Chapter 2: System Architecture

Operating System Model

Monolithic OS – most of the OS and device driver code shares the same kernel-mode protected memory space

Kernel-mode components embody object-oriented design principles – use formal interfaces to pass parameters, access/modify data structures

Most kernel-mode OS code written in C (for portability)

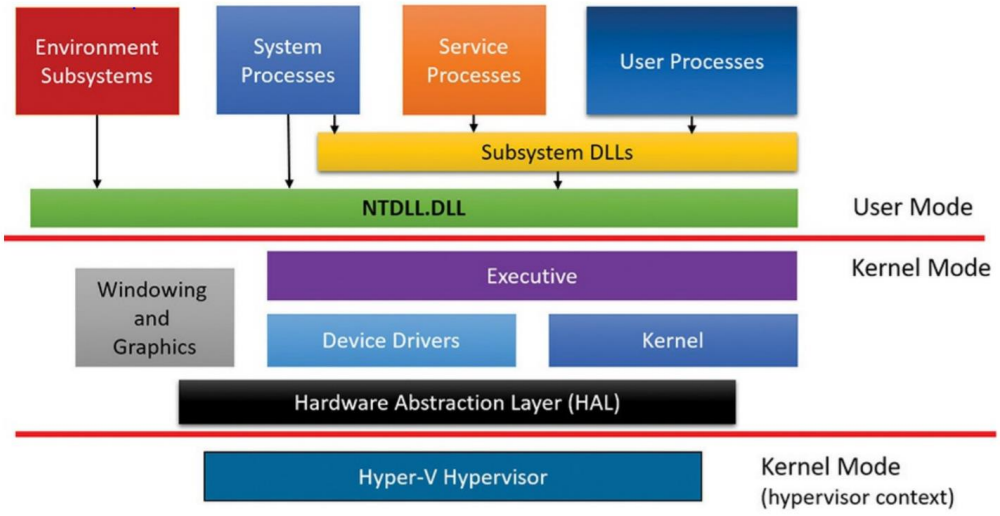


FIGURE 2-1 Simplified Windows architecture.

Architecture Overview

User-mode threads executing in kernel mode have access to system space

Hypervisor technically runs at CPU privilege level (0) but uses specialized CPU instructions to isolate from the kernel

Types of user-mode processes:

1. **User process**
   Windows 32 or 64, Windows 3.1 (16 bit), MS-DOS (16 bit), POSIX (32 or 64)
   Can only run 16-bit apps on 32-bit Windows, and POSIX apps not supported as of Win 8
2. **Service process**
   Processes that host Windows services (i.e. Task Scheduler, Print Spooler)
   Services (generally) run independent of user logons
3. **System process**
   Fixed processes (i.e. logon process, Session Manager) that are not Windows services (i.e. not started by Service Control Manager)
4. **Environment subsystem server process**
   Support for the OS environment (ex. POSIX, Subsystem for UNIX based Applications)
   Windows Subsystem for Linux (WSL) is not truly a subsystem

User apps do not call native Windows OS services directly.

They use subsystem DLLs, which translate a documented function into internal native system service calls (native calls mostly implemented in Ntdll.dll)

Kernel-mode components:

1. Executive
   Base OS services (i.e. memory, thread, process management, security, I/O, networking, inter-proc comms)
2. Windows Kernel
   Low-level OS functions (i.e. thread scheduling, interrupt & exception dispatch, multiproc synchronization)
3. Device Drivers
   Hardware device drivers: translate user I/O function calls into hardware device I/O requests
   Non-Hardware device drivers: file system, network drivers
4. Hardware Abstraction Layer (HAL)
   Isolates the kernel, device drivers, and Windows executive from platform-specific hardware differences (such as differences between motherboards)
5. Windows and graphics system
   Implements GUI functions (Windows USER, GDI)
6. Hypervisor Layer
   Composed of only the hypervisor
   No drivers or modules here (although hypervisor composed of internal layers,services)

Core Windows OS components

File Name	**Components**
Ntoskrnl.exe	Executive, Kernel
Hal.dll	HAL
Win32k.sys	Kernel-mode part of the Windows subsystem (GUI)
Hvix64.exe (Intel), Hvax64.exe (AMD)	Hypervisor
.sys files \SystemRoot\System32\Drivers	Core driver files (ex. Direct X, Volume Manager, TCP/IP, TPM, ACPI)
Ntdll.dll	Internal support functions, system service dispatch stubs to executive functions
Kernel32.dll, Advapi32.dll, User32.dll, Gdi32.dll	Core Windows subsystem DLLs

Portability

Layered Design: low-level portions specific to processor-architecture or platform are isolated into separate modules to shield upper layers from differences between arch and hardware platforms

Kernel (contained in Ntoskrnl.exe) and HAL(Hal.dll) – provide OS portability

Architecture-specific functions (i.e. thread context switching, trap dispatching) implemented in the kernel

Hardware-specific functions (functions that differ among systems within the same architecture, such as motherboard) implemented in HAL

Memory manager also has architecture-specific code

C Programming: most of Windows written in C (some in C++)

Assembly for OS components that communicate directly with system hardware (ex. interrupt trap handler) and for performance-sensitive function (ex. context switching)

Symmetric multiprocessing

Multitasking – single processor with multiple threads

Multiprocessing – multiple processors with multiple threads

Symmetric multiprocessing (SMP) OS – no master processor – OS and user threads can be scheduled to run on any proc

Asymmetric multiprocessing (ASMP) – OS selects one proc to execute kernel code, all other procs run user code

Multiprocessor systems:

Multicore

SMP code in Windows treat each core as discrete processors

Simultaneous multi-threaded (SMT)

Each logical proc has its own CPU state but share execution engine and onboard cache

"Four cores, eight threads" means up to eight threads can be scheduled → 8 logical processors

Heterogeneous

ARM versions of Windows support heterogenous multi-processing (implementation called big.LITTLE)

Not all proc cores identical in capabilitiesbut are still able to execute the same instructions

Non-uniform memory access (NUMA)

Processors grouped into smaller units called nodes

Each node has its own proc and memory and connected to larger system through cache-coherent interconnect bus

System attempts to schedule threads on procs that are in the same node as the memory being used

Windows keeps track of processors (total number, idle, busy) in bitmask (aka affinity mask) that is the size of a native data type on the machine (32-bit or 64-bit)

Processor group implemented because Windows originally limited to the number of CPUs in native word

Set of processors defined by a single affinity bitmask

Kernel and applications can choose group during affinity updates

Number of supported licensed processors depends on Windows edition, stored in system license policy file

Scalability

Deal with resource contention, performance issues

- Ability to run OS code on any available proc
- Multithreads of execution within single proc (each thread can execute simultaneously on different processors)
- Fine-grained synchronization within kernel (spinlocks, queued spinlocks, pushlocks)
- Programming mechanisms that facilitate implementation of scalable multithreaded server procs

Differences between Windows Editions

Number of total logical proc supported

Number of hyper-V containers allowed to run (server systems only)

Amount of physical memory supported (highest physical address usable for RAM)

Number of concurrent network connections supported

Support for multi-touch and Desktop Composition

Support for BitLocker, VHD booting, AppLocker, Hyper-V

Layered services (Windows Server editions) Host Guardian, Storage Spaces Direct, clustering

System knows which edition is booted by querying registry values in HKLM\SYSTEM\CurrentControlSet\Control\ProductOption key

ProductType distinguishes client system vs server system

ProductPolicy contains cached copy of tokens.dat file which differentiates between editions of Windows and features they enable

Server systems optimized for system throughput, client versions optimized for response time in interactive desktop use

Run-time policy decisions (memory manager trade offs) differ between client and server editions

Virtualization-based security architecture overview

Virtual Trust Levels (VTLs) extend processor's privilege-based separation policy

Regular kernel and drivers run in VTL 0 – cannot touch VTL 1 resources

When VBS enabled, VTL 1 contains its own secure kernel running in privileged processor mode (ring 0), and Isolated User Mode (IUM) runs unprivileged code (ring 3)

Secure kernel is its own separate binary (securekernel.exe)

IUM restricts allowed system calls from user-mode DLLs (limits the DLLs that can be loaded), and adds special system calls that can only execute under VTL 1

VTL 1 sys calls exposed in Iumdll.dll (VTL 1 version of Ntdll.dll) and use Windows subsystem-facing library called Iumbase.dll (VTL 1 version of Kernelbase.dll)

IUM share most standard Win32 API libraries

Copy-on-write mechanisms prevent VTL 0 apps from making changes to binaries used by VTL 1

Kernel-mode code at VTL 0 cannot touch user mode code running VTL 1 (because privilege), but VTL 1 user-mode code cannot touch kernel mode code running at VTL 0 (because separation of ring 3 and ring 0)

VTL 1 user-mode apps must still go through reg Windows system calls to access resources

Privilege levels (user vs kernel) enforce power

VTLs enforce isolation

VTL 1 user-mode app is NOT more powerful than VTL 0 app or driver

(They are in fact less powerful since secure kernel (aka proxy kernel) does not implement full range of system capabilities)

Prohibited from I/O and Graphics and may not communicate with drivers

Secure kernel (running at VTL 1) has complete access to VTL 0 memory and resources

Can use hypervisor to limit VTL 0 OS access to certain memory locations by leveraging Secure Level Address Translation (SLAT)

SLAT: CPU hardware support, basis of Credential Guard, stores secrets in those memory locations

Can also use SLAT to interdict and control execution of memory locations, Device Guard

I/O Memory Management Unit (MMU) prevents normal device drivers from using hardware devices to directly access memory by virtualizing memory access for devices

Prevents device drivers from using direct memory access (DMA) to access hypervisor or secure kernel's physical memory regions

Hypervisor is first system component to be launched by boot loader

Defines VTL 0 and 1 execution environment

Programs SLAT and I/O MMU

Runs the bootloader again in VTL 1 to load the secure kernel which further configures the system

Trustlets – special class of signed binaries that are allowed to execute in VTL 1

User-mode procs running in VTL-1 can leverage isolation to hide themselves, make secure system calls, sign their own secrets → could lead to bad interactions with other VTL 1 proc or smart kernel

Secure kernel has hard-coded knowledge of which Trustlets have been created so far

Impossible to create new Trustlet without access to secure kernel and cannot patch secure kernel (since that would void the Microsoft signature)

Key system components

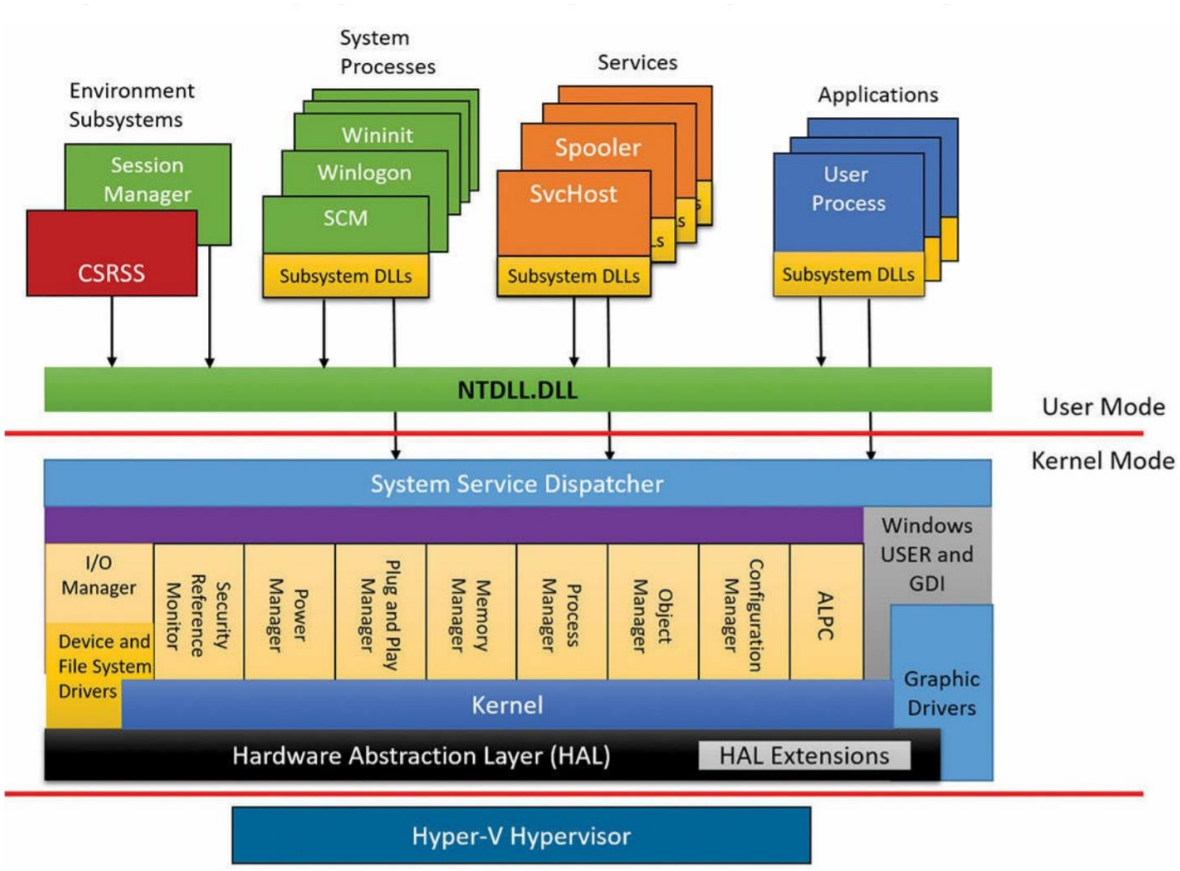


FIGURE 2-4 Windows architecture.

Environment subsystems and subsystem DLLs (/SUBSYSTEM)

Role: expose subset of Windows executive system services to application programs

Each subsystem provides access to different subsets of native services

Each executable image (.exe) bound to one subsystem

Process creation code examines subsystem type to notify that subsystem of the new process

Image links the subsystem DLL for their subsystem in order to access the subsystem interface to have access to executive system services

Windows subsystem DLL (Kernel32.dll, Advapi32.dll, User32.dll, Gdi32.dll) implement Windows API functions

When application calls function in subsystem DLL, either:

1. Function is implemented in user-mode inside subsystem DLL (ex. GetCurrentProcess)
   Does not send message to environment subsystem process
   Does not call any Windows executive system services
2. Function requires 1 or more calls to Windows executive (ex. ReadFile, WriteFile)

3. Function makes request to environment subsystem via ALPC message to perform some operation in the environment subsystem process

Environment Subsystem Processes in user mode – responsible for maintaining the state of the client applications under their control

Subsystem startup

Session Manager (smss.exe) starts the subsystems

Startup info stored in HKLM\SYSTEM\CurrentControlSet\Control\SessionManager\SubSystems

"Required" registry value: list of subsystems that load when system boots

"Windows" value: specifies the Windows subsystem, csrss.exe (Client/Server Runtime Subsystem)

"Kmode" value: contains file name of the kernel-mode portion of the Windows subsystem, Win32k.sys

Windows subsystem

Windows is the primary subsystem. All other subsystems call the Windows subsystem to perform display I/O

Windows subsystem is critical process.

Major components:

1. Each instance of the environment subsystem process (csrss.exe) loads DLLs: Basesrv.dll, Winsrv.dll, Sxssrv.dll, Csrsrv.dll, which support:

    Create/delete processes/threads

    Shutting down Windows applications (ExitWindowsEx API)

    .ini file to registry location mappings (backwards compatibility purposes)

    Sends kernel notification messages to Windows applications as Windows messages (WM_DEVICECHANGE)

    Support for 16-bit virtual DOS (VDM) processes (32-bit Windows only)

    Side-by-Side (SxS)/Fusion and manifest cache support

    Natural language support functions

Winsrv.dll hosts raw input thread and desktop thread (kernel mode code responsible for mouse cursor, keyboard input, desktop window)

Csrss.exe instances associated with interactive user sessions contain Cdd.dll, Canonical Display Driver, which communicates with DirectX support in kernel on each vertical refresh (VSync) t o draw visible desktop state without hardware-accelerated GDI support

2. Kernel-mode device driver (Win32k.sys) contains:

Windows manager – controls windows display, screen output, keyboard mouse input, devices, passes user messages to applications

Graphics Device Interface (GDI) – library of functions for graphics output devices

Wrappers for DirectX support implemented in kernel driver Dxgkrnl.sys

3. Console host process (Conhost.exe) support console applications
4. Desktop Windows Manager (Dwm.exe) – allows windows rendering into single surface through DirectX and CDD
5. Subsystem DLLs (Kernel32.dll, Advapi32.dll, User32.dll, Gdi32.dll) – translate Windows API functions into kernel mode system service calls in Ntoskrnl.exe and Win32k.sys
6. Graphics device drivers for hardware-dependent graphics display drivers, printer drivers, video miniport drivers

Windows 10 and Win32k.sys

Win32K.sys functionality now split among several kernel modules since not all features required on all Windows 10-based devices (IoT and Windows Mobile 10 do not require full window manager capabilities)

Windows 10 Mobile: Win32k.sys loads Win32kMin.sys and Win32kBase.sys

Full desktop services: Win32k.sys loads Win32kBase.sys, Win32kFull.sys

IoT: Win32k.sys might only need Win32kBase.sys

User-interface controls – windows and buttons – created by USER functions

Applications call USER functions, which causes windows manager to communicate these requests to GDI, which passes them to graphics device drivers

Device driver paired with video miniport driver to complete video display support

GDI mediates between applications and graphics devices (display drivers, printer drivers), so that application code can be independent of hardware devices and drivers

Most of the subsystem runs in kernel mode

Only process and thread creation and termination and DOS device drive letter mapping send message to Windows subsystem process

Windows app mostly context switch to Windows subsystem process if they need to draw cursor position, handle keyboard input, render screen via CDD

Console windows host

In original Windows subsystem, the subsystem process, Csrss.exe, responsible for managing console windows

Each console app communicated with Csrss.exe

Windows 7 +, Conhost.exe spawned from Csrss.exe and used for each console window on the system

Single console window can be shared by multiple console applications

Windows 8 +, Conhost.exe spawned from console-based process (rather than Csrss.exe) by console driver (\Windows\System32\Drivers\ConDrv.sys)

Console proc communicates with Conhost.exe via console driver, ConDrv.sys

Sends read, write, I/O control, requests

Conhost.exe acts as server for console proc

Allows console proc to directly receive input from console driver through read/write I/Os and reduces amount of context switching

Workhorse of Conhost.exe is \Windows\System32\ConhostV2.dll

Other Subsystems

Subsystem info extracted from PE header, so required source code of original binary to rebuild as PE exe.

Limited functionality provided by Win32 subsystem or NT kernel

Pico model – updated take on subsystems

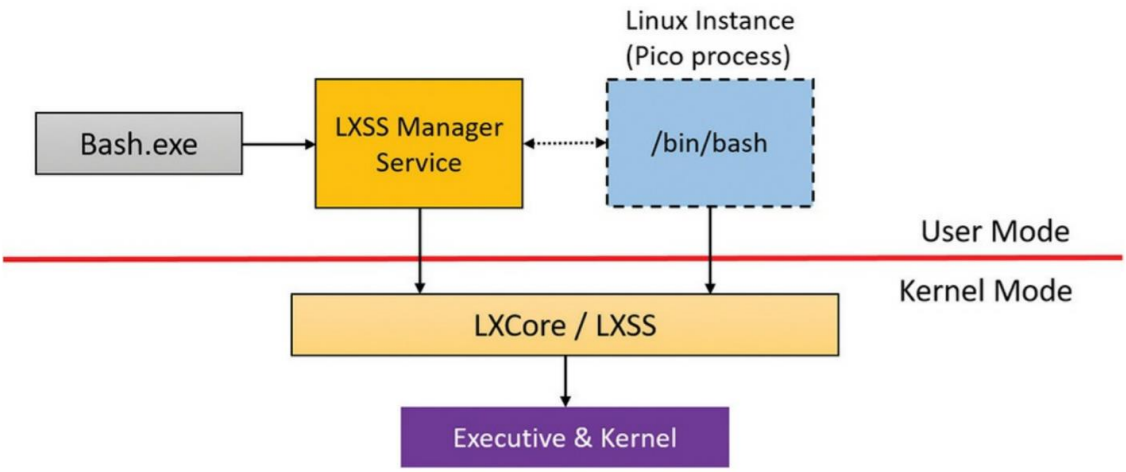
Pico provider – custom kernel-mode driver that accesses kernel interfaces via PsRegisterPicoProvider API

- Allows provider to create Pico processes/threads and customize their execution context, segments, and store data in EPROCESS and ETHREAD structures
- Allow provider to receive notifications when processes/threads engage in system actions (i.e. system calls, exceptions, APCs, page faults, termination, context changes, suspension/resume, etc)

Windows Subsystem for Linux (WSL) components: Lxss.sys and Lxcore.sys – make up the Pico provider interface for WSL

Pico provider and LXSS Manager provide support for launching Linux images without calling Windows API

Pico provider communicates with LXSS Manager, which implements COM-based interface to communicate with launcher process, Bash.exe, and with management process, Lxrun.exe.



Pico provider implements Linux Virtual File System (VFS) and support for /sys, /dev namespaces and behaviors

Pico provider leverages Windows Socket for Kernel (WSK) networking, but wraps around actual socket behavior in order to support UNIX domain sockets, Linux netLink sockets, and standard Internet sockets

Kernel Npfs.sys driver supports Windows named pipes, but differs from Linux pipes, so provider re-implemented pipes for Linux apps

Ntdll.dll

System support library for native applications and subsystem DLLs

Native: images not tied to any particular subsystem

Contains two types of functions:

1. System service dispatch stubs to Windows executive system services

   Provides user mode interface to Windows executive system services such as NtCreateFile, NtSetEvent

   Ntdll.dll creates entry point with the name of the functions in the interface

   Contains architecture-specific instruction to transition to kernel mode to invoke system service dispatcher, which calls the kernel-mode system service that contains the real code inside Ntoskrnl.exe

   If VBS enabled, Shared User Data flag will be set and execution uses int 2Eh instruction to take a different path and allow hypervisor to react to the int

2. Internal support functions for subsystems, subsystem DLLs, native images

   Image loader: functions start with Ldr

   Windows subsystem proc communication functions: start with Csr

   Run time library routines: start with Rtl

User-mode debug functions: start with DbgUi

Event Tracing for Windows: Etw

User-mode Async Proc call (APC) dispatcher and exception dispatcher

Heap Manager

C Run-Time (CRT) routines in Ntdll.dll – string and standard libraries for native applications

Native images

Executables that don't belong to any subsystem → don't link against subsystem DLLs such as Kernel32.dll for Windows subsystem

Only link to Ntdll.dll

These images usually only built by Microsoft (ex. Session Manager Smss.exe, Autochk)

Smss.exe is first user-mode proc created directly by kernel (cannot be dependent on Windows subsystem since Csrss.exe has not yet been started)

Responsible for starting Csrss.exe

Windows executive is upper layer to Ntoskrnl.exe, contains:

- Functions (system services) that are exported and callable from user mode (ex NtCreateFile)

Accessible through Windows API or API of another environment subsystem

- Device driver functions that are called through DeviceIoControl functions

Interface between user and kernel mode for device driver functions not associated with read/write (ex. ConDrv.sys)

- Documented Functions that can only be called from kernel mode (start with Io)

- Undocumented Functions that can only be called from kernel mode

- Functions defined as global symbols but not exported

- Functions internal to a module not defined as global symbols

Executive Major Components

Configuration Manager: implements and manages system registry

Process Manager: Creates/Terminates processes and threads

Security Reference Monitor (SRM): enforces security policies on the local computer

I/O manager: implements device independent I/O, dispatches appropriate device drivers

Plug and Play (PnP) manager: determines which drivers required to support a particular device and loads those drivers. Responsible for sending event notifications for device changes

Power manager: processor power management (PPM) and power management framework (PoFx) coordinate power events, generate power management I/O notifications to device drivers

Windows Driver Mode (WDM) Windows Management Instrumentation (WMI) routines: enable device drivers to public performance and configuration info and receive commands from user-mode WMI services (both local and remote machines can consume WMI info)

Memory manager: implements virtual memory

Cache manager: improves performance of file based I/O, uses memory manager's support for mapped files

Four categories of support functions that executive uses

Object manager: creates, manages, deletes Windows executive objects and abstract data types that are used to represent OS resources (i.e. processes, threads, sync objects)

Asynchronous LPC (ALPC) facility: passes messages between client and server processes on the same computer. ALPC also used for remote procedure calls (RPC) – communication for client and server processes across a network

Run-time library functions: string processing, arithmetic ops, data types conversions, security structures

Executive support routines: system memory allocation (paged and non-paged pool), interlocked memory access, sync mechanism (fast mutexes, pushlocks, executive resources)

Some executive infrastructure routines:

Kernel debugger library

User-Mode Debugging Framework

Hypervisor library and VBS library – kernel support for secure virtual machine environment

Errata manager – workarounds for nonstandard/noncompliant hardware

Driver Verifier – optional integrity checks of kernel-mode drivers and code

Event Tracing for Windows (ETW) – system-wide event tracing for kernel and user mode components

Windows Diagnostic Infrastructure (WDI) – traces system activity based on diagnostic scenarios

Windows Hardware Error Architecture (WHEA) support routines – framework for reporting hardware errors

File-System Runtime Library (FSRTL) – support routines for file system drivers

Kernel Shim Engine (KSE) – driver-compatible shims and device errata support

Kernel

Consists of set of functions in Ntoskrnl.exe

Thread-scheduling, synchronization services (for executive) and low-level architecture-dependent support (interrupt, exception dispatch)

Kernel functions documented in WDK begin with Ke

Implements OS mechanisms and avoid policy making (leaves policy to executive; exception: thread scheduling and dispatching)

Kernel objects

Represents threads and shareable resources as objects, which need object handles, security checks, resource quotas

Kernel objects support creation of executive objects

Control objects: establishes semantics for OS function

Asynchronous Procedure Call (APC) object

Deferred Procedure Call (DPC) object

I/O manager objects (such as interrupt object)

Dispatcher objects: synchronization capabilities that affect thread scheduling

Kernel thread

Mutex (mutant)

Event

Kernel event pair

Semaphore

Timer

Waitable timer

Kernel processor control region (KPCR) and control block

Kernel processor control region (KPCR) stores processor-specific data

- Processor's interrupt dispatch table (IDT)
- Task state segment (TSS)
- Global Descriptor Table (GDT)
- Interrupt control state (shared with other modules like HAL and ACPI driver)

Kernel stores a pointer to the KPCR in the fs register on 32-bit Windows and the gs register on x64 Windows

Kernel Processor Control Block (KPRCB) – private structure (embedded in KPCR) used only by kernel code in Ntoskrnl.exe and contains:

- Scheduling information such as current, next, and idle threads
- Dispatcher database for the processor (queues for priority level)
- DPC queue
- CPU vendor and id information
- CPU and NUMA topology (node info, cores per package)
- Cache sizes
- Time account information (DPC and interrupt time)
- I/O statistics
- Cache manager statistics
- DPC statistics
- Memory manager statistics
- Sometimes contains non-paged and paged-pool system look-aside lists

Hardware support

Hardware-specific variations: interrupt handling, exception dispatching, multiproc sync

Kernel provides interfaces that are implemented differently on different architectures

Interfaces to provide translation buffer and CPU cache support also arch-specific

Context-switching is arch specific since context is described by the processor state (registers), so what to save and load depends on the arch

Hardware abstraction layer

HAL is a loadable kernel-mode module that provides low-level interface to hardware platform on which Window runs

Hides hardware-dependent details (I/O interfaces, interrupt controllers, multiproc comms)

Windows internal components and device drivers maintain portability by calling HAL routines

X64 machines have the same motherboard config since they require ACPI and APIC support → ARM and x64 have only one HAL image

Windows supports HAL extensions to deal with variation in IoT devices

HAL extensions: DLLs that the boot loader may load if specific hardware requires them (registry-based configuration or through ACPI)

HAL extensions must be custom signed with HAL extension certificate

Ntoskrnl.exe linked to the following binaries:

Pshed.dll – Platform-Specific Hardware Error Driver (PSHED) hides details of platform's error-handling mechanisms

Bootvid.dll – Boot Video Driver on x86 support for VGA commands for displaying boot text and logo during startup

Kdcom.dll – Kernel Debugger Protocol (KD) comms library

Ci.dll – Integrity library

Msrpc.sys – Microsoft Remote Procedure Call (RPC) client driver for kernel mode, allows kernel (and drivers) comms with user-mode services through RPC or to marshal MES - encoded assets

API Sets described in terms of contracts such as:

Werkernel – Windows Error Reporting (WER) in the kernel

Tm – kernel transaction manager (KTM)

Kcminitcfg – custom initial registry config on specific platforms

Ksr – Kernel Soft Reboot (KST) and persistence of certain memory ranges to support KSR

Ksecurity – AppContainer policies (Windows Apps)

Ksigningpolicy – policies for user-mode code integrity (UMCI) to support non-AppContainer procs on certain SKUs or configure Device Guard/App Locker security features

Ucode – microcode update library

Clfs – Common Log File System driver used by Transaction Registry (TxR) and others

Ium Contract – Policies for IUM Trustlets

Device drivers

Can be kernel-mode or user-mode and can be hardware or non-hardware

Hardware kernel-mode device drivers:

Runs in kernel mode in one of three contexts:

1. User thread that initiated the I/O function (read operation)
2. Kernel-mode system thread (PnP manager request)
3. Interrupt – whichever thread was current when interrupt occurred

Device drivers in Windows call HAL functions to interface with hardware. They do not manipulate hardware directly

Types of device drivers:

Hardware: Use HAL to read/write from physical device or network (bus drivers, human interface drivers, mass storage drivers, etc)

File system: Windows drivers that accept file-oriented I/O requests and translate requests bound to particular device

File system filter: disk mirroring and encryption, scanning for malware, intercept I/O requests, added-value processing

Network redirectors and servers: file system drivers that transmit and receives file system I/O requests from remote machine on network

Protocol: implement TCP/IP, NetBEUI, IPX/SPX etc

Kernel streaming filter: perform signal processing on data streams

Software: perform ops that can only be done in kernel mode on behalf of user-mode procs

Windows driver model

WDM – base model for writing drivers for hardware devices since Windows 2000

Three kinds of drivers:

Bus drivers: Services bus controller, adapter, bridge, devices with child devices

Each type of bus on a system has one bus driver

Examples of buses: PCI, USB

Function driver: main device driver that accesses device-specific registers

Provides operational interface for its device

Filter drivers: Add functionality to device or driver, modify I/O requests and responses

Used to fix hardware that provides incorrect info about hardware resource requirements

No single driver controls all aspect of a device

Windows Driver foundation (WDF)

Provides frameworks: Kernel-Model Driver Framework (KMDF) and User-Mode Driver Framework (UMDF)

KMDF for Windows 2000 Sp4 +

Calls KMDF library to perform work that is not hardware-specific

UMDF for Windows XP +

Allows certain classes of drivers to be implemented as user-mode drivers (MP3, cell phone, printers, USB-based)

Universal Windows drivers

Since Windows 10, Universal Windows drivers – device drivers that share APIs and Device Driver Interfaces (DDIs) provided by Windows 10 common core

Binary-compatible for specific CPU arch

Can be used as-id on many form factors (IoT, phones, HoloLens, Xbox One, laptops, desktops)

Device drivers and Windows service processes both defined in:

HKLM\SYSTEM\CurrentControlSet\Services

Prefix	**Component**
Alpc	Advanced Local Procedure Calls
Cc	Common Cache
Cm	Configuration manager
Dbg	Kernel debug support
Dbgk	Debug Framework for user mode
Em	Errata manager
Etw	Event Tracing for Windows
Ex	Executive support routines
FsRtl	File System Runtime Library
Hv	Hive Library
Hvl	Hypervisor Library
Io	I/O manager
Kd	Kernel debugger
Ke	Kernel
Kse	Kernel Shim Engine

Lsa	Local Security Authority
Mm	Memory Manager
Nt	NT system services (accessible from user mode via system calls)
Ob	Object manager
Pf	Prefetcher
Po	Power manager
PoFx	Power framework
Pp	PnP manager
Ppm	Processor power manager
Ps	Process support
Rtl	Run-time library
Se	Security Reference Monitor
Sm	Store Manager
Tm	Transaction manager
Ttm	Terminal timeout manager
Vf	Driver Verifier
Vsl	Virtual Secure Mode library
Wdi	Windows Diagnostic Infrastructure
Wfp	Windows FingerPrint
Whea	Windows Hardware Error Architecture
Wmi	Windows Management Instrumentation
Zw	Mirror entry point for system services (beginning with Nt) that sets previous access mode to kernel (eliminates parameter validation) Nt system services validate params only if previous access mode is user

<Prefix> <Operation><Object>

Ex: KeInitializeThread

System Processes

Idle is not technically a process

System, Secure System, Memory Compression (minimal processes) are not full processes b/c they are not running user-mode executables

Idle process: One thread per CPU to account for idle CPU time

System process: Contains most kernel-mode system threads and handles

Secure System process: Contains address space of the secure kernel in VTL 1

Memory Compression process: Contains compressed user-mode processes

Session manager (Smss.exe)

Windows subsystem (Csrss.exe)

Session 0 initialization (Wininit.exe)

Logon process (Winlogon.exe)

Service Control Manager (Services.exe), child service processes such as system-supplied generic service-host process (Svchost.exe)

Local Security Authentication Service (Lsass.exe), if Credential Guard active, Isolated Local Security Authentication Server (Lsaiso.exe)

System process and system threads

System process (pid 4) contains a thread that runs only in kernel mode, kernel-mode system thread:

Can only run in kernel-mode executing code loaded in system space

Have no user process address space (must allocate from OS memory heaps)

Created by PsCreateSystemThread or IoCreateSystemThread

Owned by System process but device driver can create sys thread in any process

Balance set manager: sys thread that wakes up once per second to initiate scheduling and memory management events

System threads used for many things: polling device, waiting for I/O or objects, writing dirty pages, swapping processes, responding to I/O

Secure System Process

Home of the VTL 1 secure kernel address space, handles, and system threads

Technically only provides a visual indicator to users that VBS is active since scheduling, obj management, and mem management, are owned by VTL 0 kernel

Memory Compression Process

Stores compressed pages of memory in its user-mode address space that correspond to memory that has been evicted from the working sets of certain processes

Session Manager

Smss.exe, first user-mode proc created in the system as a Protected Process Light (PPL)

Kernel mode sys thread creates this proc after it performs the final phase of initialization of the executive and kernel

First checks whether it is the master Smss.exe, the first instance, or an instance of itself that the master Smss.exe launched to create a session

Creates multiple instances of itself during boot-up and Terminal Services session creation – enhances logon performance when multiple users connect at same time

Service Control Manager

User-mod process services are similar to Linux daemon processes – can be configured to start at boot without interactive logon, can be started manually using sc.exe (Services administrative tool) or by calling Windows StartService function

Most run as SYSTEM or LOCAL SERVICE but some run in security context as logged-in user accounts

Service Control Manager (SCM) image %SystemRoot%\System32\Services.exe responsible for start/stop/interacting with service processes including registering service's successful startup, responding to status requests, pausing/shutting down service

Services defined under HKLM\SYSTEM\CurrentControlSet\Services

Map service process to services contained in the process: tlist /s or tasklist /svc

Some services share process with other services

Winlogon, LogonUI, Userinit

Windows logon proc \System32\Winlogon.exe handles interactive user logons/logoffs

Secure Attention Sequence (SAS) keystroke: Ctrl-Alt-Delete

SAS protect users from password-capture programs that simulate logon proc (this keyboard sequence cannot be intercepted by a user-mode application)

Credential providers implement identification and authentication aspects of logon process

When Winlogon.exe detects SAS, it launches LogonUI.exe to initialize credential providers

User name and password sent to Local Security Authentication Service process (Lsass.exe) to be authenticated.

Lsass calls proper authentication package to perform actual verification (checking against Active Directory or SAM)

If Credential Guard enabled, lsass.exe communicates with Isolated LSA Trustlet (Lsaiso.exe) to obtain machine key required to authenticate the legitimacy of the authentication request

If logon successful, Lsass calls function in SRM (NtCreateToken) to generate access token object that contains the user's security profile

Lsass creates second restricted version of the token if User Account Control (UAC) enabled and user is member of administrators or has administrator priv

Winlogon uses access token to create initial process(es) in the user's session (determined by HKLM\SOFTWARE\Microsoft\Windows NT\Current Version\Winlogon "Userinit" registry value) (default is Userinit.exe)

Userinit.exe initializes user environment, looks in the registry at Shell value and creates process to run system-defined shell (Explorer.exe for example). Then Userinit exits

Explorer.exe has no parent because its parent has exited