have no such restrictions. They share many similarities, yet also many differences. In this paper, I will explore these dissimilarities and shared qualities between Windows, FreeBSD and Linux. The latter two, FreeBSD and Linux,have much in common, as they are both derivatives of Unix™. Windows has a different approach altogether in some areas, but despite these differences the three operating systems all handle three main components (among many others): processes, I/O, and memory management. I will explore these three areas, examining how Windows and FreeBSD accomplish each area and contrasting each to Linux's approach to each of the three functionalities.

2 PROCESSES

Processes have no true global definition, but a few examples include "a container for a set of resources used when executing the instance of a the [a] program" [1], or a "task or thread of execution" [2]. In either case, processes need to be managed, handled and otherwise dealt with by the operating system.

2.1 Windows

Processes in Windows are made up of several parts (at the most abstract level they can be described) [1]:

- A private virtual address space, a collection of virtual memory addresses that are private to the process
- An "executable program", essentially the code that makes up the program that the process is running
- A list of open handles to system resources. These are accessible to all threads in the process.
- A security construct called an "access token" that keeps track of the user, security groups, privileges, User Account Control virtualization state, session, and limited user account state data associated with the process. Essentially, it keeps track of the permissions the user running the process has in order to prevent them doing anything untoward.
- A process ID
- One or more *threads* of execution (more on these later).
- A pointer to the process' parent process; this information is not updated, so if the parent perishes, the pointer still points to a nonexistent parent.

Processes in Windows use virtual memory addresses when reading or writing to memory. [3] Virtual memory addresses are placeholders, and when read or write operations with virtual memory addresses are called the virtual memory address is assigned to a physical memory address. [4] Processes also have handles to system resources. Handles are tools used to interact with system resource objects, like files or threads.

Windows also has a convenient GUI for viewing and manipulating processes in the the form of its Task Manager.

2.1.1 Threads

Threads are discrete aspects of a process that are used for scheduling. In Windows, processes begin with one thread, called the primary thread. However, each thread within a process can create additional threads [3]. Threads in Windows share the virtual address space and system resource availability of the process they belong to. Threads are assembled from the following [1]:

- A set of CPU registers representing the state of the processor
- One local stack for the thread to use while executing in kernel mode, and one local stack for the thread to use while executing in local mode.
- An exclusive storage area named thread local storage, which is used for libraries and other external elements to the process the thread belongs to.
- A unique identifier called a thread ID, or TID. The TID is analogous to a PID for processes.
- Occasionally, some extra security information.

Within a process, each of its threads also have priorities associated with them, valued from 0 to 31. [5] The scheduling utility in Windows then assigns time slices to be scheduled in round-robin fashion, starting with highest priority threads then moving to the lowest priority threads [3].

2.2 FreeBSD

FreeBSD implements multitasking via processes as well. The *context* that each process has is divided into the user-level and kernel-level states. The user-level state has the address space and the run-time environment of the process, whereas the kernel-level state has unique information about the process, controls the resources allocated to the process, and deals with scheduling [2]. FreeBSD also implements unique process identifiers, or PIDs.

Process in FreeBSD are, similar to Linux, created by ”cloning” another process’s multiple contexts (user- and kernel-level both). Of note is that there is an innately hierarchical nature to a child and parent process in FreeBSD’s file system [2]. These child processes then have a concrete life cycle ahead of them, and when they exit they send 8 bits of exit information to their parent process. Processes that have children can also wait until any of its children exit using the *wait* system call.

FreeBSD also organizes groups of processes into process groups [2]. Used to control access, a process inherits its group from its parent, though the process group of a process can be changed at any time. The processes within a process group are often referred to as a job [2], and these jobs are what programs (for example shells) manage. One advantage of process groups is that users can send signals to all members of a process group at the same time.

2.2.1 Threads

FreeBSD has two methods of implementing threading. The first, 1:1 threading, implements each thread as an individual process. The second, 1:N threading, groups all threads as a single process and runs it that way. Both have benefits and detriments. [6]

1:1 scheduling implements larger threads than 1:N scheduling, and in 1:1 scheduling the scheduling itself cannot be altered by the user where as in 1:N scheduling the user can modify the scheduling. However, 1:1 threading can utilize multiple CPUs, and 1:N threading cannot. [6]

FreeBSD implements scheduling in two ways; that is, there are two schedulers that come with FreeBSD. They are M:N (hybrid) threading and 1:1 threading. [6]

2.3 Comparisons to Linux

All three of FreeBSD, Windows and Linux are multi-tasking operating systems, meaning they can all run multiple processes concurrently. As detailed above, however, they do not handle this feat in the same manner. Differences and similarities between Linux and Windows and between Linux and FreeBSD are detailed below.

2.3.1 Windows

Windows and Linux both use virtual addressing [7] [4]. However, Windows and Linux use different mechanisms to map virtual memory to physical memory. Windows uses the memory manager, and Linux uses page tables to map virtual addresses to physical memory. Both map when the memory is called for [1] [8]. Windows and Linux both also identify processes via a PID, or process identifier [1].

2.3.2 FreeBSD

In FreeBSD and Linux both, processes are created by duplicating the contents and context of other processes [2]. The ”fork” system call operates the same, and the life cycle of the process is the same. First spawned from the parent process, a child process executes, exits, may become a zombie then relays back to the parent process [2].

3 I/O

I/O is dealt with in a variety of ways between FreeBSD, Linux, and Windows. Unsurprisingly, Linux and FreeBSD deal with it in a very similar way based on descriptors, and as expected Windows has a slightly different way of dealing with things. All three however share a usage of device drivers to deal with

3.1 Windows

Windows accomplishes I/O via a series of cascading abstractions. At its heart, the I/O Manger coordinates the I/O process like a conductor, helping to keep things running smoothly and ensure everyone is playing the same song, if you will.

3.1.1 I/O

At the heart of the I/O system in Windows, the I/O Manager connects components of I/O system together to ensure their proper functioning, communicating via a packet-based system of data structures called I/O Request Packets. The I/O Manager defines the model by which I/O requests are delivered to device drivers [1].

Device drivers help smooth communication to and from connected devices. They translate high-level commands from devices into low-level commands to the operating system. They do this by relaying instructions

from the I/O Manager to devices and communicating the results of these commands back to the I/O Manager [1].

3.1.2 Data Structures

One data structure that the I/O system uses is the I/O Request Packet. An IRP stores information required to process an I/O request. Constructed when an I/O request is made, the packet allocates and initializes the IRP before storing a pointer to the *file object* of the call responsible for generating the I/O request [1].

File objects are represented by data structures implemented in the kernel that represent handles to devices. They contain a ”name” (the actual file that the file object refers to) as well as location data, permissions, information for I/O priority and scheduling, and a pointer to the device associated with it [1].

When file objects are called, Windows must determine which driver to use to handle the file object based on the its name. Driver objects are data structures that handle these operations, and in turn create device objects. These device objects are also data structures representing physical or logical devices on the system. Driver objects can create and be connected to multiple device objects in order to fulfill whatever requests the file object makes. When a driver object’s last device object has been deleted, the driver object is unloaded [1].

In summary: IRPs are created then given a file object, which is a handle to the file associated with the IRP. The file object, when called, causes a driver object to come into being, which spawns the required device object for the needs of the file object [1].

3.1.3 Types of Devices

There are several types of device drivers, divided into two broad categories: user- and kernel-mode drivers. User-mode drivers are divided into printer drivers (which unsurprisingly deal with functionalities associated with printing) and User-Mode Driver Frameworks. User-Mode Driver Frameworks are a large group of hardware devices that interact with the user [1].

The kernel-mode drivers are file system drivers, Plug and Play (PnP) drivers, and non-Plug and Play drivers. File system drivers deal with I/O to files and communicate with storage and networking devices. PnP drivers deal with hardware including mass storage devices, video and audio devices and networking adapters. Non-PnP drivers deal with kernel functionality (essentially everything else) [1].

3.1.4 I/O Scheduling

In Windows, I/O Scheduling falls under the umbrella of I/O Prioritization. I/O Prioritization not only includes scheduling but determines which processes get preferential treatment throughout the scheduling process (usually those that deal with user interface) and those that do not (background processes like antivirus or search indexing) [1]. The levels of priority available in Windows are (in order of descending urgency) critical, high, normal, low and very low.

Windows has two methods to schedule priority-ranked processes: hierarchy prioritization and idle prioritization. The former handles processes of all priority except very low, and sorts each incoming process into a queue based on its priority, grouping like to like (all processes of a similar priority are grouped together). All critical priority processes must be completed before the high priority queue is dealt with, and all high priority

processes must be completed before the normal queue comes up, and so on. Each queue is processed in a FIFO manner [1].

Idle prioritization uses a separate queue for the lowest priority I/O requests. Starvation is prevented via a mechanism that requires at least one idle priority I/O request be processed per unit time. Additionally, a delay occurs after after the last idle priority I/O before another is sought in order to prevent the low priority I/O requests from arising amidst high priority requests [1].

3.2 FreeBSD

FreeBSD attacks I/O in a different manner. Everything is treated as a file in Unix™-descended operating systems, and FreeBSD is no exception. File descriptors are king in FreeBSD, and are essentially handles that point to files or other I/O resources [2]. FreeBSD maintains a list of all the file descriptors in use, and uses that to keep track of what's going on inside the operating system.

3.2.1 I/O

In FreeBSD, all I/O is done through descriptors. There are several types of descriptors each of which point to a different type of object, a few of which are listed here:

- A *vnode* points to a file or device. Vnodes work by referencing substructures that contain information important to the file system underlying FreeBSD. Block devices are handled by other descriptors.
- A *socket* handles networking applications.
- A *pipe* is used for high-speed communication between processes. Previously this was accomplished using sockets.

Other types include crytographic handling, queues for messages, and descriptors for POSIX shared memory and POSIX semaphores.

These descriptors point to locations on a descriptor table. These descriptor tables are unique to each process and are used to locate file entries. The file entries in turn provide file types and a pointer to the underlying object of the descriptor [2]. The file entries also contain the necessary capacity to read, write, poll, and truncate file descriptors [2].

Reading, writing and other operations occur only on open file entries; file entires are only used if the file descriptor associated with them is already open. FreeBSD also maintains information in each file that contains the offset currently used in reading/writing to the file, instead of requiring that this be given to the read/write operation as a parameter.

3.2.2 Types of Devices

FreeBSD divides devices into three main categories: disk devices, character devices and network devices. Character devices are classified as almost everything except network interfaces [and disks]. They transform interfaces with hardware into byte streams. Disk devices are modeled as block devices, while network devices use an asynchronous framework [2].

3.3 Comparisons to Linux

On a broad level, Windows and Linux operate similarly; they both perform reads and writes to physical media. However, once one begins drilling down, the actual implementation differs quite substantially. Even on the level of data structures things are very different between the two operating systems [2] [1]. However, as per usual FreeBSD and Linux share much in common.

3.3.1 Types of Devices

FreeBSD and Linux both share character devices as bite stream objects [2] [8]. However, block devices exist Linux [8] but not in FreeBSD past FreeBSD 5 [2]. The paradigm of device handling is different in Windows, with a user-mode and kernel-mode separating the types of devices [1].

3.3.2 Data Structures

FreeBSD and Linux both share file descriptors as their main data structure method of interacting with files [2]. Linux however simply numbers its file descriptors that refer to files, whereas FreeBSD has a more complex naming scheme. FreeBSD and Linux also both have pipes, and share similar cryptographic interfaces through file descriptors [2]. Windows has a wildly different set of data structures, as can be reasoned; however the file object is somewhat analogous in function to the file descriptor though file objects contain more information [1].

4 MEMORY MANAGEMENT

Managing memory has always been of great concern to operating systems. Happily, the days of the ENIAC and its vacuum tubes are gone, and the memory we have to work with is millions (or billions) of times larger than it was in those times. However, fast and efficient memory management is still a priority in operating systems.

4.1 Windows

Please note: all Windows material is from [1] or a lecture given by Professor McGrath on June 01. This statement is designed to replace a citation [1] after every single sentence for reading pleasure, but feel free to mentally insert one before each period in the following sections on Windows. They have been inserted for you in some cases.

In Windows, 32-bit processes have a 2GB virtual memory space; this is increased to 314GB on ”hybrid” 32/64-bit windows. For pure 64-bit windows processes have 8192GB of virtual memory space. This number is chosen arbitrarily and could possibly be increased. Processes are also more a container for threads than a singular process.

Windows supports several other generic services in regards to memory management:

- Address mapping
- Paging
- Memory mapped files
- Copy-on-write memory
- Direct physical memory allocation and use

Windows also maintains a ”working set”, the set of pages physically present in memory at any one time. Other than this working set there are five essential components to Windows' memory management structure: the working set manger, the process and stack swapper, the modified page writer, the mapped page writer, and the zero page thread. The windows system is fully reentrant and is capable of finely grained locking, the result of Windows' broad architecture support.

4.1.1 Pages

There are two page sizes in windows to choose from; large page sand small pages. Large pages are 2MB in size, and small pages are 4KB in size. Pages can be of three types, each type controlling access to the page. Reserved pages have been requested by one page or another, but not used yet. They are set to be private. Committed pages are being used, and are therefore private. They also by default have a valid mapping to them, which is fortunate as they are (as mentioned) being used. Sharable pages can be mapped to and are resident, but not private.

4.2 FreeBSD

As with the Windows section, this is based on information from [2] and Kevin's lecture on the first of June, 2017. FreeBSD implements memory management in much the same way that Linux does, except for the way it handles virtual memory. FreeBSD has a stack section instead of a BSS section like Linux does. The stack section keeps track of a process' run time stack.

4.3 Comparisons to Linux

FreeBSD and Linux are similar in very many ways in regards to memory management. Similarly, Windows and Linux manage memory in broadly the same fashion.

4.3.1 Windows

Linux and Windows both have copy-on-write memory, and maintain a working set (or equivalent). Processes in Windows are essentially containers for threads, whereas in Linux they retain some of their process-oriented properties.

4.3.2 FreeBSD

Linux supports copy-on-write memory by default, while FreeBSD does not, and only performs copy-on-write if a process is allowed to keep a copy of the page. Additionally, as mentioned, FreeBSD does not have the block started by section, instead having a stack section.

5 Conclusion

FreeBSD, Windows and Linux are three popular operating systems for today's needs. All three have some form of process management, handle input and output, and manage memory intelligently. However, they all do so in different ways. FreeBSD and Linux are from the same spiritual ancestor, Unix™, and Windows branched off the figurative phylogenetic tree before Linux and FreeBSD parted ways. Ultimately, all three suit different needs, and selecting one is down to which of the qualities described above is most desirable for your application.

REFERENCES

[1] A. I. Mark Russinovich, David A. Solomon, *Windows Internals, Sixth Edition, Part 2*. Microsoft Press, 2012.

[2] R. N. W. Marshall Kirk McKusick, George V. Neville-Neil, *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley, 2015.

[3] Microsoft developer network - about processes and threads. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms681917(v=vs.85).aspx

[4] Microsoft developer network - virtual address spaces. [Online]. Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/virtual-address-spaces

[5] Microsoft developer network - scheduling priorities. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms685100(v=vs.85).aspx

[6] R. Divacky. A look inside. [Online]. Available: https://www.freebsd.org/doc/en_US.ISO8859-1/articles/linux-emulation/inside.html

[7] D. A. Rusling. The linux kernel. [Online]. Available: http://www.tldp.org/LDP/tlk/kernel/processes.html

[8] D. K. McGrath, "Cs444 operating systems 2 lecture and accompanying top hat textbook."