Similarities and Differences in Features of Linux, Windows, and FreeBSD Operating Systems

Soo-Min Yoo - CS444 Spring 2017

I. Introduction

Operating systems come in a variety of kinds, three well-known ones being Linux, Windows, and FreeBSD. All user-interactive computers have functionalities that are important in letting the user execute tasks, but there are similarities and differences with each of the systems that give advantages and disadvantages that go well with the kind of work that the user needs the computer to do. In this paper, we will explore some of these similarities and differences - specifically, how Linux, Windows, and FreeBSD implements processes and threads, I/O and provided functionality, and memory management.

II. Processes and Threads

A process is an entity that is a user-run application or task that has its own system resources like memory and files. A thread is a unit of work that can execute and be interrupted by its process so that other threads can run. Windows, Linux, and FreeBSD operating systems all have processes and threads, but they each have their own way of implementing them and different types of CPU scheduling algorithms. This paper will go over how each of the three operating systems implement processes, threads, and CPU scheduling, and touch on their similarites and differences.

Processes in Linux, also known as tasks, are usually run in two different ways - either by a special startup init process, or from other running processes. Linux tasks are represented by struct `task_struct` that contains information on managing tasks, such as process state, scheduling information, identifiers, timers, and file system. Below are some of the important fields in this data structure [1]:

```
volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped*/
void *stack;
unsigned int flags; /* per process flags, see here */
struct list_head tasks; /* list of all tasks to iterate over */
```

Linux processes are all schedulable and are always in one of five states - initialization, runnable, running, blocked, or exit. Processes and threads are only different in Linux by whether the SHARED flag is set or not when the process is created [1]. When a new process is created, Linux copies all of the original process's attributes over the new process using the clone() command so that they can share system resources like files and virtual memory, which makes the two processes act as threads within one process. These cloned processes can't use share user stacks, however, and so each have their own separate user stacks [2].

Processes are often in their runnable state since the OS is constantly switching between multiple processes. When they're not in the running state, they are either being blocked waiting for another process to finish, switching to another process, or exiting [1].

Linux performs various types of CPU scheduling to run their processes efficiently for different kinds of systems. It uses a technique called time division multiplexing, which makes it seem like multiple processes are being executed simultaneously. Processes can be CPU bound or I/O bound, the first being most appropriate for numerical, high performance jobs on big systems while the latter works well with more interactive systems that rely a lot on user input [1]. Compared to Windows, the two process classes in Linux are real time and normal. Examples of real time schedulers are the First Come First Serve (FIFO) or the Round Robin with Timeslice. In a FIFO scheduler, processes are scheduled on a first come, first serve basis. This doesn't work well with interactive systems that have processes that may need immediate attention. Round robin schedulier gives each process a timeslice of CPU time to run on. Once the timeslice runs out for the running process, the next process in the queue runs next - the cycle continuing until no more processes are left [1].

Linux creates processes with fork() or fork() with exec(), whereas Windows uses a function called CreateProcess(). Windows has more complicated parts to process creation, like security contexts, while Linux just adds overhead. In Linux, a fork with no exec means no hard disk access, but Windows' CreateProcess always uses disk access [3].

In Linux, when you have a parent process and create child processes from it, closing the parent process forces all of its child processes to exit as well. However, in Windows, closing a parent process does not cause its child process to exit. Instead, the child process becomes an independent process of its own with no parent and continues to run normally. Windows child processes store the PID of its parent process, but do not store information about a process that created its parent process.

Windows supports preemptive multitasking, which makes it so that multiple processes can execute multiple threads at the same time by giving each process a small time slice of the system resources to run on [3]. Processes are implemented as objects, and both processes and thread objects have synchronization capabilities [4]. A process needs to have at least one thread to execute, which can then create other threads.

Windows processes are made up of a unique PID, threads, a private virtual address where it's executed, list of open handles to system resources, and access tokens with information that determines how much account privileges to give to a user. Threads in a process are always in one of six states - ready, standby, running, waiting, transition, or terminated [4].

Each process belongs to a user who created that process, along with a session which contains all the processes, desktops, and windows stations of a user logged into the system. When a user logs on into a Windows session, the user gets a security access token with a security ID. Every process that user runs has a copy of this token, which lets the user access certain objects or functions in the system depending on the account privileges the token gives them. If a process has a handle open to its access token, it can even change its own process attributes if the token lets it [4].

When it comes to CPU scheduling, Windows implements a priority-driven preemptive scheduler with various priority levels, using round robin scheduling in each level. Depending on the thread, priority levels can be dynamically changed for some levels. There are two main classes of priority levels - real time and variable, each of which has 16 different levels. The real time class is used for threads that may need immediate attention, like communication and real-time tasks in interactive systems, so real time priority threads often have higher priority than other threads [4].

Real time priority threads are all in a round robin queue, each with a fixed priority that doesn't change. But in the variable priority class, threads have priorities that start at an initial value but can change throughout its existence, and so they have FIFO queues at each level [4]. Windows scheduling is similar to that of Linux in that process-bound threads are usually

lower priority while I/O-bound threads are often high priority.

FreeBSD processes are very similar to those of Windows and Linux - each process has a virtual address space, one or more executable threads, and system resources such as descriptors, signal status, and file system. Threads in a process run in either user mode or kernel mode. In user mode, the thread executes an application code with non-privileged protection mode, but then switches to run in kernel mode with privileged protection mode when a thread asks for the operating system's services [5].

FreeBSD creates child processes from a parent process by using fork() to duplicate the context of the parent. The child process then shares all of it's parent's system resources, such as file descriptors, memory, and signal-handling status [6]. Kind of like Windows, FreeBSD processes are organized into process groups, which are themselves contained in sessions.

FreeBSD schedules its threads using one of its various schedulers - the 4.4BSD scheduler, ULE scheduler, real time scheduler, or timeshare scheduler. FreeBSD 5.2 and up use the ULE scheduler by default. In 4.4BSD, every runnable thread has a scheduling priority that puts them in a certain run queue. The system runs the threads from highest to lowest priority [7]. In timeshare scheduling, which is good for interactive systems, the system uses multilevel feedback queues to dynamically change a thread's priority based on their resource use and consumption. The threads are moved around due to these priority changes, and the system switches to a higher priority thread as soon as the current thread finishes [5]. Real time scheduling ensures that threads finish executing within a certain time limit or by a deadline. FreeBSD is different from Linux in that there exists an idle priority thread, which runs only when there's no other runnable threads and if its idle priority is equal to or greater than other idle-priority runnable threads [5].

III. I/O and Provided Functionality

Input/Output (I/O) is an essential, basic part of an operating system. Operating systems use the help of device drivers to handle various I/O device, and scheduling I/O requests can increase overall efficiency. I/O is implemented in different ways among different operating systems, and so in this paper we will be discussing the similarities and differences between how Linux, Windows, and FreeBSD implement their I/O.

Linux provides the user with an "abstract machine" interface by using device drivers, which map hardware specific interfaces onto standard interfaces in the kernel [1].

Linux device drivers are usually implemented through kernel modules, and work in a number of different ways. One way is by direct memory access, or DMA, where tasks in the CPU are moved to the DMA controller until the DMA controller finishes the task and signals the CPU that it's finished. Other ways include the processor I/O, interrupt driven I/O, polled I/O, and memory mapped I/O [1]. In a process I/O, the CPU takes care of all the I/O, while in an interrupt driven one the I/O occurs asynchronously with an interrupt raised when it's done. Polled I/O is where the system endlessly asks for data, and memory mapped I/O maps hardware registers directly to the CPU's address space [1].

In Linux, block devices and character devices are the two main I/O devices used. Character devices had no random access, and act as streams of bytes that can't replay after they're consumed. Block devices do have random access, with fixed-size chunks of data called blocks that set a minimum addressable unit and represent the filesystem block size [1]. Block I/O (BIO) subsystems use `bio_structs` when working on memory blocks (called buffers), which allow buffers to be made up of many, non-contiguous pieces of memory spread throughout the process address space. These `bio_structs` live

inside request queues that the kernel maintains in the device. These queues are eventually merged - read and write requests in each queue are joined together with adjacent sectors on the disk [1].

Linux uses three primary data structures - linked lists, queues, and binary trees. Queues in the Linux kernel are called kfifos, and work similarly to a generic linked list. Linked lists in Linux uses a data type called a `list_head`, which is embedded in the data instead of the data in the node [1]:

```
struct data {
struct *list_head node;
int data1;
char data2[10];
long data3;
};
```

Linux has a variety of algorithms for its I/O scheduling, most of them being some kind of variant of an elevator algorithm to minimize the amount of delay in servicing requests. These algorithms all do request sorting and merging, as well as prevent starvation of requests [1]. The elevators used by Linux include the Linus elevator, Deadline, Anticipatory, No-op, LOOK/C-LOOK, and CFQ algorithms.

The Linus elevator does front and back merging, sorting queues by physical location on the disk. The deadline elevator makes sure no starvation occurs by managing three queues that each have an expiration time. The anticipatory is like the deadline elevator except it also has an anticipation heuristic. After a request is submitted, it literally pauses for a few milliseconds, during which new requests may be submitted and be serviced right away, reducing overall read latency [8]. The noop elevator just maintains the request queue in near-FIFO order, and does request merging. The LOOK/C-LOOK algorithm services sorted requests in the current direction of travel, and reverses direction and services the remaining requests. LOOK services in both directions while C-LOOK scans in only one direction. Finally, the complete fair queuing (CFQ) elevator sorts its queues sector-wise, with one queue for each process submitting I/O, and services the queues round robin [8].

The Windows I/O system is made up of several components and device drivers - the I/O manager, device driver, PnP manager, power manager, and Windows Management Instrumentation support routines.

The I/O manager is the main core of the Windows I/O system. It’s what presents an interface to all kernel-mode device drivers. Windows sends its I/O requests to drivers as I/O request packets, or IRPs [9]. Device drivers receive the commands from the I/O manager for the devices they manage, and let the I/O manger know after the commands are finished. Drivers in Windows are object-based, so the I/O manager and other parts of the system export kernel-mode support routines that devices can use to help carry out its tasks [9].

The IRPs are the main data structures similar to I/O request descriptors that Windows device drivers use to communicate with the operating system. They hold all the information that a driver needs to handle an I/O request, such as buffer size or I/O function type. An OS component or a driver sends an IRP to a driver by calling a function IoCallDriver, and its completion is reported to the I/O manager via IoCompleteRequest [9].

A bit different than Linux, the two different types of drivers that Windows uses are either user-mode or kernel-mode drivers. User-mode drivers execute in user mode and provide an interface between a Windows application and kernel-mode

drivers or other OS components, while kernel-mode drivers execute in kernel mode as part of the executive system that manages I/O, processes and threads, and so on [10].

FreeBSD, on the other hand, uses character and network device drivers. The character device driver, like in Linux, transfers data directly to and from a user process. FreeBSD does not use block devices anymore ever since they stopped supporting cached disk devices. Instead they use network drivers, used by using the system call socket(2) [11].

There are several I/O scheduling algorithms including the 4.4BSD scheduler, ULE scheduler, real time scheduler, or timeshare scheduler. In 4.4BSD, every runnable thread has a scheduling priority that puts them in a certain run queue. The system runs the threads from highest to lowest priority [12]. In timeshare scheduling, which is good for interactive systems, the system uses multilevel feedback queues to dynamically change a thread's priority based on their resource use and consumption. The threads are moved around due to these priority changes, and the system switches to a higher priority thread as soon as the current thread finishes [13]. Real time scheduling ensures that threads finish executing within a certain time limit or by a deadline.

FreeBSD has three main types of I/O, which are the character-device interface, the filesystem, and the socket interface. Character devices are like those in Linux in that they map the hardware interface into a stream of bytes. The socket interface provides a communication API for network communication [12].

FreeBSD device drivers are made up of three main sections: autoconfiguration and initialization routines, routines for servicing I/O requests (top half), and interrupt service routines (bottom half). The autoconfiguration part of the driver is used to ask if there is a device present and to set up the driver and any required software states. The I/O servicing portion is called by system calls or by the virtual-memory system, and executes synchronously in the top half of the kernel [12]. Interrupt service routines are used when there's an interrupt by a device. The interrupt service routine gets requests from the device's queue, lets the requester know that the command finished, and moves on to other requests. The I/O queues are thus the main way in which the top and bottom halves of the device driver communicate [12].

IV. Memory Management

Memory management is an essential, core part of an operating system. Operating systems use memory management to handle or manage primary memory and moves processes back and forth between the main memory and disk while executing. It keeps track of every memory location, whether it's either allocated or free. Memory management is implemented in different ways among different operating systems, and so in this paper we will be discussing the similarities and differences between how Linux, Windows, and FreeBSD implements it.

A important concept of memory management is virtual memory. Over many years, researchers noticed the growing need for more memory space for application programs, and the most successful solution was virtual memory. This makes the system appear to have more memory than it actually has, by using the hard disk to meet the extra space that's needed. Virtual memory is essentially a layer of memory addresses that map to physical addresses [14]. When a processor executes a program instruction, it reads from the virtual memory, converts the virtual memory address to a physical address, then executes it. This mapping of virtual to physical address is done based on mapping information in page tables maintained by the operating system [15].

Both virtual and physical address spaces are divided into fixed-size chunks called pages. The processor's Memory Management Unit (MMU) uses a page table to translate virtual memory address to a physical address, using a page table which specifies mapping between virtual pages and physical pages [15]. Virtual memory addresses are made up of an offset and a virtual page frame number. A processor takes this page frame number from a virtual page and translates it into a physical page frame number, and then uses the offset to put it in the correct address in the physical page [16].

There are cases where a processor tries to translate a virtual page and finds that the entry is invalid, or that since the virtual address space is much bigger than the physical space, pages are unable to be stored in the physical memory. These unstored pages are handled by the hard disk as page faults [16]. The page table associates each virtual page with a bit indicating whether or not it is present in physical memory. If not, the hard disk generates a page fault exception which returns an error if invalid or brings the required page from the hard disk into the physical memory. This time-consuming operation is called demand paging, where all memory pages in a process are not in the physical memory at any given time [14]. It's an efficient way to bring only the memory pages that are necessary at that moment into the physical memory, without the non-needed pages taking up space.

Linux, Windows, and FreeBSD operating systems have several similarities when it comes to memory management. All three systems have what's called a hardware abstraction layer, which takes care of all system-dependent work so that the remaining kernel parts are platform-independent, making it easy to port to other platforms [16]. When processes are sharing one page, a copy-on-write action occurs where a process doing a write on the page creates a private copy for that process. This action results in shadow paging, where a shadow object for the original object is created such that it alters some pages from the original but shares the rest. The three systems also each have a background daemon that runs from time to time doing tasks like page flushing and freeing unused memory. Memory mapped files can be shared between processes as inter-process communication [16].

The three systems also distribute the process virtual address space quite similarly. The higher part is used by the kernel while the lower part is used by the process. Since the kernel part of the process space use the same kernel code, switching a process involves switching the lower part's page table entries while the higher part can stay the same [16].

Data structures are handled somewhat differently among the three systems. Linux maintains a linked list of `vm_area_structs`, which are continuous memory areas. This list is searched whenever a page with a specific location needs to be found. The list is turned into a tree if the number of entries grows big enough, and this allows for the most efficient structure to be used in the appropriate situations [16].

Windows uses a balanced tree with nodes called Virtual Address Descriptors (VAD), which marks each node as committed, free, or reserved. Committed nodes are used ones with code or data mapped to it, free nodes are unused ones, and reserved nodes cannot have anything mapped to it until its reserved mark is removed. Since it's the tree is balanced, finding a node with a specific location will also take relatively low search time. The Process Control Block contains the link to the tree's root [16].

FreeBSD's data structures consist of `vmspace`, `vm_map`, `vm_map_entry`, `object`, `shadow object`, and `vm_page`. The `vm_pmap` takes care of hardware-dependent work, which leaves the other parts of the VM hardware-independent and makes it efficient to port it to different platforms [16].

Page replacement is another important part of memory management, which involves choosing which page to swap out

from memory when more free space is needed. Linux uses a demand paged system with no prepaging, where pages are only brought into memory when they're required [14]. They use the Least Recently Used algorithm to increase a page's age (a counter associated with the page) by a constant when it's used during a scan, and decrease it when it's not used. The pages with an age of 0 are removed from memory [16]. Windows uses clustered demand paging to fetch pages, and the clock algorithm to replace them. Pages are only brought into memory when they're needed, usually in a cluster of 1-8 pages. FreeBSD uses demand paging system with some prepaging for fetching and global Least Recently Used algorithm for replacing [16].

V. Conclusion

The three operating systems - Linux, Windows, and FreeBSD - each have their similarities and differences that give them both advantages and disadvantages in various situations and settings. All three are similar in that they are monolithic, i.e. a single ”master” scheduler that manages all other processes. Each of them have real time scheduling and schedulers that use timeslices or multilevel feedback queues, as they are most often efficient for user-run machines. They use device drivers to manage their I/O, with scheduling algorithms and data structures carrying out the job at the core. The three systems also handle memory using common features such as the hardware abstraction layer, copy-on-write action, shadow paging, and background daemons.

I think the similarities between these operating systems exist because all user-interactive computers have functionalities that are essential to giving the user a smooth experience in executing tasks, but there are differences with each of the systems as well to provide advantages and disadvantages that go well with the kind of work that the user needs the computer to do.

Ultimately, operating systems implement many different kinds of scheduling algorithms for processes and threads to find which algorithm will perform best for what the system will be used for. Their I/O system had to be implemented in their own ways in order to appropriately work with the respective types of I/O devices they use. In terms of memory management, Windows was developed to have better performance due to having had more effort and thought put into its design, whereas Unix-based systems like Linux and FreeBSD were designed more for the purpose of simplicity than performance. This resulted in Windows having more complex, intricate code with many features but more difficult to maintain; while Linux and FreeBSD are simpler and easier to maintain in comparison.

References

[1] D. K. McGrath, *Operating Systems Course Pack – Top Hat*. TopHat, 2016.

[2] W. Stallings, *Operating Systems: Internals and Design Principles, 6th Edition*. Prentice Hall, 2009.

[3] Microsoft. (2017) Processes and threads. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms684841(v=vs.85).aspx

[4] W. Stallings, *Operating Systems: Internals and Design Principles, 5th Edition*. Prentice Hall, 2005.

[5] M. K. McKusick, G. V. Neville-Neil, and R. N. Watson, *The Design and Implementation of the FreeBSD Operating System, 2nd Edition*. Addison-Wesley, 2014.

[6] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, 1996.

[7] M. K. McKusick and G. V. Neville-Neil, *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley, 2004.

[8] R. Love, *Linux Kernel Development Second Edition*. Sams Publishing, 2005.

[9] M. E. Russinovich, A. Ionescu, and D. A. Solomon. (2012) Understanding the windows i/o system. [Online]. Available: https://www.microsoftpressstore.com/articles/article.aspx?p=2201309

[10] Microsoft. (2017) Types of windows drivers. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/hardware/ff564864(v=vs.85).aspx

[11] T. F. D. Project, *FreeBSD Architecture Handbook*. N/A, 2013.

[12] M. K. McKusick and G. V. Neville-Neil, *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley, 2004.

[13] M. K. McKusick, G. V. Neville-Neil, and R. N. Watson, *The Design and Implementation of the FreeBSD Operating System, 2nd Edition*. Addison-Wesley, 2014.

[14] A. Himanshu. (2012) Linux memory management – virtual memory and demand paging. [Online]. Available: http://www.thegeekstuff.com/2012/02/linux-memory-management

[15] R. Dube, “A comparison of the memory management sub-systems in freebsd and linux,” Master’s thesis, Univ. Maryland, College Park, 1998.

[16] G. Khetan, “Comparison of memory management systems of bsd, windows, and linux,” Master’s thesis, Univ. Southern California, Los Angeles, 2002.