TORRENS UNIVERSITY AUSTRALIA

Computer Architecture & OS

Module 07: Types, Features, and Function of Operating Systems

Dr. Hung Ta

History of MS-DOS and Windows (1 of 2)

Year	MS–DOS	MS-DOS based Windows	NT-based Windows	Modern Windows	Notes
1981	1.0				Initial release for IBM PC
1983	2.0				Support for PC/XT
1984	3.0				Support for PC/AT
1990		3.0			Ten million copies in 2 years
1991	5.0				Added memory management
1992		3.1			Ran only on 286 and later
1993	7.0		NT 3.1		Supported 32-bit x86, MIPS, Alpha
1995		95	NT 4.0		MS-DOS embedded in Win 95 NT supports PowerPC
1996			NT		NT has Windows 95 look and feel

Figure 11.1 Major releases in the history of Microsoft operating systems for desktop PCs.

History of MS-DOS and Windows (2 of 2)

Year	MS–DOS	MS-DOS based Windows	NT-based Windows	Modern Windows	Notes
1998	--	98	--	--	
2000	8.0	Me	2000	--	Win Me was inferior to Win 98 NT supports IA-64
2001	--	--	XP	--	Replaced Win 98. NT supports x64
2006	--	--	Vista	--	Vista could not supplant XP
2009	--	--	7	--	Significantly improved upon Vista
2012	--	--	--	8	First Modern version, supports ARM
2013	--	--	--	8.1	Fixed complaints about Windows 8
2015–2020	--	--	--	10	Unified OS for multiple devices Rapid releases every 6 months Reached 1.3 billion devices
2021	--	--	--	11	Fresh new UI Broader application support Higher security baseline

Figure 11.1 Major releases in the history of Microsoft operating systems for desktop PCs. (Continued)

DEC Operating Systems by Dave Cutler

Year	DEC operating system	Characteristics
1973	RSX-11M	16-bit, multiuser, real-time, swapping
1978	VAX/VMS	32-bit, virtual memory
1987	VAXELAN	Real-time
1988	PRISM/Mica	Canceled in favor of MIPS/Ultrix

Figure 11.2 DEC operating systems developed by Dave Cutler.

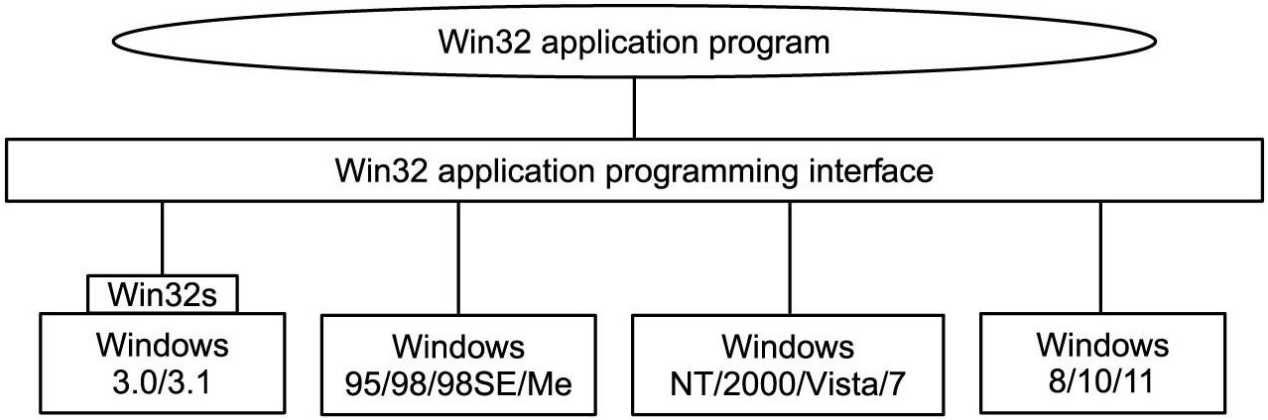

Figure 11.3 The Win32 API allows programs to run on almost all versions of Windows.

The Layers in Windows

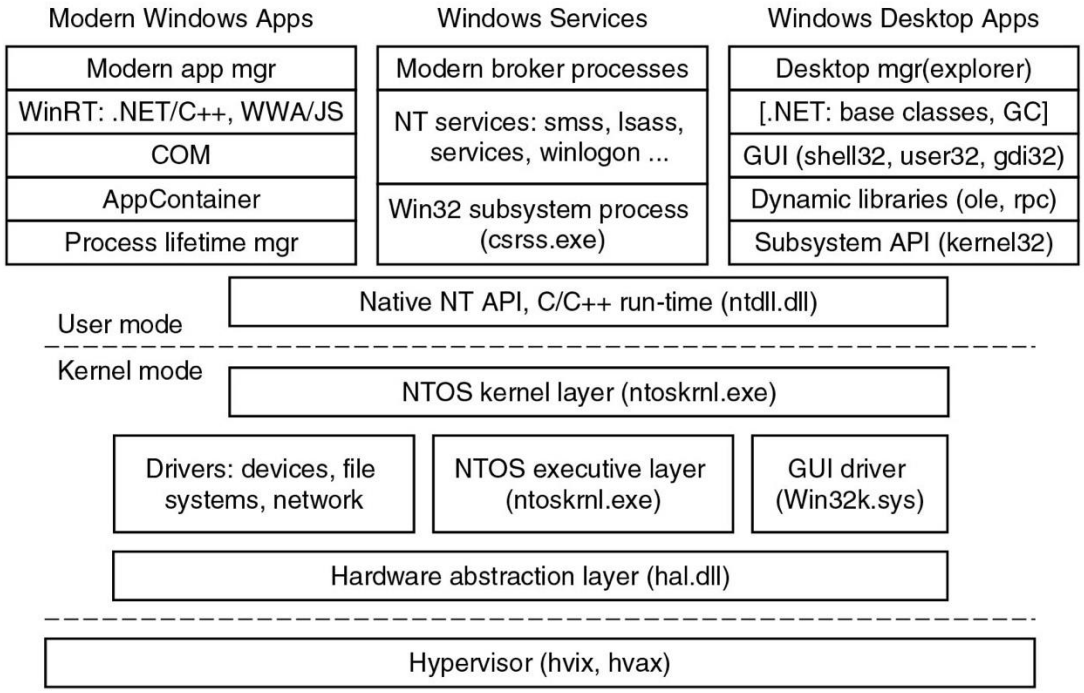

Figure 11.4 The programming layers in modern Windows.

NT Subsystems

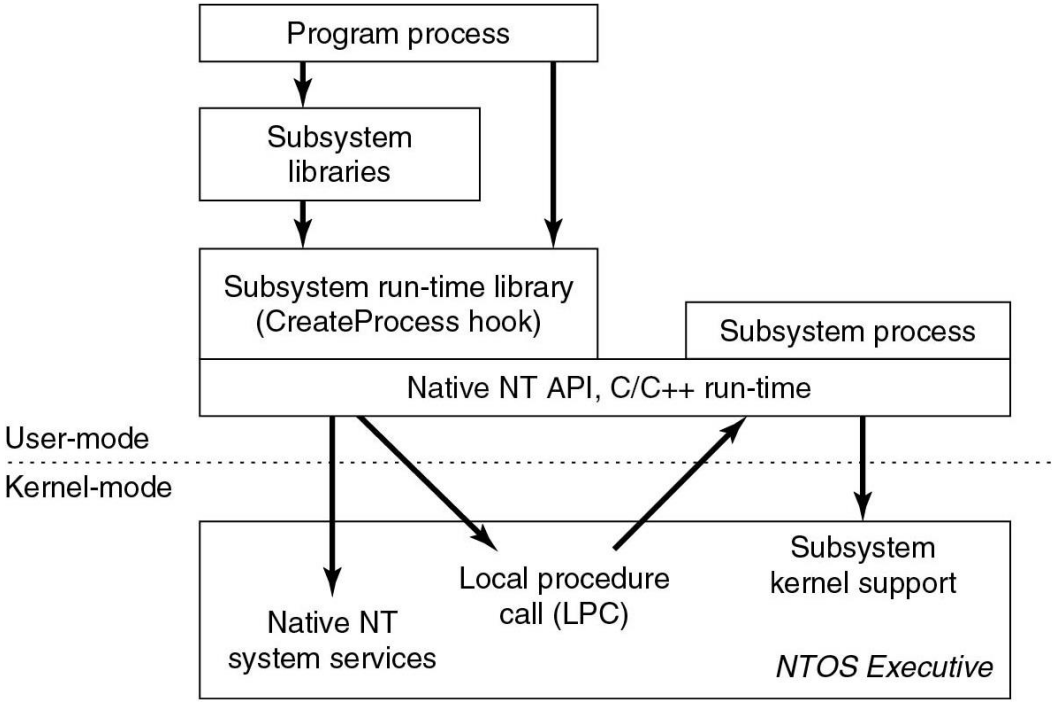

Figure 11.5 The components used to build NT subsystems.

Kernel Mode Objects

Object category	Examples
Synchronization	Semaphores, mutexes, events, IPC ports, I/O completion queues
I/O	Files, devices, drivers, timers
Program	Jobs, processes, threads, sections, tokens
Win 32 GUI	Desktops, application callbacks

Figure 11.6 Common categories of kernel-mode object types.

Examples of NT API Calls

NtCreateProcess(&ProcHandle, Access, SectionHandle, DebugPortHandle, ExceptPort Handle, ...)
NtCreateThread(&ThreadHandle, ProcHandle, Access, ThreadContext, Create Suspended, ...)
NtAllocateVirtualMemory(ProcHandle, Addr, Size, Type, Protection, ...)
NtMapViewOfSection(SectHandle, ProcHandle, Addr, Size, Protection, ...)
NtReadVirtualMemory(ProcHandle, Addr, Size, ...)
NtWriteVirtualMemory(ProcHandle, Addr, Size, ...)
Nt CreateFile(&FileHandle, FileNameDescriptor, Access, ...)
NtDuplicateObject(srcProcHandle, srcObjHandle, dstProcHandle, dstObjHandle, ...)

Figure 11.7 Examples of native NT API calls that use handles to manipulate objects.

Win32 call	Native NT API call
CreateProcess	NtCreateProcess
CreateThread	NtCreateThread
SuspendThread	NtSuspendThread
CreateSemaphore	NtCreateSemaphore
ReadFile	NtReadFile
DeleteFile	NtSetInformationFile
CreateFileMapping	NtCreateSection
VirtualAlloc	NtAllocateVirtualMemory
MapViewOfFile	NtMapViewOfSection
DuplicateHandle	NtDuplicateObject
CloseHandle	NtClose

Figure 11.8 Examples of Win32 API calls and the native NT API calls that they wrap

The Registry

Hive file	Mounted name	Use
SYSTEM	HKLM\SYSTEM	OS configuration information, used by kernel
HARDWARE	HKLM\HARDWARE	In-memory hive recording hardware detected
BCD	HKLM\BCD*	Boot Configuration Database
SAM	HKLM\SAM	Local user account information
SECURITY	HKLM\SECURITY	lsass' account and other security information
DEFAULT	HKEY USERS \.DEFAULT	Default hive for new users
NTUSER.DAT	HKEY USERS \<user id>	User-specific hive, kept in home directory
SOFTWARE	HKLM\SOFTWARE	Application classes registered by COM
COMPONENTS	HKLM\COMPONENTS	Manifests and dependencies for system. components

Figure 11.9 The registry hives in Windows. HKLM is short for HKEY_LOCAL_MACHINE.

Win32 Calls That Use the Registry

Win32 API function	Description
RegCreateKeyEx	Create a new registry key
RegDeleteKey	Delete a registry key
RegOpenKeyEx	Open a key to get a handle to it
RegEnumKeyEx	Enumerate the subkeys subordinate to the key of the handle
RegQueryValueEx	Look up the data for a value within a key
RegSetValueEx	Modifies data for a value within a key
RegFlushKey	Persist any modifications on the given key to disk

Figure 11.10 Some of the Win32 API calls for using the registry.

Windows System Structure

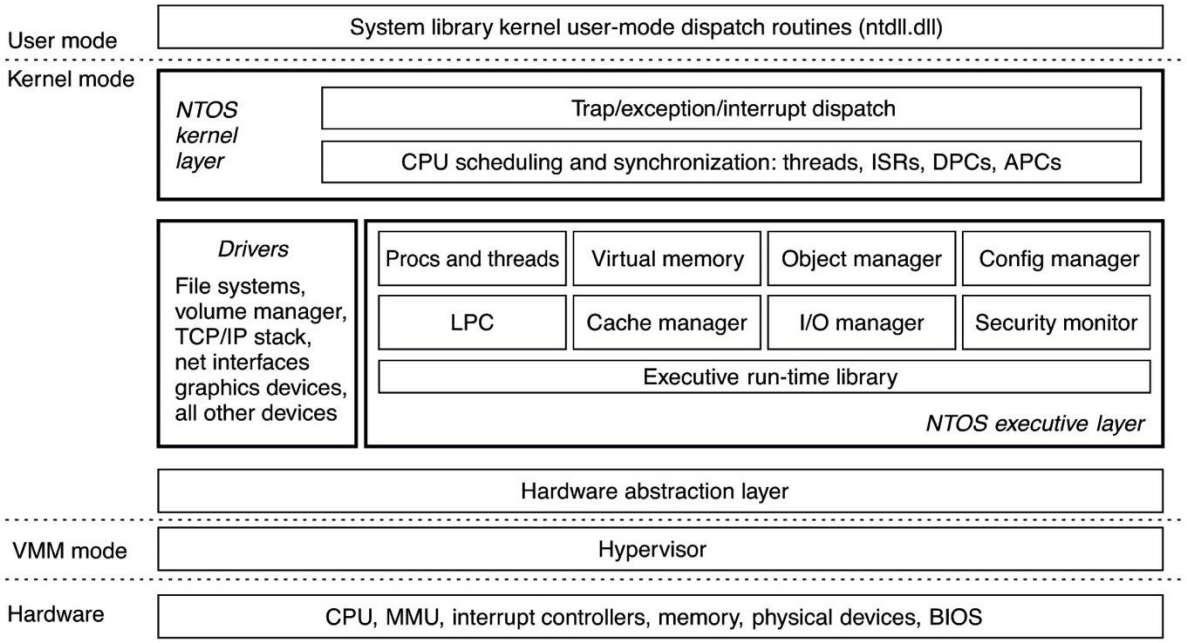

Figure 11.11 Windows kernel-mode organization.

The Hardware Abstraction Layer (HAL)

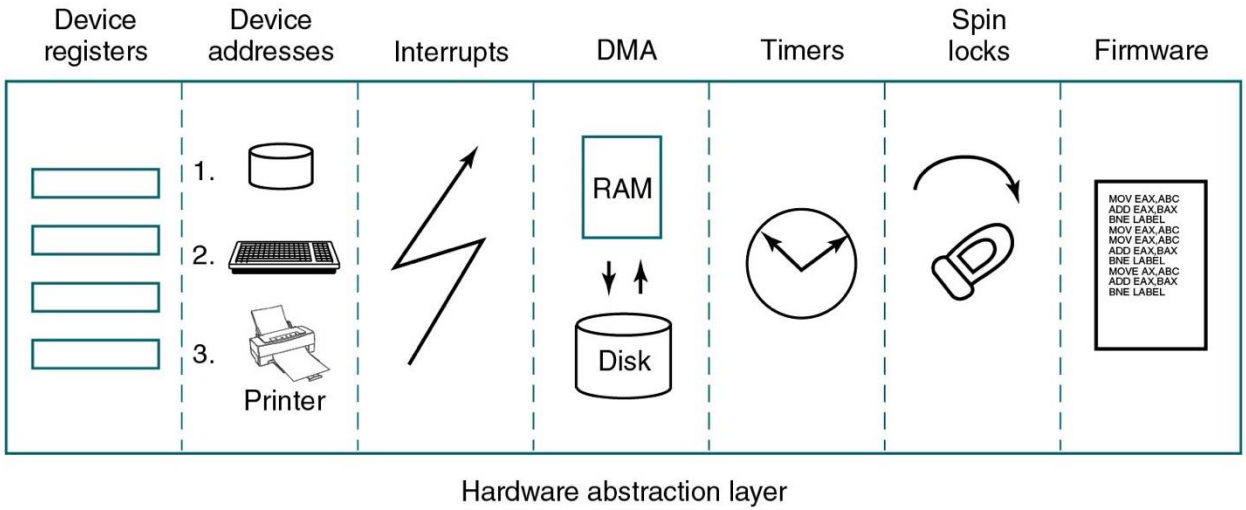

Figure 11.12 Some of the hardware functions the HAL manages.

Dispatcher Objects

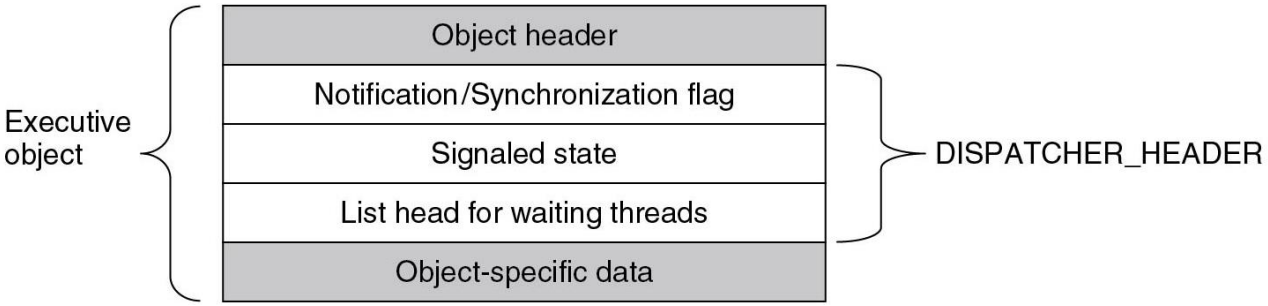

Figure 11.13 Dispatcher_header data structure embedded in many executive objects.

Device Stacks

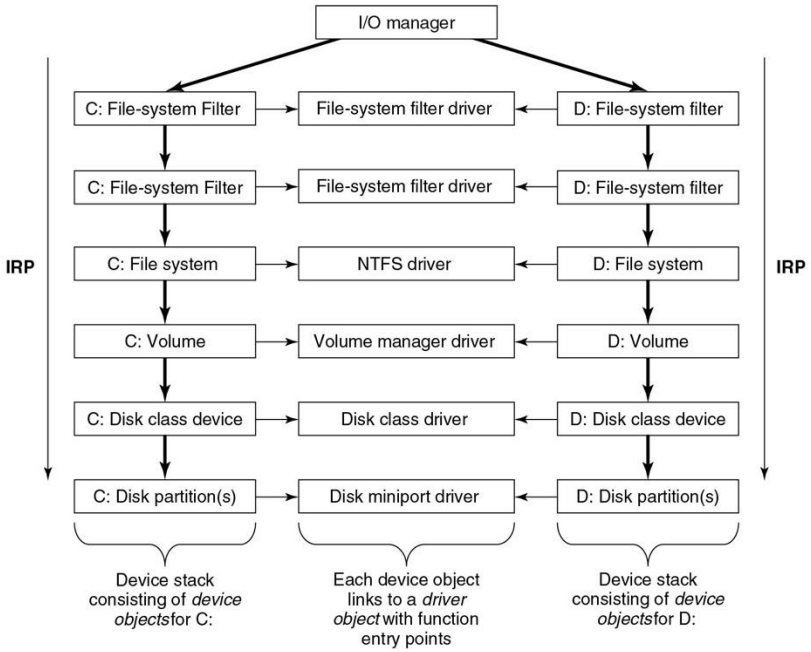

Figure 11.14 Simplified depiction of device stacks for two NTFS file volumes.

Executive objects

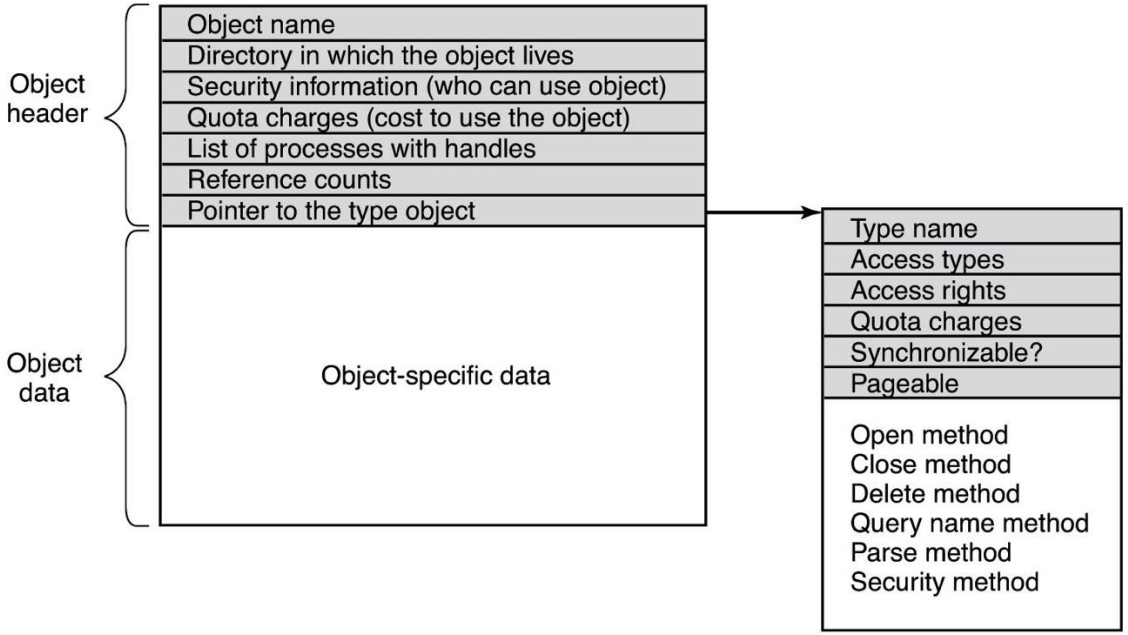

Figure 11.15 Structure of an executive object managed by the object manager.

Handles (1 of 2)

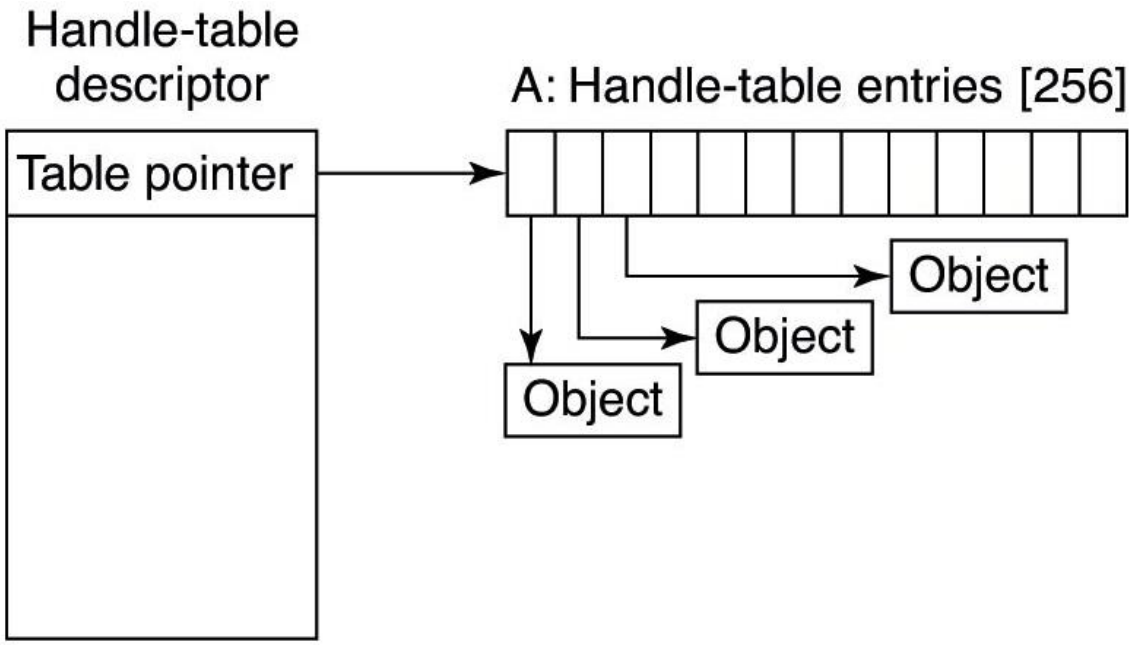

Figure 11.16 Handle table data structures foRAMinimal table using a single page.

Handles (2 of 2)

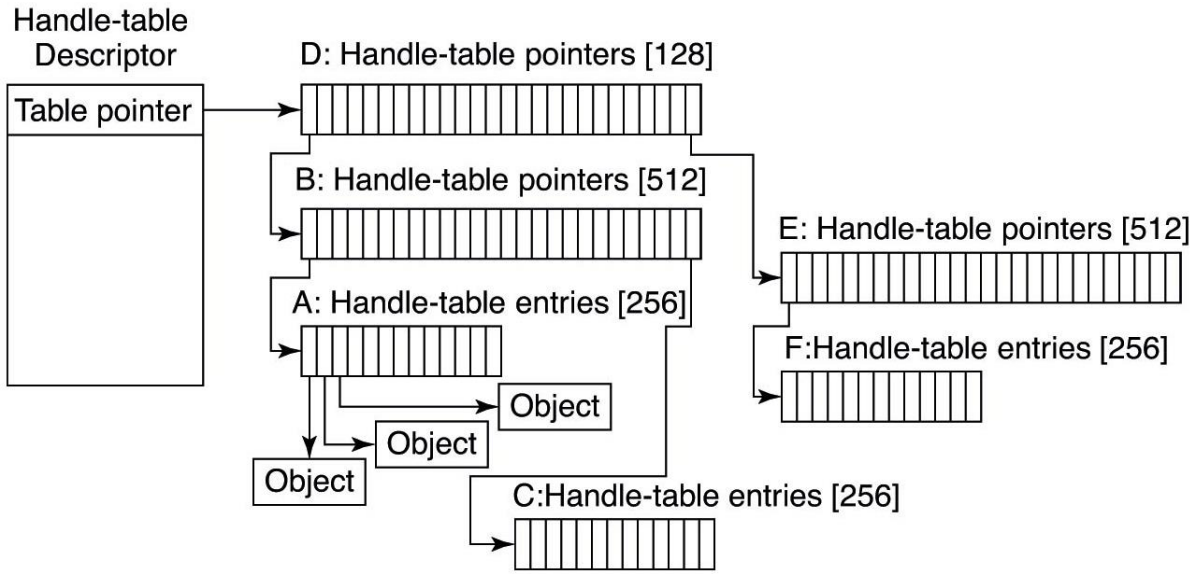

Figure 11.17 Handle table structures foRAMaximal table of up to 16 million handles.

Object Procedures

Procedure	When called	Notes
Open	For every new handle	Rarely used
Parse	For object types that extend the namespace	Used for files and registry keys
Close	At last handle close	Clean up visible side effects
Delete	At last pointer dereference	Object is about to be deleted
Security	Get or set object's security descriptor	Protection
QueryName	Get object's name	Rarely used outside kernel

Figure 11.18 Object procedures supplied when specifying a new object type.

Directories in the Object Namespace

Directory	Contents
\GLOBAL??	Starting place for looking up Win32 devices like C:
\Device	All discovered I/O devices
\Driver	Objects corresponding to each loaded device driver
\ObjectTypes	The type objects such as those listed in Fig. 11.21
\Windows	Objects for sending messages to all the Win32 GUI windows
\BaseNamedObjects	User-created Win32 objects such as events, mutexes, etc.
\Sessions	Win32 objects created in the session. Sess. 0 uses \BaseNamed Objects
\Arcname	Partition names discovered by the boot loader
\NLS	National Language Support objects
\FileSystem	File-system driver objects and file system recognizer objects
\Security	Objects belonging to the security system
\KnownDLLs	Key shared libraries that are opened early and held open

Figure 11.19 Some typical directories in the object namespace.

Opening a File

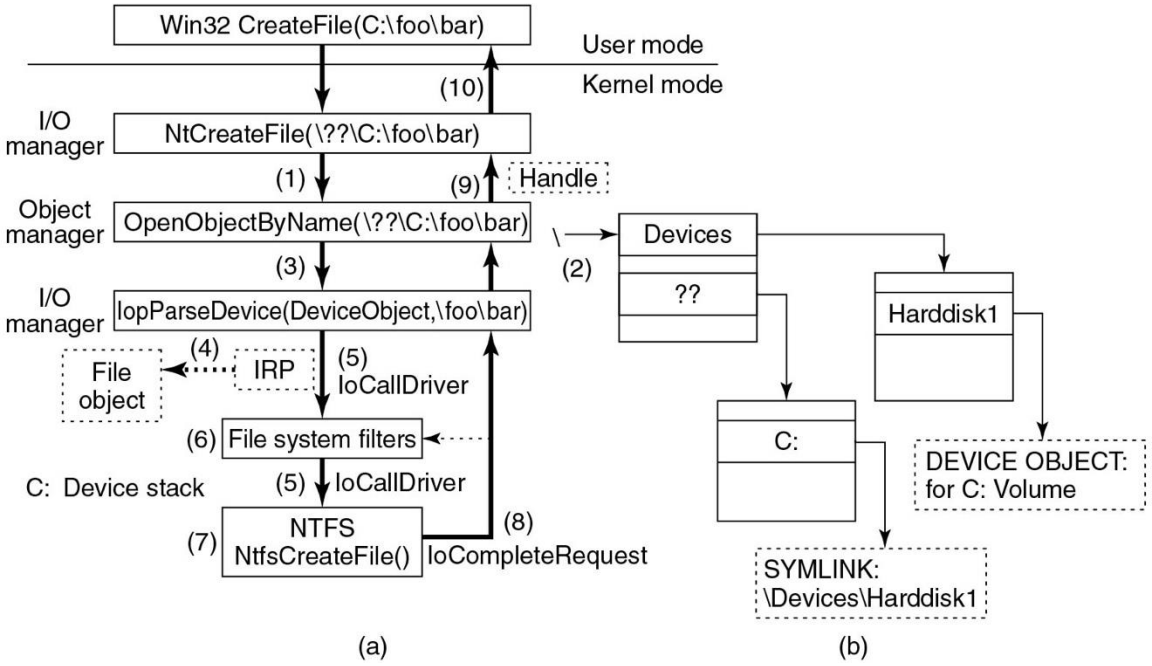

Figure 11.20 I/O and object manager steps for creating/opening a file and getting a handle.

Executive Object Types (1 of 2)

Type	Description
Process	User process
Thread	Thread within a process
Semaphore	Counting semaphore used for interprocess synchronization
Mutex	Binary semaphore used to enter a critical region
Event	Synchronization object with persistent state (signaled/not)
ALPC port	Mechanism for interprocess message passing
Timer	Object allowing a thread to sleep for a fixed time interval
Queue	Object used for completion notification on asynchronous I/O
Open file	Object associated with an open file
Access token	Security descriptor for some object

Figure 11.21 Some common executive objects types manager by the object manager.

Executive Object Types (2 of 2)

Type	Description
Profile	Data structure used for profiling CPU usage
Section	Object used for representing mappable files
Key	Registry key, used to attach registry to object-manager namespace
Object directory	Directory for grouping objects within the object manager
Symbolic link	Refers to another object manager object by path name
Device	I/O device object for a physical device, bus, driver, or volume instance
Device driver	Each loaded device driver has its own object

Figure 11.21 Some common executive objects types manager by the object manager.(Continued)

Jobs, Processes, Threads, and Fibers

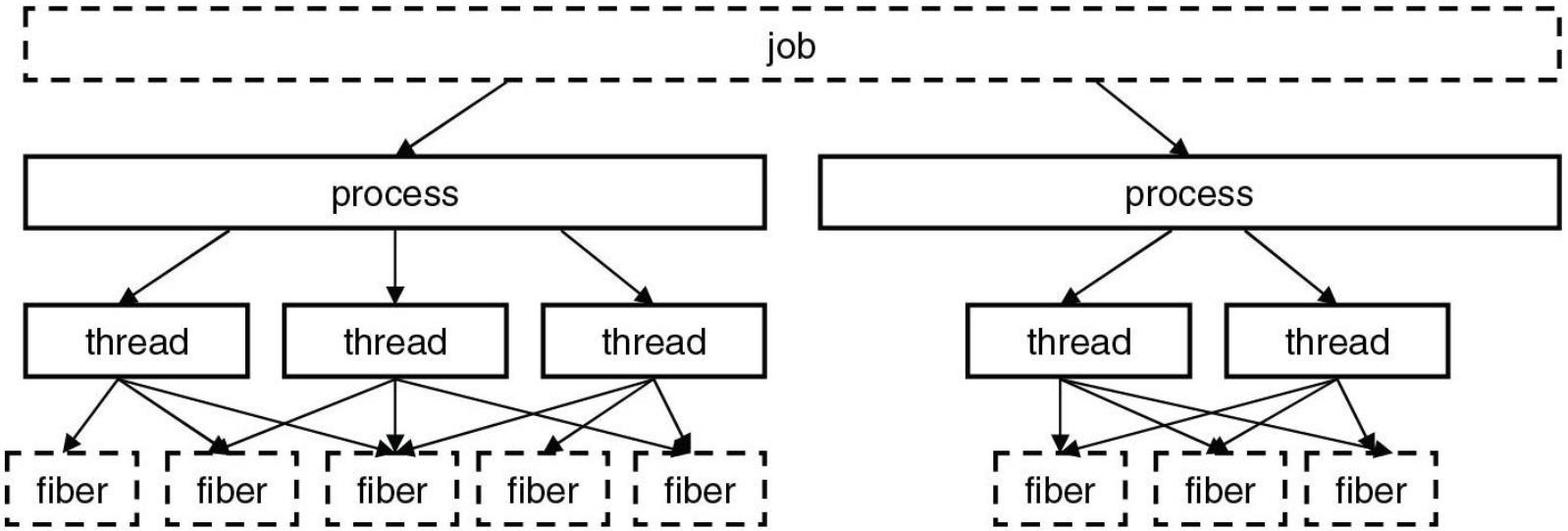

Figure 11.22 The relationship between jobs, processes, threads, and Fibers.

Concepts for CPU Management

Name	Description	Notes
Job	Collection of processes that share quotas and limits	Used in AppContainers
Process	Container for holding resources	
Thread	Entity scheduled by the kernel	
Fiber	Lightweight thread managed entirely in user space	Rarely used
Thread pool	Task-oriented programming model	Built on top of threads

Figure 11.23 Basic concepts used for CPU and resource management.

Synchronization Primitives

Primitive	Kernel object	Kernel/User	Shared/Exclusive
Event	Yes	Both	N/A
Semaphore	Yes	Both	N/A
Mutex	Yes	Both	Exclusive
Critical Section	No	User-mode	Exclusive
SRW Lock	No	User-mode	Shared
Condition Variable	No	User-mode	N/A
InitOnce	No	User-mode	N/A
WaitOnAddress	No	User-mode	N/A
EResource	No	Kernel-mode	Shared
FastMutex	No	Kernel-mode	Exclusive
PushLock	No	Kernel-mode	Shared
Cache-aware PushLock	No	Kernel-mode	Shared
Auto-expand PushLock	No	Kernel-mode	Shared

Figure 11.24 Summary of synchronization primitives.

Priorities

		Win32 process class priorities					
		Real-time	High	Above normal	Normal	Below normal	Idle
Win32 thread priorities	Time critical	31	15	15	15	15	15
	Highest	26	15	12	10	8	6
	Above normal	25	14	11	9	7	5
	Normal	24	13	10	8	6	4
	Below normal	23	12	9	7	5	3
	Lowest	22	11	8	6	4	2
	Idle	16	1	1	1	1	1

Figure 11.25 Mapping of Win 32 priorities to Windows priorities.

Thread priorities

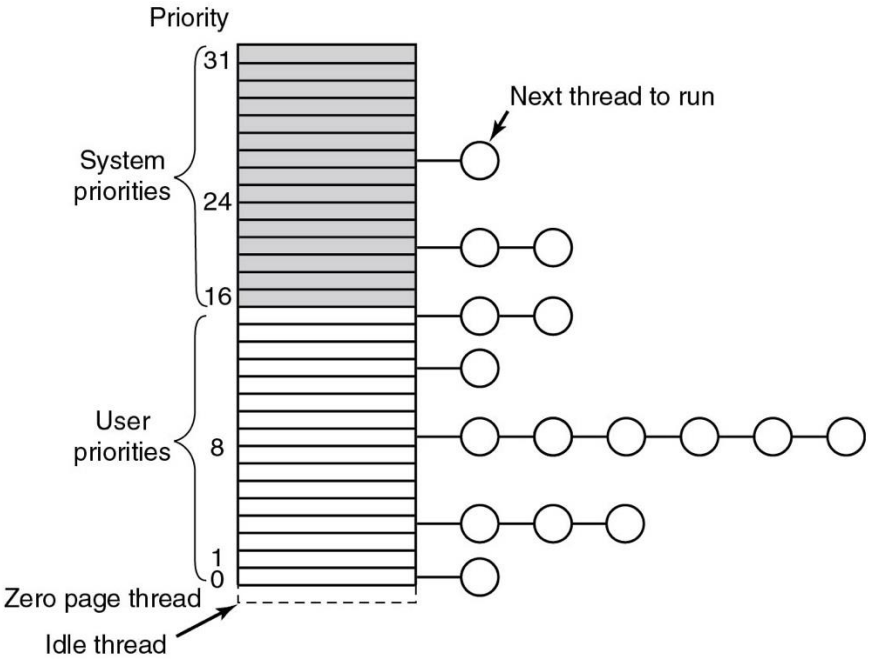

Figure 11.26 Windows supports 32 priorities for threads.

Priority Inversion

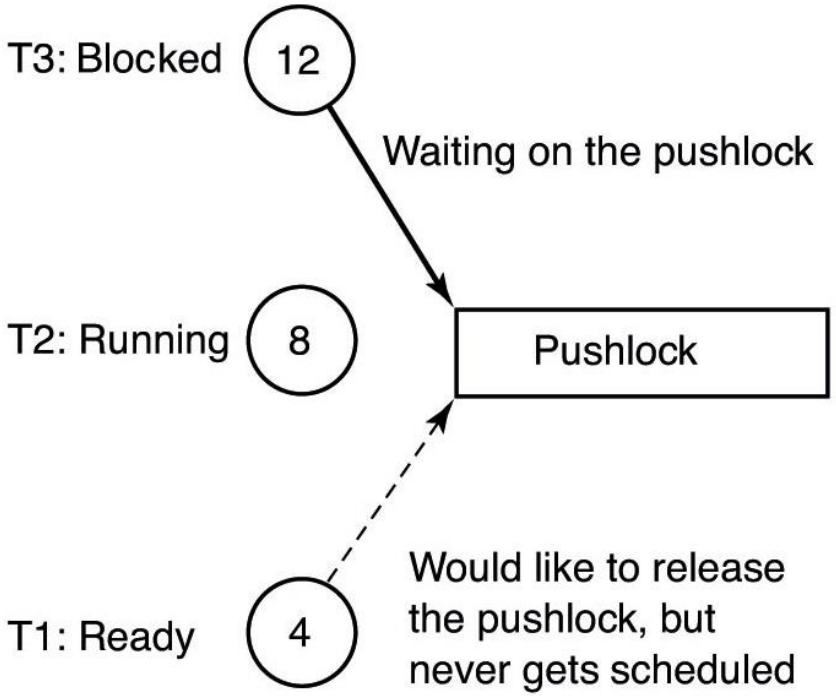

Figure 11.27 An example of priority inversion.

Windows on Windows

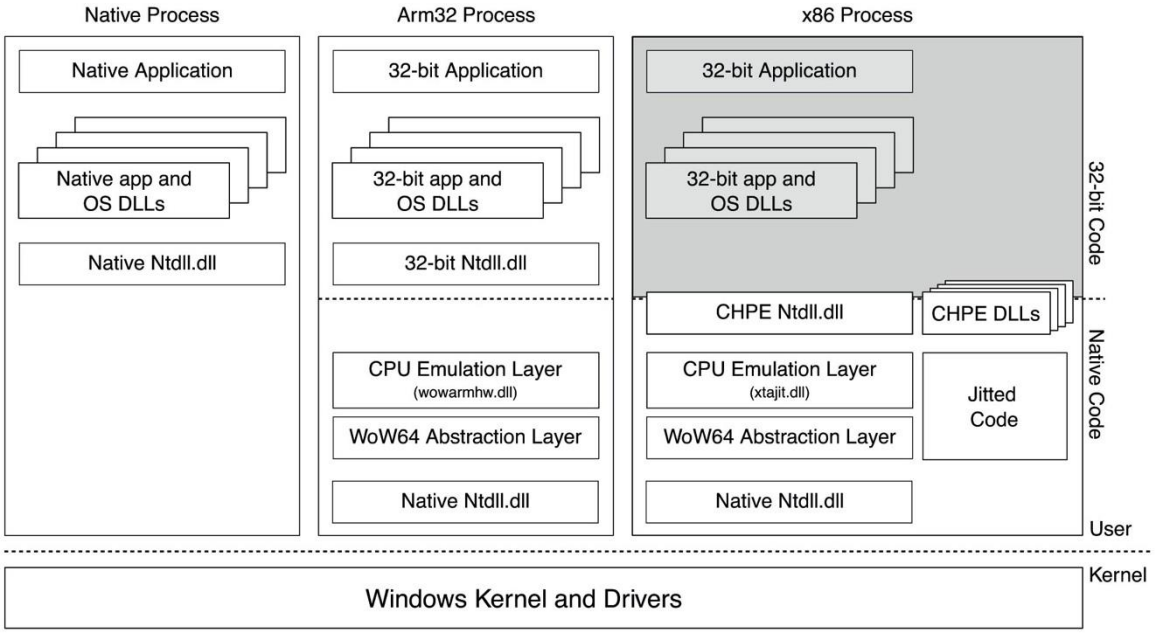

Figure 11.28 Native v.s. WoW64 on an arm 64 machine. Shaded areas indicate emulated code.

Comparison of x86 and x64 Emulation

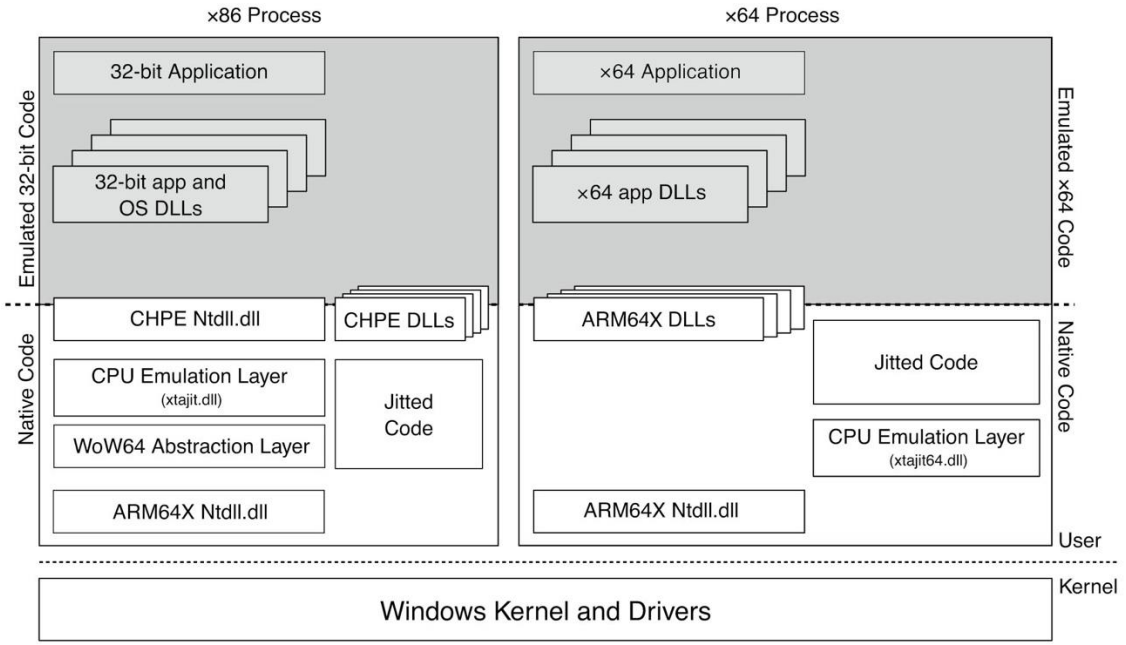

Figure 11.29 Comparison of x86 and x64 emulation infrastructure on an arm 64 machine. Shaded areas indicate emulated code.

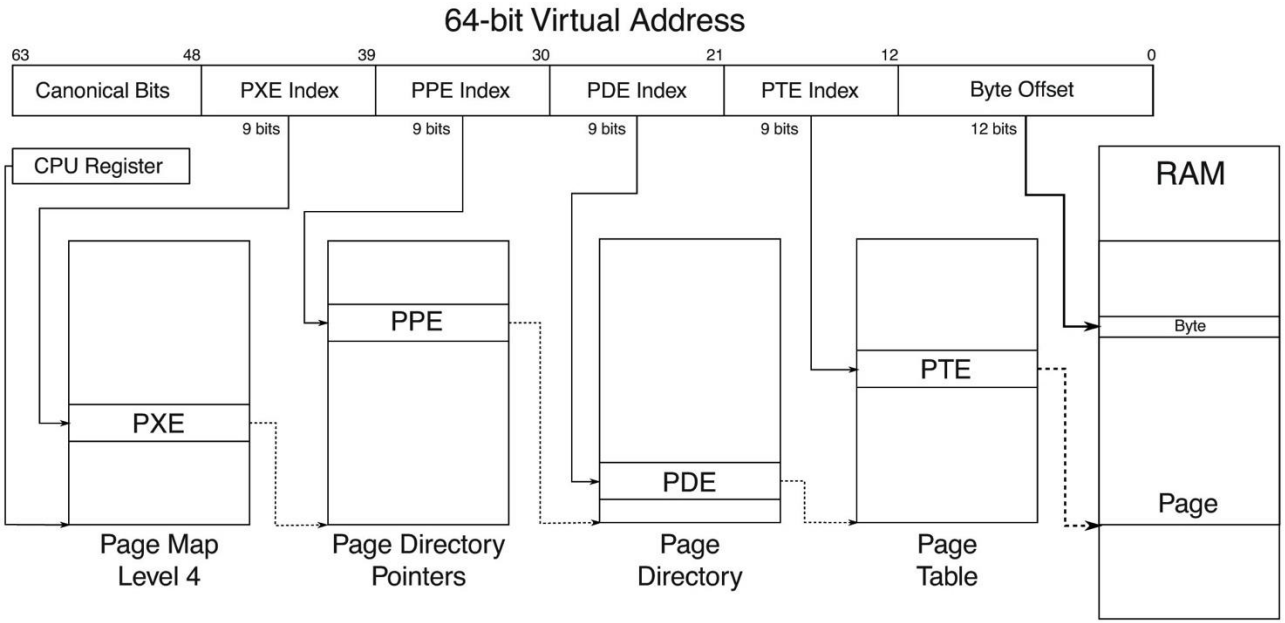

Figure 11.30 Virtual to physical address translation with a 4-level page table with 48 address bits.

Virtual Address Space Layout

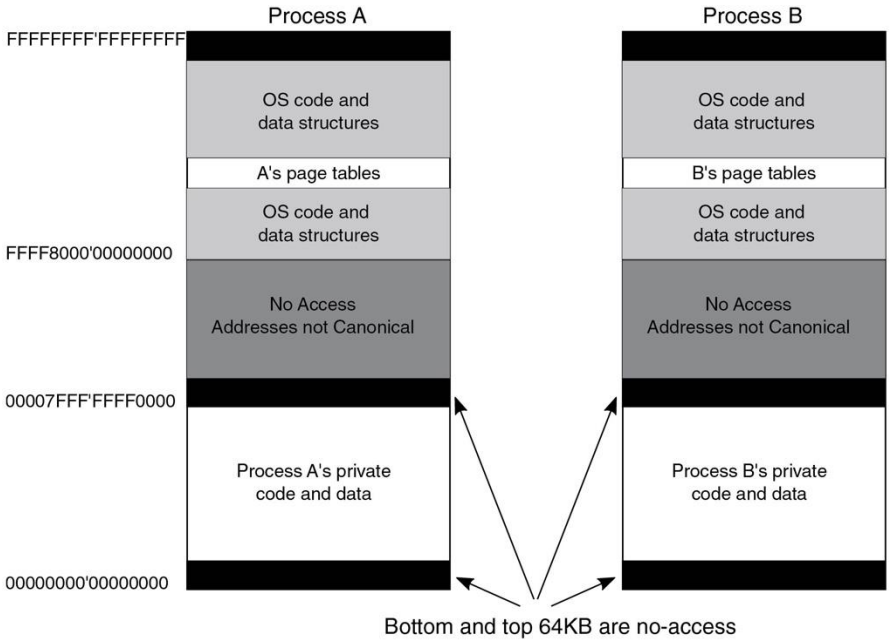

Figure 11.31 Virtual address space layout for 64-bit user processes. The white areas are private. The shaded ones are shared.

Win32 Calls for Virtual Memory

Win32 API function	Description
VirtualAlloc	Reserve or commit a region
VirtualFree	Release or decommit a region
VirtualProtect	Change the read/write/execute protection on a region
VirtualQuery	Inquire about the status of a region
VirtualLock	Make a region memory resident (i.e., disable paging for it)
VirtualUnlock	Make a region pageable in the usual way
CreateFileMapping	Create a file-mapping object and (optionally) assign it a name
MapViewOfFile	Map (part of) a file into the address space
UnmapViewOfFile	Remove a mapped file from the address space
OpenFileMapping	Open a previously created file-mapping object

Figure 11.32 The principal Win32 API functions for managing virtual memory.

Mapped Regions

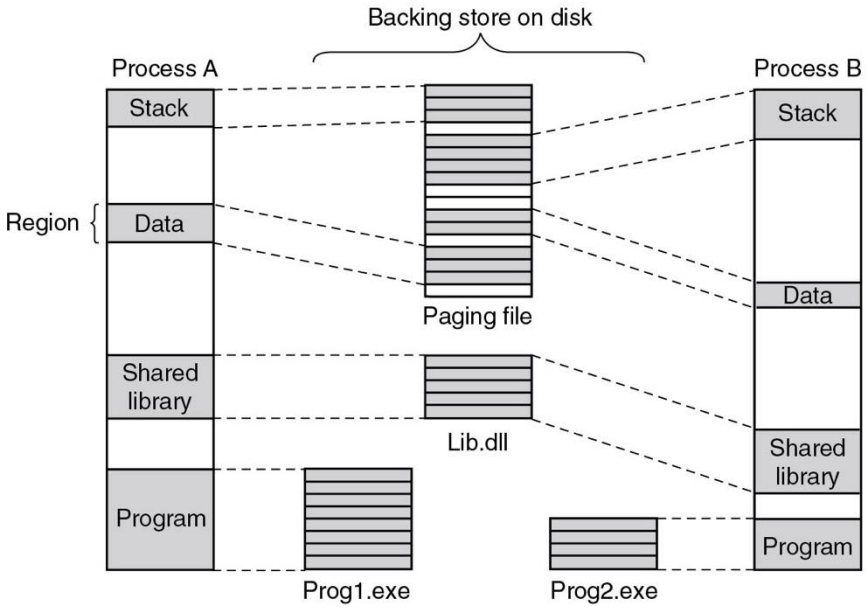

Figure 11.33 Mapped regions with their shadow pages on disk. The lib.dll file is mapped into two address spaces at the same time.

Page Table Entries

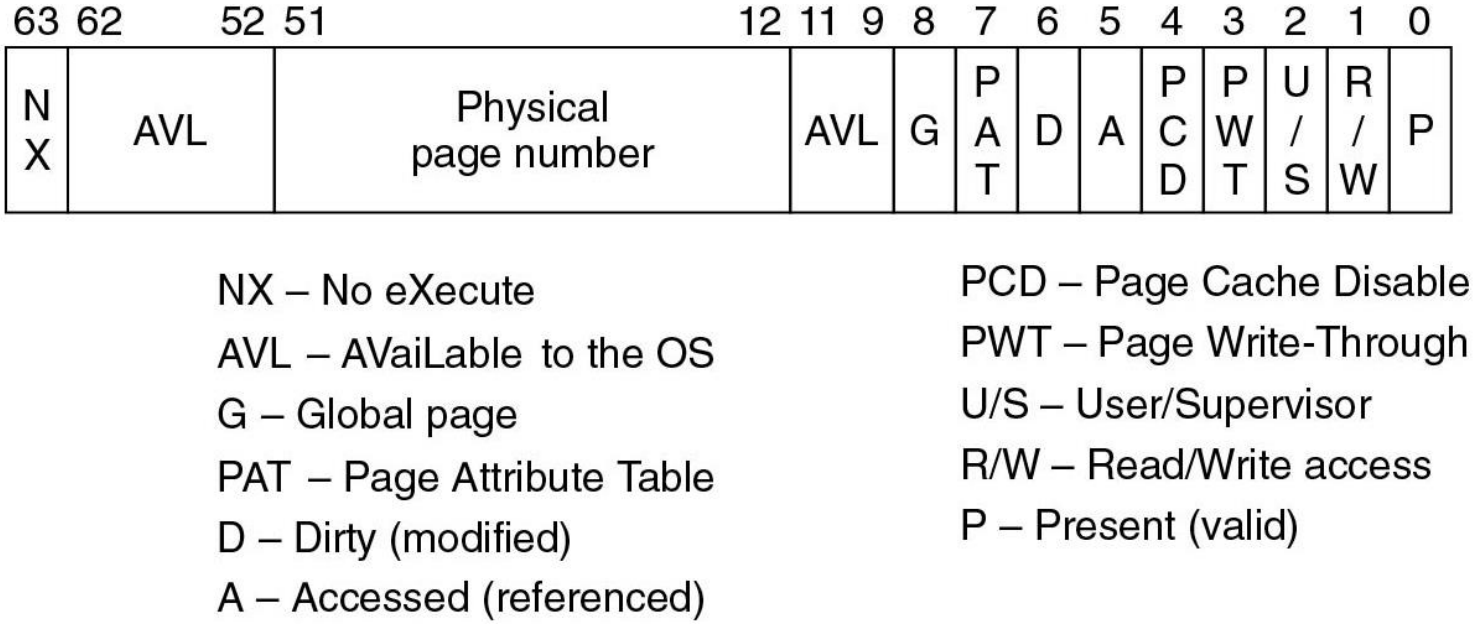

Figure 11.34 A page table entry for a mapped page on the Intel x86 and AMD x64 architectures.

Self-Map Entries

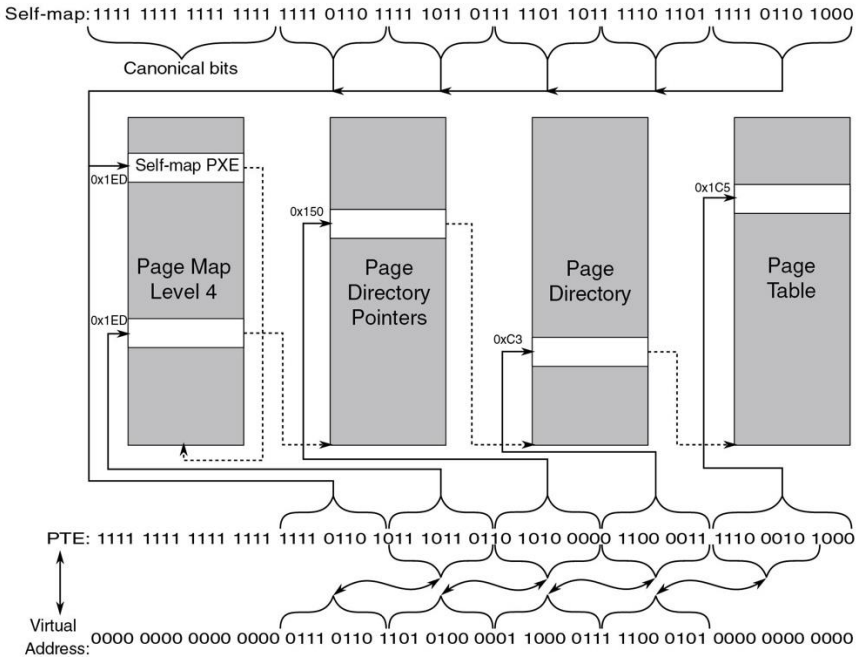

Figure 11.35 The Windows self-map entries are used to map the physical pages into kernel virtual addresses.

The Page-Frame Database

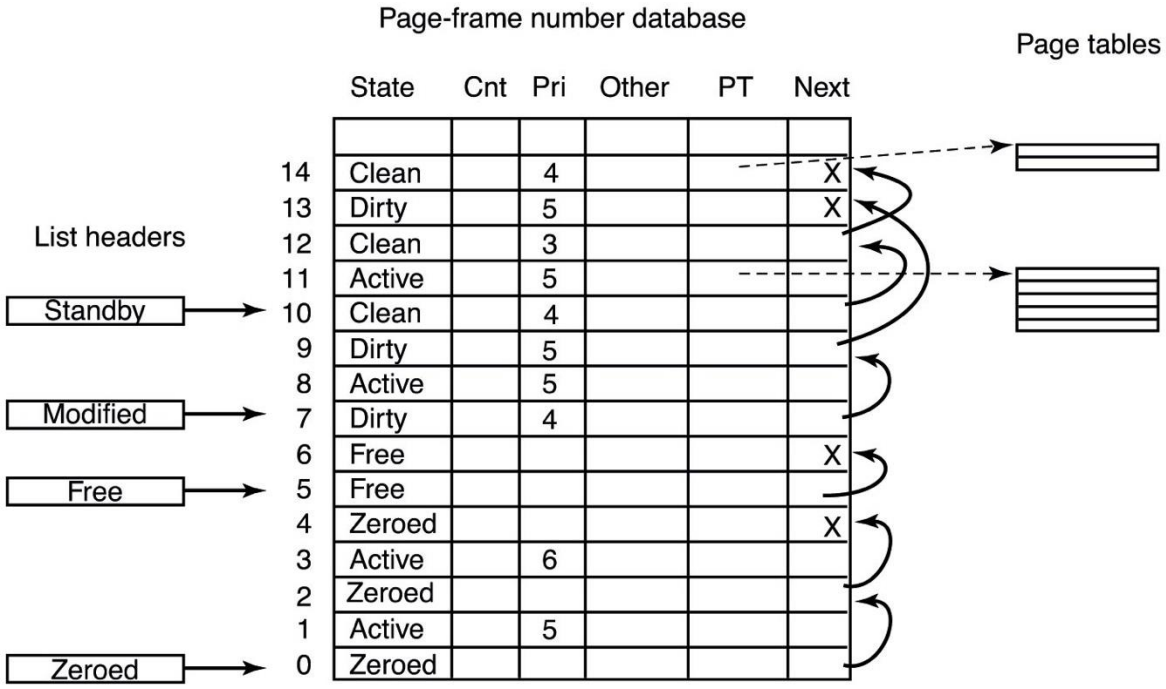

Figure 11.36 Some of the fields in the page-frame database.

Page Lists

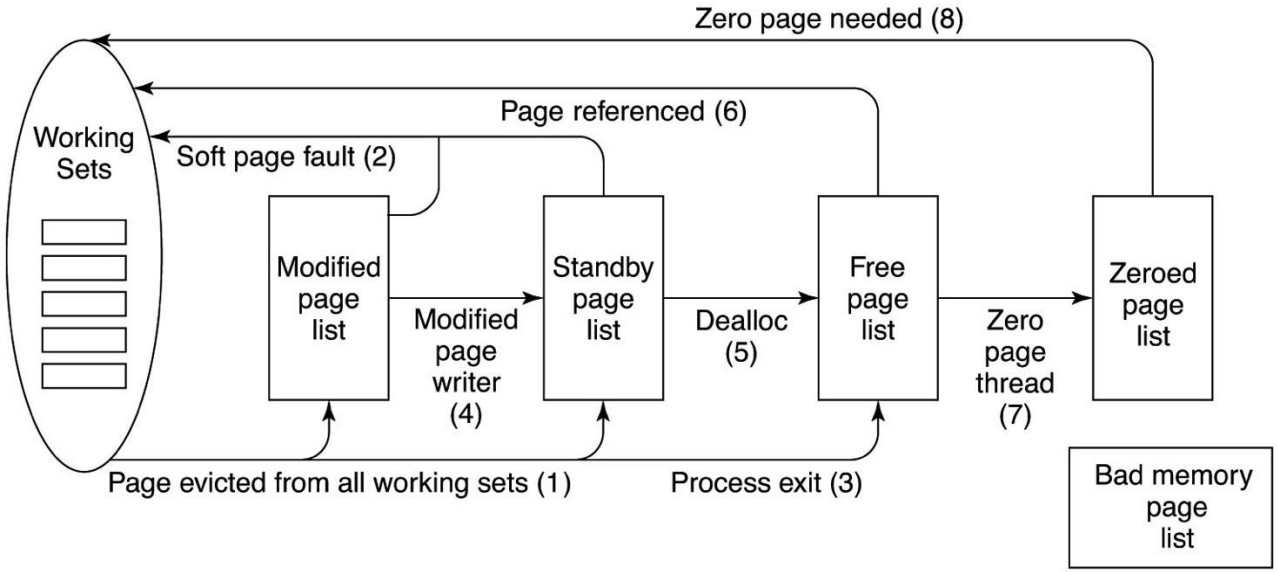

Figure 11.37 The various page lists and the transitions between them.

Page Transitions With Memory Compression

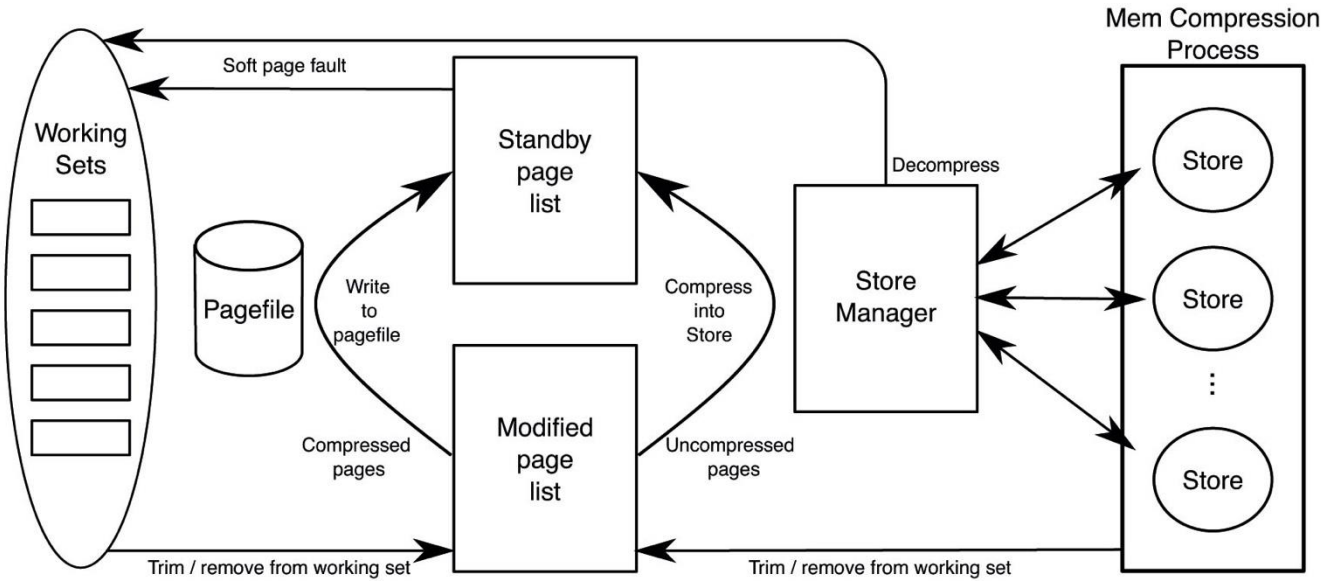

Figure 11.38 Page transitions with memory compression (free/zero lists omitted here).

Memory Partitions

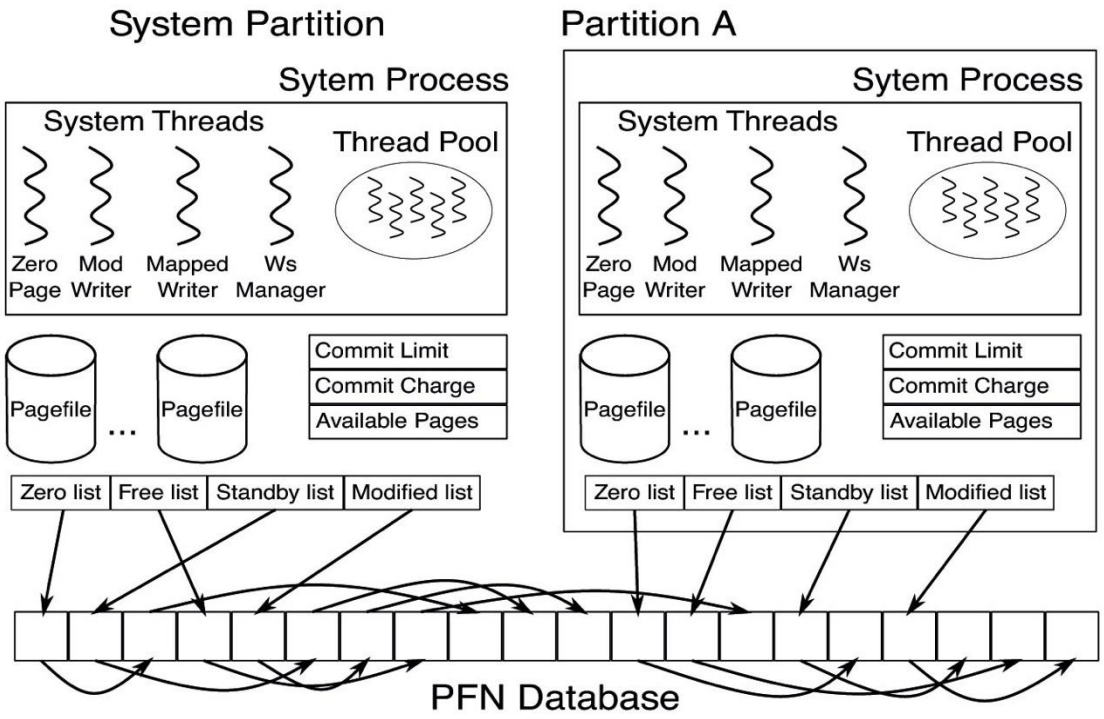

Figure 11.39 Memory partition data structures.

Native NT Calls for I/O

I/O system call	Description
NtCreateFile	Open new or existing files or devices
NtReadFile	Read from a file or device
NtWriteFile	Write to a file or device
NtQueryDirectoryFile	Request information about a directory, including files
NtQueryVolumeInformationFile	Request information about a volume
NtSetVolumeInformationFile	Modify volume information
NtNotifyChangeDirectoryFile	Finishes when any file in the directory or subtree is modified
NtQueryInformationFile	Request information about a file
NtSetInformationFile	Modify file information
NtLockFile	Lock a range of bytes in a file
NtUnlockFile	Remove a range lock
NtFsControlFile	Miscellaneous operations on a file
NtFlushBuffersFile	Flush in-memory file buffers to disk
NtCancelIoFile	Cancel outstanding I/O operations on a file
NtDeviceIoControlFile	Special operations on a device

Figure 11.40 Native NT API calls for performing I/O.

Device Drivers

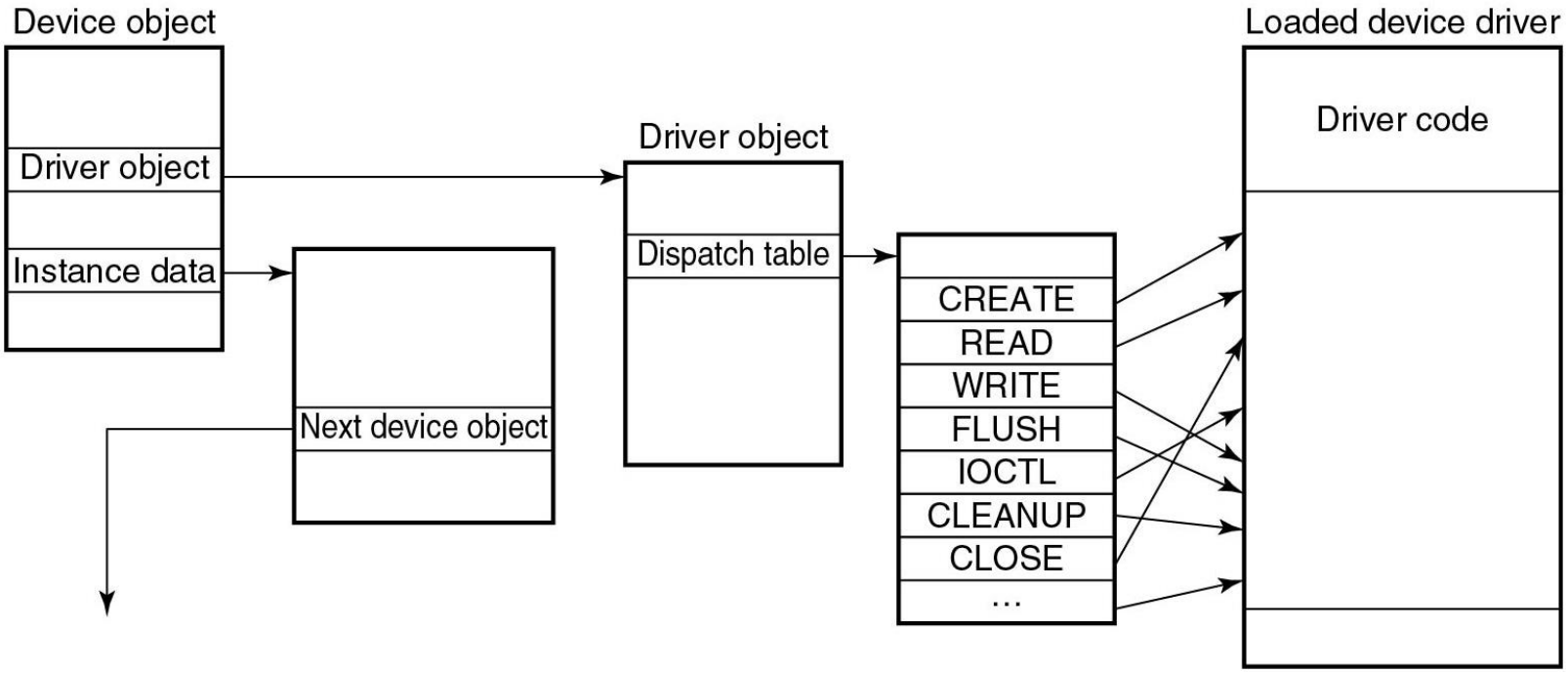

Figure 11.41 A single level in a device stack.

I/O Request Packets

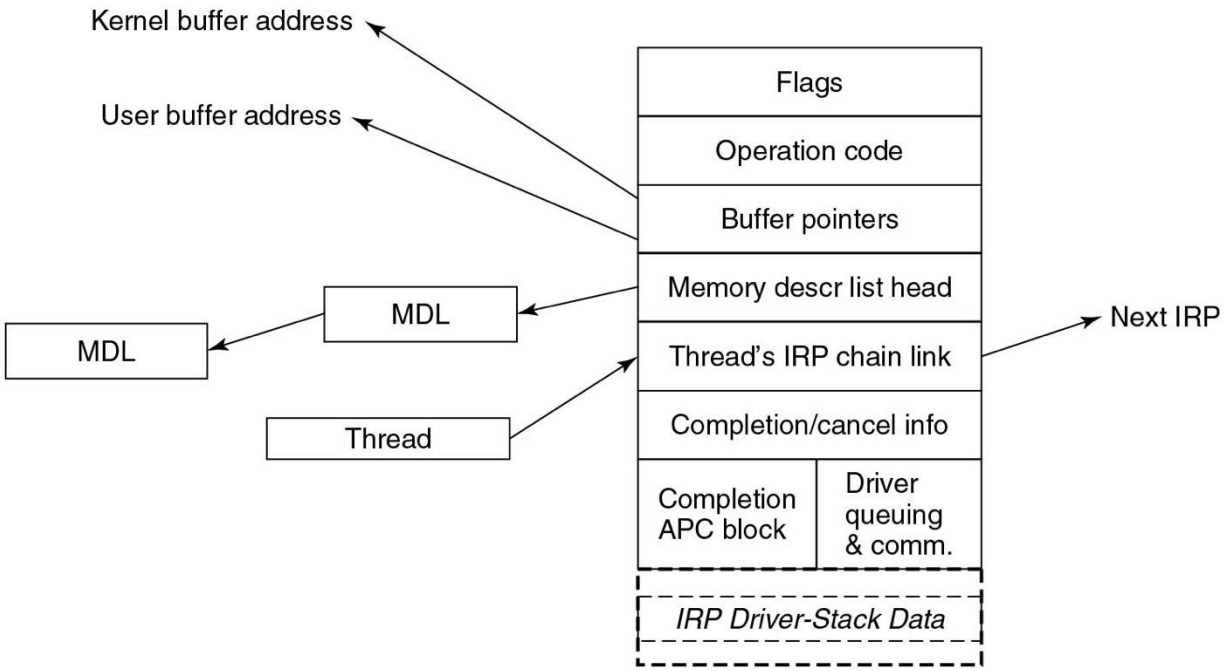

Figure 11.42 The major fields of an I/O Request Packet.

Stacked Drivers

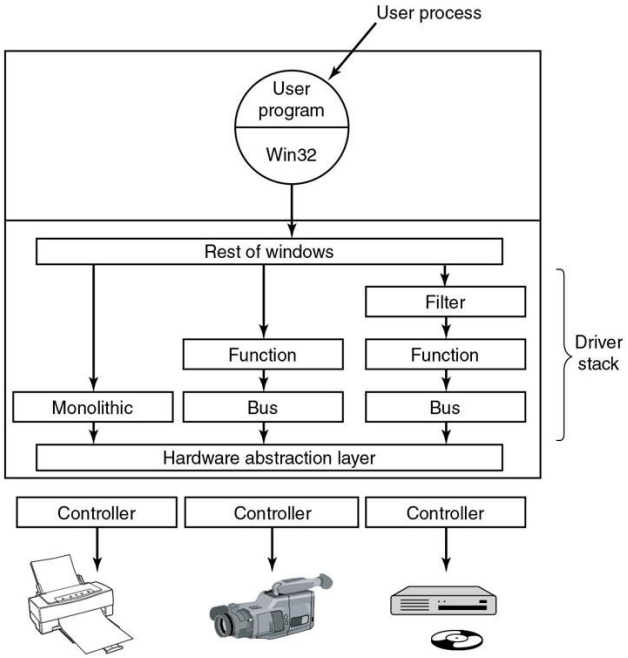

Figure 11.43 Windows allows drivers to be stacked to work with a specific instance of a device.

The NTFS Master File Table

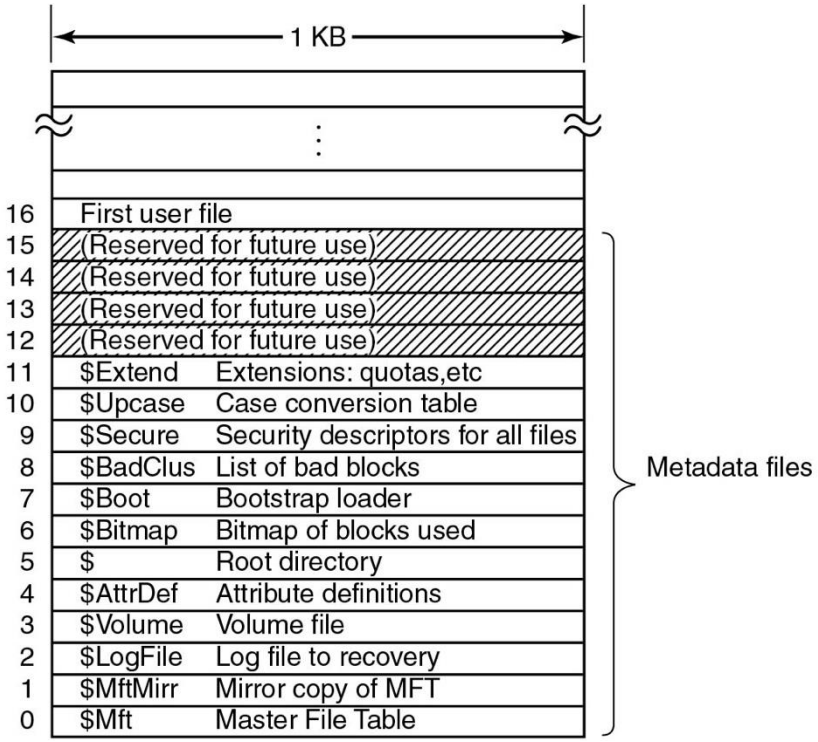

Figure 11.44 The NTFS master file table.

Master File Table Attributes

Attribute	Description
Standard information	Flag bits, timestamps, etc.
File name	File name in Unicode; may be repeated for MS-DOS name
Security descriptor	Obsolete. Security information is now in $Extend$Secure
Attribute list	Location of additional MFT records, if needed
Object ID	64-bit file identifier unique to this volume
Reparse point	Used for mounting and symbolic links
Volume name	Name of this volume (used only in $Volume)
Volume information	Volume version (used only in $Volume)
Index root	Used for directories
Index allocation	Used for very large directories
Bitmap	Used for very large directories
Logged utility stream	Controls logging to $LogFile
Data	Stream data; may be repeated

Figure 11.45 The attributes used in MFT records.

MFT Block Runs

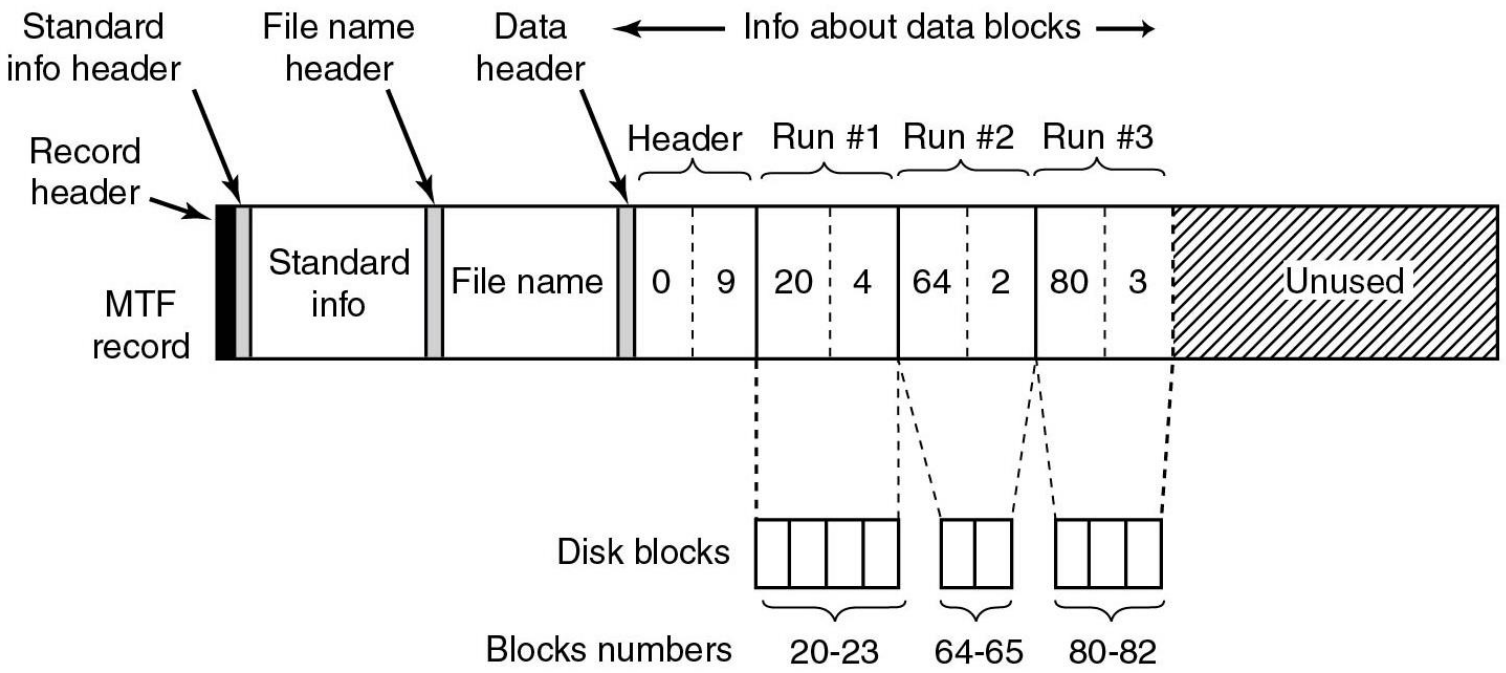

Figure 11.46 An MFT record for a three-run, nine block stream.

A File with Three MFT Records

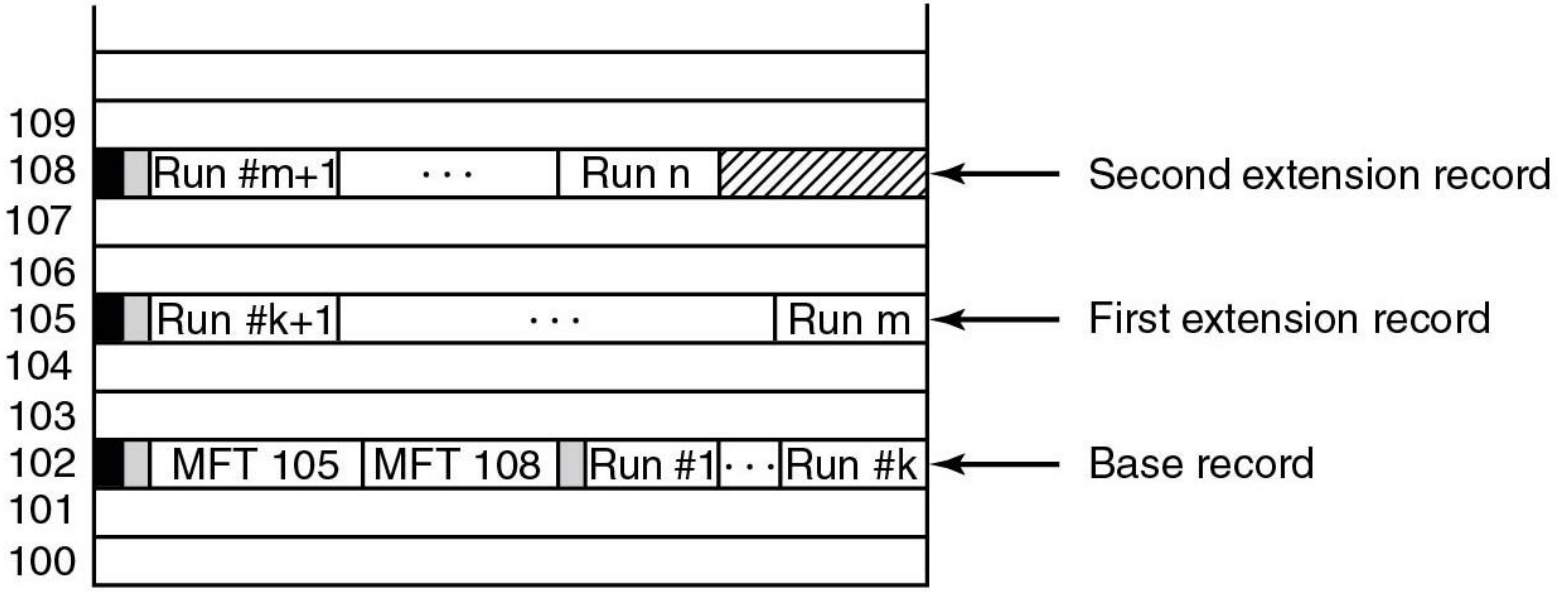

Figure 11.47 A file that requires three MFT records to store all its runs.

Directory Records

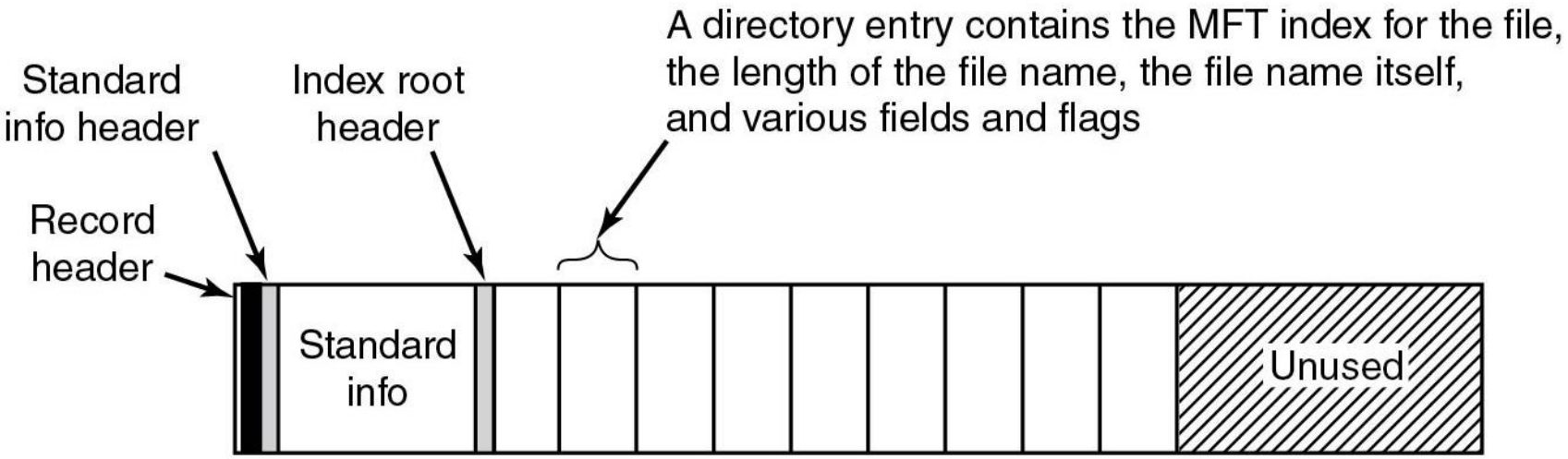

Figure 11.48 The MFT record for a small directory.

File Compression

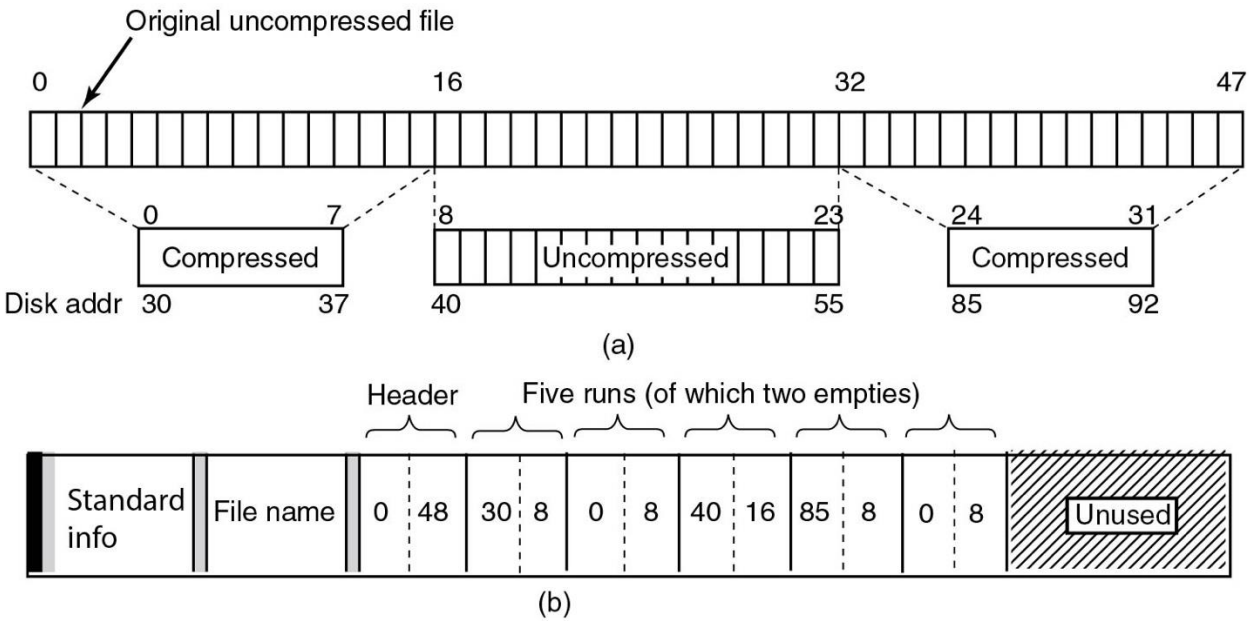

Figure 11.49 (a) A 48-block being compressed. (b) The MFT record after compression.

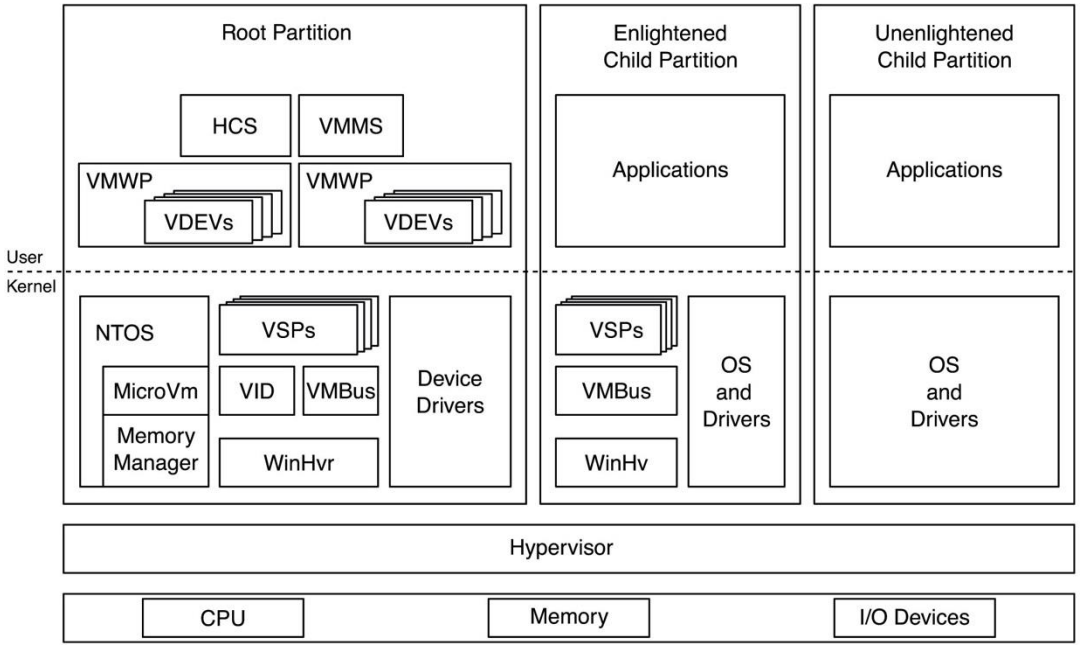

Figure 11.50 Hyper-V virtualization components in the root and child partitions.

Paravirtualized I/O

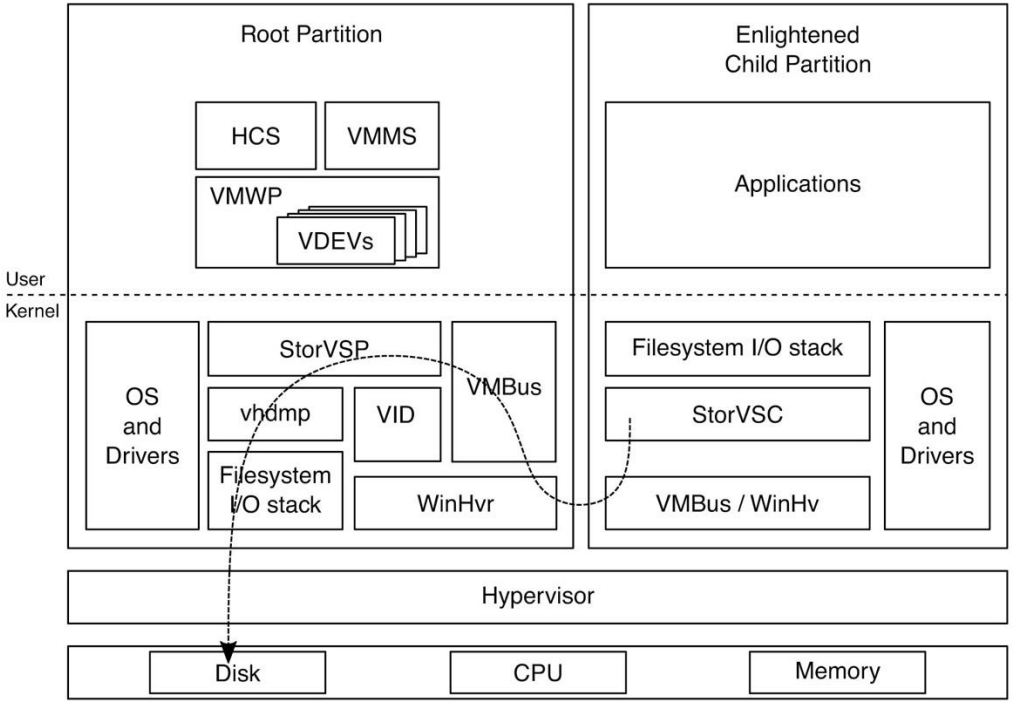

Figure 11.51 Flow of paravirtualized I/O for an enlightened guest OS.

VA-Backed VMs

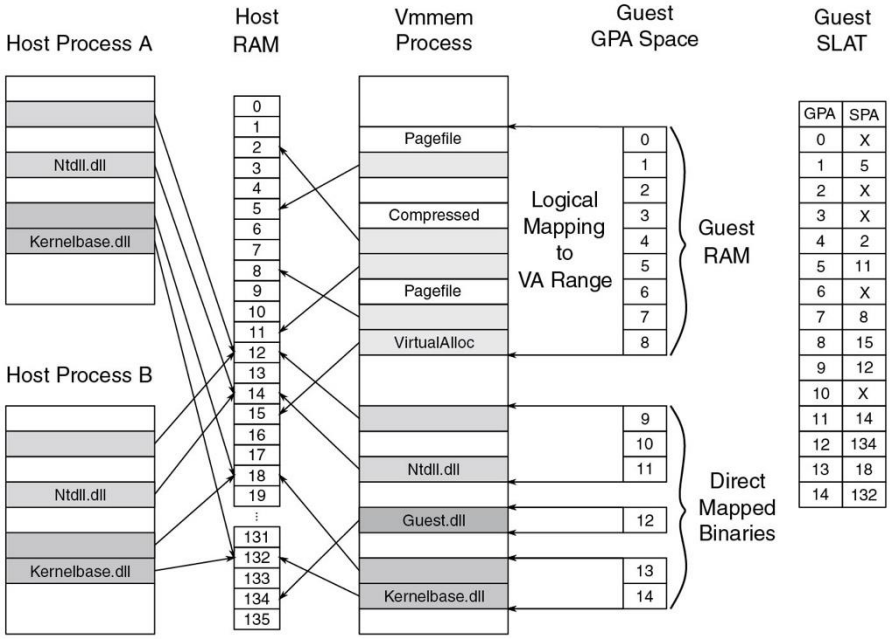

Figure 11.52 VA-backed VM's GPA space is logically mapped to virtual address ranges in its vmmem process on the host.

Virtual Hard Disks

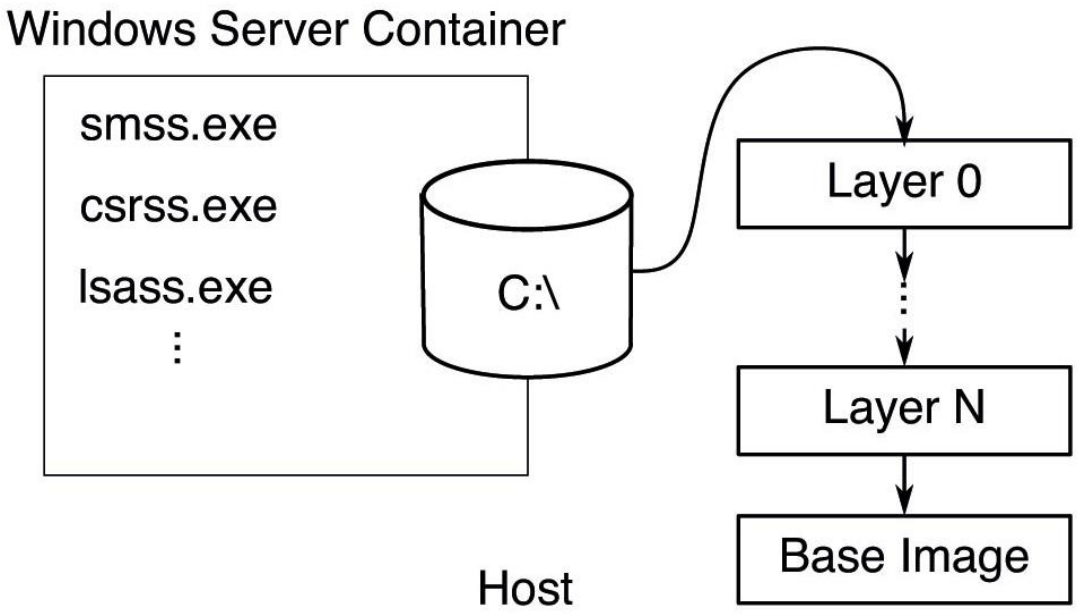

Figure 11.53 The contents of the VHD exposed to the container is backed by a set of host directories.

Virtual Secure Mode

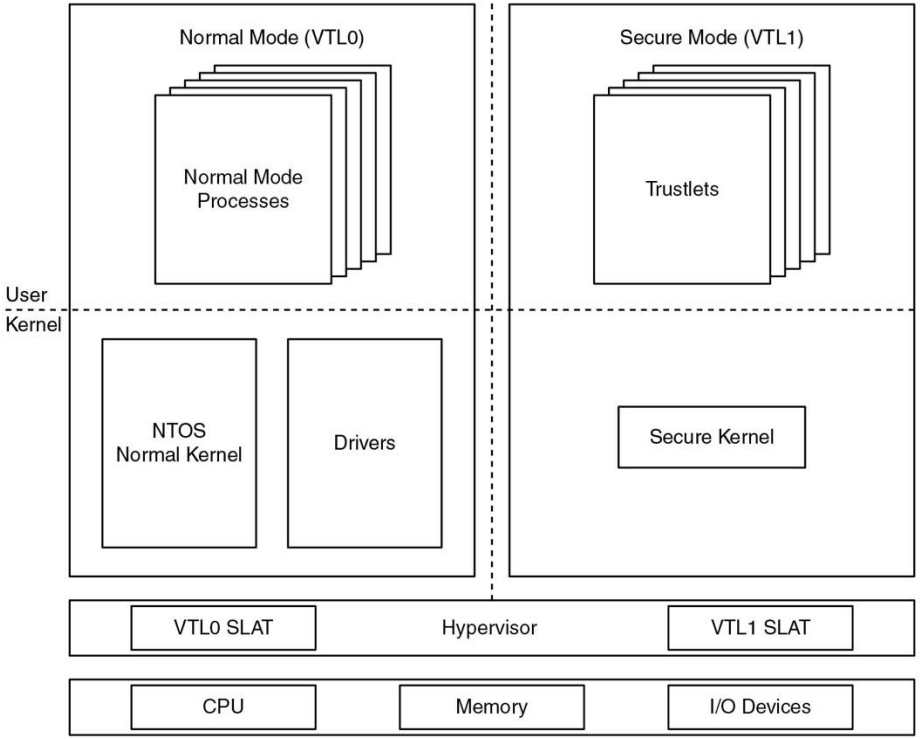

Figure 11.54 Virtual Secure Mode architecture.

Access Tokens

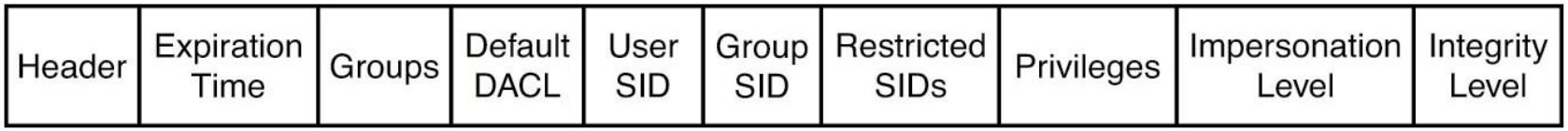

Figure 11.55 Structure of an access token.

Security Descriptors

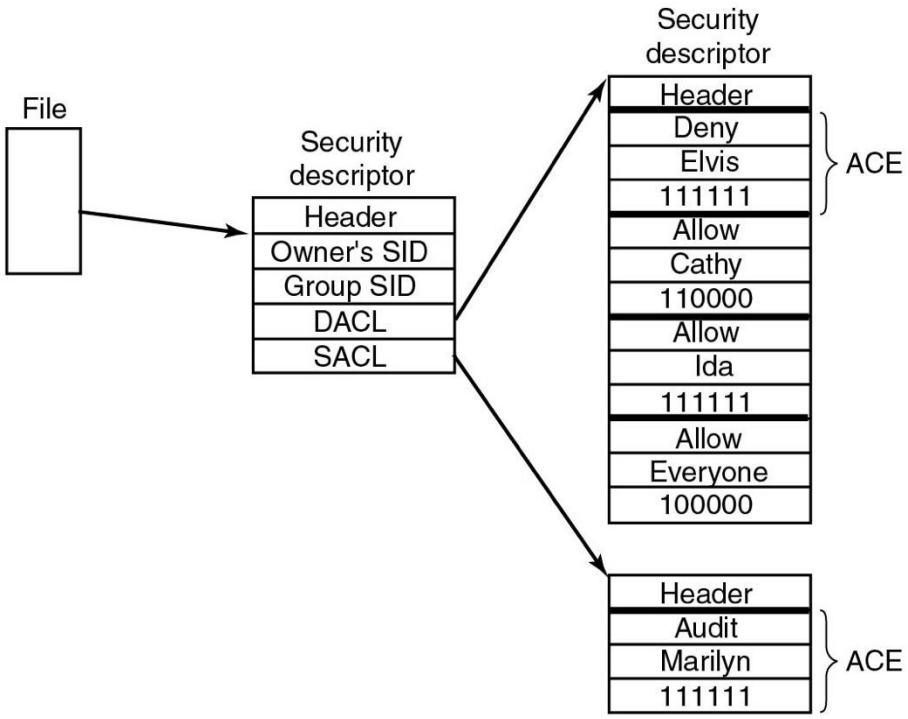

Figure 11.56 An example security descriptor for a file.

Win32 Calls for Security

Win32 API function	Description
InitializeSecurityDescriptor	Prepare a new security descriptor for use
LookupAccountSid	Look up the SID for a given user name
SetSecurityDescriptorOwner	Enter the owner SID in the security descriptor
SetSecurityDescriptorGroup	Enter a group SID in the security descriptor
InitializeAcl	Initialize a DACL or SACL
AddAccessAllowedAce	Add a new ACE to a DACL or SACL allowing access
AddAccessDeniedAce	Add a new ACE to a DACL or SACL denying access
DeleteAce	Remove an ACE from a DACL or SACL
SetSecurityDescriptorDac l	Attach a DACL to a security descriptor

Figure 11.57 The principal Win32 API functions for security.

Security Protection Mechanisms (1 of 2)

Mitigation	VBS-only	Description
InitAll	No	Zero-initializes stack variables to avoid vulnerabilities
/GS	No	Add canary to stack frames to protect return addresses
DEP	No	Data Execution Prevention. Stacks and heaps are not executable
ASLR/KASLR	No	Randomize user/kernel address space to make ROP attacks difficult
CFG	No	Control Flow Guard. Protect integrity of forward-edge control flow
KCFG	Yes	Kernel-mode CFG. Secure Kernel maintains CFG bitmap
XFG	No	Extended Flow Guard. Much finer grained protection than CFG
CET	No	Strong defense against ROP attacks using shadow stacks

Figure 11.58 Some of the principal security protections in Windows.

Security Protection Mechanisms (2 of 2)

Mitigation	VBS-only	Description
KCET	Yes	Kernel-mode CET. Secure Kernel maintains shadow stacks.
PAC	No	Protects stack return addresses using signatures
CIG	No	Enforces that code binaries are properly signed
ACG	No	User-mode enforcement for WˆX and that data cannot become code
HVCI	Yes	Kernel-mode enforcement for WˆX and that data cannot become code
PatchGuard	No	Detect attempts to modify kernel code and data
HyperGuard	Yes	Stronger protection than PatchGuard
Windows Defender	No	Built-in antimalware software

Figure 11.58 Some of the principal security protections in Windows.(Continued)

Hotpatching

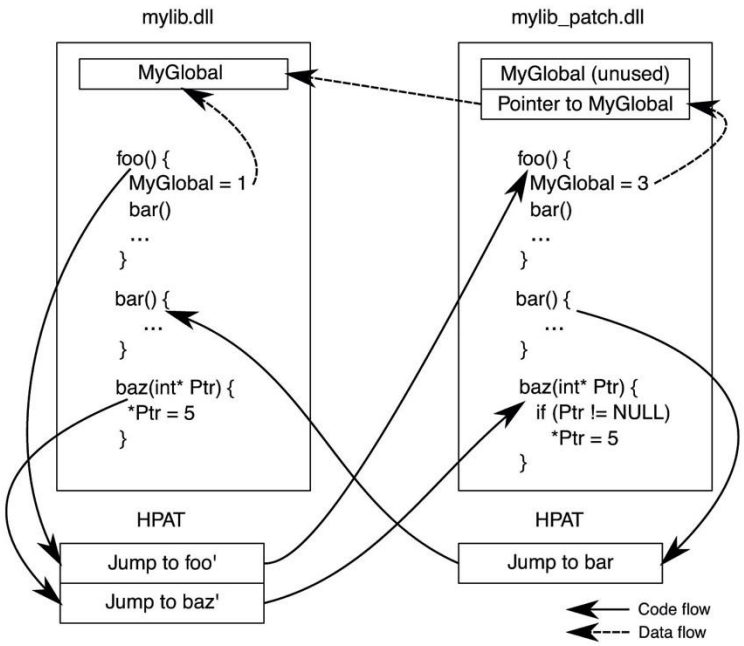

Figure 11.59 Hot patch application for my lib.dll. Functions foo( ) and Baz () are updated.