Operating System Comparison

Operating Systems II

December 7, 2018

Author:

Erin Tanaka

1 INTRODUCTION

Linux, Windows, and FreeBSD all have their own methods and implementations of kernels. On the whole, FreeBSD tends to draw heavily from the Linux kernel. Windows is often the outlier of the three with different takes on implementation of its version of abstract concepts. The main portion of this document will explore the various implementations of processes, I/O and memory management of Linux, FreeBSD and Windows,

2 LINUX

2.1 Processes

Processes are programs in execution. In Linux there are two different types of processes: foreground or interactive processes and background or non-interactive/automatic processes. Foreground processes are programs currently executing that the user interacts with. Background processes refer to programs executing but are not being interacted with by a user.

Each process is represented by a unique identifier and a task_struct data structure. The task vector points to every task_struct in the system. Linux processes each have unique numbers that identify them called process identifiers or PID. The init process is the parent of all processes on the system. It runs when the system boots up and manages all other processes on the system. The init process always has a PID of 1.

Linux processes can be in one of five states. The process state code R indicates that the process is running or runnable. If it is the current process being served by the CPU or if the process has all of they resources necessary to run but lacks CPU access respectively. S indicates that the process is currently sleeping. Sleeping processes are waiting for the resources to necessary to run. Once the resource the sleeping process is waiting on is available, the process wakes up and becomes runnable. Additionally, the process can be awoken early and become runnable if it receives a signal. Process state code D indicates a state identical to to that of S. The only difference is that it does not wake up and become runnable if it receives a signal. Stopped processes are indicated by process state code T. These processes have stopped executing. They are not running nor are they eligible to run. This state usually occurs when a process is stopped using a signal or it is being debugged. Defunct or zombie processes are indicated by process state code Z. These processes are dead and have been halted. However, they still have entry in the process table.

The top Linux command displays a real-time view to the processes currently running and being managed by the kernel. The command ps provides a static view of the current processes.

2.2 Threads

Linux implements threads the same as processes. To the kernel a thread is a process that shares resources with other processes. Each thread has a unique task_struct. Threads are created like normal tasks except the clone system call is passed flags that correspond to specific resources to be shared. In Linux you would have n processes which results in n normal task structs and some of the processes are set up to share specified resources.

The following command results in behavior identical to a normal fork with the exception that the address space, filesystem resources file descriptors and signal Handlers are shared.

```
clone (CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);
```

This new task and its parent are considered threads. The various flags provided to the CLONE() help specify the new processes behavior and what resources the parent and child will be sharing.[1]

2.3 CPU Scheduling

The Linux scheduler utilizes different algorithms to schedule different kind of processes. The scheduler is modular in this sense. This method of various algorithms matching up with different processes is called scheduler classes. Each of the scheduler classes has a priority. Whichever scheduler is of the highest priority and has a runnable process will select which process runs next. The schedulers must account for the amount of time that a process will run so that it only runs a fair share of the processor.

The CPU scheduler divides its scheduling policies into two categories: realtime and normal. Realtime threads get scheduled first and normal time ones are scheduled after. Realtime policies include SCHED_FIFO and SCHED_RR [2]. SCHED_FIFO, also know as static priority scheduling defines a fixed priority to esch thread [2]. The scheduler scans a list of threads and schedules the one with the highest priority that is ready to run. SCHED_RR is a round-robin policy derived from SCHED_FIFO. The SCHED_RR threads are also assigned a fixed priority value but threads with the

same priority are scheduled round robin style within a specified time period. THe three nomal achedling policies are SCHED_OTHER, SCHED_BATCH, and SCHED_IDLE. SCHED_OTHER or SCHED_NORMAL is the default scheduing policy. It uses the completely fair scheduler which provides fair access to all threads. The other two policies are intended to be used for very low priority jobs.

2.4 Block vs Character

In Linux, block devices necessitate more care, preparation and work that what is needed to manage character devices [1]. Block devices need the ability to navigate between any of the various locations on the media whereas character devices only have to track their current position[1]. Block devices receive their own subsystem in the Linux kernel [1]. The reasoning behind this decision is clear. Not only are block devices more complex than character devices, but their performance is also key. Optimizing the use of a hard disk is considered more important than the speed of a keyboard's input.

2.5 Data Structures

2.5.1 bio Structure

The bio structure is the container for block I/O in Linux. This structure ... represents block I/O operations that are in flight (active) as a list of segments." [1] This is the main purpose of the bio structure. The segments are best described as a "... chunk of a buffer that is contiguous in memory." [1] Since the segments are contiguous, the buffers do not need to be. This means that the kernel is able to execute block I/O operations from multiple locations in memory. The container is defined as struct bio in <linux/bio.h> This is the structure with comments for each field provided in the Linux Kernel Development textbook:

```
struct bio {
sector_t bi_sector; /* associated sector on disk */
struct bio *bi_next; /* list of requests */
struct block_device *bi_bdev; /* associated block device */
unsigned long bi_flags; /* status and command flags */
unsigned long bi_rw; /* read or write? */
unsigned short bi_vcnt; /* number of bio_vecs off */
unsigned short bi_idx; /* current index in bi_io_vec */
unsigned short bi_phys_segments; /* number of segments */
unsigned int bi_size; /* I/O count */
unsigned int bi_seg_front_size; /* size of first segment */
unsigned int bi_seg_back_size; /* size of last segment */
unsigned int bi_max_vecs; /* maximum bio_vecs possible */
unsigned int bi_comp_cpu; /* completion CPU */
atomic_t bi_cnt; /* usage counter */
struct bio_vec *bi_io_vec; /* bio_vec list */
bio_end_io_t *bi_end_io; /* I/O completion method */
void *bi_private; /* owner-private method */
bio_destructor_t *bi_destructor; /* destructor method */
struct bio_vec bi_inline_vecs[0]; /* inline bio vectors */
};
```

[1]

2.5.2 I/O Vectors

The bi_io_vec variable from the struct bio points to an array of struct bio_vec that are lists of the segments of the current operation [1]. Each structure is treated a a vector made up of the physical page it is on, an offset from the beginning of that page, and the length of the segment. This vector describes the specific location of the segment. Multiple bio_vec structures in an array form the buffer for the bio structure. Each individual block I/O operation has bi_vcnt vectors in the bio_vec buffer starting with bi_io_vec [1]. An array indexer called the bi_idx points to the current bio_vec in the array [1].

2.5.3 Request Queue

Request queues, represented by request_queue, are used by block I/O devices to hold pending requests [1]. The request queue is a doubly linked list. Each link in the list represents one block I/O request. The individual requests are are represented by struct request [1].

2.6 I/O Schedulers

The first I/O scheduler that was used in Linux was the Linus elevator. This scheduler is simpler than its descendents but has problems dealing with request starvation. The Deadline I/O scheduler was developed in order to address the starvation issues of the Linus elevator. However in its efforts to prevent starvation, the Deadline scheduler sacrifices global throughput. The Anticipatory scheduler builds upon the Deadline scheduler. With the deadline scheduler at its core the most important addition is an "...anticipation heuristic." [1] The Complete Fair Queuing (CFQ) scheduler was designed for "specialized workloads,but in that practice actually provides good performance across multiple workloads." [1]. Currently the default scheduler in Linux, the CFQ scheduler's ability to work in many scenarios makes it a valid choice. The No-op scheduler is simple. Without performing any sorting or seek pervention, this scheduler does not need any extra algorithms to function. The No-op scheduler only performs merges into a request queue. The Anticipatory, CFQ, Deadline, and No-op schedulers are available in the 2.6 kernel [1]. The current scheduler can be overridden with the option elevaator=sched_param. Where sched_param is replaced by one of the following: as for Anticipatory scheduler, cfq for Complete Fair Queuing, deadline for the Deadline scheduler, and noop for the No-op scheduler.

2.7 Pages

Physical pages are the basic units of memory in the Linux kernel [1]. Even though processor is capable of addressing a byte the memory management unit will normally deal with pages[1]. Thus in terms of virtual memory, a page is the smallest unit the matters. Each page is represented with a struct page structure defined in <linux/mm_types.h>:

```
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 struct page structure is associated with physical pages. Rather than describing the data that is contained within the pages, this structure is used by the kernel to describe the physical memory [1]. The kernel needs to keep track of all of the pages in the system in order to know if one has not been allocated. If a page has been allocated then the kernel needs to know who the page belongs to.

2.8 Zones

The Linux kernel divides pages into zones based on their physical addressed and the hardware's limitations. Some pages are limited in the different ways they can be used. There are two main hardware limitations that the Linux kernel has to overcome. The first is that some devices can "perform DMA (direct memory access) to only certain memory addresses" [1]. The second being that "some architectures are capable of physically addressing larger amounts of memory than they can virtually address. Consequently, some memory is not permanently mapped into the kernel space" [1]. There are four main zones, defined in <linux/mmzone.h>, that help alleviate these restrictions: ZONE_DMA, ZONE_DMA32, ZONE_NORMAL, and ZONE_HIGHMEM. ZONE_DMA contains pages that are able to undergo direct memory access (DMA), hence its name [1]. Like ZONE_DMA, ZONE_DMA32 contains pages that are capable of undergoing DMA but they can only be accessed by 32 bit devices [1]. Pages that are mapped regularly fall under ZONE_NORMAL [1]. Lastly, ZONE_HIGHMEM contains pages that are not permanently mapped into the kernel's address space [1]. These zones that the pages are partitioned into allow the system to have a pooling area to satisfy allocations as necessary.

2.9 Slab Layer

Kernels need to be able to allocate and free data structures. In Linux, the slab layer is used as a cache for data structures that allows for global control. The slab layer separates various objects into groups. The groups of objects are called caches, and each cache is responsible for one object type[1]. The caches get divided into slabs. The slabs are made up of at least one page or multiple physically contiguous pages. Each individual slab contains a number of the data structures that are being cached and can be in one of three states: full, partial, or empty [1]. Empty slabs contain no allocated objects, partial slabs has some alllocated and some free objects, and full slabs are completely allocated with no free objects. When a new object is requested, a partial slab, given one exists, satisfies the request [1]. If there are no partial slabs, the request is fulfilled by an empty slab. If there are no empty slabs, a new empty slab is created since full slabs

are not able to do anything.

The individual caches are represented by kmem_cache structures. Each structure contains three lists, one for each of the three states mentioned previously. The three lists, slabs_full, slabs_partial, and slabs_partial, are stored inside a kmem_list3 [1]. These lists contain all of the slabs that are associated with the cache. Each slab is represented by a slab descriptor, struct slab:

```
struct slab {
    struct list_head list; /* full, partial, or empty list */
    unsigned long colouroff; /* offset for the slab coloring */
    void *s_mem; /* first object in the slab */
    unsigned int inuse; /* allocated objects in the slab */
    kmem_bufctl_t free; /* first free object, if any */
};
```

3 FreeBSD

3.1 Processes

Similar to linux, FreeBSD processes are programs that are in execution.They have an address space, a set of kernel resources, and at least one thread that executes its code.FreeBSD processes each have a unique process identifier (PID) like Linux processes. FreeBSD processes keep track of their own PID and their parent process' PID. Processes can be in the following states: NEW, NORMAL, or ZOMBIE. Processes are marked NEW when they are created with the fork system call. they are in the NORMAL state when they have all of the necessary resources allocated for a it to begin executing. Processes stay in the NORMAL state until they terminate. Processes are in the ZOMBIE state if it is deceased until it frees its resources and tells its parent process it has terminated [3].

3.2 Threads

A FreeBSD thread is the unit of execution in a process. They need an address space and various other resources. A thread can care its resources with other threads. Threads represent a virtual processor with a full context with of register state and their own stack mapped to the address space[3]. Threads have corresponding kernel threads that have individual kernel stacks to represent the user thread [3]. Threads switch between RUNNABLE, SLEEPING, and STOPPED states. In FreeBSD, a kernel thread will be awakened by a signal only if it sets the PCATCH flag when it sleeps[3]. This is similar to Linux processes with state codes S and D.

3.3 CPU Scheduling

There are two schedulers available for use: the ULE scheduler introduced in FreeBSD 5.o and the traditional 4.4BSD scheduler. The scheduler must be selected at the time the kernel is built. This avoids the overhead that occurs in systems with a dynamic scheduler switch that must be traversed for each scheduler decision [3]. Scheduling is divided into low and high level schedulers. The low level scheduler fines frequently and the high level one only runs a few times per second. The low level scheduler is run every time a thread blocks and a new one needs to be selected to run. A set of run queues maintained by the kernel are organized by priority and help simplify the task of selecting a thread to be run [3]. There is one run queue for each CPU in the system. The high level scheduler is responsible for setting thread priorities and deciding which run queue to place them on [3].

3.4 Types of I/O

FreeBSD had three main kinds of I/O. The first is the character-device interface, then the filesystem, and lastly the socket interface with its related network devices [1]. The character interface "appears in the filesystem namespace and provides unstructured access to the underlying hardware. The network devices do not appear in the filesystem; they are accessible through the socket interface [3].

3.5 Block Devices

FreeBSD dropped support of block I/O. In FreeBSD no important applications depend on block devices. This decision prevents complications to the relevant kernel code by eliminating the need for implementation of aliasing each disk of the two devices with different semantics [4]. Supporting block I/O devices requires the kernel to cache for disk devices. In turn, the block devices become unreliable. Caching data reorders the sequence of write operations [4]. This means that the application will not know the exact contents of the disk whenever it may need.

3.6 I/O Scheduler

FreeBSD, unlike Linux, does not offer multiple scheduler options. Instead it has a Common Access Method (CAM) layer that is in between the GEOM layer and the device driver layer [3]. The Cam subsystem allows for separation of generic device drivers from the drivers controlling the I/O bus.

3.7 Filesystem

The Zettabyte filesystem (ZFS) similar to Linux's EXT4 file system is also case sensitive and journaled. The ZFS never overwrites existing data. This is beneficial because many snapshots and clones can be created easily with no affect on the system's performance [3]. With the ZFS, the state of the on-disk filesystem is always consistent. Any changes that are made to the filesystem accumulate in memory and the changes are periodically assembled and written to the disk [3]. Once the changes are on the disk, the filesystem "makes a checkpoint of the new filesystem state" [3]. This is how the ZSF is able to always be in a consistent state. It only move between two consistent states without an inconsistent middleman. One of the notable components of FreeBSD is the use of files for all of the processes and system storage. File descriptors are used to accomplish this task. The file descriptors are accessed through pipes and sockets similar to Linux[3].

3.8 Virtual-Memory System

The FreeBSD virtual memory system is based on the Mach 2.0 virtual memory system[3]. FreeBSD supports the mmap system call. This means that the address space is used in a less structured manner. For example, shared libraries might place data arbitrarily.Both the kernel and user processes use the same datastructures for managing their virtual memory systems.

- vmspace is used to encompass machine dependent and independent structures describing a process's address space [3].
- vm_map describes the machine independent virtual address space [3].
- vm_map_entry describes "... the mapping from a virtually contiguous range of addresses that share protection and inheritance attributes to the backing-store vm_object"[3]
- vm_object is a structure used to describe the sources of data [3].
- shadow vm_object is a special type of vm_object that represents a modified copy of an original piece of data [3].
- vm_page is the structure that represents the physical memory being used by the virtual memory system.

3.9 Kernel Memory Management

FreeBSD maps memory differently depending on the architecture. For a 64-bit address space architecture, memory is permanently be mapped into the high part of every process address space [3]. In 32-bit architectures the kernel will map like the 64-bit architecture or it will switch between "...having the kernel occupy the whole address space and mapping the currently running process into the address space" [3]. The first option for the 32-bit architecture is the most common. Switching processes will not affect the kernel portion of the address space if memory is permanently mapped to the higher part of the address space[3]. This method also allows the kernel to freely read and write to the address space of the user process and restricts the user process from reading and writing to kernel processes[3].

3.10 Slab Allocator

In FreeBSD a slab is "...a collection of items of identical size" [3]. Each slab is limited to the size of a single page unless the object is larger than a page in FreeBSD 11 [3]. The single page limit prevents fragmentation.

3.11 Kernel Malloc

Malloc() is the preferred method of allocating kernel memory. Its interface is similar to malloc() in C. Malloc() takes in a parameter of the size of memory that needs to be allocated. The kernel memory allocator uses a power of 2 strategy for small allocations and switches to a large block algorithm for allocations larger than a page.

3.12 Shared Memory

In FreeBSD processes create shared memory with

```
void *addr = mmap(
    void *addr, /* base address */
    size_t len, /* length of region */
    int prot, /* protection of region */
    int flags, /* mapping flags */
    int fd, /* file to map */
    off_t offset); /* offset to begin mapping */
```

This shared mapping allows processes to write to a file and those changes are able to be seen by other processes. When processes do map the same file into their address space it is important that they are viewing the same set of pages. Files being currently being used by a lcient of the virtual memory system are represented by a vm_object[3]. If a page fault occurs, the vm_map_entry structure that describes the mapping to the file is used to find the appropriate page.

3.13 Hardware Memory Management

The FreeBSD memory management unit "...implements address translation and access control when virtual memory is mapped onto physical memory."[3] There are a few different memory management unit designs that are common. The first is memory resident forward mapped page tables where the tables are large contiguous arrays [3]. The arrays are indexed by virtual addresses. The second is inverted page table also known as reverse mapped-table. The final design is is just a translation looka-side buffer. The simple hardware design gives software flexibility.

4 WINDOWS

4.1 Processes

A Windows process is a container for a set of resources used when executing the instance of a program [5]. Windows processes are represented by and executive process (EPROCESS) structure. Like Linux, processes have their own independent virtual address space and unique process IDs. Each process points to its parent or the creator process. In Windows it is possible for processes to point to parent processes that do not exist anymore. There are several methods of viewing Windows processes. One of the most popular ways it the Task Manager.

4.2 Threads

Threads are entities withing processes that get scheduled for execution. Windows threads are represented by executive thread objects. Windows threads consist of contents of a set of CPU registers representing the state of the processor, two stacks for executing in kernel mode and user mode, a thread-local storage, a thread ID and sometimes their own security tokens [5]. A windows thread is represented by and executive thread object. Threads can be in several different states: ready, deferred ready, standby, running, waiting transition, terminated, and initialized [5].

4.3 CPU Scheduling

The Windows scheduling system is priority based. This means that a thread with one of the highest priorities and is runnable always runs. However, some high priority threads may not be able to run because they may not have access to the processors they are allowed to run on. Once the scheduler selects a thread to run, the thread runs for a given amount of time. Windows uses 32 priority levels to rank threads [5].

4.4 I/O Manager

The I/O manager is the main component of the Windows I/O system. It manages the communication between an I/O request packet (IRP) and the corresponding device driver [5]. Majority of I/O requests are represented as IRPs in Windows. The manager creates IRPs in memory and passes a pointer to that IRP to the driver, then it gets rid of the IRP when the necessary operation is is complete [5]. The use of the I/O manager as a middleman for creation and removal of the I/O packets means that there is less work for each individual driver to do. This means that drivers are simpler and thus take up less space.

4.5 I/O Request Packets (IRP)

I/O request packets are used to store information necessary to process an I/O request. Since it holds all the information that the driver needs to handle I/O requests, the structure is somewhat self-contained. Additionally, IRPs hold data that is common to all drivers in the stack and information that is unique to its specific driver[5].

4.6 Scheduler

The I/O manager in Windows supports five priority categories: critical, high, normal, low, and very low with a default of normal[5]. The five priority levels are divided into two different prioritization modes. These modes are called strategies. The first strategy is hierarchy prioritization. Each priority has a queue and the strategy decides the order that each of the operations within those queues is processed. Hierarchy prioritization gives more important I/O requests priority over background tasks like indexing or virus scanning. The idle prioritization strategy implements a separate queue for non-idle I/O. Any non-idle, hierarchy operations get processed before the the idle I/O[5]. This means that it would be possible to starve an idle I/O operation if there is one non-idle request, it would be attended to before any idle ones. The idle strategy has a timer to prevent this issue. The timer monitors the queue and ensures that at least one I/O request gets removed from the queue every half second[5]. Additionally, the strategy waits for 50 milliseconds after a non-idle I/O finishes[5]. This waiting period prevents any idle I/O currently on the queue from occurring in the middle of non-idle I/O.

4.7 Memory Manager

The Windows memory manager has two main tasks. The first task of the memory manager is to translate or map a process's "...virtual address space into physical memory so that when a thread running a thread running in the context of that process reads or writes to the virtual address space, the correct physical address is referenced" [5]. The second job of the memory manager is "Paging some of the contents of memory to disk when it becomes overcommitted... and bringing the contents back into physical memory when needed"[5]. In addition to mapping virtual address space to physical and pageing and retrieving data, the memory manager provides a set of services. These services are memory mapped files, copy-on-write memory, support for applications that use a large, sparse address spaces, and provides a way for processes to allocate large amounts of physical memory[5]. The memory manager is fully re-entrantand supports simultaneous execution on multiprocessor systems[5]. This means that if two threads are running simultaneously, they can acquire the resources they need in a manner that they will not corrupt each other's data.

4.8 Pages

Pages are used to divide up the virtual address space. The processors that Windows runs on support two page sizes. The actual sizes vary by processor architecture but they are called large and small[5]. Large pages speed up address translations but attempts to allocate large pages may fail if the system has been running for too long. Pages in a process can either be free, reserved, committed, or shareable. Committed and shareable pages become valid pages in physical memory when they are accessed[5]. Committed pages cannot be shared with other processes. Shareable ones, like their label implies, can be shared with other processes but could also only be used by one.

4.9 Shared Memory and Mapped Files

Shared memory is implemented using section objects, which are exposed as file mapping objects from the memory manager [5]. The file mapping objects are used to map virtual addresses. A section can be accessed by one process or many. Mapped files can by used by applications to perform I/O operations by making the file appear in the application's address space[5].

5 Conclusion

This overview of kernel implementations of various drivers, system calls, and other constructs in the context of Windows, and FreeBSD in comparison to Linux show how the three kernels are similar and different. Each of the kernel related functionalities show the common theme of FreeBSD and Linux being very similar, with Windows taking a different route in terms of actual implementation. However, in the end, Linux, FreeBSD, and Windows keep the overall semantics of processes, I/O, memory, from an abstracted perspective the same.

References

[1] R. Love, *Linux Kernel Development: A thorough guide to the design and implementation of the Linux kernel*. Crawfordsville, IN: Pearson Education Inc, 2010.

[2] "Cpu scheduling." [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/s-cpu-scheduler

[3] M. K. McKusick, G. V. Neville-Neil, and R. N. M. Watson, *The Design and Implementation of the FreeBSD Operating System*. Westford, MA: Pearson Education, 2015.

[4] "Chapter 9. writing freebsd device drivers," 2013. [Online]. Available: https://www.freebsd.org/doc/en_US.ISO8859-1/books/arch-handbook/index.html

[5] M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals: Part one*. Redmond, WA: Microsoft Press, 2012.