**Windows Internals 6th Edition: Book Notes (Windows 7 & Server 2008-R2)** *Version 2.2*

**EPROCESS**: KPROCESS, \*PEB (User), PID, Parent, \*SAT, \*Handles, \*W32PROCESS (User), \*EJOB, \*Session, \*Station, \*VAD

**KPROCESS**: \*ETHREAD-List, Affinity, KernelTime, UserTime, BasePriority.

**PEB** (User): ModuleList, \*ImageBaseAddress, \*Heap, Subsystem, SessionID, \*RTL\_USER\_PROCESS\_ PARAM…

**RTL\_USER\_PROCESS\_PARAMETERS**: CommandLine, StdIn, StdOut, StdErr, PWD, EnvironmentVars.

**ETHREAD**: KTHREAD, PID, \*EPROCESS, \*SAT, \*PendingIRPs, \*W32THREAD.

**KTHREAD**: \*TEB, \*KernelStack, \*UserStack, \*ServiceTable, \*KTRAP\_FRAME, WaitList, MutantList, APC-List.

**TEB** (User): TIB (User), \*PEB (User), ThreadID, \*ThreadLocalStorage.

**TIB** (User): StackBase, FiberData

Csrss.exe maintains its own list of Windows subsystem processes and threads (object types?).

Win32k.sys maintains W32THREAD list for threads calling USER or GDI functions.

Win32k.sys maintains W32PROCESS list for processes calling USER or GDI functions.

KTHREAD is also called a "Thread Control Block (TCB)"; KPROCESS also called a "Process Control Block (PCB)".

KTHREAD also has a dispatcher header which lists the objects the thread is waiting for (objects "signal").

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ARCHITECTURE \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Executive** (upper ntoskrnl.exe) = system services in Windows API (Nt for ntdll, Zw from kernel), configuration manager (Cm), process and thread manager (Ps), security reference monitor (Se), I/O manager (Io), PnP manager (Pp), power manager (Po), cache and memory manager (Cc,Mm), event tracing (Etw), transaction manager (Tm), kernel debugger library (Kd); object manager (Ob), ALPC (Alpc), run-time library (Rtl), synchronization (resouces, fast mutexes, pushlocks).

**Kernel** (lower ntoskrnl.exe) = thread scheduling, kernel threads, interrupt and exception dispatching, synchronization (mutexes, events, semaphores, timers), APC and DPC objects, interrupt objects, and KPCR; the processor control region (KPCR) contains CPU's interrupt table (IDT), task-state segment (TSS), global descriptor table (GDT), and the kernel processor control block (KPRCB), which contains current/next/idle threads, dispatcher database, DPC queue, CPU identifiers, NUMA data, time accounting, CPU statistics (KPCR address in *gs* register on x64, in *fs* register on x86).

**Windows Subsystem** = csrss.exe; win32k.sys (session); kernel32.dll, advapi32.dll, user32.dll, gdi32.dll (in-process).

**Ntdll.dll** = executive system services wrapper, csrss.exe wrapper, image loader, heap manager, event tracing, user APC, Rtl\*; so a stack trace might go from program.exe 🡪 kernel32.dll 🡪 ntdll.dll 🡪 ntoskrnl.exe; however, USER and GDI calls do not involve ntdll.dll at all, they go through user32.dll and gdi32.dll to ntoskrnl.exe and win32k.sys directly.

**Process Tree** = smss.exe creates session 0, loads itself, runs csrss.exe, runs wininit.exe, then wininit runs services.exe, lsass.exe and lsm.exe, then that smss exits; smss.exe creates session 1, loads itself, runs csrss.exe and winlogon.exe, then winlogon runs logonui.exe (after ctrl-alt-del) and userinit.exe, userinit runs explorer.exe, then that smss exits.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* INTERRUPTS, EXCEPTIONS, SYSTEM CALLS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Interrupt Dispatch Table (IDT)** defines trap handlers for hardware interrupts, but also for software interrupts, exceptions and system calls (which are handled as exceptions). 256 IRQs with an APIC controller. Each CPU has its own IDT. HAL imposes an IRQ Level (**IRQL**) priority scheme: 0-31 on x86, 0-15 on x64. 0 = Passive/Low/Normal-Threads, 1 = APC, 2 = DPC/Dispatch, 3+ = Hardware Interrupts. CPU's IRQL determines which interrupts it can receive (lower IRQs are masked away). Only non-paged memory can be accessed at DPC/Dispatch (2) or higher. KINTERRUPT objects for device drivers specify IRQL, IDT entry, and service routine. Each CPU has a DPC queue. At IRQL 1 (APC level), page faults, Windows API calls, and requesting object handles are all permitted (unlike at DPC/Dispatch IRQL 2). Each thread has its own attached APC queue, while each CPU core has its own DPC queue. APCs always execute in the context of a particular thread, while DPCs are independent of any thread and may only execute non-paged system space code. If a process does not handle its own exception, then the OS will (ntdll.dll starts most new threads). User-mode code always runs at IRQL 0.

A thread runs for its *quantum* of time until another thread of the same or higher priority level is ready. Thread priorities are 0 - 31. At a given IRQL, threads compete by priority, but a higher IRQL thread will interrupt lower IRQL threads, no matter their priorities. A kernel thread can raise the IRQL.

On x64, a system call number is put in EAX, then *syscall* (*sysenter* on x86). EAX stores call number in the **System Service Dispatch Table (SSDT)**, which on x64 contains offsets to the routines (x86 uses absolute addresses) plus the number of arguments the system call takes (it's a 32-bit number = 28 bits of offset + 4 bits of arguments information).

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*OBJECTS, HANDLES, SESSIONS, MUTEXES\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

Only the object manager can create handles or resolve handles to their targets. A user-mode process must obtain a handle before its threads can use an object. Each EPROCESS has its own handle table. The **kernel handle table** is accessible only from kernel mode, but from within any process; it is the handle table of the System process. A handle includes a pointer, access mask, and inheritable flag.

Four types of objects: kernel, executive, USER and GDI. Every object has a header: handle count, pointer count, ACL, type, optional name, and optional methods for the object manager. The object manager handles the kernel and executive objects, while USER/GDI objects are handled by a separate mini object manager inside win32k.sys.

USER and GDI objects support only one handle per object. A GDI handle is private to the process which created it. Handles to USER objects are public to all processes, but the ACL on these objects limit access. Processes in one session cannot reference a USER handle in another session.

Services use the global namespace of all objects. User sessions are transparently redirected to per-session copies of \BaseNamedObjects, \Windows, and \DosDevices (the other directories are inaccessible to users). For a user, \Sessions\x\DosDevices is a symbolic link to **\GLOBAL??**. If an object is found or created in the per-session copy, then that object is used; if it is not found, then the real directories are searched. With the exception of section objects, a process must have SeCreateGlobalPrivilege to create a global object. An object path prefixed with "**\??**" tells the object manager to redirect to the per-session directory first. The "\Global" prefix indicates that the per-session copy should not be used. Session directories have ACLs to secure them from other session users. An object referenced through the Windows API must be under the \?? directory; the other WINOBJ.EXE directories are for lower-level use. (Is \?? the same as \GLOBAL?? -- not sure) To specify a long literal Unicode file path, prepend "\\?\" to the path,

e.g., "\\?\c:\folder" or "\\?\UNC\server\share". To specify a path to a device object, prepend with "\\.\" as in

"\\.\CdRom0".

A **mutex** is signaled when it is not owned, nonsignaled when it is owned by a thread; a thread opens a handle to the mutex and then calls a wait function until the thread can own the mutex. A mutex can only be used in kernel mode, while a **mutant** can be used in both kernel and user mode. A **semaphore** maintains a counter, between max and zero, which is decremented with each new waiting thread, until zero is reached and no additional threads can wait on it. An **event** can be (non)signaled manually by a thread or automatically by the OS. A **timer** signals after a period of time.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*SECURITY\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Kernel Patch Protection** (PatchGuard) crashes on changes to ntoskrnl.exe, hal.dll, ci.dll, kdcom.dll, pshed.dll, clfs.sys, ndis.sys, tcpip.sys, GDT, IDT, SSDT, MSRs, kernel stacks, object type definitions, various debugging function pointers, cache of exception directories, PatchGuard's own code, and a few other things. **Code Integrity** checks signatures of drivers at load time and also the hashes of their pages in memory; Microsoft and other CAs are supported; fully implemented on x64, but not on x86 systems where policies can be set to disable it entirely or just for PnP drivers.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*MEMORY\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

On x64, **pages** are either 4KB or 2MB in size; most kernel images use large pages for TLB optimization. Pages of virtual memory are either commited, reserved or free. Committed pages are either private or mapped to section objects, which are shareable. **Section objects** are the basis of both shared memory and file mappings.

Processes are protected from each other by 1) preventing user-mode threads from accessing kernel memory, 2) preventing user-mode threads in one process from accessing non-shared memory in other processes, 3) pages have read/write/execute access masks, and 4) section objects have ACLs.

Nonpaged and paged "**pools**" are kernel-mode heaps, and they are mapped into the system space of every process. There can be multiple pool objects of each type, not just one of each, and are dynamically sized. Pool allocations can be "tagged" to identify the driver or source of the allocation (pooltag.txt).

A "**look-aside list**" is like a pool, but faster, contains only fixed-sized blocks, kernel only (true?), and never uses spinlocks.

Allocating pages can be wasteful. The heap manager handles small allocations within a larger set of pages (or a section object) called a "**heap**". Each process has at least one heap, but more can be created, and heaps are dynamically sized. Heaps built on section objects can be shared across processes or between kernel and user mode (with restrictions). Heaps can exist in user mode ("heap") and kernel mode ("pool"). The heap manager, which is a memory manager client, handles multi-thread synchronization to heap contents, (de)committing memory, free memory tracking, metadata integrity checking, and heap resizing. The C runtime uses the heap manager when calling *malloc* and *free*.

The virtual address space of an x86 32-bit process is 4GB, while the x64 64-bit space is 16EB (17.2 billion GB), though not all of x64 virtual space is currently addressable due to hardware and software limitations (only 48 bits used today):

x86 user space = 0x00000000 - 0x7FFFFFFF

x86 system space = 0x80000000 - 0xFFFFFFFF (session space is a part of system space)

x64 user space = 0x0000000000000000 - 0x7FFFFFFFFFFFFFFF

x64 system space = 0x8000000000000000 - 0xFFFFFFFFFFFFFFFF (session space is a part of system space)

Each process in a "**session**" has an associated per-session csrss.exe, winlogon.exe, win32k.sys, object manager namespace, window station, desktop(s), and GUI windows. A paged pool in system space is mapped into the session space of every process sharing the same session ID number (optionally, shareable section objects too).

**Working Set (WS)** = subset of virtual pages resident in physical memory. Three types of WS: system, per-session, and per-process. Hard fault requires disk access, a soft fault only requires a virtual memory mapping to a page already in physical memory. WS of a process can actually get to zero, e.g., winlogon.exe.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*MEMORY ADDRESS TRANSLATION\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

The **Page Frame Number (PFN)** database describes each page of physical memory. PFN entries identify each physical page as Active/Valid, Standby, Modified, Modified No Write, Bad, Free, or Zeroed. PFN entries also keep a reference count pointing towards them, such as when part of a working set; and when the count goes to zero, the PFN can be freed or put on standby. Each PFN also has the virtual address of the PTE pointing to it. Standby PFNs are ranked into eight priorities, 0 - 7 (lower number = next to be freed/zeroed).

Each EPROCESS has a **Virtual Address Descriptor (VAD)** tree to track which addresses have been reserved. When a process allocates memory, a new node in the VAD tree is created to define the address range, as shared/private, inheritable, and its protection bits (read-only, read/write). VAD information is used to help create PTEs as needed.

32-bit PAE kernel uses a three-level page table, 32-bit non-PAE only two, and x64 uses a four-level table.

A virtual address is divided into regions: map level 4 (x64 only), directory pointer (32-bit PAE/x64 only), directory index, table index, and byte index (*4PDTB*). **CR3 register** contains physical address of the topmost table for the current process; this physical address is read from KPROCESS to update CR3 when switching to a new process. The page directory (*D*) is composed of **Page Directory Entries (PDEs)**, each of which contains the PFN of a page table (*T*). The page table (*T*) is composed of **Page Table Entries (PTEs)**, each of which contains the PFN for the virtual address of the *page*. Then the byte index (*B*) is the offset of the byte within that physical page. Note that page faults trigger PTE creation, not reservation or commitment. PTE contains a PFN and also flags to indicate valid, read/write, kernel mode only (owner flag), part of system space (global flag), copy on write, accessed, dirty, no caching, etc.

Page tables (*T*) for system space are shared among all processes, while the tables for session space are shared only among processes in the same session. More than one page table (*T*) is needed to map the entire address space of a process. A **Prototype PTE** is a layer of indirection between various processes' page tables (*T*) and the PFN database; it's used to manage virtual pages which are shareable across multiple processes, such as for section objects, memory-mapped files and swap file contents, hence, a process' PTE might point to a Prototype PTE, not to a PFN entry.

The **Translation Lookaside Buffer (TLB)** is the CPU's cache of virtual-to-physical page mappings, plus the PTE flags. The **Memory Management Unit (MMU)** is a *hardware* component involved in virtual memory page address translation.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*CACHE MANAGER\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

NTFS streams are cached in system space 256KB "views", each represented by a Virtual Address Control Block (**VACB**). **Fast I/O** is satisfying read/write requests entirely in cache instead of creating an IRP. Cached memory for unmapped views is not zeroed, it goes into the standby list (the "System Cache" in Task Manager and Process Explorer is the sum of the system working set, standby list and modified page list all together).

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*SESSIONS, WINDOW STATIONS & DESKTOPS\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

A session may contain multiple stations, and a station may contain multiple desktops (lsass.exe creates “logon sessions”, but these are different). The Service-0x0-3e7$ window station is for any non-interactive service that runs as Local System (Winsta0 for interactive), Service-0x0-3e5$ is for Local Service, Service-0x0-3e4$ for Network Service, and msswindowstation\mssretricteddesk for Microsoft Search Indexer. All processes in the same session map their session space to the same pages. **Session space** has has four areas: image space (win32k.sys and its per-session data, video drivers), structure (per-session working set list), view space (desktop heap, mapped views), and paged pool. A window **station** includes a clipboard and a global atom table. Each session ID number is unique, but window station and desktop names can be identical across sessions. Each session can have only one window station (Winsta0) which can interact with a display, keyboard and mouse. Each process is associated with a particular session\station\desktop path (see its handles). Once a process or thread opens a graphical window object, that process or thread is then stuck to the station and desktop containing the window object. Non-interactive stations are created for services that require a station’s context, but shouldn’t, because a service should not need a desktop. **Winsta0** will have at least three desktops: winlogon, disconnect and default. Window messages can only be sent between processes on the same desktop, including messages delivered via hooks. Only one desktop can be active at a time, i.e., receive keyboard and mouse input. USER objects are allocated from the desktop heap, while GDI objects are allocated from the session pool.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*I/O MANAGER, DRIVERS, DEVICES\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

I/O Manager handles most requests as either **Fast I/O** requests (satisfied without an IRP, such as from cache) or as **I/O Request Packets (IRPs)** given to a driver as a work order. With layered drivers, the IRP is given to each driver in turn, then finally back to the I/O Manager.

Drivers are either user-mode (16-bit VDD, printer, UMDF) or kernel-mode (filesystem, PnP, non-PnP OS extensions). **Windows Driver Model (MDM)** drivers include ***bus drivers*** (PCI, USB, ISA, 1394) which present a device to a ***function driver*** via the PnP Manager, while ***filter drivers*** may reside above or below the device's function driver. Some functional drivers are categorized into **class/port/miniport** relationships; for example, storage drivers are layered into class (disk.sys), port (storport.sys, scsiport.sys, ataport.sys) and miniport (vendor-supplied). The class and port drivers are Microsoft's, the miniport driver is from the vendor.

A device driver is mainly a collection of routines to be called by the I/O Manager for tasks like initialization (DriverEntry), adding PnP devices, IRP dispatch, Interrupt Service Routine (ISR), DPC data transfer, layered completion notification, Fast I/O, unload, and shutdown. The **Driver object** provides the addresses of these routines to the I/O Manager. The Driver object has a linked list of the **Device objects** for which it is responsible, and each of these Device objects has a pointer back to its Driver. IRP includes status of the request, the functions to be called for each layered driver necessary, and the target device. A Device object includes the current IRP, pending IRPs, and device-specific extensions.

A handle to a file or to a Device object is really a handle to a **File** type object, which virtualizes interaction with the real file or device. Interaction with a simple device is modeled on file interaction. In the handles of a process, a handle to a \Device\\* is listed as a handle of type File. A per-process File object has a list of IRPs, access flags, synchronization lock, and the current offset for the next read operation for that process.

PnP drivers create Device objects when the PnP Manager notifies them of new devices. Non-PnP drivers typically create Device objects at initialization. Device objects under the **\Device namespace** are inaccessible to the Windows API unless symbolic links are added underneath \Global??.

Filesystem recognizer driver (fs\_rec.sys) registers with I/O Manager to identify any new volume by its first sector. Disk.sys class driver creates device objects as \Device\Harddisk\*\DR\* (where \* = 0,1,2,3,…).

For **SMB**, srv.sys and srv2.sys are the server-side drivers; on the client, rdbss.sys is the port driver, while mrxsmb.sys is the miniport wrapper for mrxsmb10.sys and mrxsmb20.sys (and mrxdav.sys is the WebDAV miniport).

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*NETWORKING\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

In user mode, ws2\_32.dll implements the **Winsock API** and calls through the **Service Provider Interface (SPI)** to Winsock Service Providers (like mswsock.dll), also in user mode, for transport and name resolution services. Service Providers call protocol-specific ***helper libraries*** (like wshtcpip.dll) to call the kernel-mode **protocol drivers** (like tcpip.sys). Winsock maintains a catalog of its Service Providers (netsh.exe winsock show catalog). Winsock can also represent sockets as file handles (msafd.dll -> afd.sys -> tcpip.sys) to integrate with the I/O Manager.

For kernel-mode clients, like http.sys, **Winsock Kernel (WSK)** replaces the legacy Transport Driver Interface (TDI) API.

On the **RPC** client side, rpcrt4.dll is the run-time library, implementing marshaling functions, local RPC, and it calls either Winsock or named pipe APIs for remote RPC (msrpc.sys is the client from within kernel mode). On the RPC server side, rpcss.dll in svchost.exe is a service, with rpcepmap.dll for end-point mapping.

To encapsulate HTTP and FTP usage, there are the **WinInet**, **WinHTTP**, and **HTTP Server API**s. Windows Explorer and IE use WinInet. WinHTTP is more advanced than WinInet. IIS uses HTTP Server API (httpapi.dll -> http.sys). HTTP Server API allows multiple services to share port 80/443 by each service registering its URLs.

For IPC over SMB, **named pipes** (reliable, bidirectional, unicast, npfs.sys, \Device\NamedPipe, \\*server*\Pipe\*pipename*) and **mailslots** (unreliable, unidirectional, unicast or broadcast, msfs.sys, \Device\Mailslot, \\*server*\Mailslot\*slotname*, or \\*domain*\Mailslot\*slotname*, or \\\*\Mailslot\*slotname* where \* means local domain). Named pipes have ACLs and support server-side impersonation of clients. Named pipe and mailslot functions implemented by kernel32.dll using familiar *CreateFile*, *ReadFile*, *WriteFile*, etc. Search on a pipe name to find the process which has a handle to it.

The ***Netbios*** function is implemented through netapi32.dll -> ntdll.dll -> netbios.sys -> netbt.sys -> tcpip.sys.

NDIS drivers do not process or use I/O Request Packets (IRPs).

A "provider" is a wrapper for a redirector for the purpose of determining which redirector should be used to access a share, printer or similar network resource; providers are polled to find the correct redirector for a given resource (Control Panel > Network Connections > Advanced menu > Provider Order tab). The polling is performed by both the **Multiple Provider Router** (mpr.dll), for when clients use the Windows Networking (WNet) API, and by the **Multiple UNC Provider** (mup.sys), for when clients use the standard Windows I/O API. The MPR is a DLL for loading into user processes, while the MUP is a kernel-mode filesystem driver. The SMB redirector includes the Workstation service, and the MPR provider of the SMB redirector is ntlanman.dll. There are also providers for RDP (drprov.dll) and WebDAV (davclnt.dll). The MPR mainly uses drive letter or device name symbolic links (like Z:) to associate a path with a redirector, while the MUP resolves UNC paths. A UNC path is prepended with "\Global??\UNC", which is actually a link to the "\Device\Mup" filesystem driver. Modern redirectors are actually mini-redirectors that register with the **Redirected Driver Buffering Subsystem** (rdbss.sys); it's the RDBSS which actually talks to MUP, I/O Manager and Cache Manager.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*GRAPHICS\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**DirectX** = Direct3D, DirectDraw, DirectMusic, DirectPlay, DirectSound (but mainly Direct3D). The Direct3D runtime in user processes is d3d\*.dll (like d3d10.dll), and the kernel drivers are dxapi.sys, dxgkrnl.sys and dxgmms1.sys.

**Windows Display Driver Model (WDDM)** is the DirectX driver model. WDDM virtualizes video memory so that system memory (and even paging files) can be used when video memory is exhausted.

Each GUI application draws itself on an offscreen surface (or "texture"), then **Desktop Window Manager** (dwm.exe) composits these surfaces together to form the desktop a user sees. DWM is built on top of Direct3D: the visible desktop is a fullscreen Direct3D application. GDI applications write to a buffer in system memory, this is translated into DirectX format and copied into video memory; but on Windows 7 and DirectX 11, there is partial hardware acceleration for GDI, so not all GDI surfaces must be held in system memory, which reduces dwm.exe memory utilization on Win7 over Vista.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*BIOS BOOT-UP AND LOGON (Not UEFI)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

1) BIOS reads MBR into memory, passes control to MBR.

2) Master Boot Record (MBR) = first sector of the entire hard drive, includes boot code and the partition table.

3) Boot Sector = first sector of the *system partition* identified from the partition table (MBR != Boot Sector).

4) Bootmgr: switches to protected mode, enables paging, reads \Boot\BCD, shows menu to choose.

5) Winload.exe: loads ntoskrnl.exe and hal.dll and their dependencies, reads SYSTEM hive, loads boot drivers.

6) Ntoskrnl.exe: Reads "loader parameter block" created by Winload; phase 0 init, then phase 1.

Phase 0: executive loaded, driver verifier, first DPC stack, Idle and System "processes", Idle starts Phase 1.

Phase 1: interrupts enabled, progress screen, start other cores, further executive initialization, driver and device object creation, boot driver init, system drivers loaded and initialized, then smss.exe is started.

7) Smss.exe: delayed file rename/delete, runs any BootExecute commands (like Autochk), initialize the rest of the registry, create section objects for known DLLs, creates session zero with csrss.exe/win32k.sys and wininit.exe.

8) Wininit.exe: creates initial window station and desktop; starts services.exe, lsass.exe and lsm.exe.

9) At user logon, smss.exe creates itself in session 1+, loads csrss.exe and winlogon.exe; winlogon.exe runs logonui.exe with credential provider DLLs; winlogon.exe runs userinit.exe; userinit.exe runs logon scripts, Group Policy and launches explorer.exe for the user; session 1+ smss.exe terminates, but the original session 0 smss.exe remains running.