Dylan Porter

Operating Systems

Dr. Drozdek

OS Paper

The Designs and Principles of Windows

The prospects of the Windows operating system are complex and therefore require complex algorithms to handle potential processes, threads, and jobs in its functionality. The entire model of this OS, naturally, will require a robust ability to handle most problems facing computers – and will also need to focus on efficiency in its handling. Although there are many ways in which the Windows operating system handles certain problems, the entirety of its design has provided countless amounts of people with stability and security in a compact and useful operating system. In this paper, I intend to address the major operations and functions ascribed to the operating system and I hope to elaborate on how such techniques of implementation work efficiently for a desirable user experience. Throughout this research, I will be covering many of the subsections of the Concepts and tools, system mechanics, processes, threads, and jobs concepts of the operating system. In addition, I will also analyze the I/O System, and Crash Dump Analysis implementations to understand other important concepts as well.

With the idea of efficiency and process handling, we must first analyze the concept of virtual memory in the Windows operating system. With any access of memory that is outside of main memory, Windows needs to attribute a memory manager to aide in the execution of various processes. Most notably, the Windows operating system must control the mapping of virtual memory to physical memory to ensure safe operation. As Russinovich mentions in Windows Internals 6th edition, “At run time, the memory manager, with assistance from hardware, translates, or maps, the virtual addresses into physical addresses, where the data is actually stored. By controlling the protection and mapping, the operating system can ensure that individual processes don’t bump into one another or overwrite operating system data” (Russinovich, 15). With this illusive technique, each process has the illusion of a large and private address space. Even with this illusion, an effective mapping system can pull off a resourceful handling of virtual memory if the mapping from virtual to physical memory is contained. It is also definitively important that the memory manager is useful in other regards when considering the proportional space of main to virtual memory.

With the limits of storage in main memory, the Windows memory manager needs to take into consideration ways to free up access. The Windows memory manager will ultimately be tasked to free up main memory so that other operating processes can be stored and executed. It is clear, from Russinovich’s consideration, that the memory manager will need to transfer memory to the disk in a case such as memory management. This technique consideration, described as paging in the Windows operating system, is an underlying property of how Windows effectively manages processes in the generic sense. The text mentions that “Paging data to disk frees physical memory so that it can be used for other processes or for the operating system itself” (15). In this implementation, the Windows operating system needs to incorporate the previously mentioned techniques of handling virtual memory into the concept of paging. If for example, a request is generated for an address that was moved to disk, the software and hardware designated will ultimately bring the physical contents of that masked address back to main memory for access. This method of swapping, although abstracted, will become useful when main memory is needed most.

In the concept of virtual memory, the understanding of how Windows deals with memory specifically is critically important to the entirety of the OS. From Windows Internals Part 2 by Russinovich, Windows seemingly contains a total of eight components for memory management including a set of executive system services for memory allocation, a translation-not-valid and access fault trap handler, and six key top-level routines tasked with memory management as well (188). The first component is serves with allocating, deallocating, and managing virtual memory. They are integral to the memory management of the Windows operating system. The translation-not-valid and access fault trap handler, on the other hand, resolve memory management exceptions and make virtual pages resident on behalf of a process (188). The concept of page faults and memory problems are ultimately mitigated through these components’ operations in the system. Finally, the six top-level routines are important to memory management as well. They include the balance set manager, the process-stack swapper, the modified page writer, the mapped page writer, the segment dereference thread, and the zero page thread. The balance set manager is tasked with calling the working set manager when memory falls below a certain threshold. The working set manager then is tasked with the entirety of the memory management policies. The process-stack swapper, on the other hand, “performs both process and kernel thread stack inswapping and outswapping” (188). In addition, the modified page writer and mapped page writer are both responsible for writing dirty pages in the interaction of the process-stack swapper. The only difference is that the modified page writer provides the dirty pages to certain paging files while the mapped page writer provides the dirty pages to disk. Alternatively, the segment dereference thread is tasked with cache reduction and page file change. From that, it would follow that the zero page thread is responsible for freeing pages to satisfy future needs in the system (188-9). With the addition of these components in the operating system, virtual memory mapping is easy, efficient, and secure for a variety of purposes. Specifically, these components help keep track of freed-up memory availability, virtual memory swapping, trimming the page cache for better allocation of fast memory, and mitigating page faults in the system (189). With these component abilities, memory management is fast, efficient, and useful in its entirety. Ultimately, the Windows operating system uses these components to translate or map a process’ virtual address space into physical memory accurately and to page some contents of the memory to disk when there is not enough main memory available (187).

Considering data access, Windows Internals Part 1 shows that the Windows operating system has provided a kernel mode and user mode in its functionality. These separate modes are designated to potentially limit access to critical operating system data that could cause physical or internal harm to the device. The clear designations and privileges ascribed to kernel mode are quite unique and useful. The text makes it clear that such things like system services and device drivers will require execution in kernel mode. In a stark contrast to the normal memory access of processes, access to kernel mode defines that “kernel-mode operating system and device driver code share a single virtual address space” (17). This designation also shows that there are requirements to read or write to a page that is of higher access privilege. Specifically, the processor will need to be in kernel mode if there are pages in system space, for example. There is, in addition, a read-only specification for static data that is not writable in any operating mode. This read-only data may contain information that could cause devastating effects if modified. Kernel mode in 32-bit Windows, however, can write over “system space memory and can bypass… security to access objects” (17). This can obviously create many problems if the programs running in kernel mode are not designed properly.

With the distinction of user and kernel mode purpose, it is also important to define the differences of use and architecture. In Windows, the architecture of the user and kernel modes are noticeably different. The user mode, for example, contains system support processes, service processes, user applications, environment subsystems, and subsystem DLLs (35). On the other hand, kernel mode contains the executive, kernel, device drivers, hardware abstraction layer (HAL), along with Windowing and graphics. As mentioned before, the processes running in kernel mode have, in most cases, access to system space whereas the processes running in user mode do not. System support processes, service processes, user applications, and Environment subsystem server processes are all examples of processes that do not require kernel access. Examples of these categories can include the logon and Session manager processes or the Task Scheduler and Print Spooler processes (36). In summary, anything that is not a specific Windows service or not started by the control manager will most likely not require kernel access. The executive, kernel, device drivers, and hardware abstraction layer, on the other hand, will require kernel access. The windows executive, the text mentions, can contain services such as memory management, process and thread management, security, I/O, networking, and process communication (36). The kernel, in addition, consists of low-level operating system functions such as thread scheduling and interrupt / exception dispatching. The device drivers and hardware abstraction layer, which is a layer of code that isolates the kernel, will systematically require kernel functionality. From that, the windowing and graphics system will ultimately require kernel access as well.

With these ideas in mind, the Windows operating system has always been focused on compatibility with many different hardware architectures. The achievement of portability in Windows is accomplished through a layered design and a functionality with the programming language of C. First, the layered design of Windows considers the differences between potential architectures and hardware platforms. As the text mentions, “The lower-level portions of the system that are processor-architecture-specific or platform-specific are isolated into separate modules so that upper layers of the system can be shielded from the differences between architectures and among hardware platforms” (38). The two components that are tasked with providing portability are the kernel and the hardware abstraction layer. Functions that are specifically for architecture such as thread context switching are implemented in the kernel whereas functions that differ among systems within the same architecture are implemented in the HAL (38). In addition to this method of portability, Windows encompasses an extended ability because of the programming language used in its creation. Specifically, the clear majority of Windows is written in C whilst some portions are in C++. Because of this, greater portability to different architectures is realized in this specific operating system.

With respect to portability, multiprocessing was also a key concern and attraction for the Windows operating system. Since an original requirement for Windows was to run on multiprocessor computer systems, the operating system needed to encompass ways to accomplish efficient multiprocessing. From the text, it is clear the operating system and individual user threads can be run on any processor with proper access. In addition, it is the case that these mentioned processors share only one memory space. The contrast to ASMP (asymmetric multiprocessing) is shown in that ASMP typically designates one processor for kernel code and the rest for the user. This is a clear distinction that defines a more relaxed method of multiprocessing. Clearly, the Windows operating system is not considered in the designation of one process to kernel activity. This symmetric and asymmetric viewpoint accompany something very uniquely different in the choice that Windows has made for multiprocessing. Specifically, given a processor A and B, an asymmetric system would designate operating system tasks to processor A and user tasks to processor B. With a symmetric system, however, both operating system and user tasks can be implemented by both processors.

With the concept of symmetric multiprocessing in mind, Windows also supports other types of multiprocessor systems. Specifically, Windows supports multicore, Hyper-Threading enabled, and NUMA systems. Hyper-threading, mentioned in the text, “provides two logical processors for each physical core. Each logical processor has its own CPU state, but the execution engine and onboard cache are shared” (39). This enables a specific progression of the CPU considering some arbitrary mistake or such as cache miss or a bad branch prediction. The previously mentioned scheduling algorithms are used in Windows to increment the usefulness of the processing approach. In NUMA systems, the author defines, “processors are grouped in smaller units called nodes. Each node has its own processors and memory and is connected to the larger system through a cache-coherent interconnect bus” (39). The Windows system, naturally, could make use of more efficient thread scheduling requests on processors through this technique of same node allocation. Finally, the support of multicore systems is definitively purposeful to the Windows operating system because of its originally intended purpose of portability. As the author notes, “[since] these systems have real physical cores, the original SMP code in Windows treats them as discrete processors, except for certain accounting and identification tasks that distinguish between cores on the same processor and cores on different sockets” (39). With the combination of these various techniques of multiprocessing, the Windows operating system provides many different methods for efficiency and practicality in this discipline.

With the understanding of multiprocessing in the Windows operating system, it is also important to look at interrupt and exception handling as well. In Windows, interrupt handling is uniquely useful in both software and in hardware applications. With the interaction from devices to specific threads, waiting for a response is impractical – and that useful work must be completed in the interim. The Windows operating system must ascribe a way to handle process and hardware interrupts in a useful manner – since these specific interrupts can be of both natures. The kernel, at least in its present implementation, contains trap handlers tasked to react to the interrupts of devices. When a response is necessary, the kernel transfers control to an external handler, that is, a handler that will respond to this interrupt, or to a routine that is internal to the specific kernel. Any external service routines will ultimately be tasked with responsibly handling that specific interrupt if necessary (81). In between these two events of service and request, there are important steps to take careful note of. In hardware interrupt processing, for example, the use of an interrupt controller is necessary to interrupt the processor’s execution of the current process. Once execution is halted, the control will get the interrupt request. When translated, the corresponding output is that of an interrupt number which will ultimately be used to index into the interrupt dispatch table (82). When it is clarified, the designated elimination in that table will direct to an execution of the interrupt dispatch routine found in the structure. Once the interrupt dispatch routine is completed, the processor resumes execution of the halted process.

With the structure of interrupts in the Windows operating system, analyzing the priority system associated with its handling turns out to be useful. Specifically, the Windows operating system uses Software Interrupt Request Levels to prioritize a given interrupt throughout execution. Although these Interrupt Request Levels vary from different architectures, the representative IRQLs service to signify higher priority interrupts. On x86, for example, the IRQLs contain a list from 0 to 31 in terms of priority. The highest priorities of this list are shown in the ideas of power failure, interprocessor interrupts, clock interrupts, and device interrupts whereas lower priorities pertain to interrupts through software and normal thread execution. The operating system, naturally, handles these interrupts by their priority. If for example, a higher priority interrupt is issued while an IDR is in progress containing less priority, the new interrupt will ultimately be accommodated. When an interrupt is restored, the operating system will also load back its saved machine state if there are no other higher priority interrupts in place. Of course, each processor has its own settings for handling higher priority interrupts – and what constitutes a higher priority in the first place (86-7). For example, x64 and IA 64 only contains 16 priorities whereas the previously mentioned x86 contains 32.

In addition to hardware interrupts, the Windows operating system needs to also systematically handle software interrupts that may occur. In the Windows operating system, interrupt handling is necessary for initiating thread dispatching, non-time-critical interrupt processing, handling timer expiration, Asynchronously executing a procedure in the context of a particular thread, and supporting asynchronous I/O operations (104). Ultimately, the Windows operating system will need to consider IRQL standards for kernel structure access and, in addition, consider efficient context switching for software interrupts. On a thread’s completion or pause, the kernel will either immediately provide a context switch, or it will request a context switch and wait for certain circumstances to proceed. Activities, for example, that need to be completed will ultimately cause a deferment among context switching. In the previously mentioned statement regarding interrupt handling in software, the kernel will need to raise the processor’s IRQL to DPC/dispatch level or above if synchronization with shared kernel structures is required (104). If an interrupt is present with a higher priority than dispatch, the kernel will halt dispatching and allow the progress of this interrupt to take place. This practice, naturally, follows suit to the previous methods of hardware interrupt handling. If a system function is necessarily executed in kernel mode, a deferred procedure call will be issued -- initially with a less critical status than a current DPC issued. This concept can be used for rescheduling a thread if its period of quantum expires. In total, an expired timer may generate a DPC, releasing threads waiting on that specific timer. Once a software interrupt is generated, the thread waits for the IRQL to drop below DPC/dispatch level. Once this is completed, the thread will be extended a new quantum of time and will resume execution.

From interrupt handling, the very idea of what constitutes a representative process in the Windows operating system is also an important idea to understand. With every process, the Windows operating system designates a stream of characteristics and ways to represent it to the overall system. To start, the EPROCESS structure is used to represent each process in Windows. This structure is integral to representing the internal representation of a process – and its other related data structures. This is expanded upon in the purpose of interactive threads under that specific process. For example, this process object shows the Windows operating system some important information for saved state restoration and interrupt handling control. In that consideration, the EPROCESS block contains many necessary fields for interrupt handling specifically. These fields include the Process Control Block, Process ID, Parent process ID, Exit status, Active process link, Quota block, Process flags, Process counters, and many other attributes necessary for system interaction. As mentioned before, the EPROCESS block can also contain links to other EPROCESS blocks that may represent a separate thread linked to the parent process. It is important to note that the Windows operating system considers the process control block to be of type KPROCESS – referring, of course, to the kernel. In its entirety, the EPROCESS block is mostly restricted to kernel access. This is due to the strenuous need of the operating system to handle these processes and their threads in an appropriate manner (365 – 7).

In the context of interrupt handling, priority, and context saving within this operating system, the analyzation of the thread internals is vitally important in understanding the broader concept of process handling. In its entirety, the Windows operating system needs to deal with and properly analyze threads in an efficient and effective manner. To start, each thread that is present in the operating system is represented through an executive thread object. This object contains items like the thread control block, create and exit times, process ID, EPROCESS block, start address, etc. By its nature, the executive thread object helps maintain vital information about the thread needed for the operating system. It is seen that although this object requires many localized fields to properly function with the OS, it is still vitally necessary in terms of scheduling, priority, and access information. The interaction between processes, in other terms, will ultimately rely on this information, especially at the kernel access level. Naturally, this thread control block will need to point to the parent process, security information, impersonation information, and pending I/O requests. The overall object containing this thread information, as mentioned before, extends to a variety of purposes in the operating system. Specifically, we will need this object to schedule threads, enhance synchronization, and deal with timekeeping functions (393). Clearly, the need of the KTHREAD to store credible and useful information for system interaction is important for the overall functionality and performance of the operating system in its entirety. These localized fields in the KTHREAD also extend to scheduling factors such as total user time, kernel time, timer / waiting blocks, and the list of objects in which the thread is waiting on for further execution (392). With these fields in place, the system will make responsible decisions on thread scheduling or preemption when it needs to.

With the idea of an executive thread object, it suffices to also analyze the birth of a thread in the Windows operating system. With the idea of how complex a thread can be, Windows will need to provide many specific commands and actions to complete the birth of a thread. In total, however, this operating system needs to find and allocate the appropriate resources before a thread can be created. We start with the idea of an original thread request. If a program desires to spawn a thread into the system, it will be up to the process manager to allocate the necessary space for the thread to execute alongside a given process. After some time, the thread will be given certain localized resources to function and perform its tasks in the OS. Once the thread receives its designated resources needed for execution, the operating system will issue a variety of steps to begin execution. In total, the OS will initiate CreateThread and NtCreateThreadEx to simulate requests and responses for an arbitrary thread. The CreateThread command will primarily serve as a receiver of information necessary to create an executive thread object for the system to manage. It can, in addition, allocate an activation context for the thread, and notify the Windows OS about the new thread. (398). The NtCreateThreadEx command is used to create a user-mode context while then calling PspCreateThread to define a suspended executive thread object (398). Once all the necessary commands are issued, the thread handle and ID are returned to the calling request and the thread is ready for execution.

After the thread is ready for execution, the operating system must decide how to schedule a given process efficiently while considering the priority and other needs as well. Overall, the Windows OS will ultimately be a priority and preemptive scheduling system with thread scheduling. As analyzed previously, the thread execution in Windows will be derived from preemptive tendencies. This idea will ultimately cross into the idea of a quantum scheduling system. Because the system is preemptive, a given thread will have a period for execution – once this time is completed, the thread will be stopped from execution and the operating system will continue to the next. It is important to note that the quantum of time is very important to decide because of the overhead swapping can contain in a preemptive system. In this idea, the OS will give a fair amount of time to a given process without any discrimination in execution. If a process is preemptive, it will be put on the queue to eventually resume execution with a new time quantum when the thread’s turn arrives. In addition to preemptive scheduling, the Windows operating system must also consider many other reasons for preemption of a process. This can extend back to the idea of giving priority to a scheduled thread if needed. When a thread has a higher priority than one in execution, it takes its place ultimately causing a context switch and a new thread in execution. With this method, the highest priority threads will receive consideration by the OS and will quite often jump to the front to ensure appropriate execution. It should also be noted that a thread’s priority can change at any given time – and the operating system will accommodate. The final cause of preemption, quite obviously, would be the completion of a thread. In this, a thread’s completion symbolizes that the next available thread may start execution through the OS when available. In the culmination of these thread scheduling aspects, the Windows operating system provides a unique perspective on how to schedule threads given a time quantum, priority, and early preemption (408 – 9).

In the case of thread scheduling, the Windows operating system needs to also define a set of states to specify a thread’s current condition in execution. Ultimately, this specific OS provides a variety of ways to define a thread’s execution state both when it is being executed and when it is not in any execution by the operating system. To start there are eight different states of execution for threads in Windows. These states include ready, deferred ready, standby, running, waiting, transition, terminated, and initialized. Naturally, they are all very different and serve mild, differentiated purposes for the OS. The ready state, quite obviously, signals that a thread is ready to be executed. The deferred ready state, however, defines a process that is ready to run and has ultimately been selected but has not begun execution quite yet. These two states clearly define an important transition for the OS – most notably in the idea of a context switch for threads. Apart from that idea, the standby state defines a selected process that is waiting on the right conditions for execution. Once ready, the OS will preempt a thread and start this thread next. If a thread is Running, it would follow that a thread with that designated state is currently in execution. This is, of course, after a context switch has been made from the preempted thread to the new thread. Once a thread is done executing, it would then enter a Terminated state defining that there is no need for further execution. Although these are obvious, the waiting transition and initialized states are a bit more unique and complex. The Waiting state, for example, can occur if a thread is perhaps waiting on an IO operation to complete. Either way, this suspension designates a Waiting state for the operating system’s consideration. The transition state, on the other hand, occurs only when the needed information of the thread is paged out of memory while the initialized state just symbolizes the creation of a thread. With these specific states, the Windows operating system has created a complex way to analyze when a thread is ready for execution. In the ladder, the OS can understand each individual state of the thread and, ultimately, consider problems like paging and suspension (416-17).

Although the considerations of both process and thread scheduling are important, the idea of concurrency in the Windows operating system is, perhaps, the most important concept in its operation. With the problems that a lack of concurrency causes, Windows deals with the problems of mutual exclusion, deadlock, and starvation in unique ways. To start, the problem of mutual exclusion props up as an immediate concern because of memory access and modification. To mitigate this problem of mutual exclusion, the Windows OS has defined a way to bar other processes or threads from modifying a resource if it is designated as being used by another process or thread. To stop this potential problem, Windows needed to focus on High-IRQL Synchronization the most because of interrupts. With the access of global variables on the kernel access level, the Windows operating system needs to bar interrupted processes from modifying delicate resources if in use by another. This idea is implemented using spinlocks. If a vital resource is needed by a kernel-level implementation, it must acquire the spinlock associated with that resource. If the spinlock is not available, a kernel-level application will need to continue requesting the spinlock until it is received to access and modify a resource (178). It should be noted that these spinlocks are unique to IRQL and kernel-mode processes or threads. There are, however, spinlocks that operate differently for other instances. Specifically, the queued spinlock now maintains a running list of requesting processes or threads to access a resource. If the resource is blocked, the identifier associated with the requestor will be stored at the end of the queue. Once the preceding requestors are completed, the process or thread will acquire the spinlock and access the critical section. This ensures that a requester is served on a first-come-first-served basis much like the scheduling methods mentioned before. (180-81).

Although the spinlock provides a comprehensive way to ensure mutual exclusion, the method does not often suffice for non-kernel level processes or threads. Alternatives to the spinlock method include kernel dispatcher objects, fast mutexes and guarded mutexes, pushlocks, executive resources, condition variables, and Reader-Writer locks. Kernel dispatcher objects, for example, is a mechanism were the objects requiring synchronization will acquire that ability through the kernel dispatcher objects (184). Much like a provision for mutual exclusion, the kernel dispatcher objects provide an interactive way for coordination. Fast and Guarded Mutexes, on the other hand, help increase efficiency by serving as regular Mutexes while only going through the dispatcher (196). This increases efficiency and timeliness of access. Of the same nature, Pushlocks act on gate objects while also having the ability to be acquired in special modes such as shared or exclusive settings (199). Additionally, executive resources are derived from that same idea. Specifically, executive resources provide support for shared and exclusive access. Any thread waiting on an executive resource would ultimately need an event to continue (198). Another form of mutual exclusion, the condition variable, is defined as a Windows implementation for synchronizing threads relying on a specific conditional test. This implementation can be accessed by calling *InitializeConditionVariable* and *SleepConditionVariableCS* (202). Alternatively, Reader-Writer locks use the same concept of condition variables while using atomic operatings for release and acquire. This ensures that trips through kernel mode will not violate the problem of mutual exclusion (203). From those concepts of mutual exclusion, the given object type can implement these methods to define a signaled state. For example, the termination of processes and threads act alone as a signal to other processes and threads for future execution. The gate, semaphore, and mutex objects act specifically as signals for one waiting thread to take the critical place. A keyed event, on the other hand, helps release dependent threads or processes from the suspended waiting state and allows access to a potential critical section (186). When an object is signaled, any waiting threads are released even if a thread is not chosen next. Clearly, these concepts can be ascribed to both kernel-level and user-level modes for mutual exclusion and ultimately help maintain security for any critical memory accesses in a process or thread.

Although these ways to stop problems regarding mutual exclusion are effective, the problem of deadlock still exists. Currently, the Windows operating system uses the Driver Verifier to detect cases of problematic deadlock through the same methods that ensured mutual exclusion. Windows Internals Part 2 by Russinovich makes it clear that through the previously mentioned objects used for detecting a mutual exclusion violation, the problem of deadlock can be automatically detected by the Windows operating system through monitor problems used for concurrency. To start, a detected deadlock can occur when a device driver does not return from a routine, a high priority real-time thread causes system problems, or there is a kernel-level or user-level indefinite pause. Automatically, the operating system uses a feedback-based resource allocation to determine if the possibility of deadlock can exist. If a deadlock exists in reality or even potentially, the driver verifier will crash the system ultimately returning what caused the deadlock in the first place (577). The preemption of these processes allows the user to ultimately see the occurring problem and, hopefully, stop it from occurring again. With that said, the Windows operating system does not exclude the possibility of deadlock for all cases – it can still occur in many instances, but the preemptive crash is a useful tool.

In the understanding of deadlock, the Windows operating system also needs to account for starvation. Using its scheduling tactics, this specific OS guarantees that starvation will never occur because of a given priority in processes and threads. When a process is put in the waiting queue, the elapsed time quantum will serve as an integral component to eliminating the possibility of starvation. Specifically, each quantum of time could increment a processes priority if it has been waiting long enough. To avoid starvation, in other words, the Windows operating system may bump up the priority of the concerned process or thread to allow for future execution. This problem arises more so when the concept of higher priority is needed. When processes of a given priority need to execute, in other words, there may be a case were a lower priority process is never executed because of the constant influx of higher priority applications. Therefore bumping up the priority of a process allows the Windows operating system to eliminate all possibilities of starvation. Also, this ensures to the user a more stable environment for execution and reliability for non-kernel mode activity (62).

From the idea of concurrency, the Windows file systems are also critically important for analyzation. To start, Windows systems have included a variety of system formats across various platforms. Specifically, CDFS, UDF, FAT12, FAT16, FAT32, exFAT, and NTFS have all been implemented in a certain fashion previously in Windows. Though the CDFS file system is dated, there are some noticeable restrictions that should be observed. Specifically, the CDFS file system format was restricted in a variety of ways. To start, the maximum file size of this format was limited to a tiny 4 GB of memory. In addition, there was also a 65k directory limit as well. Since the CDFS system uses the Joliet format, however, there are no effective character length limitations (392). Regarding the UDF format, the restrictions of file systems were lifted somewhat to include longer file or directory names, availability for spare files, increase file sizes, and support for ACLs and ADSs. Although these two specified file systems are important to describe, the FAT12, FAT16, and FAT32 systems help extend the Windows operating system even further. As the author notes, the FAT file system served a variety of compatibility purposes with other multiboot systems. Because of limitations of the Windows operating system, the FAT12 format may contain a volume size of 32 MB while also having the ability to address 4,096 clusters. In contrast, the FAT16 format may contain a volume size of 4 GB and can address 65,536 clusters. The transition from FAT12 to FAT16 clearly signifies more significant advantages and fewer access restrictions to the overall system. Following that same pattern, the next format, FAT32, can address up to an astounding 16 TB volume size (393-96), a clear distinction from the FAT16 file system. The last FAT file system, exFAT, often called FAT 64 would, quite obviously, signify an exponential increase in capability as well. This specific file system increases the file size limit to an approximate 16 exabytes – ultimately increasing the maximum cluster size to 32 MB. Finally, the NTFS file system, often realized as the native file system to the Windows operating system, uses 64-bit cluster numbers – which yields an ability to address 16 exaclusters. The volume size, however, is only 256 TB. The NTFS also contains noticeable advantages in its implementation. Specifically, it contains file or directory security, ADSs, disk quotas, sparse files, file compression, soft links, and the ability for recovery.

In the instance of a need for recovery, the Windows operating system uses a variety of ways to prevent data loss. With the problem of hardware backup or corruption prevention, Windows uses three different setups to mitigate the problem. To start, Windows provides support for RAID-0. The system data, as the author outlines, “tends to be distributed evenly among the disks” through its concept and implementation (149). In addition, it is possible that disk loading and disk management can become much easier and efficient. There will, however, be a need for redundant storage schemes that contain mirrored volumes to fix file corruption, per the author (149). Since RAID-1 can restore data in a higher, more precise manner, the method is preferred over RAID-0 typically. It should be noted, however, that data loss can be found in a more extreme manner through the failure of RAID-1 prevention methods. Finally, the last recovery system used for Windows, RAID-5, can prove useful in the entirety of corruption deterrence. In contrast to other methods, this implementation’s useful approach to corruption prevention is a modern method for operating systems. Using three disks, the use of arithmetic operations will provide a general guide to fixing most problems tasked with the corrupted data. This provides an easy and efficient way to deter many problems in a short period of time.

With the culmination of the characteristics mentioned in the Windows operating system, the design and implementation of this specific operating system are clearly complex and considerate to the ascribed rules and problems facing modern day operating system requirements. Through the analyzation of its basic Concepts and tools, the fundamental building blocks necessary for operation were realized and attributed to the broader concepts of the Windows operating system. From the ideas of virtual memory, page swapping, and kernel / user mode, the concepts of architecture use, portability, and multiprocessing became clear. Further along, the analyzation of exception handling, priorities, and interrupts also proved useful to understanding the operation of Windows with respect to processes and threads. We also analyzed how the operating system effectively stored the necessary information needed for process or thread interaction through both the EPROCESS and KPROCESS blocks. From that, we analyzed the most important aspect of the Windows operating system. The problems related to concurrency were ultimately realized as effectively mitigated in the operating system through its use of complex and assistive techniques. These techniques of dealing with starvation, mutual exclusion, and deadlock were helpful to most arbitrary processes or threads that needed access to system resources. Finally, we analyzed the functionality of Windows file systems and data corruption prevention. These concepts were pivotal to the understanding of data allocation and retrieval. With these concepts in mind, the Windows operating system clearly brings together the most effective components for system operation into a single operating system. Because of this, the concepts of Windows are deterministic and highly useful in most regards. Considering its efficient structure and capabilities, the techniques implemented in Windows have been widely used throughout other critically important operating systems.

Works Cited

Russinovich, Mark E., Alex Ionescu, and David A. Solomon. *Windows Internals, Part 1*. Redmond,   
 Wash. : Farnham: Microsoft ; O'Reilly, 2012. Print.

Russinovich, Mark E., Alex Ionescu, and David A. Solomon. *Windows Internals, Part 2*. Redmond,

WA: Microsoft, 2012. Print.