Overview of Windows Internals

Windows System Architecture

OneCore Kernel (Shared platform)

OneCore is Microsoft's foundational base for its Operating Systems, designed to unify the underlying infrastructure across multiple devices. This means that devices (PCs, Xbox One, Series, HoloLens...) share the same underlying platform. The kernel, drivers, and base platform binaries are the same.

Windows API (Windows Application Programming Interface)

Windows API is the user-mode system programming interface for the Windows OS family. This provides a way for software developers to write programs that can perform tasks such as accessing files, managing memory, handling I/O operation, interacting with hardware devices, etc. The interfaces abstract the complexities of interacting with the underlying hardware and system resources.

Services, functions, and routines

- **Windows API functions**: These are documented, callable subroutines in the Windows API. Examples: CreateProcess, CreateFile, GetMessage, SendMessage, MessageBox and CreateWindow.
- **System Calls:** These are services in the OS that are callable from user mode. Examples:
  - **NtCreateUserProcess:** Is the lowest-level API responsible for creating a new process.
  - **NtCreateFile:** Is an internal service used by the Windows kernel to create or open files.
- **Kernel support functions (or routines):** These subroutines can be called only from kernel mode.

- **Windows services:** The windows services are processes initiated by the Windows service control manager.
- **Dynamic link libraries (DLLs):** These are collections of callable routines bundled together as a single binary file. DLLs are widely used by user-mode applications in Windows. One key advantage of DLLs compared to static libraries is that they allow applications to share code, and Windows ensures that only one copy of a DLL's code is loaded into memory, regardless of how many applications are using it. They essentially convert documented functions into the necessary internal system calls.

Virtual Memory

Virtual Memory is a technique used by OS to manage memory efficiently and allow programs to believe they have access to more memory than is physically available in the system. Its primary function is to provide each process with an "illusion" of a large and private memory address space, called a virtual address space. This enables programs to operate as if they have access to a large amount of memory, even if the actual physical memory in the system is limited.

In the Windows OS, virtual memory works as follows:

- **Virtual Address Spaces:** Each process has its own virtual address space, which is a set of addresses it can use.
- **Address Translation:** Is the translation of the virtual addresses into physical addresses.
- **Paging:** Since physical memory is limited, the memory manager moves parts of memory that are not actively being used to a storage device. This is known as paging and frees up physical memory for other processes or for the OS.
- **Protection and Mapping:** The OS controls the protection and mapping of virtual addresses to ensure that individual processes do not interfere with each other or overwrite OS data.
- **Data Relocation:** When a program accesses a virtual address that has been moved to disk, the virtual memory manager reloads that information back into memory from disk as needed.

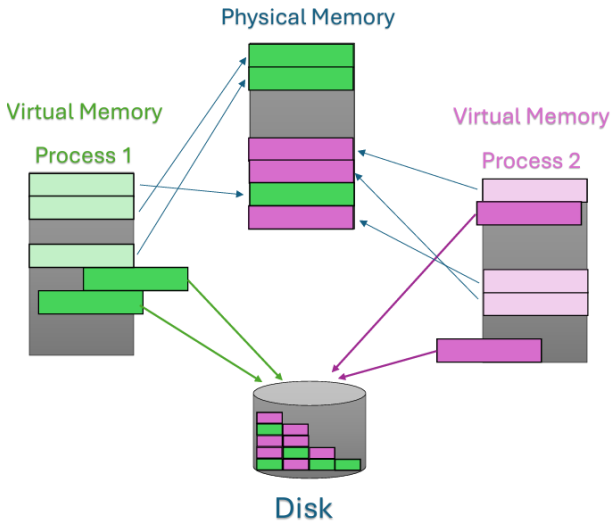

Figure 1: Mapping virtual memory to physical memory with paging

Based on 'Windows Internals Part 1' by Pavel Yosifovich, Alex Ionescu, Mark E. Russinovich, and David A. Solomon.[1]

Kernel Mode vs User Mode

User applications code operates in user mode, whereas OS code such as device drivers functions in kernel mode. Kernel mode denotes an execution state in a processor that provides access to all system memory and CPU instructions.

Processor access modes are established to ensure the security and stability of the OS. User mode restricts applications access to critical parts of the system, such as kernel memory and CPU instructions, preventing unauthorized modifications or actions that could compromise system operation. Conversely, kernel mode grants full access to all system resources and allows for the execution of OS services and device drivers, thereby ensuring smooth and secure operation of the system as a whole.

The figure 2 illustrates the communication between kernel mode and user mode and some examples:

- **User-Mode Drivers:** The printer driver facilitates printing from applications. In this case, it would facilitate printing from the Microsoft Word application.
- **Kernel-Mode Drivers:** The ntoskml.exe located in the Windows Kernel, operates at the lowest level of the OS and manages tasks such as memory management and communication with hardware.
- **File System Driver:** For example the NTFS.sys. These drivers manage access to file systems (NTFS, FAT32, etc.)

- **Exported Driver Support Routines:** IoCreateDevice is a support routine used by drivers to create device objects in the Windows Kernel. A device object is an abstract representation of a physical or logical device in the system. It is associated with a driver and is used to manage communication between software and hardware.
- **Hardware:** The processor is a fundamental component of a computer's hardware. The processor interprets and executes instructions from programs and applications running on the computer, following the sequence of instructions contained within a program.
- **Windows API:** CreateFile is a function of the Windows API used for creating or opening files in Windows. It enables applications to interact with fuels and storage devices.
- **Hardware Abstraction Layer (HAL):** Is a component within the Windows OS that acts as a barrier between the kernel, device drivers, and other parts of the Windows executive, and the specific hardware of the computer. Its primary function is to abstract away the hardware differences, ensuring that the OS can interact with various hardware configurations.

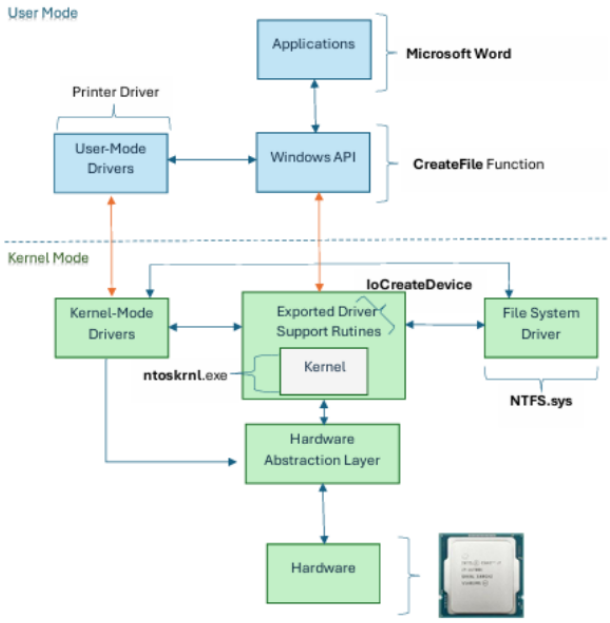

Figure 2: Communication between kernel mode and user mode

Based on the content from the official Microsoft Learn page:'User mode and kernel mode'.[2]

The kernel-mode operating system and device-driver code share a unified virtual address space. Each page within the virtual memory is labeled to specify the processor's required access mode for reading or writing the page. Pages situated in the system space are exclusively accessible from kernel mode, whereas all pages within the user address space are accessible from both user mode and kernel mode. Once in kernel mode, the OS and device-driver code has complete access to system-space memory and can circumvent Windows security measures to reach objects. Given that the majority of Windows OS code operates in kernel mode, it becomes imperative for components executing in kernel mode to undergo meticulous design and testing, thereby ensuring they refrain from breaching system security protocols or inducing system instability.

In Windows 10, all newly developed Windows 10 drivers must bear signatures from only two of the recognized certification authorities, utilizing a SHA-2 Extended Validation (EV) Hardware certificate. Once a driver obtains an EV signature, it must undergo submission to Microsoft via the SystemDevice (SysDev) portal for attestation signing, resulting in the driver receiving a signature from Microsoft. Consequently, the kernel will solely recognize drivers for Windows 10 that carry signatures issued by Microsoft, except for the Test Mode.

Registry

The Windows registry is a crucial part of the OS. It acts as a central database storing information necessary for booting up the system, configuring system-wide settings, managing security, and storing individual user preferences. It also holds real-time data about the current hardware status, like which device drivers are in use and their allocated resources.

Scheduling

Scheduling within the Windows OS is a fundamental function that manages multitasking by determining which threads, among those competing, are allocated the next slice of processor time. This allocation is based on priorities assigned through scheduling.

In a preemptive scheduling model, utilized by Windows, the OS has the capability to interrupt the execution of a running process, pausing its operation momentarily to allocate CPU resources to another process.

- **Scheduling priorities:** Each thread is designated a scheduling priority, graded on a scale from zero (representing the lowest priority) to 31 (the highest). Solely the zero-page thread may possess a priority of zero, responsible for zeroing free pages when no other threads necessitate execution.
- **Context Switches:** This occurs when the kernel transitions the processor from one thread to another.

- **Priority Boosts:** Threads awaiting execution subsequent to user events, I/O operations, or the allocation of executive resources are afforded a priority boost by the system.
- **Priority Inversion:** Temporarily, the system elevates the priority of a low-priority thread when it holds a synchronization object essential to a high-priority thread.
- **Multiple Processors:** The scheduler determines which thread runs on which processor.
- **Thread Ordering Service:** This ensures that high-priority threads execute without being preempted by lower-priority ones.

Driver Architecture

What Is a Driver?

The purpose of a driver is to manage communication between applications and a device. Since the Windows kernel itself cannot directly interact with devices, it relies on drivers to detect and interact with them.

Drivers possess the following characteristics:

- They run in the background.
- They have the same lifespan as their devices. This means that the driver starts when Windows detects the device and shuts down when the device is removed.
- They respond to I/O requests.
- They can communicate directly with components in kernel mode. Even if it is a UMDF (USer-Mode Driver Framework), it still needs to communicate with kernel-mode drivers to transfer data to or from a device.

How do drivers fit into the Windows Os?

We can observe in the Figure 3, there is a dividing line that segments two modes, called User Mode and Kernel Mode. The applications we use are running in User Mode, while the core of the OS is running in Kernel Mode. Programs running in User Mode have a set of restrictions that forbids them to access certain parts of the system, to prevent compromising parts of the OS.

In the case of programs running in Kernel Mode, they are imposed minimal restrictions and basically have access to the kernel. Applications in User Mode can only communicate with the kernel through the Windows API. However, for drivers in Kernel Mode,

they can communicate directly with applications in User Mode. Nevertheless, this would compromise security and pose a risk. So, generally, what happens is that Kernel Mode drivers, when they need to communicate or send data to applications, do so through the **Kernel Subsystems**.

On the User Mode side, we have **Applications** and also the **Windows API**. Applications typically make I/O requests. Since they are in User Mode, they need the functions provided by the Windows API to make these I/O requests. These functions pass the request to the kernel mode, which relays it to the corresponding component.

Kernel Subsystems are responsible for managing a significant portion of the core functionality of Windows. They also offer routines that drivers can use to interact with the system.

Drivers and devices: Drivers have an interface between the device and the Kernel Subsystems. This interface facilitates communications and interaction between the device's hardware and the OS software. In the case of I/O, the Kernel Subsystems pass the request to the drivers to process the request and communicate it to the device. Similarly, when the device sends data, the driver retrieves it and passes it to the Kernel Subsystem.

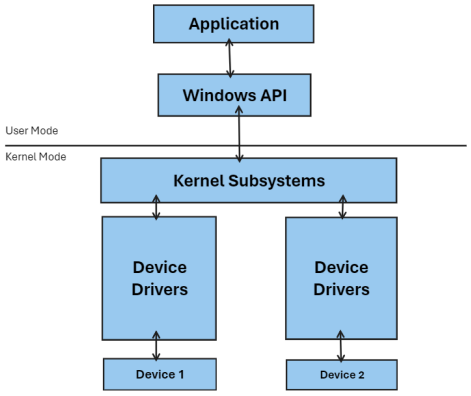

Figure 3: Simplified diagram of the Windows architecture

Based on 'Developing Drivers with the Windows Driver Foundation' by Penny Orwick and Guy Smith.[3]

Driver Architecture

The driver architecture in Windows include:

- **Modular Layered Architecture:** This means that a device can be serviced by

several layers of drivers, each responsible for specific aspects. This layered architecture is organized through a stack.

- **Packet-based I/O:** This means that requests are handled within packets, and these packets are passed back and forth within the stack until they reach the driver that can handle them.

- **Asynchronous I/O:** Drivers can handle requests without waiting. For instance, if a driver is receiving a read request from an application that takes a long time to process, the driver initializes the necessary operations to inform the I/O manager that the request is pending. The driver is then free to process additional requests. When the read operation is completed, the device notifies the driver, which informs the I/O manager, allowing the application to process the read data.

  The advantage of asynchronous I/O is that it improves performance in multithreaded applications. Threads don't have to wait idly for a request to complete; they can perform other tasks while waiting.

- **Dynamic Loading and Unloading:** The lifespan of a driver depends on the lifespan of the device. Therefore, drivers are not loaded until the device arrives, and similarly, when the device is removed, the drivers are unloaded.

Device Objects and the Device Stack

When the kernel Subsystem sends an I/O request, one or more drivers process this request. Since it's not just one driver processing it, this is where the Device Object (DO), comes into play. Each driver is associated with one DO.

A Device Object is a data structure containing pointers to functions within the driver that handle or manage device-related requests. This Device Object acts as a participation enabler, letting the system know it's participating in the request.

These Device Objects are organized in the Device Stack. Each device has its own separate stack. This stack contains these Device Objects plus the associated drivers; however, drivers may be utilized by various devices stacks

How a Device Stack is built?

These three types of device objects work together to enable the system to process I/O requests.

- **Bus Driver and Physical Device Object:** At the bottom of the stack, we find the Physical Device Object. This is linked to a Bus Driver. Devices are typically connected to hardware through a bus like PCI or USB. The Bus Driver's role is to manage these hardware pieces connected to the physical bus.

- **Function Driver and Functional Device Object:** The Function driver translates the Windows abstraction of a device into commands required to transfer data to a real device. It provides an interface called the "upper edge", which applications and services use to interact. It also controls how the device responds to Plug and Play changes. For the "lower edge", it handles communication with the device or other drivers.

- **Filter Drivers and Filter Device Objects:** These filters are optional. The general purpose of a filter driver is to modify some I/O requests to add value, depending on the vendor. For example, drivers can encrypt or decrypt read or write requests. The filter device object (FDO) is associated with a filter device. Multiple filter device objects can exist within the device stacks, either before or after the functional device object. For instance, in the illustration, we see a device object above the FDO and another DO below the FDO.

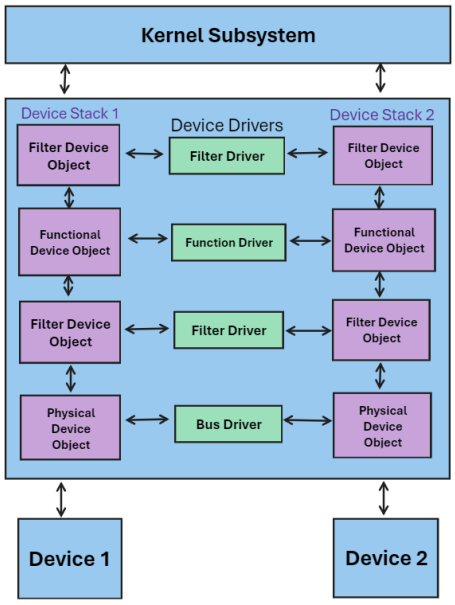

Figure 4: The device stack

Based on 'Developing Drivers with the Windows Driver Foundation' by Penny Orwick and Guy Smith.[3]

The Windows I/O Model

I/O Request

The most common clients of drivers are user-mode applications. The most common types of I/O request are:

- **Write requests:** These requests pass data to the driver to be written to the device.
- **Read requests:** These requests pass a buffer to the driver, which gets filled with data from the device.
- **Device I/O control requests:** These are requests passed to the driver for purposes other than reading or writing.

How a Device Stack Handles IRPs

When a client sends an I/O request, the I/O Manager packs the request into an IRP (I/O Request Packet). Then, the I/O Manager locates the pointer to the appropriate routine that will handle the request at the top of the Device Object within the stack, then calls the routine and passes the IRP to the associated driver for processing.

If the driver at the top manages to fulfill the IRP request, it then informs the I/O Manager that the request has been completed. The I/O Manager returns the IRP to the client along with the requested data.

However, there are cases where the top-level driver cannot process the request alone. In such cases, the driver processes the IRP request and returns it to the I/O Manager, which then passes it to the next lower level of the device stack. Subsequently, the next driver does what it can, and then the IRP is passed down to lower levels until eventually reaching a driver that can fully process it. In the scenario where no driver can process the request(IRP), what would happen is that an error code would be returned indicating that the request state was not processed or supported.

Kernel-Mode Programming

Programming in Kernel Mode requires us to be very careful. Here we can notice a significant difference between User Mode and Kernel Mode. For example, in User Mode, when there is a page error, such as a program attempting to access a memory address that is not in the memory, User Mode applications are not affected. There might be a slight delay while the pages are re-read into memory. However, in the case of Kernel Mode, if a page error occurs, the system crashes.

Interrupts

An interrupt is something that falls outside the realm of normal processing and must be serviced as quickly as possible.

When a hardware event occurs, an interrupt is generated within the OS. Windows delivers the interrupt associated with that event to the processor. Once the interrupt has been delivered to the processor, Windows briefly interrupts an existing thread to handle the event. Subsequently, once it has finished handling this task, the system returns the thread to its original state to continue what it was doing.

IRQL

An IRQL is a level associated to each interrupt. The System uses the IRQL values to ensure that the most important interrupts are handled first. The highest IRQL is prioritized.

When a processor receives an interrupt, the following sequence of events occurs:

- If an interrupt's IRQL is greater than that of the processor, the system elevates the processor's IRQL to match the interrupt's level. Consequently, the ongoing code execution on that processor halts and remains paused until the service routine concludes and the processor's IRQL returns to its initial state.

  The service routine may also be interrupted by another service routine possessing an even higher IRQL.

- When an interrupt's IRQL matches the processor's IRQL, the service routine must wait until any previous routines with the same IRQL have completed.

- If an interrupt's IRQL is lower than the processor's current IRQL, the service routine must wait until all interrupts with a highest IRQL have been processed.

Only specific IRQLs are allocated for use by drivers, the rest are designated for system operations. Below are the commonly used IRQLs by drivers, arranged in ascending order from the lowest value:

- **PASSIVE_LEVEL:** Is the default IRQL for regular thread processing and is not associated with an interrupt. User-Mode applications and low-priority drivers run at this level.

- **DISPATCH_LEVEL:** Represents the highest IRQL linked to a software interrupt. Driver routines with higher priority operate at this level.

- **DIRQL:** Is the highest IRQL that drivers commonly deal with. This refers specifically to the set of IRQLs associated with hardware interrupts.

Memory

Kernel-mode and user-mode drivers have a virtual address space mapped to physical addresses by the system.

- Kernel-mode components share a virtual address space. This means that the drivers on kernel-mode can corrupt each other's memory.
- Each process of User-mode has its own virtual address space.
- User-mode processes cannot access Kernel-mode addresses. But, Kernel-mode processes can access user-mode addresses. We need to be careful if we want to access a user-mode address.

Memory Management

A driver is responsible for the following tasks:

- **Allocating and freeing memory:** This involves managing memory resources efficiently, ensuring that memory is allocated when needed and released when no longer required.
- **Using stack memory appropriately:** The driver must utilize the stack memory efficiently, taking care not to overflow the stack or cause memory corruption.
- **Handling faults:** An invalid access fault causes a bug check and crashes the system. The impact of a page fault depends on IRQL. The driver should gracefully handle them.

References

[1] P. Yosifovich, A. Ionescu, M. E. Russinovich, and D. A. Solomon, *Windows Internals Part 1.* Redmon, Washington: Microsoft Press, 2017.

[2] Microsoft. (2024) User mode and kernel mode. [Online]. Available: https://learn.microsoft.com/es-es/windows-hardware/drivers/gettingstarted/user-mode-and-kernel-mode

[3] P. Orwick and G. Smith, *Developing Drivers with the Windows® Driver Foundation.* Redmon, Washington: Microsoft Press, 2007.