

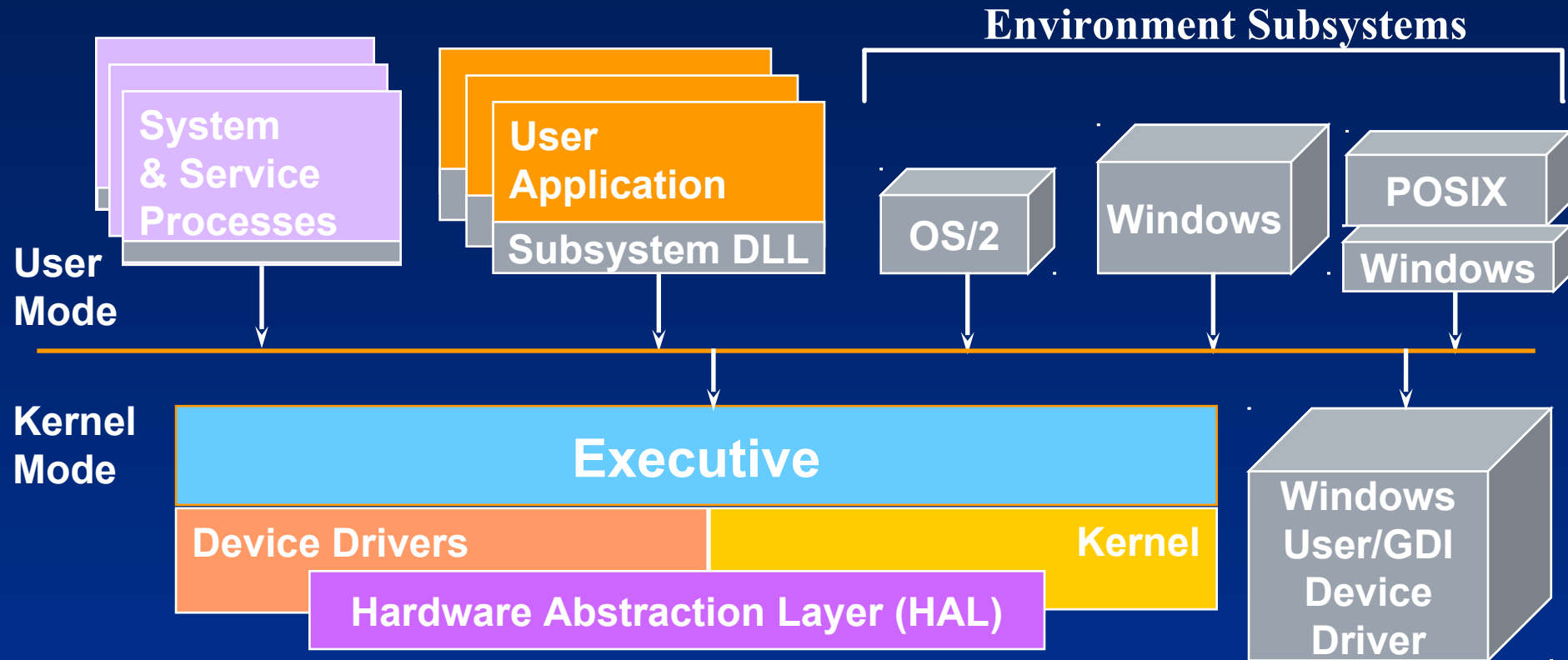
Unit 2: Operating System Principles

2.3. Windows on Windows - OS Personalities

Roadmap for Section 2.3.

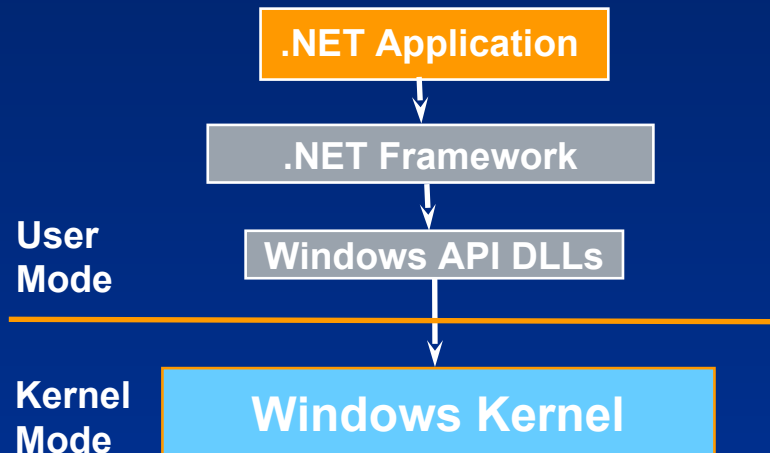
- Environment Subsystems
- System Service Dispatching
- Windows on Windows – 16 bit
(running 16-bit apps on 32-bit OS)
- Windows on Windows – 64 bit
(running 32-bit apps on 64-bit OS)

Multiple OS Personalities



What about .NET and CLR?

- .NET is a software framework for writing apps on Windows, which provides:
 - A large class library known as Framework Class Library (FCL)
 - An application virtual machine, called Common Language Runtime (CLR)
- .NET Framework is built on standard Windows APIs
 - It is not a subsystem
 - It does not call undocumented Windows system calls



.NET Framework

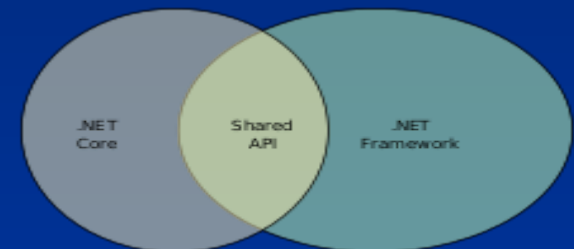
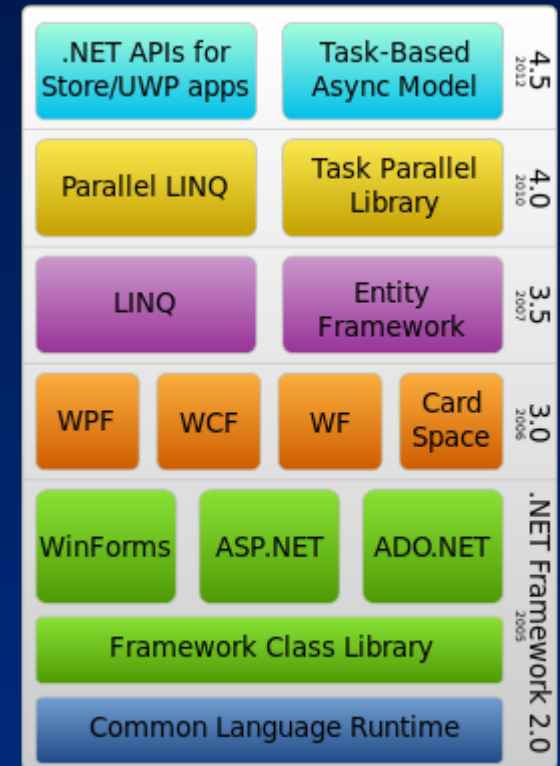
• .NET Framework components and history →

• Alternative implementations:

- Mono
- .NET Micro Framework
- .NET Core

• .NET Core:

- It is a cross-platform free and open-source managed software framework similar to .NET Framework
- Version 1.0 was released on 27 June 2016
- It consists of CoreCLR, a complete cross-platform runtime implementation of CLR, the virtual machine that manages the execution of .NET programs, and CoreFX, which is a partial fork of FCL
- It shares a subset of .NET Framework APIs, but it comes also with its own API that is not part of .NET Framework



Environment Subsystems

- Three environment subsystems originally provided with NT:
 - Windows – Windows API (originally 16-bit, then 32-bit, now also 64-bit)
 - OS/2 - 1.x character-mode apps only
 - Removed in Windows 2000
 - Posix - only Posix 1003.1 (bare minimum Unix services - no networking, windowing, threads, etc.)
 - Removed in Windows XP/Server 2003 – an enhanced version ships as Services For Unix 3.0 with Windows Server 2003 R2
 - A new and enhanced variant was introduced in Windows 10 version 1607, called Windows Subsystem for Linux (WSL)
- Of the three, Windows provides access to the majority of OS native functions
- Of the three, Windows is required to be running
 - System crashes if Windows subsystem process (CSRSS.EXE) exits
 - POSIX and OS/2 subsystems are actually Windows applications
 - POSIX & OS/2 start on demand (first time an app is run)
 - Stay running until system shutdown

Environment Subsystems

- Environment subsystems provide exposed, documented interface between application and Windows native API
 - Each subsystem defines a different set of APIs & semantics
 - Subsystems implement these by invoking native APIs
 - i.e., subsystem “wraps” and extends Windows native API
 - Example: Windows CreateFile in Kernel32.Dll calls native NtCreateFile
- .exe's and .dll's you write are associated with a subsystem
 - Specified by LINK /SUBSYSTEM option
 - Cannot mix calls between subsystems



Subsystem Information in Registry

- Subsystems configuration and startup information is in:

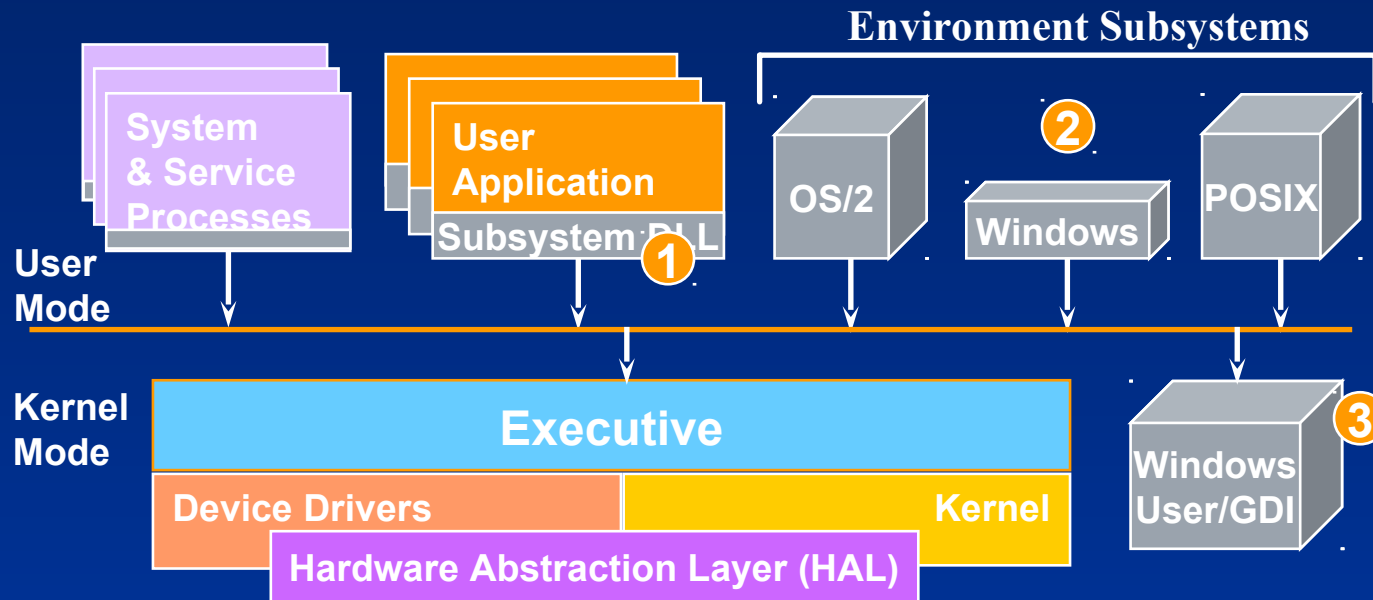
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control
 \Session Manager\SubSystems

Values:

- Required - list of value names for subsystems to load at boot time
- Optional - list of value names for subsystems to load when needed
- Windows - value giving filespec of Windows subsystem (csrss.exe)
 csrss.exe Windows APIs required - always started when Windows boots
- Kmode - value giving filespec of Win32K.Sys
 (kernel-mode driver portion of Windows subsystem)
- Posix - file name of POSIX subsystem
 psxss.exe Posix APIs optional - started when first Posix app is run
- Some Windows API DLLs are in “known DLLs” registry entry:
 HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs
 - Files are opened as mapped files
 - Improves process creation/image startup time

Subsystem Components

- ① API DLLs
 - for Windows: Kernel32.DLL, Gdi32.DLL, User32.DLL, etc.
- ② Subsystem process
 - for Windows: CSRSS.EXE (Client Server Runtime SubSystem)
- ③ For Windows only: kernel-mode GDI code
 - Win32K.SYS - (this code was formerly part of CSRSS.EXE)



Role of Subsystem Components

1. API DLLs

- Export the APIs defined by the subsystem
- Implement them by calling Windows “native” services, or by asking the subsystem process to do the work

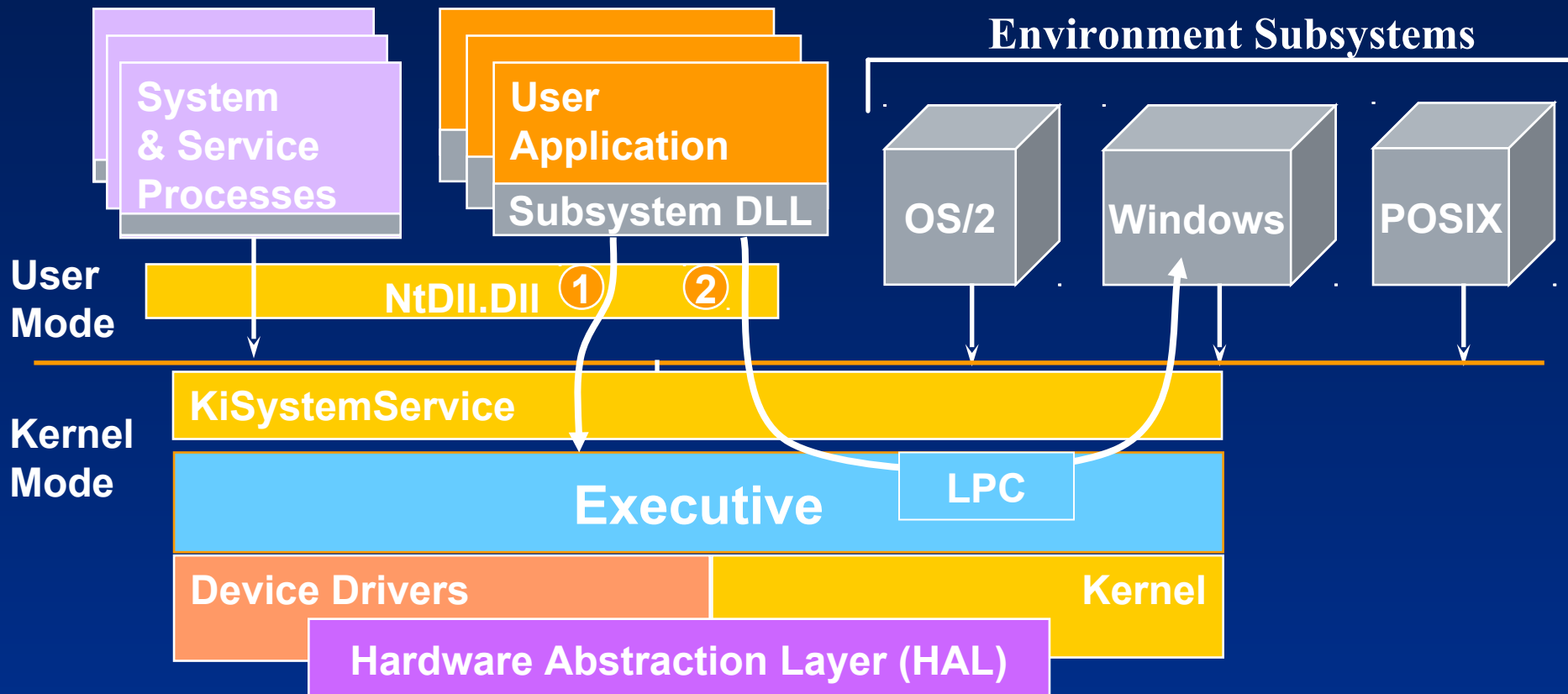
2. Subsystem process

- Maintains global state of subsystem
- Implements a few APIs that require subsystem-wide state changes
 - Processes and threads created under a subsystem
 - Drive letters
 - Window management for apps with no window code of their own (character-mode apps)
 - Handle and object tables for subsystem-specific objects

3. Win32K.Sys

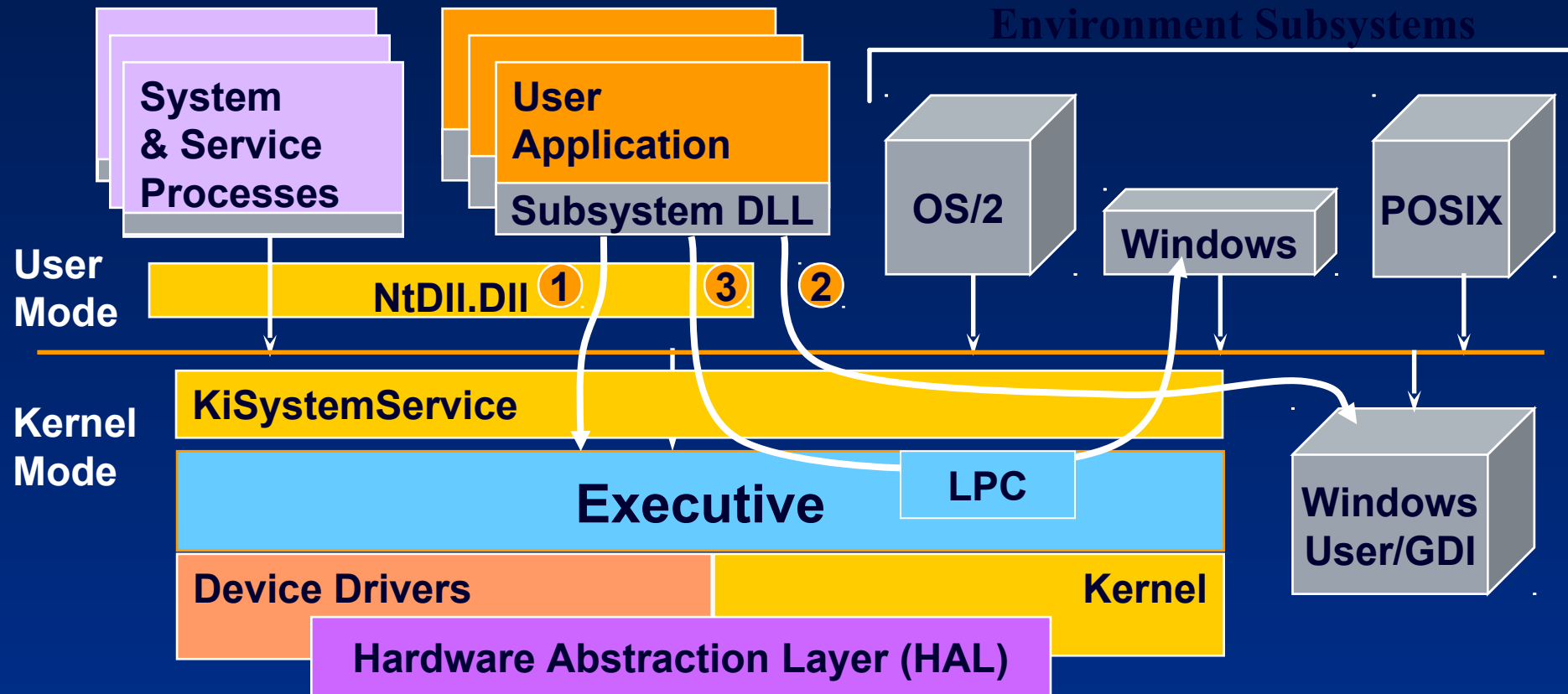
- Implements Windows User & GDI functions; calls routines in GDI drivers
- Also used by Posix and OS/2 subsystems to access the display

Simplified Architecture (3.51 and earlier)



- ① most Windows Kernel APIs
- ② all other Windows APIs, including User and GDI APIs

Windows Simplified Architecture



- ① most Windows Kernel APIs
- ② most Windows User and GDI APIs (these were formerly part of CSRSS.EXE)
- ③ a few Windows APIs

Role of CSRSS.EXE

(Windows Subsystem Process)

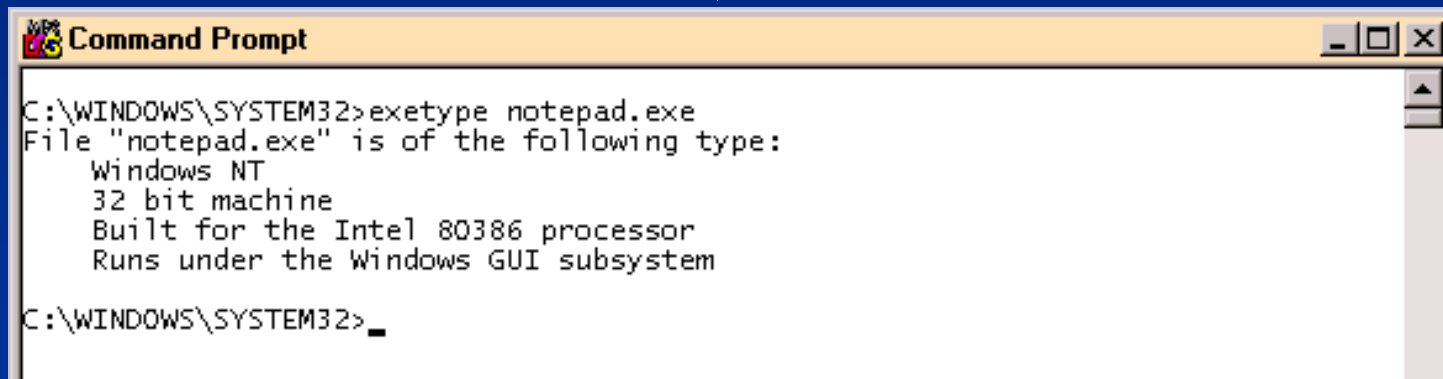
- A few Windows APIs are implemented in this separate process
 - In 3.51 and earlier:
 - Nearly all User and GDI APIs were implemented in CSRSS
 - CSRSS had a thread for every application thread that created a window
 - GDI drivers (video, printer) were user mode, mapped into this process
 - This was done for protection, esp. to keep GDI drivers in user mode
- CSRSS in NT 4.0 and later: role is greatly diminished
 - Maintains system-wide state information for all Windows “client” processes
 - Several Windows services LPC to CSRSS for “setup and teardown” functions
 - Process and thread creation and deletion
 - Get temporary file name
 - Drive letters
 - Security checks for file system redirector
 - Window management for console (character cell) applications ...
 - ... including NTVDM.EXE

Header of Executable File Specifies Subsystem Type

- Subsystem for each .exe specified in image header
 - see winnt.h (in Platform SDK)

```
IMAGE_SUBSYSTEM_UNKNOWN      0    // Unknown subsystem
IMAGE_SUBSYSTEM_NATIVE      1    // Image doesn't require a subsystem
IMAGE_SUBSYSTEM_WINDOWS_GUI  2    // Windows subsystem (graphical app)
IMAGE_SUBSYSTEM_WINDOWS_CUI  3    // Windows subsystem (character cell)
IMAGE_SUBSYSTEM_OS2_CUI      5    // OS/2 subsystem
IMAGE_SUBSYSTEM_POSIX_CUI    7    // Posix subsystem
```

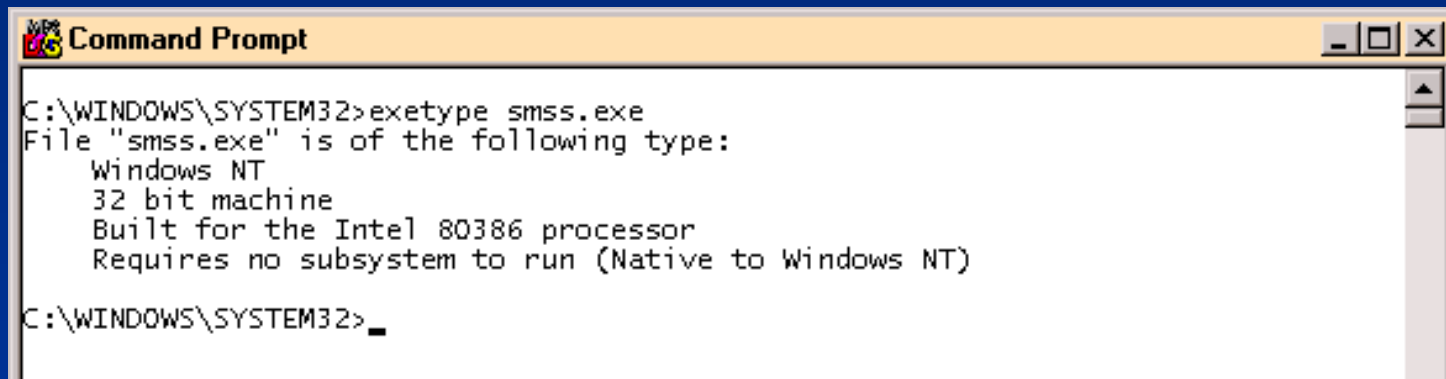
- or exetype image.exe (Windows 2000 Resource Kit)



```
Command Prompt
C:\WINDOWS\SYSTEM32>exetype notepad.exe
File "notepad.exe" is of the following type:
  Windows NT
  32 bit machine
  Built for the Intel 80386 processor
  Runs under the Windows GUI subsystem
C:\WINDOWS\SYSTEM32>
```

Native Images

- .EXEs not linked against any subsystem
 - Interface to Windows executive routines directly via NTDLL.DLL
- Two examples:
 - smss.exe (Session Manager -- starts before subsystems start)
 - csrss.exe (Windows subsystem)



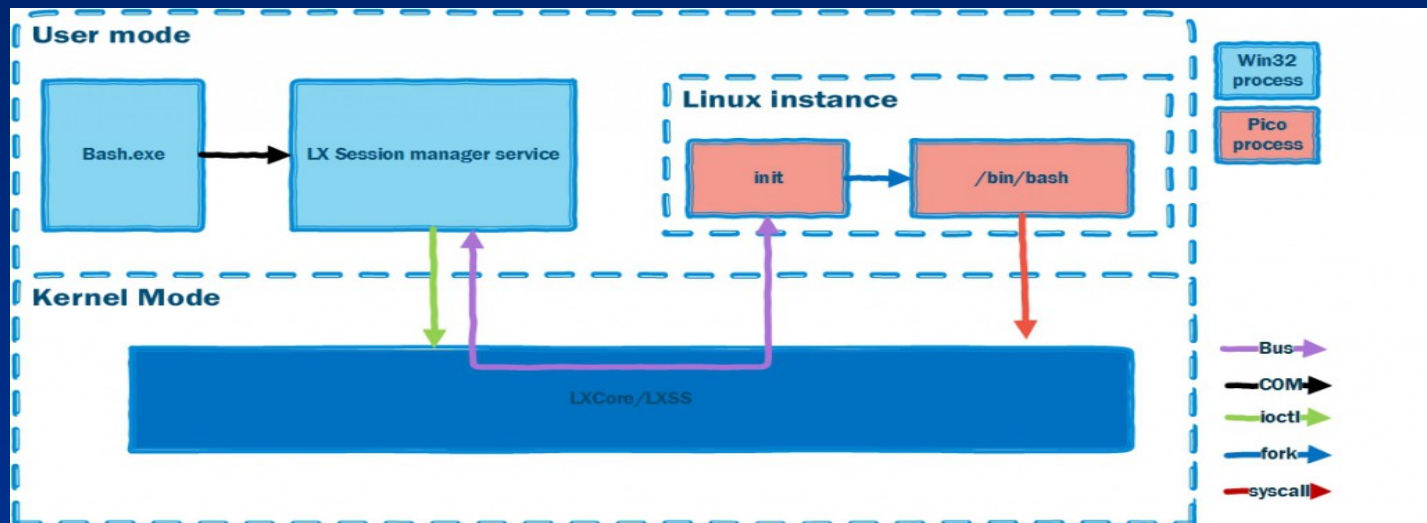
```
Command Prompt
C:\WINDOWS\SYSTEM32>exetype smss.exe
File "smss.exe" is of the following type:
    Windows NT
    32 bit machine
    Built for the Intel 80386 processor
    Requires no subsystem to run (Native to Windows NT)
C:\WINDOWS\SYSTEM32>
```

POSIX.1 Subsystem

- Original POSIX subsystem implemented only POSIX.1
 - ISO/IEC 9945-1:1990 or IEEE POSIX standard 1003.1-1990
 - POSIX.1 compliance as specified in Federal Information Processing Standard (FIPS) 151-2 (NIST)
 - POSIX Conformance Document in \HELP in Platform SDK
- Support for implementation of POSIX.1 subsystem was mandatory for NT
 - fork service in NT executive
 - hard file links in NTFS
- Limited set of services
 - such as process control, IPC, simple character cell I/O
 - POSIX subsystem alone is not a complete programming environment
- POSIX.1 executable cannot
 - create a thread or a window
 - use remote procedure calls (RPCs) or sockets

Windows Subsystem for Linux

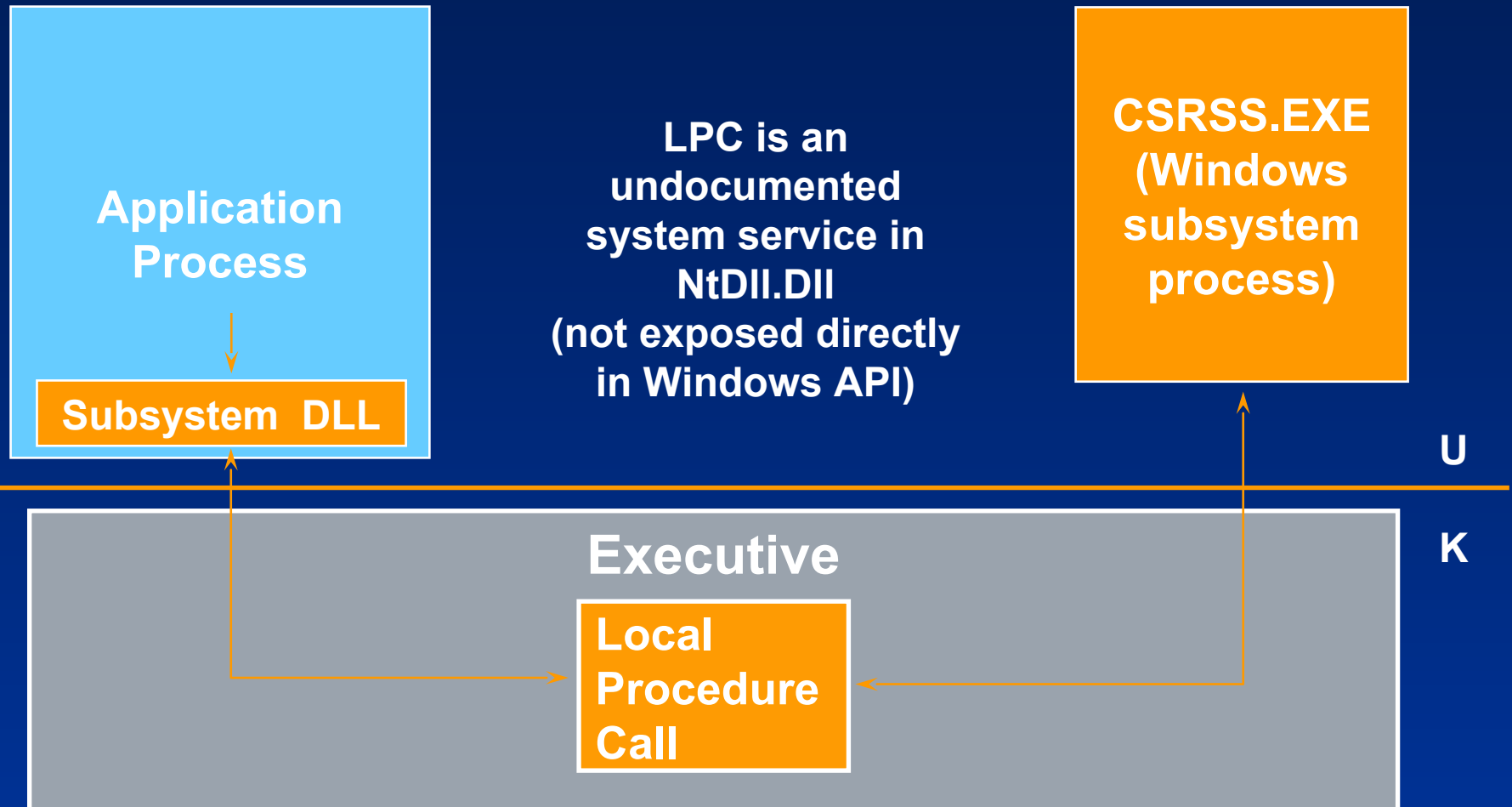
- WSL is a new and enhanced alternative for the POSIX subsystem, introduced in Windows 10 version 1607. As of version 1709 is not considered beta anymore.
- WSL is a collection of components that enables native Linux ELF64 binaries to run on Windows. It contains both user mode and kernel mode components, of which the primary ones are:
 - User mode session manager service that handles the Linux instance life cycle
 - Pico provider drivers (lxss.sys, lxcore.sys) that emulate a Linux kernel by translating Linux syscalls
 - Pico processes that host the unmodified user mode Linux (e.g. /bin/bash)



References: <https://blogs.msdn.microsoft.com/wsl/> and <https://msdn.microsoft.com/commandline/wsl/about>

Invoking (a few) Windows Services

- ◆ Some system calls still require communication with the Windows subsystem process





System Call Dispatching

- NTDLL.DLL provides interface for native system calls

Dependency Walker - [notepad.exe]

File Edit View Window Help

NOTEPAD.EXE

- COMDLG32.DLL
- SHELL32.DLL
- MSVCRT.DLL
- ADVAPI32.DLL
- KERNEL32.DLL
- NTDLL.DLL**
- GDI32.DLL
- USER32.DLL

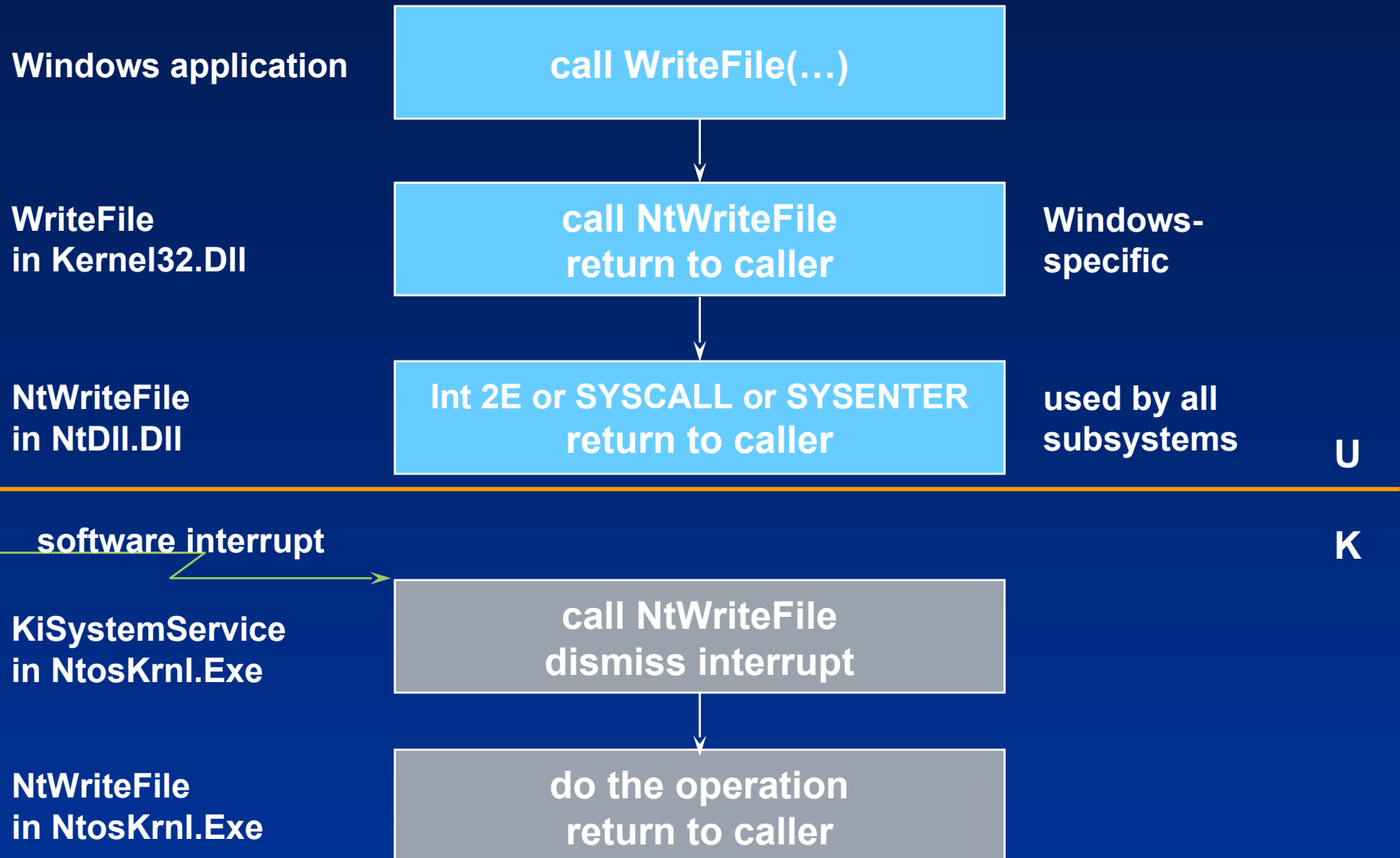
| Ordinal ^ | Hint | Function |
|-----------|------------|--------------------------|
| N/A | 1 (0x0001) | CsrAllocateCaptureBuffer |
| N/A | 3 (0x0003) | CsrAllocateMessagePort |
| N/A | 4 (0x0004) | CsrCaptureMessageBuffer |
| N/A | 5 (0x0005) | CsrCaptureMessageString |
| N/A | 7 (0x0007) | CsrClientCallServer |

| Ordinal ^ | Hint | Function |
|------------|--------------|-----------------------------|
| 1 (0x0001) | 0 (0x0000) | ?Allocate@CBufferAlloc |
| 2 (0x0002) | 266 (0x010A) | PropertyLengthAsVariant |
| 3 (0x0003) | 310 (0x0136) | RtlCompareVariants |
| 4 (0x0004) | 315 (0x013B) | RtlConvertPropertyToVariant |
| 5 (0x0005) | 320 (0x0140) | RtlConvertVariantToProperty |

| Module ^ | Time Stamp | Size | Attributes | Machine | Subsystem |
|--------------|----------------|---------|------------|-----------|---------------|
| GDI32.DLL | 05/01/97 1:00a | 165,648 | A | Intel x86 | Native |
| KERNEL32.DLL | 05/01/97 1:00a | 372,496 | A | Intel x86 | Win32 console |
| MSVCRT.DLL | 05/09/97 1:00a | 280,576 | A | Intel x86 | Win32 GUI |
| NOTEPAD.EXE | 10/13/96 9:38p | 45,328 | A | Intel x86 | Win32 GUI |

For Help, press F1

Example: Invoking a Windows API





Invoking System Functions from User Mode

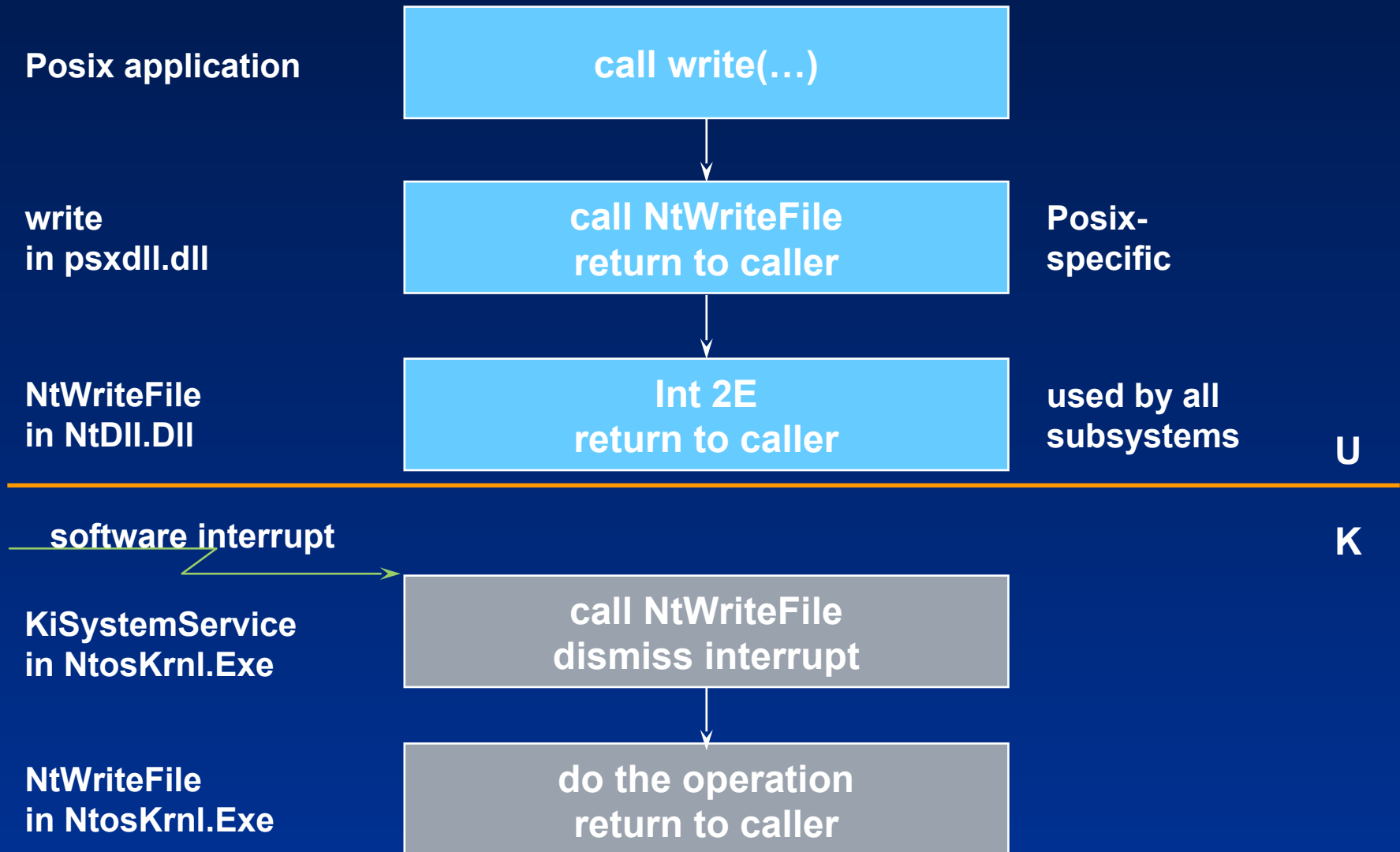
- Kernel-mode functions (“services”) are invoked from user mode via a protected mechanism
 - x86: INT 2E (as of Windows XP, faster instructions are used where available: SYSENTER on Intel x86, SYSCALL on AMD)
 - i.e., on a call to an OS service from user mode, the last thing that happens in user mode is this “change mode to kernel” instruction
 - Causes an exception or interrupt, handled by the system service dispatcher (KiSystemService) in kernel mode
 - Return to user mode is done by dismissing the interrupt or exception
- The desired system function is selected by the “system service number”
 - Every Windows function exported to user mode has a unique number
 - This number is stored in a register just before the “change mode” instruction (after pushing the arguments to the service)
 - This number is an index into the system service dispatch table (SSDT)
 - This table gives kernel-mode entry point address and argument list length for each exported function



Invoking System Functions from User Mode

- All validity checks are done after the user to kernel transition
 - KiSystemService probes argument list, copies it to kernel-mode stack, and calls the executive or kernel routine pointed to by the table
 - Service-specific routine checks argument values, probes pointed-to buffers, etc.
 - Once past that point, everything is “trusted”
- This is safe, because:
 - The system service table is in kernel-protected memory; and
 - The kernel mode routines pointed to by the system service table are in kernel-protected memory; therefore:
 - User mode code can't supply the code to be run in kernel mode; it can only select from among a predefined list
 - Arguments are copied to the kernel mode stack before validation; therefore:
 - Other threads in the process can't corrupt the arguments “out from under” the service

Example: Invoking a Posix API



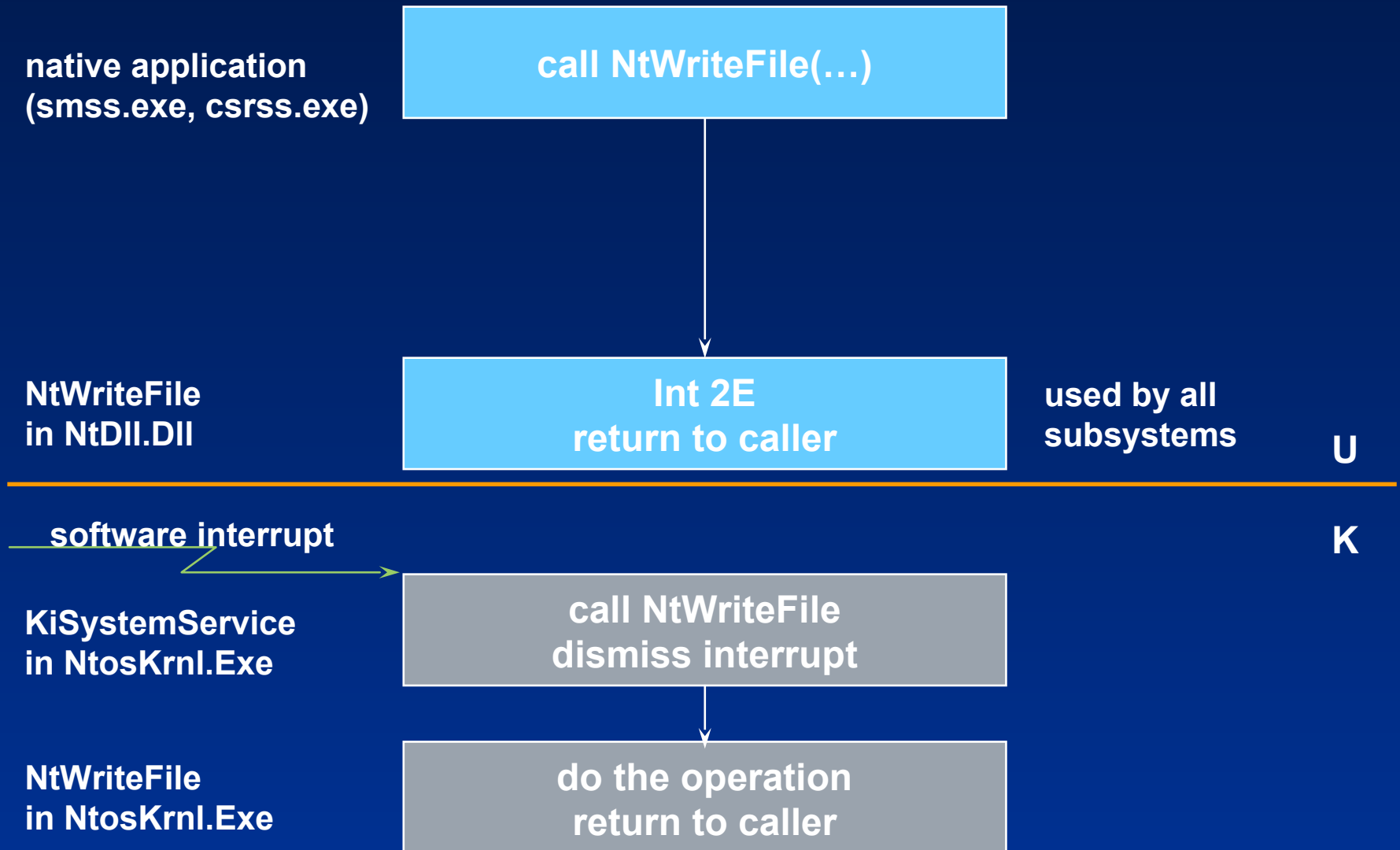
Native API – exported by Ntdll.dll

- A set of +450 functions starting with “Nt” (most have “Zw” aliases) – an interface to the Windows executive system services (i.e., Windows system calls)
 - Most of these are user-mode callable (e.g. through Windows API) routines that have the same function names and arguments as the kernel-mode routines they invoke
 - e.g. NtWriteFile in Ntdll.dll invokes NtWriteFile in NtosKrnI.Exe
 - Majority are not supported or documented
 - Some are (partially) documented in the Platform SDK:
 - NtQuerySystemInformation, NtQuerySystemTime, NtQueryInformationProcess, NtQueryInformationThread, NtCreateFile, NtOpenFile, NtWaitForSingleObject, etc.
 - The DDK describes most of them as “Zw” routines (such as ZwReadFile)
 - These entry points call the corresponding “Nt” interface via the system call interface
 - Thus, “previous mode” is kernel-mode, which means no security checks
 - Kernel-mode code could also call NtReadFile directly

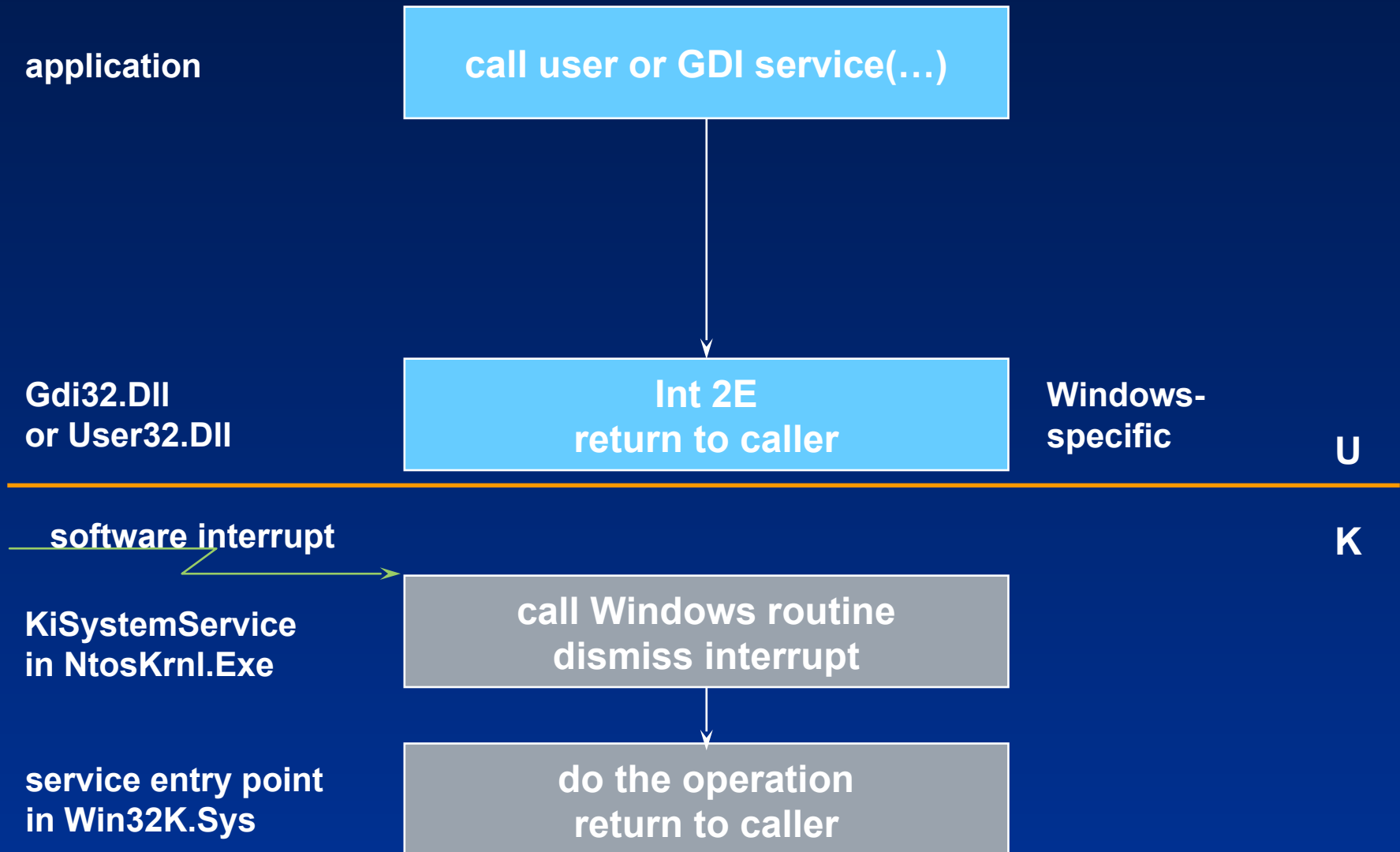
References: <https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/ntxxx-routines>

- Other user-mode callable routines – internal support functions used by subsystems, subsystems DLLs, and other native applications (i.e. native images)
 - “Rtl” functions – the (extended) C run-time library routines
 - “Csr” functions – CSRSS support routines (for communications with CSRSS.EXE process)
 - “Dbg” functions – debugging functions (such as a software breakpoint)
 - “Ldr” functions – image loader functions for PE file handling and starting of new processes
 - “Etw” functions – event tracing for Windows routines (introduced in Windows Server 2003)
 - some other groups of support routines

Calling a “Native” API from User Mode

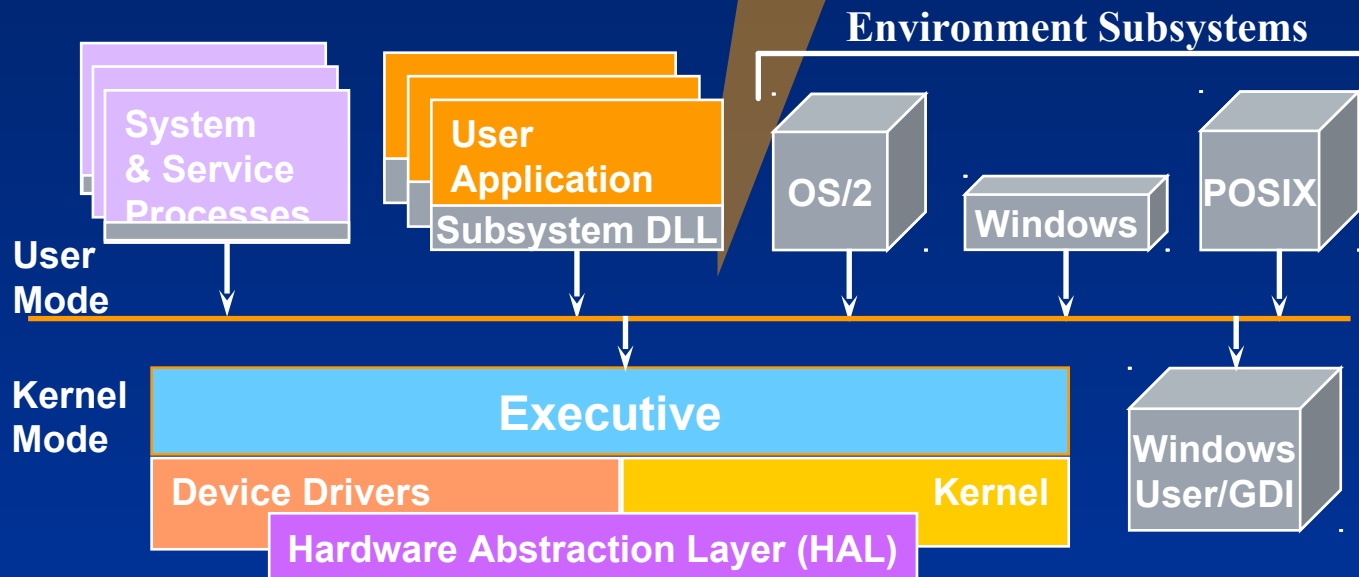
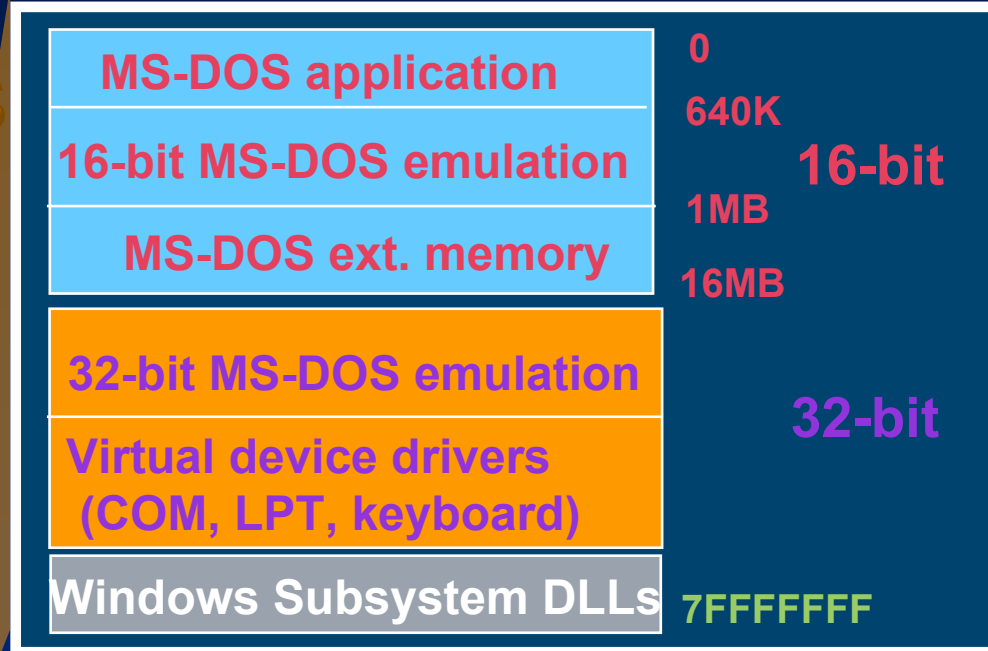


Invoking (most) User and GDI Services



16-bit Applications on 32-bit Windows

- Windows runs NTVDM.EXE (NT Virtual Dos Machine)
 - NTVDM is a Windows image
 - No “DOS subsystem” or “Win16 subsystem”

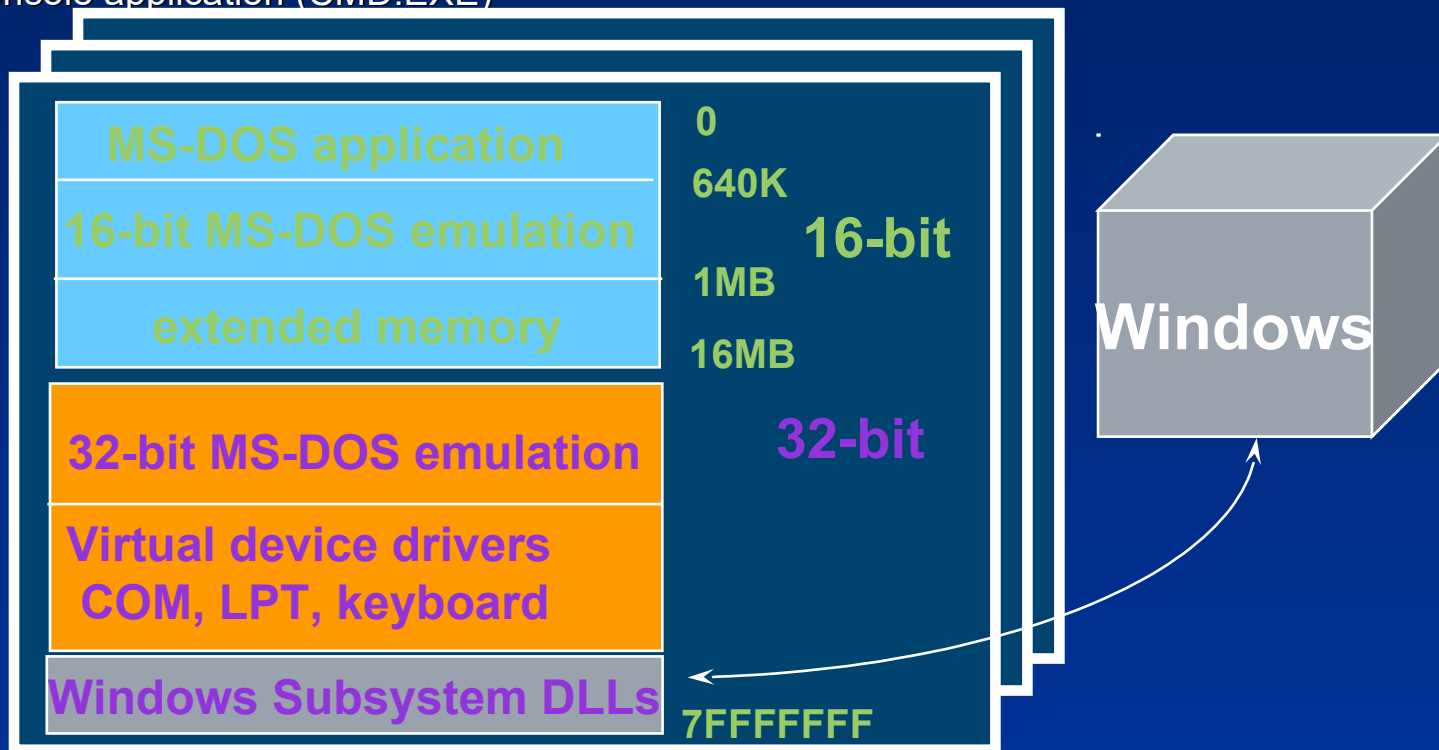




1) DOS 16-bit Applications

e.g. command.com, edit.com (NT4 had qbasic.exe)

- Windows runs NTVDM.EXE (NT Virtual DOS Machine)
 - See \System\CurrentControlSet\Control\WOW\cmdline
- Each DOS app has a separate process running NTVDM
 - DOS & Windows 16-bit drivers not supported
 - Note: Windows “command prompt” is not a “DOS box”, despite icon; it’s a Windows console application (CMD.EXE)



Example:
three DOS
apps
running in
three NTVDM
processes

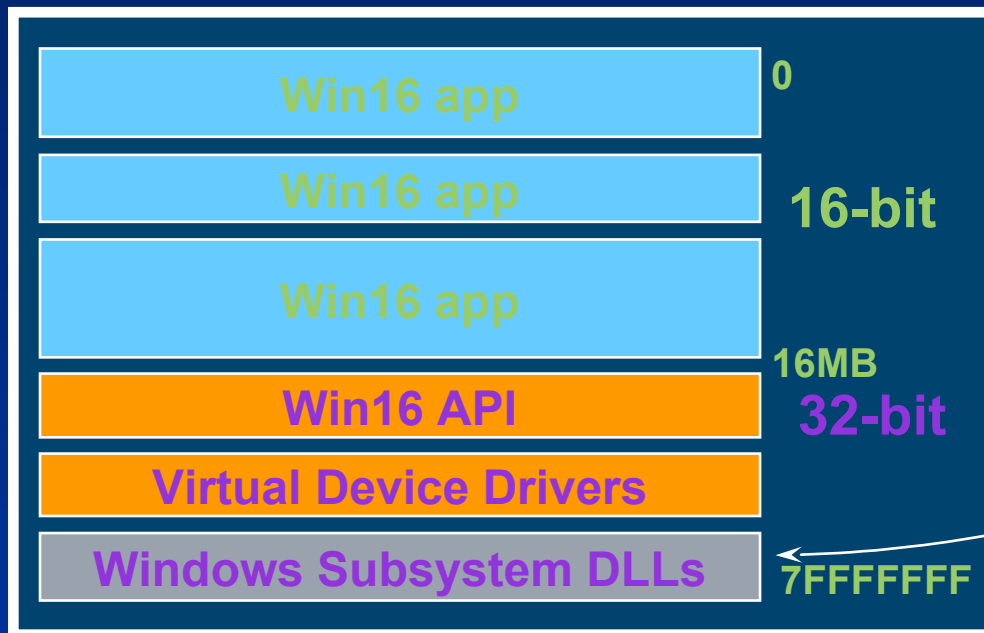


2) Windows 16-Bit Applications

e.g. sysedit.exe, winhelp.exe

- Windows also runs NTVDM.EXE
 - See \CurrentControlSet\Control\WOW\wowcmdline
- NTVDM loads wowexec.exe
 - WOW = “Windows on Windows”
 - Win16 calls are translated to Win32 (Windows API)

Example:
three Win16
apps (and
wowexec.exe)
running in
one NTVDM
process





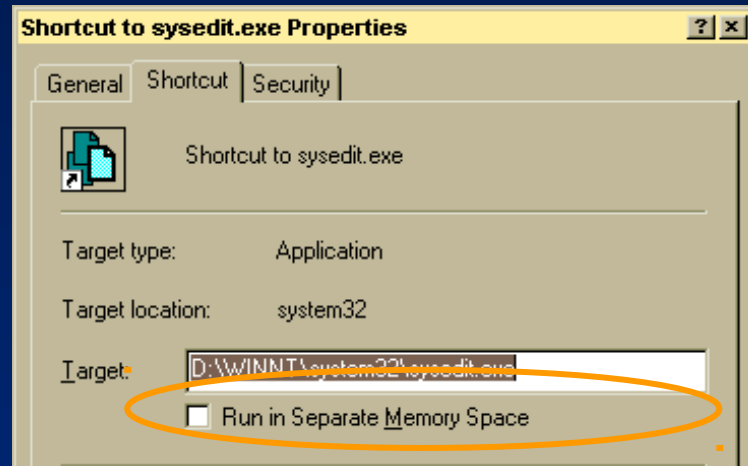
Windows 16-bit Applications Multitasking Details

By default:

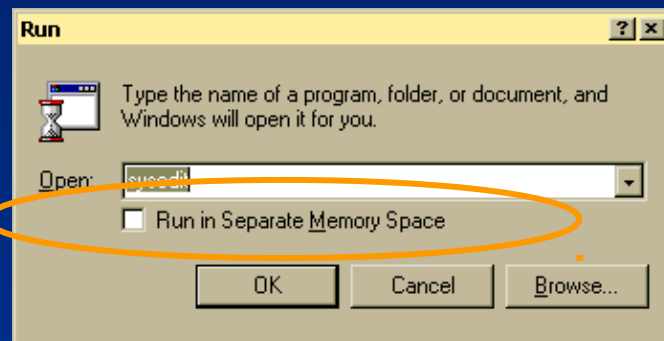
- Each Win16 app runs in a separate thread in the common NTVDM process
- They cooperatively multitask among themselves (Win16 Yield API)...
- ...and the one (if any) that wants to run, preemptively multitasks with all other threads on Windows
- necessary to meet serialization assumptions of some Win16 apps

Option to run Win16 apps in separate VDMs

- “Run in Separate Memory Space” = run in separate process
- default set by \CurrentControlSet\Control\WOWDefaultSeparateVDM
- Win16 apps run this way preemptively multitask with all other threads, including the un-Yield’ed thread in a shared Win16 NTVDM (if any)



NT4 only:





Monitoring 16-bit Applications

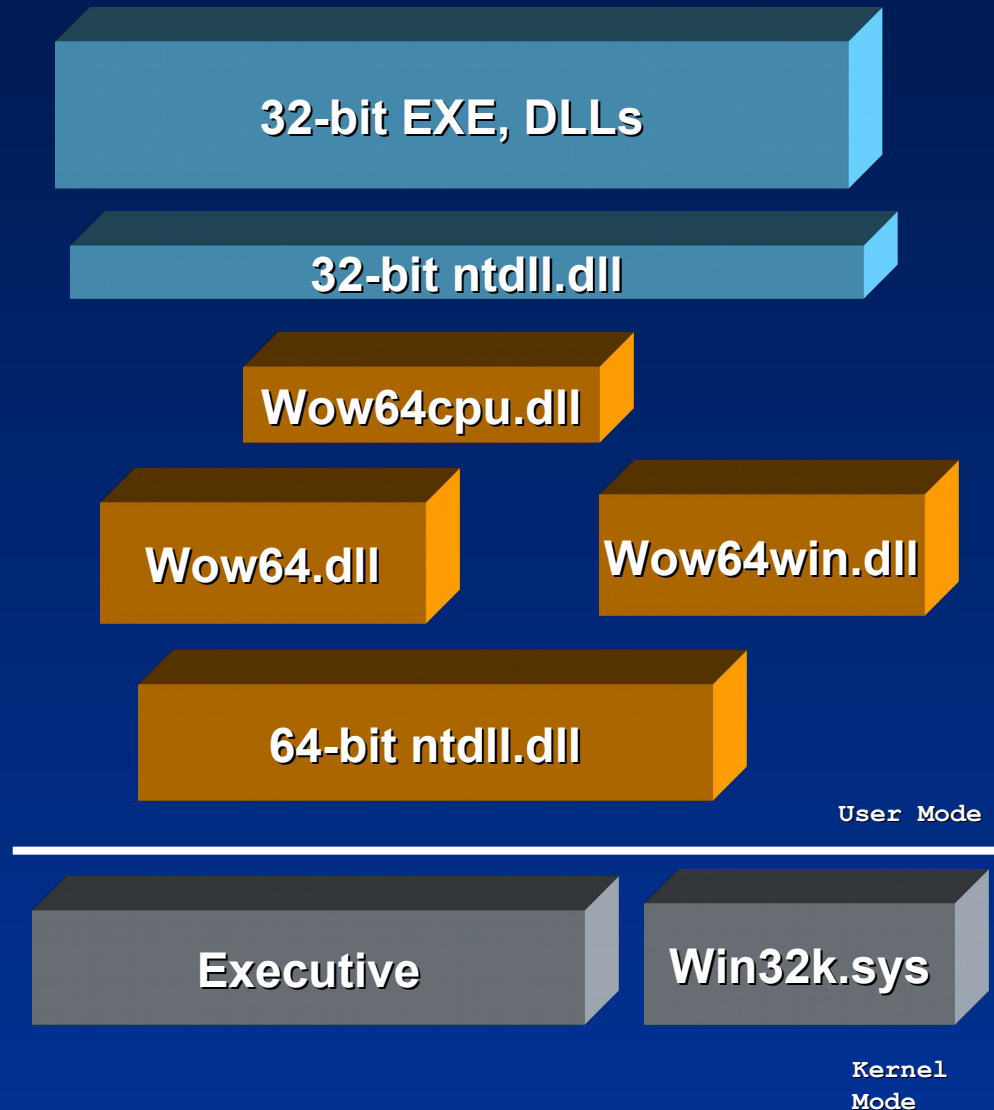
- To most of Windows, an NTVDM process is just another process
- Task Manager
 - “tasks” are simply the names of top-level windows - Win16 windows included
 - “processes” display identifies Win16 apps within NTVDM processes
 - by reading the NTVDM process’s private memory (undocumented interface)
 - does not identify the DOS apps within each NTVDM process
- TLIST (resource kit)
 - does identify the DOS apps within each NTVDM process (by window title)
 - but for a shared Win16 NTVDM process, only shows one window title
- QuickView, exetype
 - identifies DOS, Win16, etc., application .exe’s

Wow64

- Allows execution of Win32 binaries on 64-bit Windows
 - Wow64 intercepts system calls from the 32-bit application
 - Converts 32-bit data structure into 64-bit aligned structures
 - Issues the native 64-bit system call
 - Returns any data from the 64-bit system call
- *IsWow64Process()* function can tell a 32-bit process if it is running under Wow64
- Performance
 - On x64, instructions executed by hardware
 - On IA64, instructions have to be emulated
 - New Intel IA-32 EL (Execution Layer) does binary translation of Itanium to x86 to speed performance
 - Downloadable – bundled with Server 2003 SP1

Wow64 Components

- Wow64.dll - provides core emulation infrastructure and thunks for Ntoskrnl.exe entry-point functions; exception dispatching
- Wow64win.dll - provides thunks for Win32k.sys entry-point functions
- Wow64cpu.dll – manages thread contexts, supports mode-switch instructions



Wow64 Limitations

- Cannot load 32-bit DLLs in 64-bit process and vice versa
- Does not support 32-bit kernel mode device drivers
 - Drivers must be ported to 64-bits
 - Special support required to support 32-bit applications using DeviceIoControl to driver
 - Driver must convert 32-bit structures to 64-bit

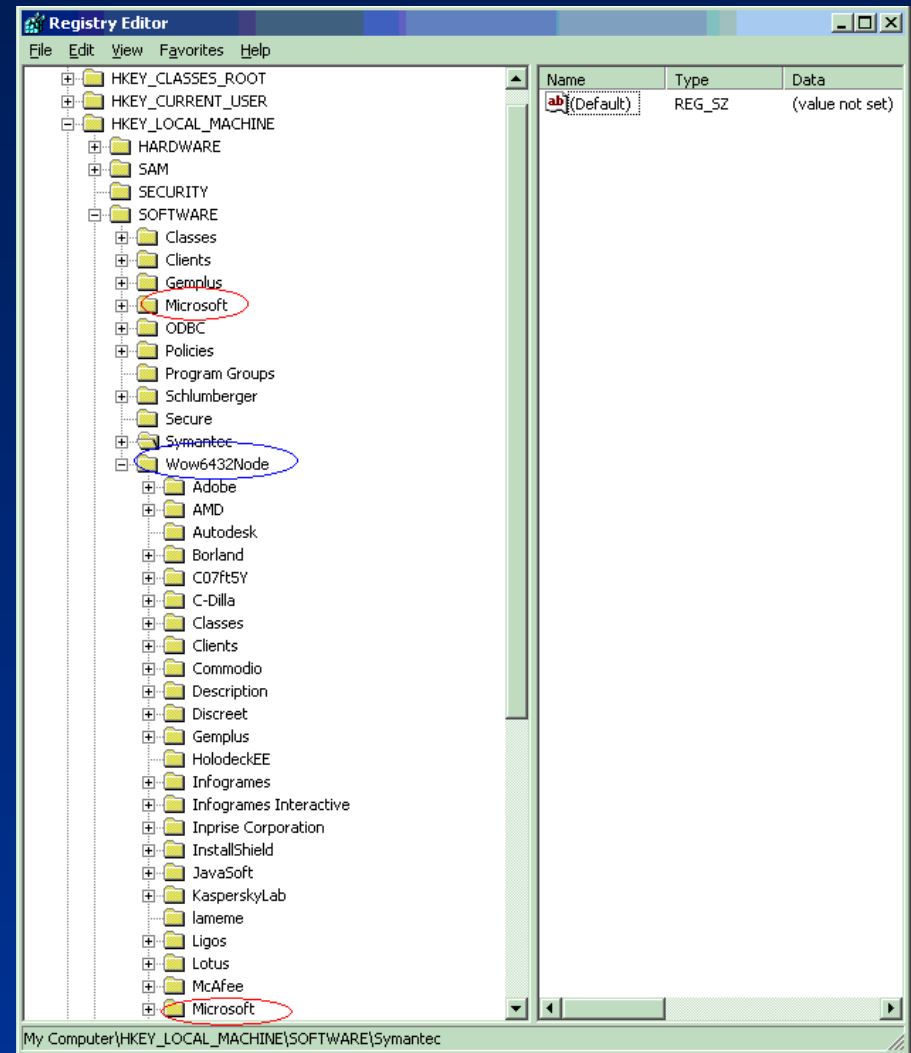
| Wow64 Feature Support on 64-bit Windows | Platforms | |
|---|-------------------------|-----|
| | IA64 | x64 |
| 16-bit Virtual DOS Machine (VDM) support | N/A | N/A |
| Physical Address Extension (PAE) APIs | N/A | Yes |
| GetWriteWatch() API | N/A | Yes |
| Scatter/Gather I/O APIs | N/A | Yes |
| Hardware accelerated with DirectX version 7,8 and 9 | Software-Emulation Only | Yes |

Wow64 File Locations

- Location of system files
 - 64-bit system files are in \windows\system32
 - 32-bit system files are in \windows\syswow64
 - 32-bit applications live in “\Program Files (x86)”
 - 64-bit applications live in “\Program Files”
- File access to %windir%\system32 redirected to %windir%\syswow64
- Two areas of the registry redirected (see next slide)

Wow64 Registry Redirection

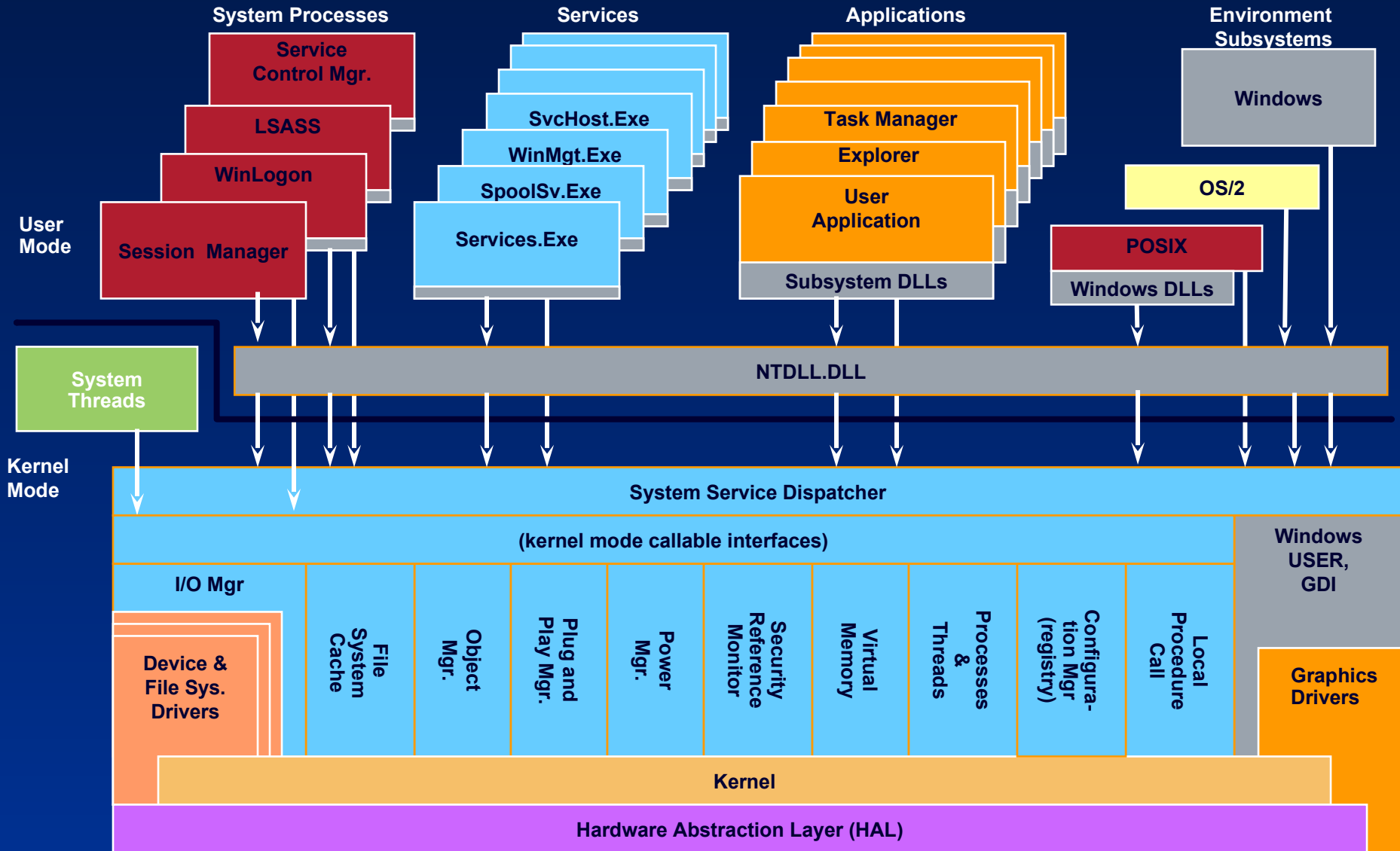
- Two registry keys have 32-bit sections:
 - HKEY_LOCAL_MACHINE\Software
 - HKEY_CLASSES_ROOT
 - Everything else is shared
- 32-bit data lives under \Wow6432Node
 - When a Wow64 process opens/creates a key, it is redirected to be under Wow6432Node



Four Contexts for Executing Code

- Full process and thread context:
 - User applications
 - Windows Services
 - Environment subsystem processes
 - System startup processes
- Have thread context but no “real” process:
 - Threads in “System” process
- Routines called by other threads/processes:
 - Subsystem DLLs
 - Executive system services (NtReadFile, etc.)
 - GDI32 and User32 APIs implemented in Win32K.Sys (and graphics drivers)
- No process or thread context
 - (“arbitrary thread context”)
 - Interrupt dispatching
 - Device drivers

Windows Architecture



hardware interfaces (buses, I/O devices, interrupts, interval timers, DMA, memory cache control, etc., etc.)

Original copyright by Microsoft Corporation. Used by permission.

Further Reading

- Pavel Yosifovich, Alex Ionescu, et al., “Windows Internals”, 7th Edition, Microsoft Press, 2017.
- Chapter 2 - System Architecture
 - Environment Subsystems and Subsystem DLLs (from pp. 104)
 - NTDLL.DLL (from pp. 115)
 - Executive (from pp. 118)