Florida Atlantic University
Department of Computer & Electrical Engineering, & Computer Science
Computer Operating Systems – COP4610

An Architectural Evaluation of Windows and Linux

Submitted by:
Joshua Cidoni-Walker
jcidoniwalke2017@fau.edu

Friday, April 24, 2020

Table of Contents

Introduction

There are several computer operating systems available on the market today. Two popular systems are Windows and Linux. Each of these systems possess their own unique purpose, design, and implementation.

Microsoft's "Windows" (1.01) operating system was introduced on November 20th, 1985. It marked the company's first public attempt at incorporating a graphical user interface into its computer operating system, "MS-DOS". MS-DOS was a non-graphical, command-line interface; which was (and still is) not consumer friendly. The introduction of the graphical user interface appealed to many, and eventually led Microsoft's capitalization during the start of the "personal computing" era.

In September 17, 1991, Linus Torvald released Linux. Linux is an open source, computer operating system. Although contentious, Linux is believed to have been created as a free alternative to Unix, a then-popular, closed-source operating system. Linux is most popular amongst technologists due to its licensing, reliability, stability, security, and architecture.

Like any product, an operating system serves a purpose. The purpose it serves is driven by many factor; it is primarily driven by its audience. For example, an operating system used in an embedded system will be driven by different requirements than an operating system used in a personal computer.

The purpose of an operating system drives its architectural design. There are several common designs, such as monolithic, microkernel, and hybrid (combinational), all which offer their own unique advantages and disadvantages. For example, monolithic kernels—used in Linux—maintain a separation between system components and user components (applications); however, they are often tightly coupled. Microkernel's allow users to implement system-specific functionality; lacking (practically) any separation between the core of the system, and user. The hybrid design—used in Windows—combines the principles of both a monolithic design and a microkernel design.

An operating system's implementation follows the principles set forth by its design. Each design requires a unique implementation to satisfy its principles. A monolithic kernel requires interface mechanisms to interact with kernel (due to the supportive nature). A hybrid kernel also requires interface mechanisms to interact with the kernel, but also provides facilities to interact with system components in user space.

This paper will discuss the design and implementation of both Linux and Windows computer operating systems.

Windows

General Architecture

Windows is a hybrid (by design), multiuser multitasking operating system ("Structuring of the Windows Operating System"). That is, the bulk of the operating system and device drivers share kernel-mode protected memory space, multiple users are able to use the system at the same time, and the system facilitates mechanisms for process concurrency.

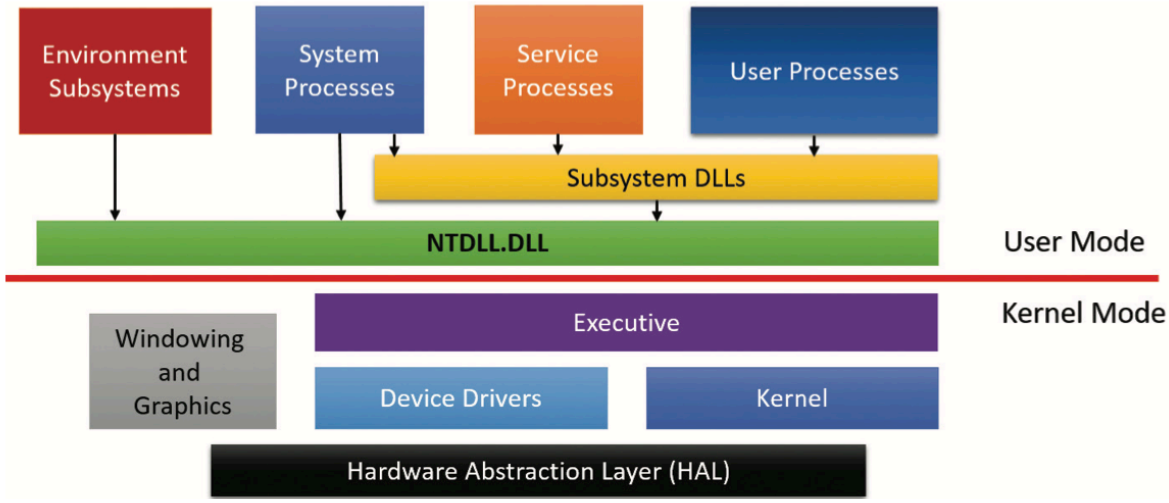


Figure 1-1 Yosifovich et al. Simplified Windows architecture. 2017

In **Figure 1-1**, first recognize the distinction made (by the red line) between user mode and kernel mode. The processes, subsystems, and libraries in user mode execute under a different set of privileges than those within kernel mode.

The four basic types of user mode processes are described below:

- **User Processes** These are processes that are not owned by the operating system.
- **Service Processes** These are processes that run independent of user logins.

- **System Processes** These are processes that are fixed, such as the session manager.
- **Environment Subsystems** These implement part of the support for the operating system environment; they allow non-native processes to execute.

The *Subsystem DLL* component translates documented functions into the appropriate internal (and undocumented) system calls.

The five components of kernel mode are described below:

- **Executive** This component contains the base operating system services, such as memory management, process and thread management, security, I/O, networking, and inter-process communication (IPC).
- **Kernel** This component contains functions pertaining to thread scheduling, interrupt and exception dispatching, and multiprocessor synchronization.
- **Device Drivers** This component contains both hardware device drivers, which handles the translation from user input/output to hardware-specific instructions), and non-hardware device drivers, such as file system and network drivers.
- **The Hardware Abstraction Layer (HAL)** This component abstracts the kernel and device drivers from the hardware (such as differences between motherboards).
- **Windowing and Graphics** This component handles the graphical user interface (GUI).

Processes

A process is a container for a set of resources needed to execute a program. A process has a private virtual address space, an executable program, a list of open handles to system object, a security context, a process identification number, and at least one thread of execution. Each process is started with a single thread, often called the primary thread, and can create additional threads from any of its threads.

Process Internals

The instance of a process is represented internally by an executive process object (**EPROCESS**). This structure holds, and points to data pertaining to the process.

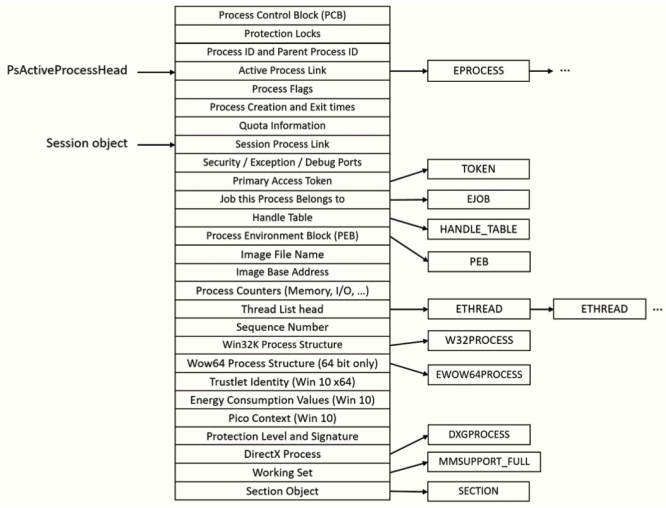


Figure 1-2 Yosifovich et al. Important fields of the executive process structure.

Threads

A thread is the entity within a process to which an operating system allocates processor time ("About Processes and Threads", 2018). All threads of a process share its virtual address space and system resources. A thread maintains priority, a unique identification code, stack space, and more ("About Processes and Threads", 2018).

Thread Priority

There are thirty-two (32) internal priority levels that the operating system uses (Yosifovich et al., 2017, pg. 215). They range in value from zero (0) to thirty-one (31). They are divided as follows:

- Sixteen real-time levels (16 through 31)
- Sixteen variable levels (0 through 15), with zero being reserved for a zero page thread.

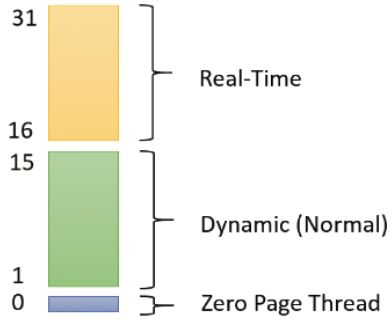


Figure 1-3 Yosifovich et al. Thread priority levels.

Thread States

There are eight distinct states a thread can be in (Yosifovich et al., 2017, pg. 223). They are described below:

- **Initialized** A thread is being created.
- **Ready** A thread is waiting to execute.
- **Deferred Ready** A thread has been selected to run on a specific processor, but has not yet started running there.
- **Standby** A thread has been selected to run next on a particular processor.
- **Running** A thread is currently being ran.
- **Waiting** A thread is waiting on a resource before it can run.

- **Transition** A thread is ready to run, but its kernel stack has been paged out of memory.
- **Terminated** A thread has finished executing and is awaiting deallocation.

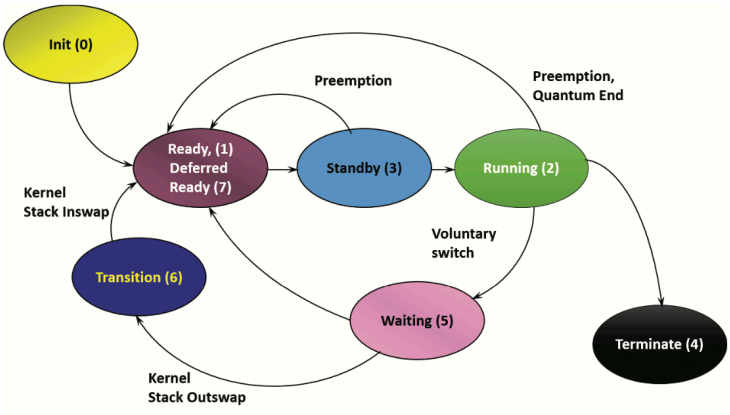


Figure 1-4 Yosifovich et al. Thread states and transitions

In **Figure 1-4**, the main state transitions for threads are shown. The numeric values shown represent the internal state values. Threads do not occupy one state for long periods, they quickly transition to ready, running, or waiting states.

Thread Scheduling

Threads are scheduled for execution based on their priority. This is due to the fact the operating system implements a priority-driven, preemptive scheduling system (Yosifovich et al., 2017, pg. 214). Generally, at least one of the highest-priority runnable threads always runs—with consideration for processor affinity (Yosifovich et al., 2017, pg. 214).

After a thread is selected to run, it runs for a period of time referred too as a time quantum (quantum for short). The time quantum is variable, and is most often determined by the type of process it is (e.g. a foreground or background process). It is possible that a thread does not execute for its full quantum due to the preemptive nature of the scheduling system—if a higher priority thread becomes runnable, it will interrupt the current execution and begin executing.

The code associated with thread scheduling exists in kernel space (kernel mode). There is no single scheduling module, rather scheduling code is spread throughout the kernel in places where scheduling-related events occur (Yosifovich et al., 2017, pg. 214). The most basic events are as follows:

- A thread becomes ready to execute
- A thread leaves the running state
- A thread's priority changes

At each juncture, a collective group of processes—known as the thread dispatcher—determines which thread should run next using the criteria discussed above (thread priority and processor affinity).

Linux

General Architecture

Linux is a monolithic, multiuser multitasking operating system (Wolfgang, 2008, pg. 3). That is, the bulk of the operating system and device drivers share kernel-mode protected memory space, multiple users are able to use the system at the same time, and the system facilitates mechanisms for process concurrency.

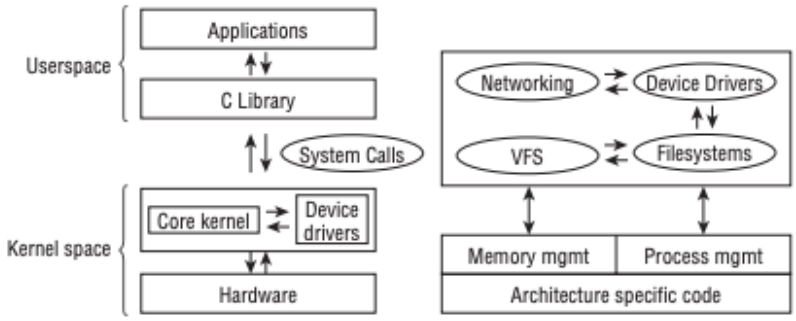


Figure 2-1 Wolfgang. High-level overview of the structure of the Linux kernel and the layers in a complete Linux system.

In **Figure 2-1**, the separation between user space and kernel space are made apparent by the labeled curly brackets. The processes within user space execute with different privileges than those of kernel space.

The types of processes and components comprising user space are described below:

- **Applications** These are processes that are not owned by the operating system.
- **C Library** This component implements standard C functionality.

As seen in **Figure 2-1**, processes in user mode interact with processes in kernel mode (and vice versa) through system calls.

The core kernel is comprised of processes relating to filesystems, networking management, process management, and memory management (Jones, 2007). The core kernel is allowed the privilege of interacting directly with hardware, which enables the interface between hardware and software.

Processes

A Linux *process* refers to a program in execution, and employs a hierarchal scheme in which it is dependent on its parent process. All processes are assigned address space in the virtual memory, rendering it effectively independent from any other process in the system. For a process to communicate with other processes, special kernel mechanisms must be used.

Process Internals

In Linux, processes are represented by a data structure named the *task_struct* (task structure). The task structure maintains a large number of attributes and elements that link the process to kernel subsystems. The basic components maintained are described below (Wolfgang, 2008, pg. 44):

- State and execution information
- Virtual memory information
- File handle information
- Thread information

In **Figure 2-2**, it is shown that many elements within the *task_struct* are pointers to other data structures such as *thread_info, mm_struct* (memory descriptor), *tty_struct* (teletype), and more.

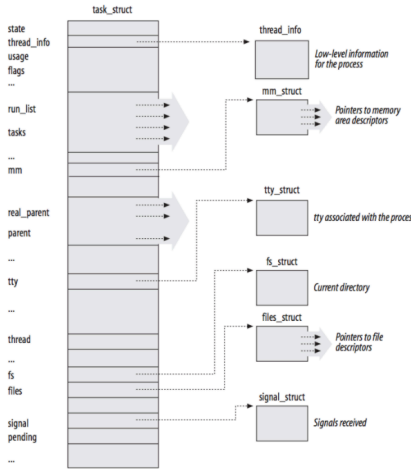


Figure 2-2 "Linux Process Management". Representation of a task_struct object.

Process Priority

Processes can be classified as either real-time processes, non-real-time processes, or normal processes (Wolfgang, 2008, pg. 36). They are described below:

- **Hard real-time processes** are subject to strict execution time constraints.
- **Soft real-time processes** require quick results, but can tolerate processor tardiness.
- **Normal processes** have no specific time constraint, but are still assigned priority.

Although normal processes require no specific time constraint, it is still necessary for them to possess priority. For example, consider the foreground process of a web browser; it is of higher priority that this process is more responsive rather than a background compiler process.

Linux maintains two values of priority for all processes, they are described below:

- **Priority** is a dynamic value that reflects the priority level seen by the kernel and can range in value from 0 to 139. The lower the value, the more likely a process will be scheduled next.

- **Niceness** is a static value that suggests how the kernel should prioritize the process and can range in value from -20 to +19. The lower the value, the more likely a process will be scheduled next.

Priority is a function of *niceness* such that $P(N) = 20 + N_i$. For all P(N) $0 \leq 30$, $P(N) = P(N) + 100$. Consider the following:

$$N_i = -20, \text{then } P(N) = 20 + (-20) = 0 \leq 30 \text{ therefore, } P(N) = 0 + 100 = 100$$

Therefore, actual *priority* is equal to 100.

Process States

A process in Linux can be in one of several states (at a time). They are described below:

- **Running** A process is executing and using the central processing unit (CPU).
- **Runnable** A process is ready to be executed by the CPU.
- **Sleeping** A process is waiting on a resource before it can be executed again.
- **Interruptible Sleep State** A process is waiting for an event to occur. The process will be removed from this state once the event occurs.
- **Uninterruptible Sleep State** A process will not respond to interrupts until a waited-upon resource becomes available, or until a time-out occurs.
- **Terminated/Stop State** A process has finished (either successfully or unsuccessfully).

Process Scheduling

General

Linux implements a Completely Fair Scheduler (CFS) (Jones, 2018). The goal of its scheduler is to maintain balanced processor time for each process. To determine balance, the scheduler records the time given to a process in a structure called the 'virtual runtime'. The smaller a process's virtual runtime, the greater its need for execution time.

Internals

The Completely Fair Scheduler (CFS) maintains a time-ordered, self-balancing red-black tree to build a timeline of process execution (Jones, 2018). By doing such, processes can be quickly selected, inserted, and deleted.

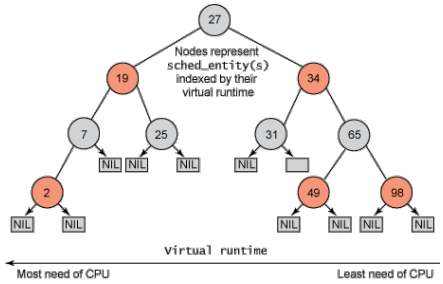


Figure 2-3 Jones. Example of a red-black tree.

Processes are represented by *sched_entity* objects within the tree. The processes with the greatest need for execution time are stored toward the left side of the tree, and processes with the least need are stored toward the right. To be fair, the scheduler removes the left most node of the tree and executes it. If needed (the process hasn't finished), the scheduler then re-calculates the virtual runtime, and re-inserts it into the tree.

Priority in Scheduling

Priority is used indirectly by the Completely Fair Scheduler; however, it is used as an aging factor for the time a process is permitted to execute (Jones, 2018). For example, the execution time of a low priority process will expire quicker than that of a higher priority process.

Evaluation

Windows

Limitations

There are limitations to the Windows operating system. A major limitation is that it is a closed-source, proprietary product. This presents as a limitation (in my opinion) for several reasons; the most important reasons are as follows:

- **Customization** Modifications nor expansions are permitted to the core operating system due to the closed-source nature, and the legal aspect of propriety software. Thus, users that require unique functionality from internal operating system components (kernel, subsystems, etc) are left at the mercy of the Microsoft Windows development team.
- **Privacy** By nature of closed-source products, users are unable to investigate the integrity of a product's source code. Thus, Microsoft should make an effort to provide more transparency in regards to user privacy.

Improvements

As mentioned above, due to the closed source nature of the Windows system, users entrust Microsoft to handle their information appropriately. This can be information such as photos, videos, documents, and more. Recently, Microsoft has been forceful in their efforts to maintain user information on the *cloud*. For example, users of Windows 10 are unable to create a local (offline) account during installation—if they are connected to the internet. Many users, who are not technically inclined, are often unaware that (by default) their

information will be stored outside of their local computing environment. These mechanisms of cloud management should be more transparent to the user.

Linux

Limitations

There are many use cases that require a system with the versatility that Linux provides; however, that does not mean Linux is not without its limitations. A critical limitation of the system is the lack of driver support.

Linux systems are (and historically have been) largely unsupported by hardware manufactures. As a result of Linux's market share percentages—and audience—manufactures do not feel that it is profitable enough for them to develop Linux drivers. For this reason, Linux lacks compatibility with many hardware devices.

Improvements

There are improvements to be made to Linux systems. A major improve that could be is:

- **Unified Package Manager** A unified package manager would ease a lot of confusion amongst users of Linux distributions. As it stands, there are many implementations of package managers; some are distribution dependent. For example, Debian manages .deb packages; whereas, RedHat manages .rpm packages. Thus, a different process must be ensued to install the same package on two different distributions.

References

About Processes and Threads. https://docs.microsoft.com/en-us/windows/win32/procthread/about-processes-and-threads

Linux Process Management. https://sites.google.com/site/bletchleypark2/3-operating-system/linux/linuxprocess

Structuring of the Windows Operating System. http://www.cs.sjtu.edu.cn/~kzhu/cs490/3/3_Win-Structuring.pdf

Babar Y. 2012, Understanding Linux Process States, 31 August 2012, https://access.redhat.com/sites/default/files/attachments/processstates_20120831.pdf

Jones M. 2007, Anatomy of the Linux kernel, IBM, 7 April 2020, https://developer.ibm.com/technologies/linux/articles/l-linux-kernel/

Jones M. 2018, Inside the Linux 2.6 Completely Fair Scheduler, IBM, 19 September 2018, https://developer.ibm.com/tutorials/l-completely-fair-scheduler/

Wolfgang M. 2008, *Professional Linux Kernel Architecture,* Wiley Publishing, Indianapolis.

Yosifovich P., Ionescu A., Russinovich M.E., Solomon D.A, 2017, *Windows Internals Part 1 System architecture, processes, threads, memory management, and more,* Microsoft Press, Redmond, Washington.

Presentation

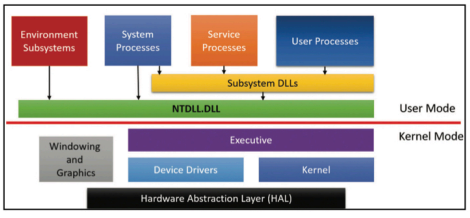
An Architectural Evaluation of Windows and Linux

Joshua Cidoni Walker - 12 April 2020

Windows Architecture

A brief overview of system architecture

- A monolithic kernel.
- A multi-user, multi-tasking computer operating system.
- Enforces the separation of user space and kernel space.



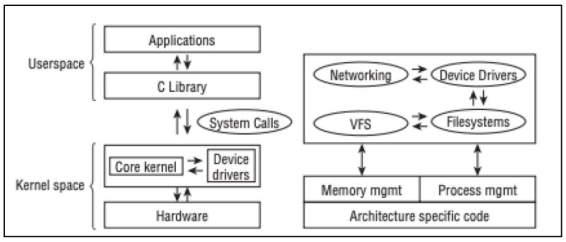
Simplified Windows Architecture

Image taken from "Windows Internals Part 1, System architecture, processes, threads, memory management, and more". By Yosifovich P., Ionescu A., Russinovich M., Solomon D.

Linux Architecture

A brief overview of system architecture

- A monolithic kernel.
- A multi-user, multi-tasking computer operating system.
- Enforces the separation of user space and kernel space.



Simplified Linux Architecture

Image taken from "Professional Linux Kernel Architecture", by Wolfgang Mauerer

Processes in Windows

A brief overview of process structure

- Processes are seen as **sets of resources**—or containers—for programs to execute.
- A process is maintained by a parent data structure known as the EPROCESS.
- Processes hold information (within—or pointed to by—the EPROCESS structure) such as private virtual address space, handles to open system objects, security contexts, the program, identification numbers, and at least one thread of execution.

Threads in Windows

A brief overview of threads

- Threads are entities (within a process) that are eligible for scheduling and execution.
- Threads can take on different priorities ranging in values from 0 to 31.
- Threads can take on different states such as initialized, ready, deferred ready, standby, running, waiting, transition, and terminated.

Thread Scheduling in Windows

A brief overview of thread scheduling

- A priority-driven, preemptive scheduling algorithm.
- A collective group of processes—the thread dispatcher—determines which thread will execute next.
- The scheduling code is not modularly contained; rather, it is spread throughout the kernel in places where scheduling-related events occur.

Processes in Linux

A brief overview of processes

- A process is a program in execution; a schedulable entity.
- A process can take on different priorities (both in user space and kernel space).
- A kernel can take on different states such as running, runnable, sleeping, interruptible sleep state, uninterruptible sleep state, and terminated.
- A process is maintained by a data structure known as the task_struct, which holds data (within—or pointed to by—the task_struct) pertaining to state and execution, virtual memory, file handles, and threading.

Process Scheduling in Linux

A brief overview of process scheduling

- A Completely Fair Scheduler (CFS)
- Priority has an indirect effect on scheduling, it is used as an aging factor for execution time.
- A self-balancing, time ordered red-black tree is maintained to determine the next process to execute.

Limitations of Windows

A brief discussion on limitations

- Not customizable because it is proprietary, closed source software.
- Dependent upon a single entity which creates a concern for privacy.

Limitations of Linux

A brief discussion on limitations

- Lack of executable programs
- Lack of hardware driver support

Improvements to Windows

A brief discussion on improvements

- Increase the transparency regarding the use of cloud-storage; for example, do not force users to create an “online” account during system installation.

Improvements to Linux

A brief discussion on improvements

- A greater push for manufacture hardware driver support.