Comparing and Contrasting Essential System Operations in the Linux, Windows, and FreeBSD Operating Systems

Alex Hoffer

CS 444
June 2017
Spring 2017

Contents

1 Introduction

Computer operating systems have a rich tapestry of history. When we observe the development of the modern operating systems from the ones responsible for handling giant, room-filling computers (performing what we could consider today as ho-hum computations) to the current climate of tiny processors coordinating and tackling lofty challenges in our laptops and mobile devices, it is important as computer scientists to understand which components of an operating system emerge as being essential to the proper functioning of a computer. Through this study, we can see that processes and threads, I/O and provided functionality, and memory management are three operations of an operating system that are crucial to making a computer that is intended for widespread human usage work correctly. While being able to pinpoint these three features as important operating systems topics is useful, it is only when we see how they can become reality that things become truly interesting. It is the purpose of this paper to elucidate this by explaining how Linux, Windows, and FreeBSD each implement these core functions in their operating systems. Along the way, we will be able to see where they differ in such implementations and where they run parallel to each other.

2 Processes and Threads

A *process* is an "instance of an executing [computer] program" [3]. Each process is given its own memory space to perform operations within. One central task of an operating system is to synchronize process execution, as well as to decide which process gets ran when (this is called *scheduling*). A *thread* is similar to a process in that it is subject to similar synchronization and scheduling constraints, though threads exist within the process space and each thread running within such a space shares resources with other threads in that same space. Processes and threads are essential to an operating system because they greatly increase the amount of work being done by providing an illusion of multiple tasks being completed at once. This section is devoted to explaining how processes and threads are implemented in Linux, Windows, and FreeBSD, with the manner in which they are scheduled for CPU time also being covered.

2.1 Windows

2.1.1 Similarities to Linux

Process Creation When a new process is created in either Windows or Linux, the new process is also given a newly created thread. Processes in both Windows and Linux can have 1 or more threads assigned to them, though multithreaded programs in Windows are easier to program (the explanation for this can be found in Windows' *Differences from Linux* subsection). Additionally, newly created processes in Windows are allocated similar memory resources to that of Linux. Each process on both systems has specially allocated memory segments corresponding to the text segment, initialized data segment, uninitialized data segment, stack, and heap of the running program that are unique for that specific process [8].

Process Identification Both offer the ability to get the currently running process's ID, with *GetCurrentProcessId()* in Windows and *getpid()* in Linux [8] [3]. Windows and Linux are also both subject to *race conditions*, or the tendency for a program to rely on one process to complete a sequence of actions before another process completes its own sequence, since both of their processes are subject to nondeterministic scheduling (more on CPU scheduling of processes later).

Process Termination/Exiting A process can terminate in Windows and Linux in either a normal or abnormal fashion. Termination in normal or abnormal fashion for both of these systems yields an exit value that other processes can access. When a process dies in either Windows or

Linux, if it has a child that has not properly been waited for and no terminating signal has been sent by the user, then the child process will continue to run. This orphaned child process is called a *zombie*.

Thread creation Creating a thread complete with stack size specification and security attributes can be done in one complex function call in Windows, but Linux can specify the exact same things for a thread as Windows can, just through the use of four methods, found below courtesy of [3]:

- pthread_create
- pthread_attr_init
- pthread_attr_setstacksize
- pthread_attr_destroy

[2] claims POSIX Pthreads, which are used prominently in the Linux operating system, provides similar features to Windows threads, but Windows allows for a "broader collection of functions". In fact, Windows threads are similar enough with POSIX Pthreads in how threads are managed that there are open source Pthreads libraries available for Windows system developers.

Thread synchronization Threads in both Windows and Linux share resources with other threads. Additionally, both Windows and Linux use semaphores and mutexes to restrict access to shared resources between threads.

Thread termination/exiting Normal thread exiting can be done in a single function in both Windows and Linux. The function for this in Windows is *ThreadExit*, and in Linux it is *pthread_exit* [8] [3].

Timeslice properties Both Windows and Linux use timeslices, or quantums, of about 10 milliseconds to in the hundreds of milliseconds (though Windows has traditionally lower timeslices, likely in an effort to favor higher priority threads, which is discussed in a later section). Both operating systems also institute *reentrant* timeslices, which means that if a thread is interrupted during its execution, then it is returned with what remains of its timeslice after interruption, rather than a fresh timeslice.

Base priorities Both operating systems assign a *base priority* to a process, that is, a priority calculated according to a scheduling algorithm. Although Windows and Linux both favor different priorities and utilize different scheduling algorithms, the idea of attaching an initial priority subject to change to a process is universal across them both.

2.1.2 Differences from Linux

Process Creation To create a new process in Windows, we must execute a *CreateProcess* call. To do this same function in Linux, we traditionally use the *fork* system call. The differences between these calls is indicative of how the two operating systems handle the parent and child relationship inherent in a process hierarchy. Specifically, *CreateProcess* takes 10 arguments, including (but not limited to) desired security attributes, creation flags, and environment. The *CreateProcess* function signature is listed below [7]:

```
BOOL WINAPI CreateProcess(
  _In_opt_      LPCTSTR                lpApplicationName,
  _Inout_opt_   LPTSTR                 lpCommandLine,
```

```
  _In_opt_      LPSECURITY_ATTRIBUTES lpProcessAttributes,
  _In_opt_      LPSECURITY_ATTRIBUTES lpThreadAttributes,
  _In_          BOOL                  bInheritHandles,
  _In_          DWORD                 dwCreationFlags,
  _In_opt_      LPVOID                lpEnvironment,
  _In_opt_      LPCTSTR               lpCurrentDirectory,
  _In_          LPSTARTUPINFO         lpStartupInfo,
  _Out_         LPPROCESS_INFORMATION lpProcessInformation
);
```

A Linux developer, on the other hand, can spawn a child process by simply using *fork()*.
Another important difference in process creation between these two systems is that the *CreateProcess* call returns a *bool* that is set to true if the process and subsequent thread are successful in creation, and false otherwise. Alternatively, *fork()* returns an integer value that, if the process is successfully created, corresponds to the process's ID. Thus, we can see that with Windows, the act of specifying a process's attributes are quite important, while with Linux, the programmer's awareness of the process hierarchy is emphasized, since they can easily store and access process IDs in order to identify which process is running at a given time. Meanwhile, Windows does not maintain the relationships between parent and child processes [2]. [2] goes on to claim that the reason behind this lack of interest in parent and child process relationships in Windows is because the act of forking in Linux is, in practice, dreadful when creating multithreaded programs. This is because the *fork* system call makes it so that the newly created child process is an exact replica of the parent process, retaining precise copies of the parent's threads and synchronization objects, making management between multiprocessors a nightmare.

Process Identification In Windows, processes can be identified by both handles AND process IDs, while in Linux, we can only identify a process by its ID, which we can access using the *getpid()* system call. With Windows, we can get a process handle by using the *OpenProcess* function, and get a process ID by using the *GetCurrentProcessId*. The *OpenProcess* function allows for you to enumerate that handle's access rights and contains the ability to set whether or not the process is inheritable. I think Windows includes this handle as well as the process ID in a strategy to make security a priority in the act of process management, since you can specify what a process can access through its handle. In this sense, the identity of a Windows process is a much more complex entity, since it can include a large quantity of information. In fact, both the ID and the handle of a process is stored in a *PROCESS_INFORMATION* structure. Access control information of processes in Linux can be found in a process's *process credentials*.
Of note is that Windows does not provide a function to discover a process's parent's ID, while Linux processes maintain records of their parents' process IDs [3]. Again, this is likely because of Windows' different methods of handling process hierarchies.

Process Termination/Exiting In Linux, normal process termination occurs with the *_exit()* system call [3], while in Windows the function responsible for normal process termination is *ExitProcess* [7]. The act of process termination is reflected upon system resources differently across these two operating systems. When a process terminates either normally or abnormally in Linux, synchronization resources such as semaphores or memory locks are closed and unlocked, respectively. Like [2] claims, with Windows, it is essential that resources shared across processes like semaphores and memory locks are deliberately freed by the programmer in the act of termination. That is, Windows offers less garbage collection in process termination when it comes to synchronization resources than Linux does.

Thread creation In Linux, threads are not built-in and must be implemented using the POSIX Pthreads library. Windows, on the other hand, has built-in threads which serve as the kernel's basic unit of scheduling. Several different parameters present in thread creation function for Windows than Linux, just like how a Windows process requires more parameters in its creation. This is because Windows intends for you to specify more about process and thread functionality during the genesis of processes and threads than Linux does. In Windows, threads are crucially important. The thread creation function signature for Windows is as follows:

```
HANDLE WINAPI CreateThread(
  _In_opt_   LPSECURITY_ATTRIBUTES     lpThreadAttributes,
  _In_       SIZE_T                    dwStackSize,
  _In_       LPTHREAD_START_ROUTINE    lpStartAddress,
  _In_opt_   LPVOID                    lpParameter,
  _In_       DWORD                     dwCreationFlags,
  _Out_opt_  LPDWORD                   lpThreadId
);
```

Meanwhile, to create a thread in Linux, we simply use the *pthread_create* function, which takes only four parameters. Again, we can see through this example that Windows emphasizes the programmer's specification of attributes such as access control in the creation of threads and processes.

Thread synchronization An important distinction in Windows vs. Linux thread management is the inclusion of *events* in Windows as a method of thread synchronization. Events are objects that are used to notify waiting threads when an event, like I/O operations on files, are completed [7]. Windows also utilizes *critical sections* while Linux implements *conditional variables*, like *pthread_cond_wait*.

Scheduling Classes In Windows, there are two scheduling classes, *real time* and *dynamic*. Windows favors threads that have a higher priority value (e.g. 32 rather than 0) attached to it [8]. Linux, on the other hand, uses three scheduling classes, *normal*, *Fixed Round Robin*, and *Fixed FIFO*, though the default scheduling class is the *Fixed Round Robin*, or the *Completely Fair Scheduler (CFS)* [3]. Linux favors threads that have a lower priority value attached to it in an effort to institute a level of class equality in their scheduling methodology.

2.2 FreeBSD

2.2.1 Similarities to Linux

UNIX-like operating systems Since FreeBSD is a UNIX-like operating system, FreeBSD and Linux are very similar in how they handle processes and threads. First and foremost, like Linux and Windows, FreeBSD processes contain information that an operating system needs to have in order to handle program execution, such as process ID, registers, stack, heap, etc.

Preservation of the Parent-Child Relationship Like Linux, FreeBSD uses a *fork()* system call to create a new child process. This call returns the pid of the new process, and that new process can retain the ID of its parent using a function call [5]. In this way, FreeBSD and Linux share a common interest in preserving an easy and convenient way for a developer to understand the process hierarchy of a program's execution. Thus, FreeBSD does not contain the *handles* or complex forking procedure that Windows utilizes. Like both Windows and Linux, a programmer can get the running process's ID with a *getpid()* function call. Unlike Windows but like Linux, a child process is the exact duplicate of the context of its parent [3], which, as stated previously, is a concern in the context of multiprocessing.

POSIX-Compliance Like Linux, FreeBSD is POSIX-compliant, meaning threads can be managed using the Pthreads library. This means that the following sample of two fundamental thread functions are the same between FreeBSD and Linux:

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
        void*(*start_routine)(void *), void *arg)

int pthread_join(pthread_t thread, void **value_ptr)
```

It follows that since FreeBSD utilizes the Pthreads library, nearly all of its thread management is identical to that of Linux: there are not many major differences between thread creation, destruction, and synchronization between the two operating systems.

Nice values and similar priority model Like Linux, FreeBSD sets large *nice values* (numbers that correspond to job priority) such as the range between 0 to 20 as being of low priority, while small nice values such as the range between -20 - 0 are set for higher priority for execution [1].

Getting, setting priority Both Linux and FreeBSD programmers can include the *sys.resource.h* file in order to use functions that can get the nice value of a specified process or set the nice value of a specified process. These functions are included below, courtesy of [6].

```
int getpriority(int which, id_t who);
int setpriority(int which, id_t who, int prio);
```

2.2.2 Differences from Linux

No considerable differences in processes There are no significant differences in the way processes themselves are created, monitored, or destroyed between FreeBSD and Linux. There are, however, differences in how processes are chosen for CPU time (outlined in a further section of this paper).

No considerable differences in threads Like with processes, the emergence of both FreeBSD and Linux from the UNIX operating system led to few discernible differences between the two in managing threads.

Default schedulers The default scheduler for the FreeBSD is the ULE, while the default scheduler for Linux is the CFS. The ULE is built on low-level and high-level scheduling subsystems. The low-level one is frequently running and quickly grabs the next thread to run based off what the highest prioritied task is in the system's *run queues* [5]. Meanwhile, CFS does not use run queues to choose the next process to run, instead this scheduling strategy computes the next process based on round-robin time-sharing, as described in the section on Linux vs. Windows for scheduling.

3 I/O and Provided Functionality

Input/Output, or I/O, operations are crucial to an operating system because a computer cannot perform particularly interesting computations without receiving data from an exterior source or writing data to an exterior source. To take a lighthearted example, try to think of a way that I could type this LaTeX file and generate a PDF from it without the operating system (at the very least) receiving data from my keyboard. Provided functionality is important to the programmers responsible for writing programs that must utilize kernel operations in some way. Without provided functionalities such as kernel data structures, for example, how would we conveniently store and access memory pages?

3.1 Windows

3.1.1 Similarities to Linux

I/O Both operating systems have a design goal to provide what [7] calls an "abstraction of devices", meaning a universal set of software tools by which the programmer can manage I/O functions. Additionally, both operating systems implement, in their own way, a *Hardware Abstraction Layer* (HAL), which is a generalized interface that allows programs to directly manipulate device hardware [8]. Note, with the HAL as a perfect example, both operating systems' inclusion of abstraction in their design decisions. Linux and Windows both desire to be able to manage the I/O requests of a large variety of devices and file systems, since they are intended to be commercially used operating systems. While they share this design principle, the way in which they approach this abstraction is the source of a lot of differences between the two systems.

Provided functionality In both operating systems, their respective *Application Programming Interfaces* (APIs) are written to respond to *events*, which is when a user wants something to happen or the device wants to pass a message to the I/O manager. Windows has three provided functionalities for programmers to allow communication between Userland and I/O drivers: *buffered I/O*, *direct I/O*, and *memory mapping* [2]. Linux also provides these same three functionalities [**?**]. Both operating systems also provide system-defined data structures, algorithms, and cryptographic protocols, though there is some differences in which options are provided. For example, Linux and Windows kernels both have linked lists, though Windows provides both singly and double linked lists, while Linux only provides a doubly linked circular list by default. The Windows linked lists are included in the file "Ntdef.h", while the Linux kernel's list is included in the "list.h" file. In terms of cryptography, both use things like *access control lists* (ACLs), PKI, and such to provide basic security.

3.1.2 Differences from Linux

I/O A key difference between Windows and Linux that explains further differences is the fact that Windows and its kernel are separated and are only combined through the use of calls to the Application Binary Interface (ABI) [2], while Linux's device drivers are located within the kernel itself and not relying on calls to the ABI [3]. Programmatically speaking, in Windows, files and devices are managed as *objects* in the object-oriented paradigm, while in Linux files and devices are managed through the use of *file descriptors*, further exemplifying the classic UNIX saying that "Everything is a file". Additionally, Windows' I/O has three device driver layers that each I/O request can be subjected to (though a request can be handled by merely one of these layers). These three layers are *filter, function,* and *bus* [7]. Consider the following block of Microsoft C code courtesy of Microsoft's Github that initializes a function driver for a Bluetooth device:

```
NTSTATUS
DriverDeviceAdd(
    IN  WDFDRIVER        _Driver,
    IN  PWDFDEVICE_INIT  _DeviceInit
    )
```

On the other hand, Linux's drivers are defined as being *block*, *character*, or *network*, and an I/O request can't be processed by all three. Since Windows' kernel doesn't have any device driver development, it instead relies on the *Windows Driver Model* (WDM) to provide device drivers. When a piece of software in Windows requests an I/O operation, the Windows kernel dispatches the request to its *I/O Manager*, which translates the request into an *I/O Request Packet*, or IRP, which is then sent to the device driver layers. The function layer consists of the predominant drivers that map programming interfaces to specific devices, the bus layer helps out device hosting bus

controllers, and the filter layer offers extra IRP processing. Meanwhile, in LinuxLand, these device drivers are wildly different: the character device driver type handles simple devices that can be read one byte at a time, the block device driver type handles complex devices whose requests come in the form of blocks of data, and networking options are there for moving data to and from a network.

Provided functionality Windows linked lists come in either doubly linked or singularly linked varieties, while Linux only offers doubly linked lists by default, which means you must use a Linux patch file to change this functionality in the Linux kernel. Naturally, there are some differences in the two operating systems' APIs for when it comes to manipulating lists. For example, Windows offers routines like *RemoveHeadList()*, whose function signature is listed below, which can perform convenient operations such as removing the first element of a list.

```
PLIST_ENTRY RemoveHeadList(
  _Inout_ PLIST_ENTRY ListHead
);
```

The same removal of a list head can only be completed in Linux using its "List.h" file by doing the following:

```
if (!list_empty(myQueue->queue))
    list_del(myQueue->queue.next);
```

As we can see, there are different routines provided in the two different systems for performing basic data structure manipulation. In addition to differences in data structure implementation between the two kernels, there is a lot of difference in how the two systems handle encryption. Some of these differences are difficult to compare since Linux is open source and Windows is proprietary. For example, while both kernels provide things like access control lists to institute a level of security, Windows also uses security measures such as the *Microsoft crypto application programming interface* [2]. Therein lies a fundamental and interesting difference between Linux and Windows that is microcosmic of a common theme expressed throughout my writing this quarter: while we can look directly at how Linux implements things like security and critize or laud them, Windows' source code is on lock down, so all we can do is look at helper routines and things of the like in order to assess how they might be used.

3.2 FreeBSD

3.2.1 Similarities to Linux

I/O Since both operating systems are derived from UNIX, there are of course plenty of similarities between FreeBSD and Linux in terms of how I/O is handled and which functionalities are provided. For example, both FreeBSD and Linux specify that devices should be accessed through device-nodes, which are a special type of file [5] [3]. Another similarity is that both FreeBSD and Linux have character and network device driver types. These device driver types are essentially the same in both of these operating systems, though FreeBSD emphasizes the character device driver type, while Linux emphasizes the block device driver type (described more in the "Differences from Linux" piece of this section).

Provided functionality Both Linux and FreeBSD offer a lot of built-in functionalities for programmers to use when doing systems programming. For example, both have convenient and easy-to-use methods that allocate memory within the kernel. Here is the function signature for allocating memory in the FreeBSD kernel:

```
void* malloc(unsigned long size, struct malloc_type *type, int flags);
```

Meanwhile, in Linux, here is the simple method we use to allocate memory in the kernel: [3]

```
void * kmalloc(size_t size, int flags);
```

As we can see, these two routines to allocate memory are very similar. In fact, FreeBSD and Linux provided functionalities universally tend to be somewhat close, possibly because of their shared inheritance from UNIX. Both FreeBSD and Linux have data structures for kernel usage including lists, queues, and trees, for instance.

3.2.2 Differences from Linux

I/O The principal difference between these two UNIX-derived systems is that in FreeBSD, the block device driver type is not present. This is manifested in the fact that FreeBSD has two device driver types, which are character and network. Meanwhile, Linux has three device driver types: character, block, and network. On the surface, this may seem like a superficial difference, but in fact, this fundamentally changes the way these two systems handle I/O requests. FreeBSD treats the character device driver type as its cardinal type [1]. Unlike Linux, where the character device driver type is merely used to handle the requests of simple devices where I/O can be managed one byte at a time, FreeBSD handles all devices using this one-byte-at-a-time strategy. Meanwhile, in Linuxland, the block device driver type handles devices where requests are represented by a block of memory (hence the name). Oddly enough, the FreeBSD organization's programming handbook really looks down on the block device driver type, calling this disk device type "unusable, or at least dangerously unreliable", due to the kernel's caching of its operations.

Provided functionality The primary difference between what these two operating systems offer in terms of provided functionalities, in my opinion, is found in how they implement cryptography, which has important ramifications for their respective securities. Specifically, FreeBSD uses the *Crypto* interface in its kernel [1], while Linux uses its own *Crypto* interface in its kernel [6]. Original, right? Regardless, Linux's Crypto interface is far more robust. Though FreeBSD's interface focuses solely on the user space, Linux's focuses on both the user space interfacec and the programming interface, and has loads of different cryptographic protocols built into its API. The breadth of Linux's different crypotgraphic protocols may be due to its popularity. Let us take the topic of built-in Cipher algorithms, for instance. FreeBSD includes the following functions that implement Cipher algorithms for security purposes: CRYPTO_AES_CBC, CRYPTO_AES_NIST_GCM_16, CRYPTO_AES_ICM, and CRYPTO_AES. Meanwhile, documenting the number of Cipher algorithms that Linux offers here would be silly: there are far, far too many. In fact, Linux offers a multitude of different cryptographic protocols by which the programmer can pretty much set up the security of his system to his fancy. It is in the vastness of Linux's provided security functionalities compared to the relative sparseness of FreeBSD's provided security functionalities where we find the most profound differences, I think.

4 Memory Management

Memory management is a crucial aspect of any operating system. Memory management's primary functions are to provide virtual memory spaces to different processes, properly map these virtual memory spaces to their corresponding physical addresses, and allocate and de-allocate pages of virtual memory and assign such pages safely and responsibly to processes as they request them. Without the ability to manage memory, an operating system would descend quickly into chaos.

4.1 Windows

4.1.1 Similarities to Linux

Virtual memory addresses Both systems support the use of 32-bit and 64-bit virtual memory addresses. [8] [3]

Hardware Abstraction Layer Of course, a common similiarity that we encounter in the comparison of any two operating systems is their use of the *Hardware Abstraction Layer* (HAL). The HAL is beneficial to programmers because it handles system-specific operations and everything else in the kernel is built on top of it, which means kernel developers can implement things without being concerned about what system it's running on. Since we're discussing two widely used operating systems in Windows and Linux, the fact that both use a HAL is unsurprising, after all, they wouldn't be that popular if their kernel operations weren't portable to a myriad of different computers. In the context of memory management, the HAL lets the programmer in both operating systems to write methods that can do things like memory paging in a universal manner.

Data Structures Naturally, the way in which kernels generally keep track of what segments of memory are occupied are through the use of data structures. There are some similarities in data structure usage to do this between these two systems. In Windows, the tree data structure is exclusively used, and each node of the memory tree consists of variables that are called *Virtual Address Descriptors* (VADs) [2]. These descriptors hold whether a given node is free to write to, occupied with data, or reserved to be filled with data. In Linux, a different data structure IS used by default (noted below in the "Differences" section, along with implications), but when this data structure reaches 32 items, it is converted into a tree [6]. This means that a programmer in both systems can access free, taken, and about-to-be-taken memory addresses using tree traversal methods. Here is the function signature for allocating memory to be placed in a memory tree in the Windows kernel: [2]

```
void __RPC_FAR * __RPC_USER midl_user_allocate (size_t cBytes);
```

Meanwhile, here is the function signature for allocating memory to be placed in either a memory tree or the data structure used when there are less than 32 items needed:

```
void * kmalloc(size_t size, int flags);
```

4.1.2 Differences from Linux

User space vs Kernel space memory allowance In Linux, 3 GB of memory is allocated for the user space and 1 GB of memory is allocated for the kernel space, while in Windows, only 2 GB of memory is allocated for the user space and 2 GB is allocated for the kernel space [8] [3]. This is an intriguing difference, since it means that the Linux memory management system favors the user space, while Windows places equal emphasis on both. Perhaps this is because Windows expects its kernel to handle larger, more robust computations where additional memory space is necessary. Another difference is that while both systems support 32-bit or 64-bit virtual memory addressses, only Linux has both built-in. For Windows, using 64-bit virtual memory addresses is only possible through the use of the *Windows 64-bit Edition*.

Data Structures As mentioned earlier, in Linux, if there are less than 32 items in a memory data structure, it is by default not a tree. It is instead a linked list. In this sense, memory management in Linux could be construed as being less rigid for the programmer than in Windows. Since linked lists are by reputation less efficient than trees, Linux likely switches to trees once the data structure hits the 32 item limit because if there are less than 32 items in a linked list, its difference in performance

when compared to the tree data structure is negligible. But, since there are both trees and linked lists within the Linux memory management system, this means there are provided functionalities within the Linux kernel that aren't present in the Windows kernel related to linked list operations. Let's consider one that I encountered when Linux kernel hacking for Project 4 of Kevin McGrath's Operating Systems 2 course:

```
list_for_each_entry(sp, slob_list, list){}
```

Found in the slob.c file of the memory management folder in Linux, this method iterates through a list of pages as defined by the argument *slob_list* and places each passing page into the variable *sp* so the programmer can see what's going on a given slab of memory.

4.2 FreeBSD

4.2.1 Similarities to Linux

Paging system Like in the section on Windows' memory management similarity to Linux, it's important to note that a major similarity between FreeBSD and Linux is their emphasis on virtualized memory. Within both kernels' paging systems, they use what is called the "buddy system" when allocating memory at the page level [5] [6]. The "buddy system" in this context is when a block of memory is divided, or *partitioned*, into smaller chunks to better fit the size requirements of memory requests. This means that our favorite Best-Fit algorithm from Project 4 of Operating Systems 2 could just have easily been written into FreeBSD. Additionally, both kernels' paging systems use *on-demand paging* and *swapping* [1] [3]. On-demand paging refers to a construct of virtualization where "pages of data are not copied from disk to RAM until they are needed". It's a kind of swapping, which is when you copy everything related to a process from main memory to an auxiliary type of storage. Of course, these are necessary methods from a memory management perspective since occupying a massive amount of main memory would greatly slow down computing speed.

4.2.2 Differences from Linux

Portability Since FreeBSD runs only on x86 architectures, all computers running it must conform to its *Pmap module* [5]. The Pmap module is basically an abstracted layer that handles the physical hardware manipulations necessary to manage memory programmatically, although only with computers of the x86 architecture type. This means that FreeBSD's memory management system doesn't have a HAL of the breadth like we've seen with Linux (or even Windows!). Linux transforms architecture specific data structures for use with memory management into kernel page tables when code is compiled, giving its memory management system portability.

Allocating memory We've seen what allocating memory in the kernel looks like in Linux and Windows, so let's take a look at how to allocate kernel memory in FreeBSD: [1]

```
void* malloc(unsigned long size, struct malloc_type *type, int flags);
```

Notice the additional argument separating it from its Linux equivalent. Interestingly, in FreeBSD, the call to allocate memory in Userland is basically the same as it is to allocate memory in the kernel.

Different memory allocator algorithms One key difference programmers may notice between memory management in Linux and FreeBSD is the memory allocators they use. FreeBSD utilizes what is called the *zone* allocator, which is considered a *region-based* kind of memory management [4]. This means that modules in the kernel will tell the zone allocator the sizes of memory zones (think of these zones as chunks in memory) before running, and the zone allocator will split up memory

to correspond to these zones. Then, when running, processes in need of memory are given these pre-sliced zones of memory. Linux, on the other hand, typically uses the *slab* allocator [6]. The slab allocator, which is like a version of the zone allocator that has been adapted to use methods that function essentially as constructors and destructors in object-oriented programming paradigms. Imagine that, object-oriented programming in the C language!

5 Conclusion

Through our research of the Linux, Windows, and FreeBSD operating systems, we have seen that there are a multitude of different ways programmers can implement processes and threads, I/O and provided functionality, and memory management. Look no further than the difference in how the parent-child process relationship is handled in Windows vs. Linux or how FreeBSD lacks the block device driver type that Linux relies heavily on. These examples make it clear that there is no one simple recipe to produce a working operating system, and that design decisions and tradeoffs are as present in kernel development as they are in any other worldly subject. But programmers who desire to write good code in all three of these operating systems need not to worry, because there are a great number of similarities between them as well. Consider, for example, how similar thread synchronization constructs are across the three systems, or how similarly the data structures within each of the three kernels resemble each other. It is only with the knowledge of the similarities and differences of how these three essential features are implemented in popular operating systems that we can confidently call ourselves programmers.

References

[1] Nathan Boeger and Mana Tominaga. Freebsd system programming, 2001.

[2] Johnson M. Hart. *Windows System Programming (3rd Edition)*. Addison-Wesley Professional, 2004.

[3] Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, San Francisco, CA, USA, 1st edition, 2010.

[4] Greg Lehey and Marshall Kirk McKusick. *The Complete FreeBSD*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 4 edition, 2003.

[5] Marshall Kirk. McKusick, George V. Neville-Neil, and Robert N. M. Watson. *The design and implementation of the FreeBSD operating system*. Addison-Wesley/Pearson, 2015.

[6] Evi Nemeth, Garth Snyder, Trent R. Hein, and Ben Whaley. *Unix and Linux System Administration Handbook*. Prentice Hall, 2011.

[7] Mark E. Russinovich, Alex Ionescu, and David A. Solomon. *Windows internals, Part 2*. Microsoft, 2012.

[8] Mark E. Russinovich, David A. Solomon, and Alex Ionescu. *Windows internals*. Microsoft Press, 2012.