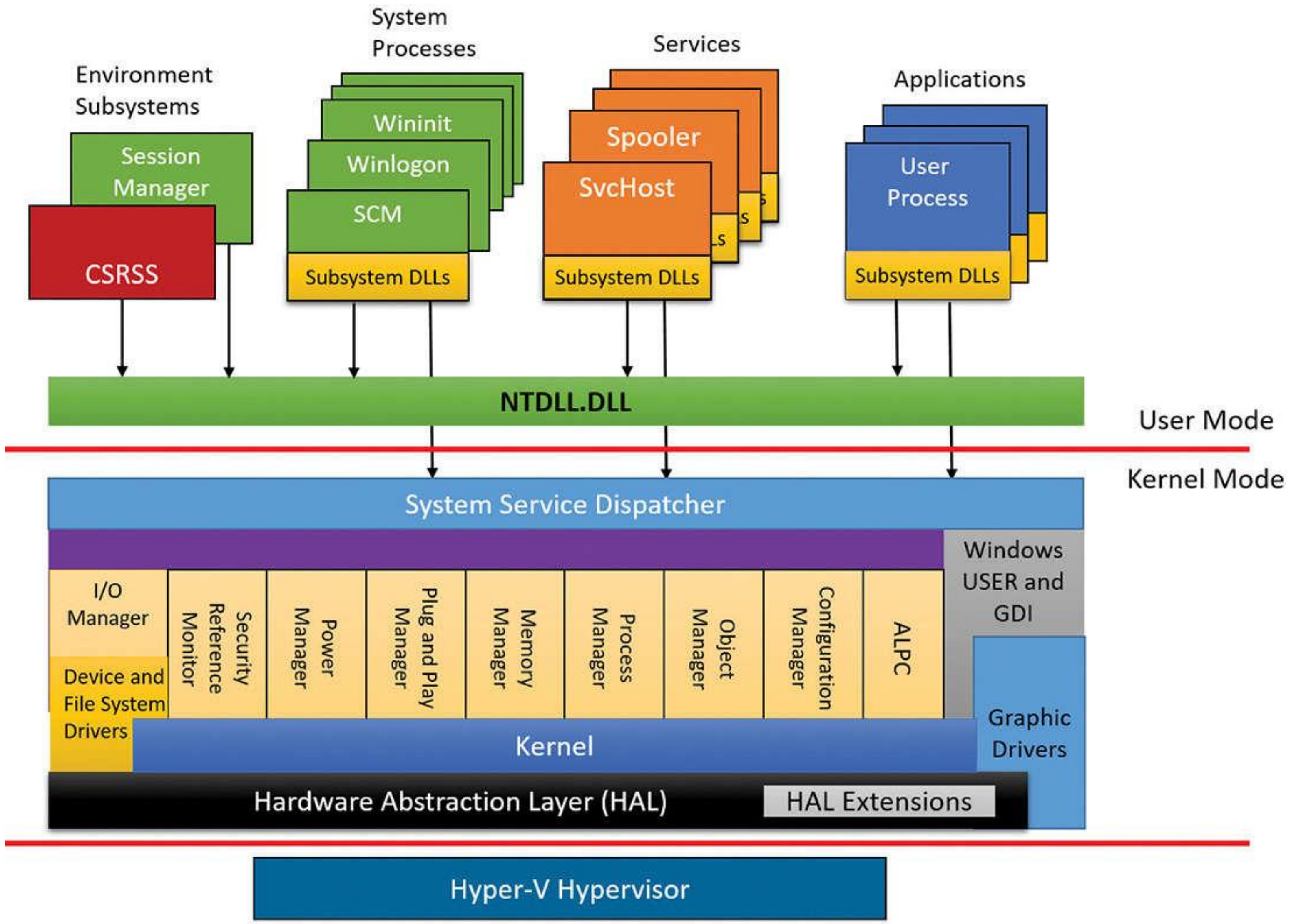
Windows architecture



Environment subsystems and subsystem DLLs

- To expose some subset of the base Windows executive system services to application programs
- Each subsystem can provide access to different subsets of the native services in Windows
- Each executable image (.exe) is bound to one and only one subsystem.
- When an image is run, the process creation code examines the subsystem type code in the image header so that it can notify the proper subsystem of the new process.
- This type code is specified with the /SUBSYSTEM linker option of the Microsoft Visual Studio linker

Environment subsystems and subsystem DLLs

- User applications don’t call Windows system services directly.
- Instead, they go through one or more subsystem DLLs.
- These libraries export the documented interface that the programs linked to that subsystem can call.
- For example,
  - the Windows subsystem DLLs (such as Kernel32.dll, Advapi32.dll, User32.dll, and Gdi32.dll) implement the Windows API functions
  - The SUA subsystem DLL (Psxdll.dll) is used to implement the SUA API functions (on Windows versions that supported POSIX)

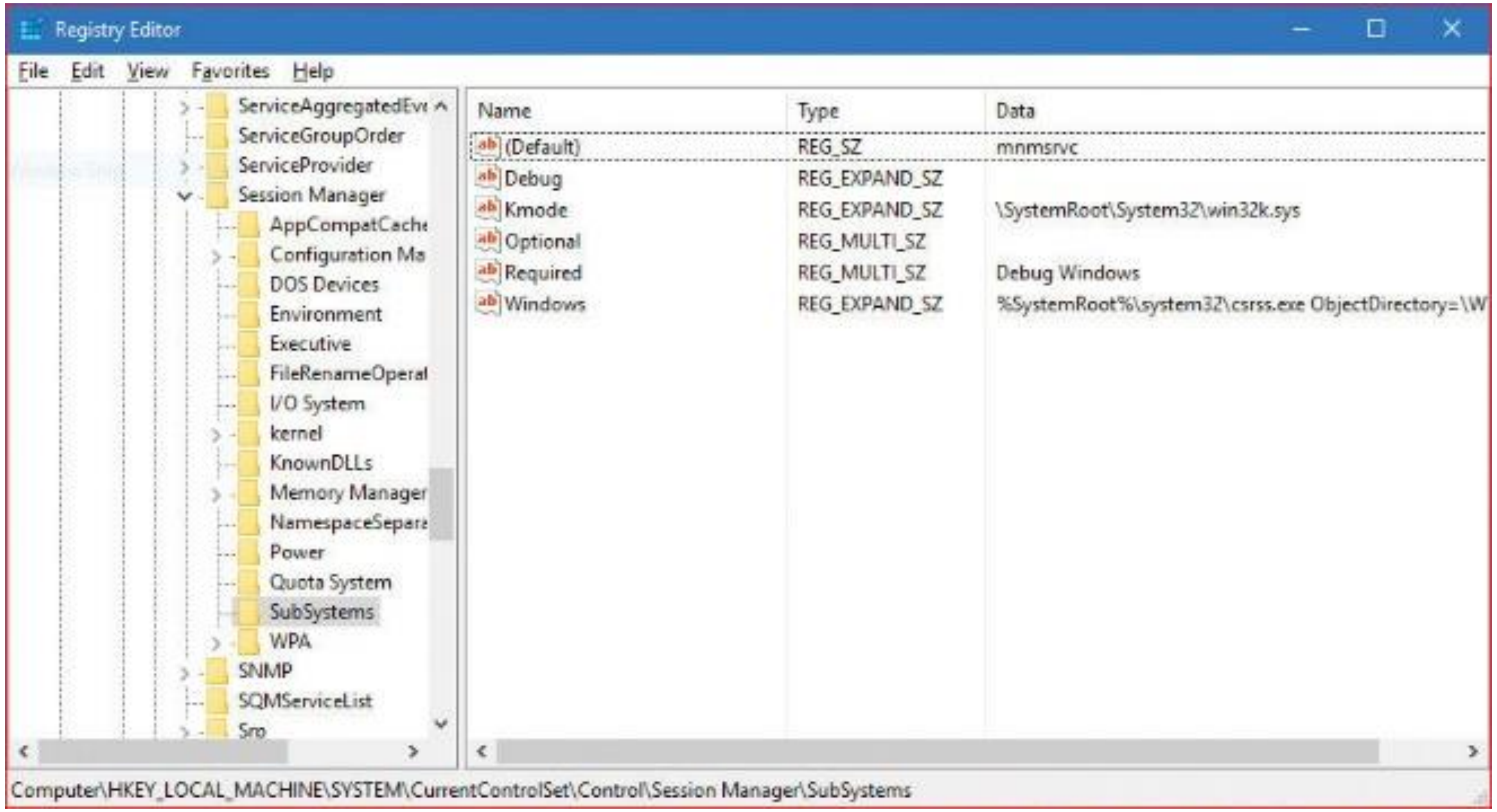
Environment subsystems and subsystem DLLs debugging

- When an application calls a function in a subsystem DLL, one of three things can occur:
  - The function is entirely implemented in user mode inside the subsystem DLL (`GetCurrentProcess` and `GetCurrentProcessId`).
  - The function requires one or more calls to the Windows executive (`ReadFile -> NtReadFile` and `WriteFile -> NtWriteFile`).
  - The function requires some work to be done in the environment subsystem process. (ALPC)
  - Some functions can be a combination of the second and third items just listed (`CreateProcess` and `ExitWindowsEx`).

Environment subsystems and subsystem DLLs debugging

Subsystem startup

- are started by the Session Manager (Smss.exe) process.



Name	Type	Data
(Default)	REG_SZ	mnmsrvc
Debug	REG_EXPAND_SZ	
Kmode	REG_EXPAND_SZ	\SystemRoot\System32\win32k.sys
Optional	REG_MULTI_SZ	
Required	REG_MULTI_SZ	Debug Windows
Windows	REG_EXPAND_SZ	%SystemRoot%\system32\csrss.exe ObjectDirectory=\W

Computer\HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\SubSystems

Environment subsystems and subsystem DLLs debugging

Windows subsystem

- Windows was designed to support multiple independent environment subsystems.
- Having each subsystem implement all the code to handle windowing and display I/O would result in a large amount of duplication of system functions!
- Because Windows was the primary subsystem, the Windows designers decided to locate these basic functions there and have the other subsystems call on the Windows subsystem to perform display I/O.
- Windows subsystem is a required component for any Windows system, even on server systems with no interactive users logged in. Because of this, the process is marked as a critical process.

Environment subsystems and subsystem DLLs debugging

The Windows subsystem consists of the following major components:

- For each session, an instance of the environment subsystem process (Csrss.exe) loads four DLLs (Basesrv.dll, Winsrv.dll, Sxssrv.dll, and Csrsrv.dll).
- A kernel-mode device driver (Win32k.sys)
- The console host process (Conhost.exe)
- The Desktop Window Manager (Dwm.exe)
- Subsystem DLLs
- Graphics device drivers

Environment subsystems and subsystem DLLs debugging

Windows 10 and Win32k.sys

- On phones (Windows Mobile 10) Win32k.sys loads Win32kMin.sys and Win32kBase.sys
- On full desktop systems Win32k.sys loads Win32kBase.sys and Win32kFull.sys system service calls in Ntoskrnl.exe and Win32k.sys
- On certain IoT systems, Win32k.sys might only need Win32kBase.sys

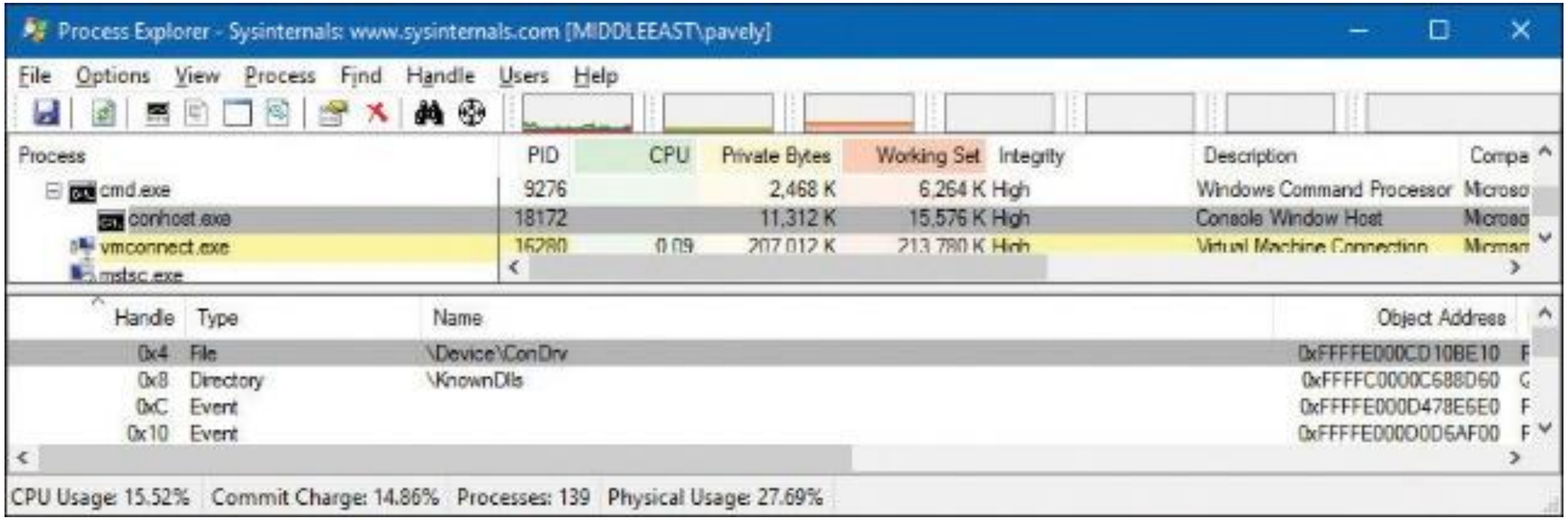
Environment subsystems and subsystem DLLs debugging

Console window host

- In the original Windows subsystem design, the subsystem process (Csrss.exe) was responsible for managing console windows and each console application.

Starting with Windows 7	With Windows 8 and Later
console window host (Conhost.exe) spawned from Csrss.exe	console window host (Conhost.exe) spawned from console-based process by the console driver (\Windows\System32\Drivers\ConDrv.sys)
Csrss.exe receives keyboard input (as part of the raw input thread), send it through Win32k.sys to Conhost.exe, and then use ALPC to send it to Cmd.exe	By obviating the need for Csrss.exe to receive keyboard input, the process in question communicates with Conhost.exe using the console driver (ConDrv.sys), by sending read, write, I/O control and other I/O request types avoiding needless context switching

Environment subsystems and subsystem DLLs debugging



Process Explorer - Sysinternals: www.sysinternals.com [MIDDLEEAST\pavely]

Process	PID	CPU	Private Bytes	Working Set	Integrity	Description	Compa
cmd.exe	9276		2,468 K	6,264 K	High	Windows Command Processor	Microso
conhost.exe	18172		11,312 K	15,576 K	High	Console Window Host	Microso
vmconnect.exe	16280	0.09	207,012 K	213,780 K	High	Virtual Machine Connection	Microso
mstsc.exe							

Handle	Type	Name	Object Address
0x4	File	\Device\ConDrv	0xFFFFE000CD10BE10
0x8	Directory	\KnownDlls	0xFFFFC0000C688D60
0xC	Event		0xFFFFE000D478E6E0
0x10	Event		0xFFFFE000D0D6AF00

CPU Usage: 15.52% Commit Charge: 14.86% Processes: 139 Physical Usage: 27.69%

- Conhost.exe is a child process of the console process
- Conhost creation is initiated by the image loader for Console subsystem images or on demand if a GUI subsystem image calls the `AllocConsole` Windows API.
- GUI and Console are essentially the same in the sense both are variants of the Windows subsystem type.)
- Conhost.exe loads a DLL (\Windows\System32\ConhostV2.dll) that includes the bulk of code that communicates with the console driver.

Other subsystems

- Windows originally supported POSIX and OS/2 subsystems.

- No longer provided with Windows general concept of subsystems remains, however, making the system extensible to new subsystems if such a need arises in the future.

Other subsystems

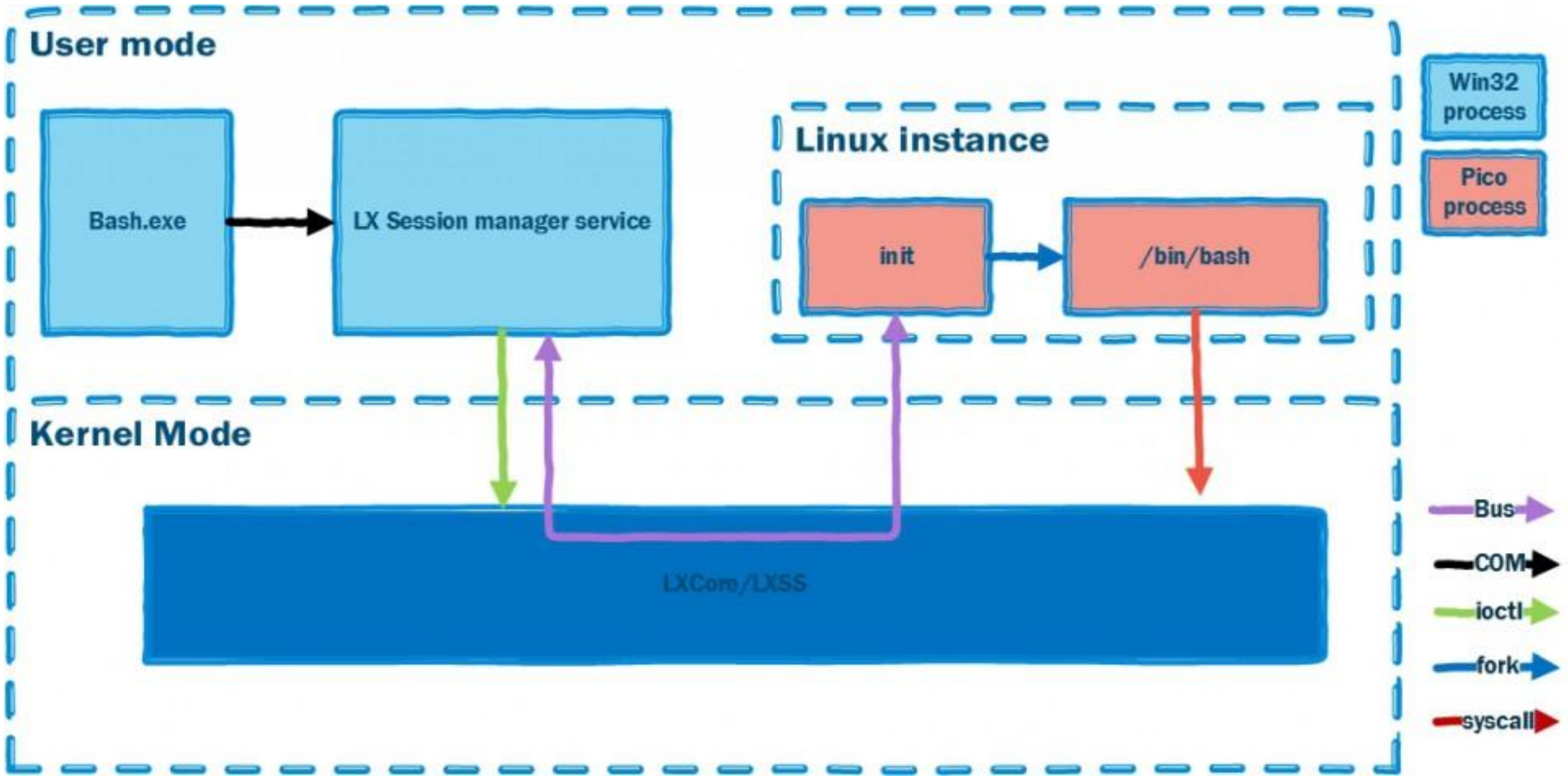
Pico providers and the Windows subsystem for Linux.

- Two disadvantages of the traditional subsystem model
  - Because subsystem information is extracted from the Portable Executable (PE) header, it requires the source code of the original binary to rebuild it as a Windows PE executable file (.exe). This will also change any POSIX-style dependencies and system calls into Windows-style imports of the Psxdll.dll library.
  - It is limited by the functionality provided either by the Win32 subsystem (on which it sometimes piggybacks) or the NT kernel. Therefore, the subsystem wraps, instead of emulates, the behavior required by the POSIX application. This can sometimes lead to subtle compatibility flaws.
- POSIX subsystem/SUA was designed with POSIX/UNIX applications in mind, which dominated the server market decades ago, not true Linux applications, which are common today.

Other subsystems

- Solving these issues required a different approach to building a subsystem—one that did not require the traditional user-mode wrapping of the other environments' system call and the execution of traditional PE images.
- The Drawbridge project from Microsoft Research provided the perfect vehicle for an updated take on subsystems. It resulted in the implementation of the Pico model.
- Pico provider: A custom kernel-mode driver that receives access to specialized kernel interfaces through the `PsRegisterPicoProvider` API
- Benefits of Pico provider:
  - They allow the provider to create Pico processes and threads while customizing their execution contexts, segments, and store data in their respective EPROCESS and ETHREAD structures
  - They allow the provider to receive a rich set of notifications whenever such processes or threads engage in certain system actions such as system calls, exceptions, APCs, page faults, termination, context changes, suspension/resume, etc.

Other subsystems



Other subsystems

Ntdll.dll

- a special system support library primarily for the use of subsystem DLLs and native applications
- Contains two types of functions:
  - System service dispatch stubs to Windows executive system services
  - Internal support functions used by subsystems, subsystem DLLs, and other native images
- For each of these functions, Ntdll.dll contains an entry point with the same name.
- The code inside the function contains the architecture-specific instruction that causes a transition into kernel mode to invoke the system service dispatcher.
- After verifying some parameters, this system service dispatcher calls the actual kernel-mode system service that contains the real code inside Ntoskrnl.exe.

Other subsystems

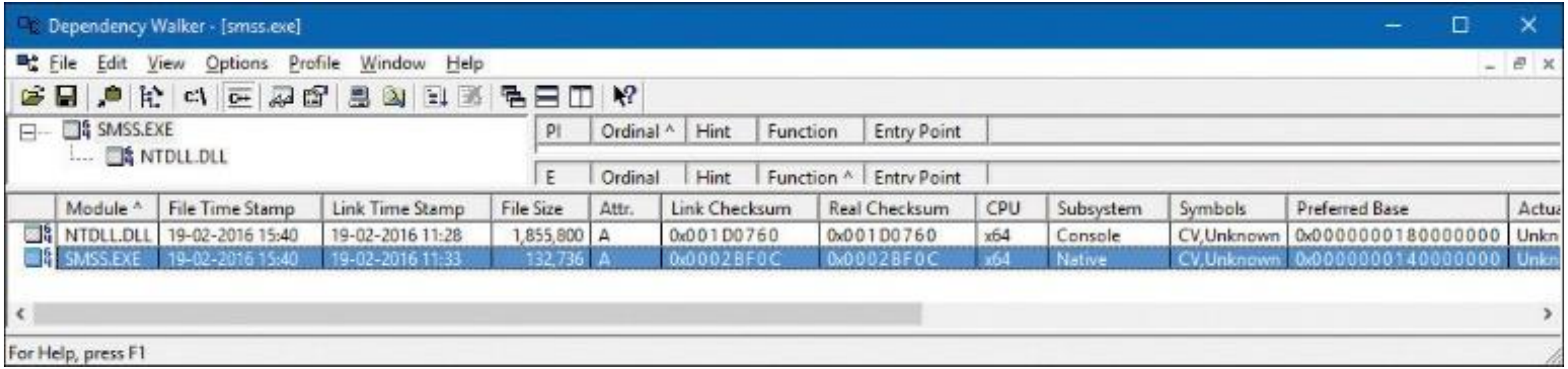
- IUM applications can leverage another binary similar to Ntdll.dll, called IumDll.dll
- This library also contains system calls, but their indices will be different.
- Ntdll Contains many support functions, such as:
  - image loader (functions that start with `Ldr`)
  - heap manager
  - Windows subsystem process communication functions (functions that start with `Csr`)
  - general run-time library routines (functions that start with `Rtl`)
  - user-mode debugging (functions that start with `DbgUi`)
  - Event Tracing for Windows (functions starting in Etw)
  - user-mode asynchronous procedure call (APC) dispatcher
  - exception dispatcher
  - a small subset of the C Run-Time (CRT) routines in Ntdll.dll, limited to those routines that are part of the string and standard libraries (such as memcpy, strcpy, sprintf, and so on)

Other subsystems

Native images

- Some images (executables) don’t belong to any subsystem (they don’t link against a set of subsystem DLLs, such as Kernel32.dll for the Windows subsystem).
- They link only to Ntdll.dll, which is the lowest common denominator that spans subsystems.
- The native API exposed by Ntdll.dll is mostly undocumented.
- These kind of images are typically built only by Microsoft.
- One example: Session Manager process (Smss.exe)
  - Is responsible for launching Csrss.exe
  - First user-mode process created (directly by the kernel), so it cannot be dependent on the Windows subsystem because Csrss.exe (the Windows subsystem process) has not started yet
- Another example: Autochk
  - Runs at system startup to check disks
  - Because it runs relatively early in the boot process (launched by Smss.exe, in fact), it cannot depend on any subsystem

Other subsystems


Dependency Walker - [smss.exe]
File Edit View Options Profile Window Help
SMSS.EXE
NTDLL.DLL
PI Ordinal ^ Hint Function Entry Point
E Ordinal Hint Function ^ Entry Point
For Help, press F1

Module ^	File Time Stamp	Link Time Stamp	File Size	Attr.	Link Checksum	Real Checksum	CPU	Subsystem	Symbols	Preferred Base	Actu
NTDLL.DLL	19-02-2016 15:40	19-02-2016 11:28	1,855,800	A	0x001D0760	0x001D0760	x64	Console	CV,Unknown	0x0000000180000000	Unkn
SMSS.EXE	19-02-2016 15:40	19-02-2016 11:33	132,736	A	0x0002BF0C	0x0002BF0C	x64	Native	CV,Unknown	0x0000000140000000	Unkn

Executive

- The Windows executive is the upper layer of Ntoskrnl.exe. (The kernel is the lower layer)

- The executive includes the following types of functions:
  - Functions that are exported and callable from user mode
  - Device driver functions that are called through the DeviceIoControl function
  - Functions that can be called only from kernel mode that are exported and documented in the WDK
  - Functions that are exported and can be called from kernel mode but are not documented in the WDK
  - Functions that are defined as global symbols but are not exported
  - Functions that are internal to a module that are not defined as global symbols

Executive

- The executive contains the following major components:
  - Configuration manager
  - Process manager
  - Security Reference Monitor (SRM)
  - I/O manager
  - Plug and Play (PnP) manager
  - Power manager
  - Windows Driver Model (WDM) Windows Management Instrumentation (WMI) routines
  - Memory manager
  - Cache manager

Executive

- The executive contains four main groups of support functions:
  - Object manager
  - Asynchronous LPC (ALPC) facility
  - Run-time library functions
  - Executive support routines
- The executive also contains a variety of other infrastructure routines, some of which are mentioned only briefly:
  - Kernel debugger library
  - User-Mode Debugging Framework
  - Hypervisor library and VBS library
  - Errata manager
  - Driver Verifier
  - Event Tracing for Windows (ETW)
  - Windows Diagnostic Infrastructure (WDI)
  - Windows Hardware Error Architecture (WHEA) support routines
  - File-System Runtime Library (FSRTL)
  - Kernel Shim Engine (KSE)

Kernel

- The kernel consists of a set of functions in Ntoskrnl.exe that provides fundamental mechanisms:
  - Thread-scheduling and synchronization services, used by the executive components
  - Low-level hardware architecture–dependent support, such as interrupt and exception dispatching, which is different on each processor architecture

- The kernel code is written primarily in C, with assembly code reserved for those tasks that require access to specialized processor instructions and registers not easily accessible from C.

- A number of functions in the kernel are documented in the WDK (and can be found by searching for functions beginning with Ke) because they are needed to implement device drivers.

Kernel

Kernel objects

- The kernel provides a low-level base of well-defined, predictable OS primitives and mechanisms that allow higher-level components of the executive to do what they need to do.
- The kernel separates itself from the rest of the executive by implementing OS mechanisms and avoiding policy making.
- It leaves nearly all policy decisions to the executive, with the exception of thread scheduling and dispatching, which the kernel implements.
- Outside the kernel, the executive represents threads and other shareable resources as objects.
- These objects require some policy overhead, such as object handles to manipulate them, security checks to protect them, and resource quotas to be deducted when they are created.
- This overhead is eliminated in the kernel, which implements a set of simpler objects, called kernel objects, that help the kernel control central processing and support the creation of executive objects.

Kernel

- Most executive-level objects encapsulate one or more kernel objects, incorporating their kernel-defined attributes.
- One set of kernel objects, called control objects, establishes semantics for controlling various OS functions.
- This set includes the `Asynchronous Procedure Call (APC)` object, the `Deferred Procedure Call (DPC)` object, and several objects the I/O manager uses, such as the `interrupt` object.
- Another set of kernel objects, known as dispatcher objects, incorporates synchronization capabilities that alter or affect thread scheduling.
- The dispatcher objects include the kernel thread, mutex (called mutant in kernel terminology), event, kernel event pair, semaphore, timer, and waitable timer.
- The executive uses kernel functions to create instances of kernel objects, to manipulate them, and to construct the more complex objects it provides to user mode.

Kernel

Kernel processor control region and control block

- The kernel uses a data structure called the kernel processor control region (KPCR) to store processor-specific data.
- To provide easy access to the KPCR, the kernel stores a pointer to it in the `fs` register on 32-bit Windows and in the `gs` register on an x64 Windows system.
- The KPCR contains
  - basic information such as the processor's interrupt dispatch table (IDT), task state segment (TSS), and global descriptor table (GDT).
  - It also includes the interrupt controller state, which it shares with other modules, such as the ACPI driver and the HAL.
  - an embedded data structure called the kernel processor control block (KPRCB).
- Unlike the KPCR, which is documented for third-party drivers and other internal Windows kernel components, the KPRCB is a private structure used only by the kernel code in Ntoskrnl.exe.

Kernel

- The KPCRB contains
    - Scheduling information such as the current, next, and idle threads scheduled for execution on the processor
    - The dispatcher database for the processor, which includes the ready queues for each priority level
    - The DPC queue
    - CPU vendor and identifier information, such as the model, stepping, speed, and feature bits
    - CPU and NUMA topology, such as node information, cores per package, logical processors per core, and so on
    - Cache sizes
    - Time accounting information, such as the DPC and interrupt time

- The KPRCB also contains all the statistics for the processor, such as:
    - I/O statistics
    - Cache manager statistics
    - DPC statistics
    - Memory manager statistics

Kernel

Hardware support

- The other major job of the kernel is to abstract or isolate the executive and device drivers from variations between the hardware architectures (handling variations in functions such as interrupt handling, exception dispatching, and multiprocessor synchronization).
- The kernel supports a set of interfaces that are portable and semantically identical across architectures.
- Most of the code that implements these portable interfaces is also identical across architectures. Some of these interfaces are implemented differently on different architectures or are partially implemented with architecture-specific code.
- Some kernel interfaces, such as spinlock routines are implemented in the HAL because their implementation can vary for systems within the same architecture family.

Hardware abstraction layer

- The hardware abstraction layer (HAL) is a key part of making portability of Windows possible.
- The HAL is a loadable kernel-mode module (Hal.dll) that provides the low-level interface to the hardware platform on which Windows is running.
- It hides hardware-dependent details such as I/O interfaces, interrupt controllers, and multiprocessor communication mechanisms—any functions that are both architecture-specific and machine-dependent.
- Rather than access hardware directly, Windows internal components and user-written device drivers maintain portability by calling the HAL routines when they need platform-dependent information.
- Although a couple of x86 HALs are included in a standard desktop Windows installation, Windows has the ability to detect at boot-up time which HAL should be used.

Hardware abstraction layer

- Windows supports modules known as HAL extensions, which are additional DLLs on disk that the boot loader may load if specific hardware requiring them is needed
- Creating HAL extensions requires collaboration with Microsoft, and such files must be custom signed with a special HAL extension certificate available only to hardware vendors.
- Additionally, they are highly limited in the APIs they can use and interact through a limited import/export table mechanism that does not use the traditional PE image mechanism.

Device drivers

- Loadable kernel-mode modules (files typically ending with the .sys extension) that interface between the I/O manager and the relevant hardware.
- Run in kernel mode in one of three contexts:
  - In the context of the user thread that initiated an I/O function (such as a read operation)
  - In the context of a kernel-mode system thread (such as a request from the Plug and Play manager)
  - As a result of an interrupt and therefore not in the context of any particular thread but rather of whichever thread was current when the interrupt occurred
- There are several types of device drivers:
  - Hardware device drivers
  - File system drivers
  - File system filter drivers
  - Network redirectors and servers
  - Protocol drivers
  - Kernel streaming filter drivers
  - Software drivers

Device drivers

- From the WDM perspective, there are three kinds of drivers:
    - Bus drivers
    - Function drivers
    - Filter drivers
- The Windows Driver Foundation (WDF) simplifies Windows driver development by providing two frameworks: the Kernel-Mode Driver Framework (KMDF) and the User-Mode Driver Framework (UMDF)
- KMDF provides a simple interface to WDM and hides its complexity from the driver writer without modifying the underlying bus/function/filter model.
- UMDF enables certain classes of drivers—mostly USB-based or other high-latency protocol buses, such as those for video cameras, MP3 players, cell phones, and printers—to be implemented as usermode drivers
- UMDF runs each user-mode driver in what is essentially a user-mode service, and it uses ALPC to communicate to a kernel-mode wrapper driver that provides actual access to hardware.

Device drivers

- If a UMDF driver crashes, the process dies and usually restarts. That way, the system doesn’t become unstable; the device simply becomes unavailable while the service hosting the driver restarts.

- Universal Windows drivers are binary-compatible for a specific CPU architecture (x86, x64, ARM) and can be used as is on a variety of form factors, from IoT devices, to phones, to the HoloLens and Xbox One, to laptops and desktops.

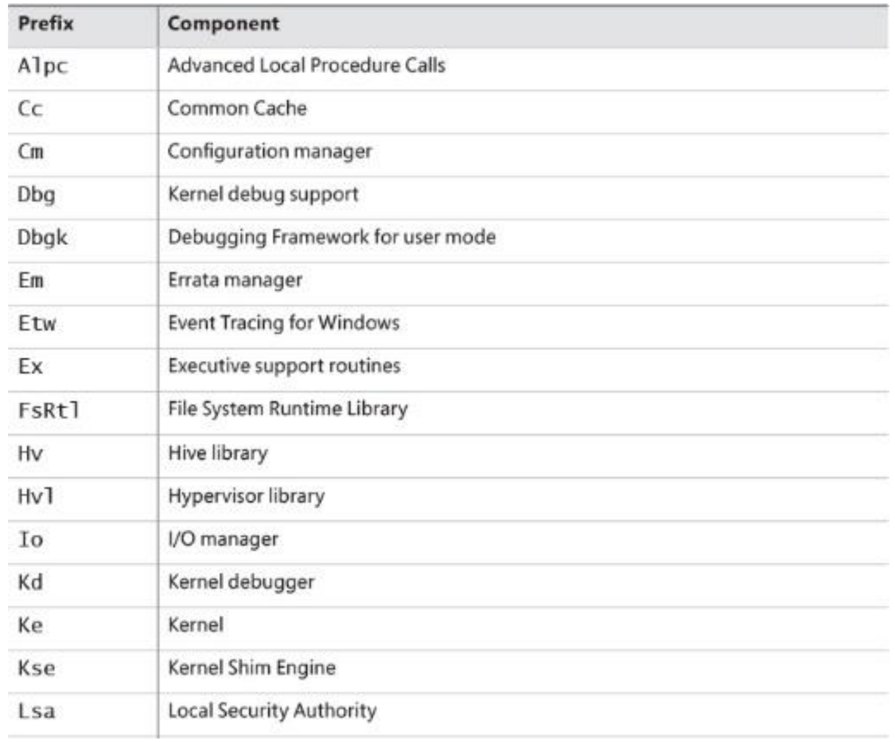
- Universal drivers can use KMDF, UMDF 2.x, or WDM as their driver model.

Device drivers

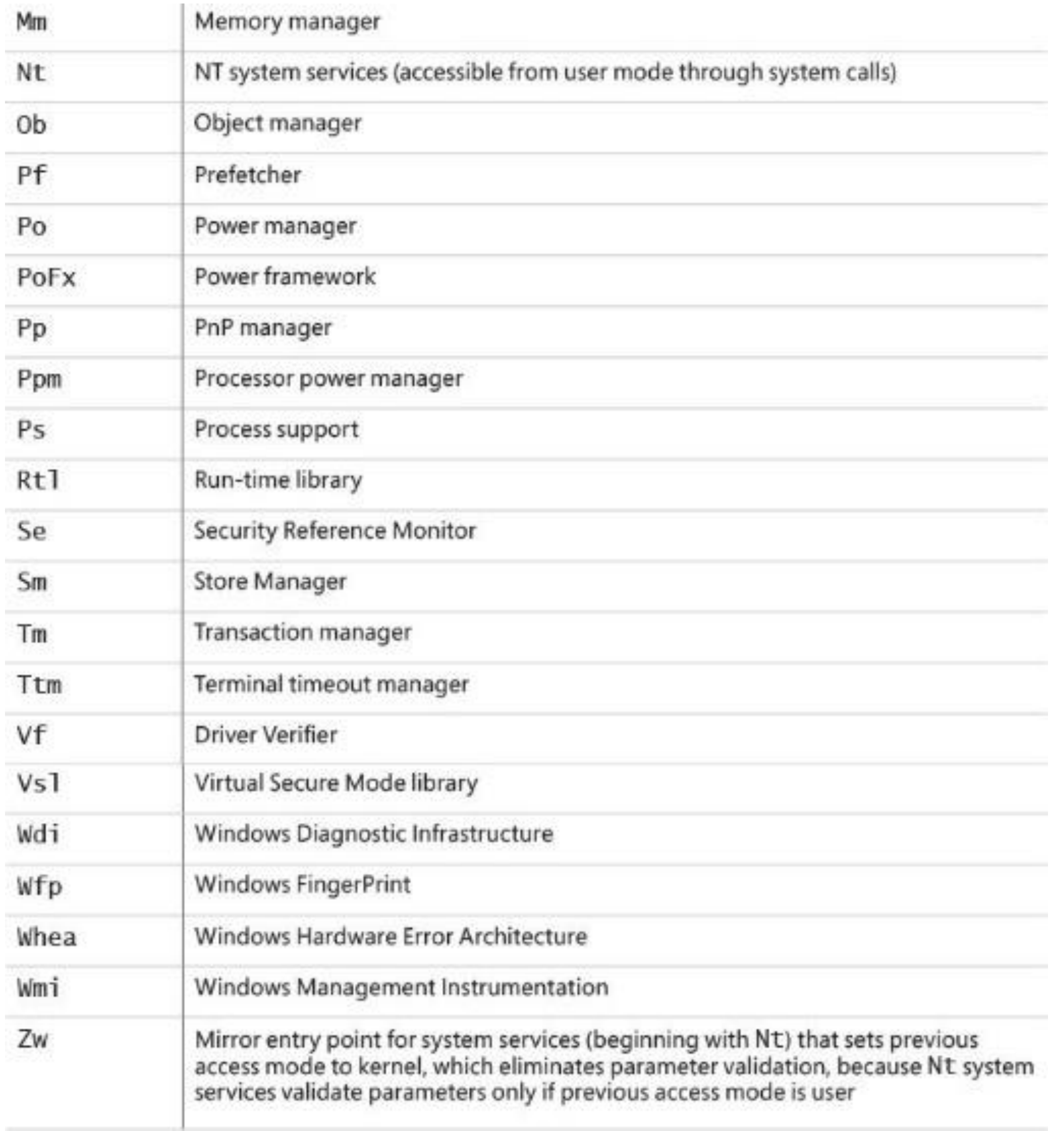
<Prefix><Operation><Object>

```
ExAllocatePoolWithTag

KeInitializeThread
```



Prefix	Component
Alpc	Advanced Local Procedure Calls
Cc	Common Cache
Cm	Configuration manager
Dbg	Kernel debug support
Dbgk	Debugging 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 through 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, which eliminates parameter validation, because Nt system services validate parameters only if previous access mode is user

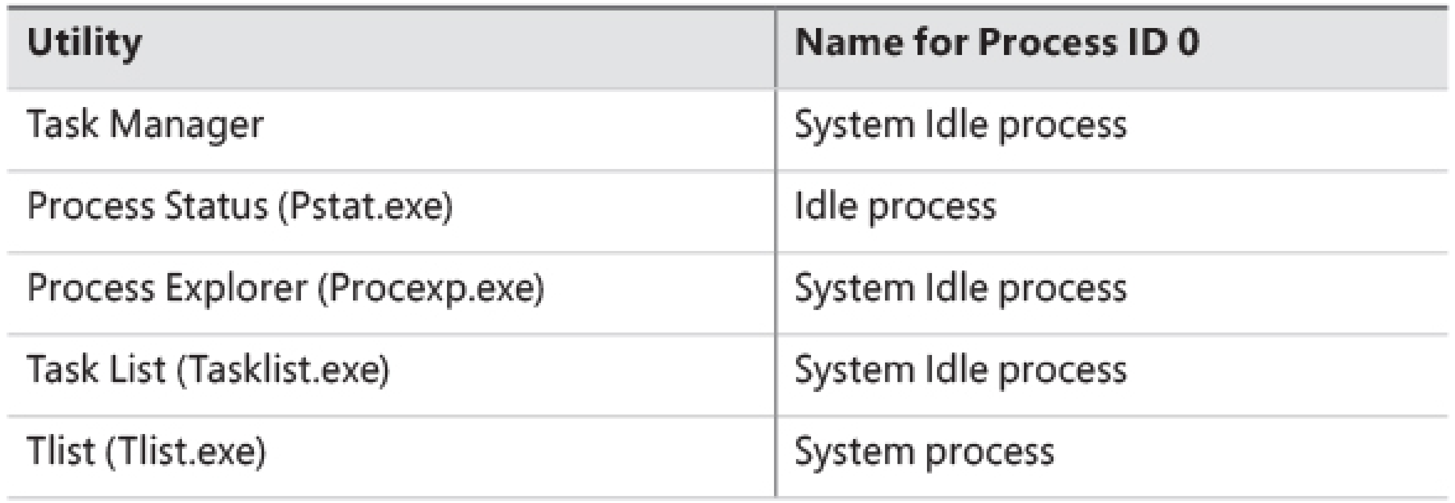
System processes

- Idle process
- System process
- Secure System process
- Memory Compression process
- Session manager (Smss.exe)
- Windows subsystem (Csrss.exe)
- Session 0 initialization (Wininit.exe)
- Logon process (Winlogon.exe)
- Service Control Manager (Services.exe)
- Local Security Authentication Service (Lsass.exe)

System processes

Idle process

- Contains one thread per logical processors to account for idle CPU time with PID 0.
- There is no a real user-mode image and “System Idle Process.exe” in the \Windows directory.



Utility	Name for Process ID 0
Task Manager	System Idle process
Process Status (Pstat.exe)	Idle process
Process Explorer (Procexp.exe)	System Idle process
Task List (Tasklist.exe)	System Idle process
Tlist (Tlist.exe)	System process

System processes

System process and system threads

- Contains the majority of the kernel-mode system threads and handles with PID 4.
- They run only in kernel-mode executing code loaded in system space, whether that is in Ntoskrnl.exe or in any other loaded device driver.
- System threads don’t have a user process address space and hence must allocate any dynamic storage from OS memory heaps, such as a paged or non-paged pool.
- System threads are created by the PsCreateSystemThread or IoCreateSystemThreadUMDF
- A device driver can create a system thread in any process

System processes

Secure System process

- Contains the address space of the secure kernel, handles, and system threads in VTL 1, if running, with variable PID.
- Because scheduling, object management, and memory management are owned by the VTL 0 kernel, no such actual entities will be associated with this process.
- Its only real use is to provide a visual indicator to users (for example, in tools such as Task Manager and Process Explorer) that VBS is currently active (providing at least one of the features that leverages it).

System processes

Memory Compression process

- Uses its user-mode address space to store the compressed pages of memory that correspond to standby memory that's been evicted from the working sets of certain processes.
- Unlike the Secure System process, the Memory Compression process does actually host a number of system threads, usually seen as SmKmStoreHelperWorker and SmStReadThread (Both of these belong to the Store Manager that manages memory compression)
- Unlike the other System processes, this process actually stores its memory in the user-mode address space. This means it is subject to working set trimming and will potentially have large visible memory usage in system-monitoring tools.

System processes

Session Manager

- The Session Manager (%SystemRoot%\System32\Smss.exe) is the first user-mode process created in the system.
- The kernel-mode system thread that performs the final phase of the initialization of the executive and kernel creates this process as a Protected Process Light (PPL).
- When Smss.exe starts, it checks whether it is the first instance (the master Smss.exe) or an instance of itself that the master Smss.exe launched to create a session. If command-line arguments are present, it is the latter.
- Init based on keys under HKLM\SYSTEM\CurrentControlSet\Control\Session Manager

System processes

Service Control Manager

- Is a special protected system process running the image %SystemRoot%\System32\Services.exe that is responsible for starting, stopping, and interacting with service processes.
- Services can refer to either a server process or a device driver.
- Can be configured to start automatically at system boot time without requiring an interactive logon.
- Can also be started manually, such as by running the Services administrative tool, using the sc.exe tool, or calling the Windows `StartService` function.
- Most services run in special service accounts (such as SYSTEM or LOCAL SERVICE), others can run with the same security context as logged-in user accounts.

System processes

Service Control Manager

- Services are defined in the registry under HKLM\SYSTEM\CurrentControlSet\Services.
- Services have three names:
  - the process name you see running on the system
  - the internal name in the registry
  - the display name shown in the Services administrative tool.
- There isn’t always one-to-one mapping between service processes and running services, Because some services share a process with other services.
- A number of Windows components are implemented as services, such as the Print Spooler, Event Log, Task Scheduler, and various networking components.

System processes

Winlogon, LogonUI, and Userinit

- The Windows logon process (%SystemRoot%\System32\Winlogon.exe) handles interactive user logons and logoffs.
- Winlogon.exe is notified of a user logon request when the user enters the secure attention sequence (SAS) keystroke combination (Ctrl+Alt+Delete).
- The reason for the SAS is to protect users from password-capture programs that simulate the logon process because this keyboard sequence cannot be intercepted by a user-mode application.
- The identification and authentication aspects of the logon process are implemented through DLLs called credential providers.
- The standard Windows credential providers implement the default Windows authentication interfaces: password and smartcard.
- Windows 10 provides a biometric credential provider: face recognition, known as Windows Hello.

System processes

Winlogon, LogonUI, and Userinit

- Developers can provide their own credential providers to implement other identification and authentication mechanisms.
- Because Winlogon.exe is a critical system process on which the system depends, credential providers and the UI to display the logon dialog box run inside a child process of Winlogon.exe called LogonUI.exe.
- When Winlogon.exe detects the SAS, it launches LogonUI.exe which initializes the credential providers.
- When the user enters their credentials (as required by the provider) or dismisses the logon interface, the LogonUI.exe process terminates.
- Winlogon.exe can also load additional network provider DLLs that need to perform secondary authentication (multiple network providers to gather identification and authentication information all at one time during normal logon.).

System processes

Winlogon, LogonUI, and Userinit

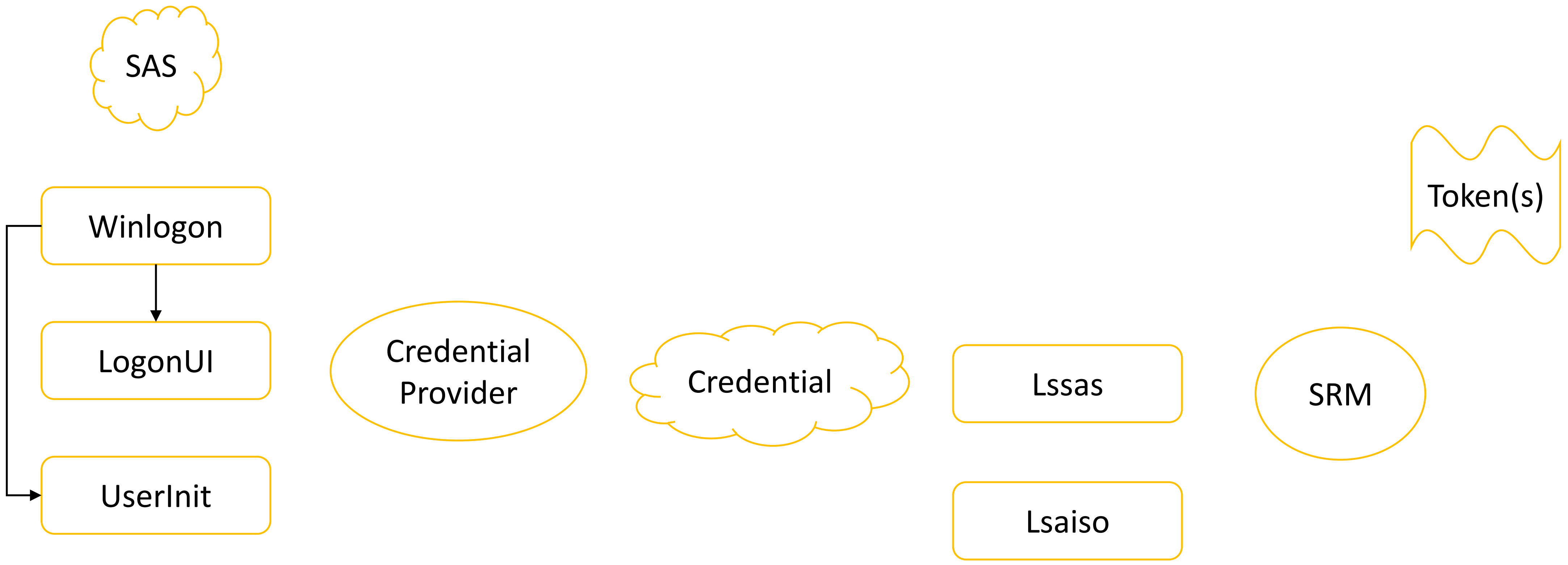
- Provided information are sent to the Local Security Authentication Service process (Lsass.exe) to be authenticated.
- Lsass.exe calls the appropriate authentication package, implemented as a DLL, to perform the actual verification
- If Credential Guard is enabled, and this is a domain logon, Lsass.exe will communicate with the Isolated LSA Trustlet (Lsaiso.exe)
- Upon successful authentication, Lsass.exe calls a function in the SRM (for example, `NtCreateToken`) to generate an access token object that contains the user’s security profile.
- If User Account Control (UAC) is used and the user logging on is a member of the administrators group or has administrator privileges, Lsass.exe will create a second, restricted version of the token.

System processes

Winlogon, LogonUI, and Userinit

- This access token is then used by Winlogon to create the initial process(es) in the user's session.
- The initial process(es) are stored in the `Userinit` registry value under the HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon registry key. The default is Userinit.exe, but there can be more than one image in the list.
- Userinit.exe performs some initialization of the user environment:
  - running the login script
  - reestablishing network connections
  - looking in the registry at the Shell value (under the same Winlogon key mentioned previously) and creates a process to run the system-defined shell (by default, Explorer.exe)

System processes


SAS
Winlogon
LogonUI
UserInit
Credential Provider
Credential
Lssas
Lsaiso
SRM
Token(s)