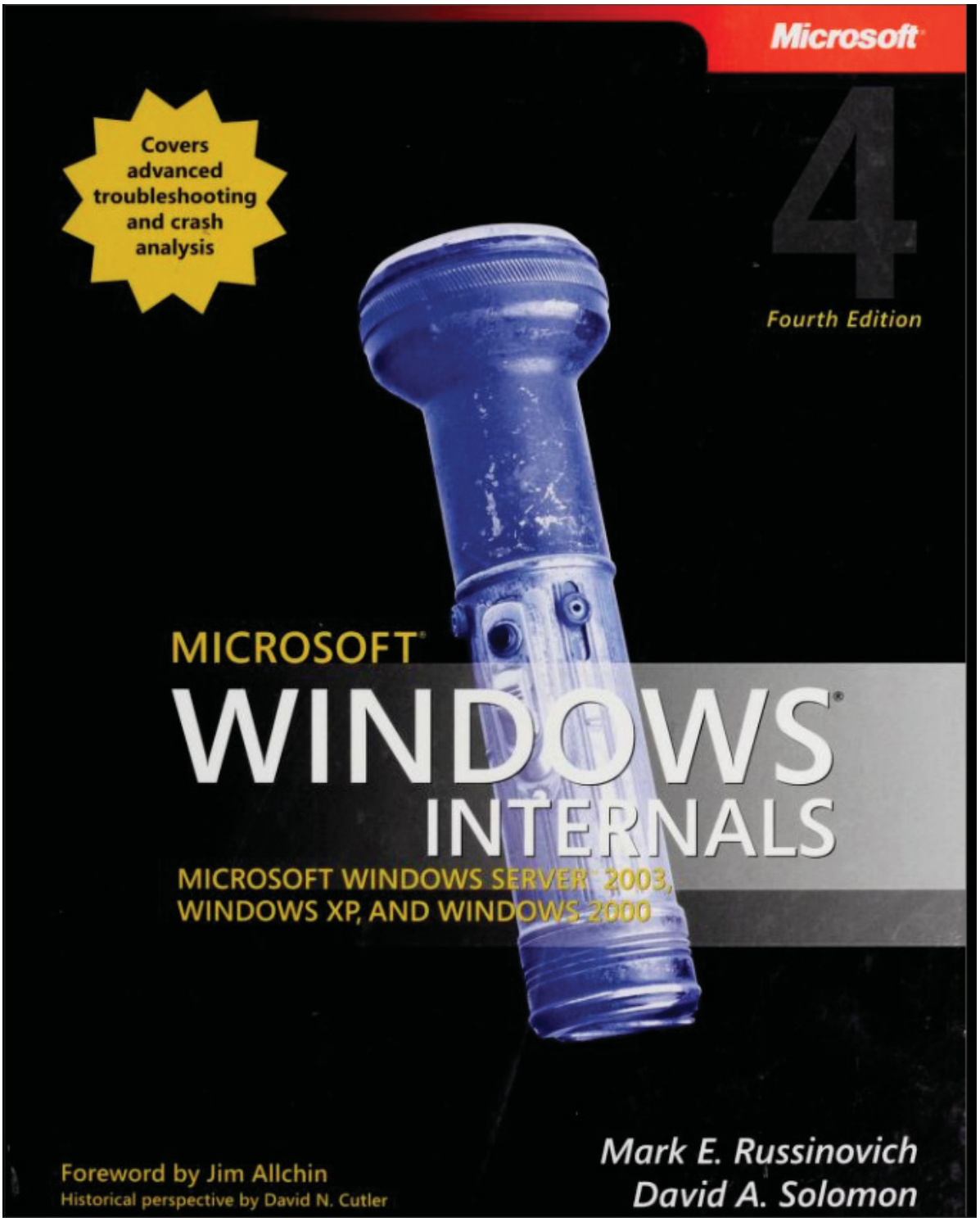
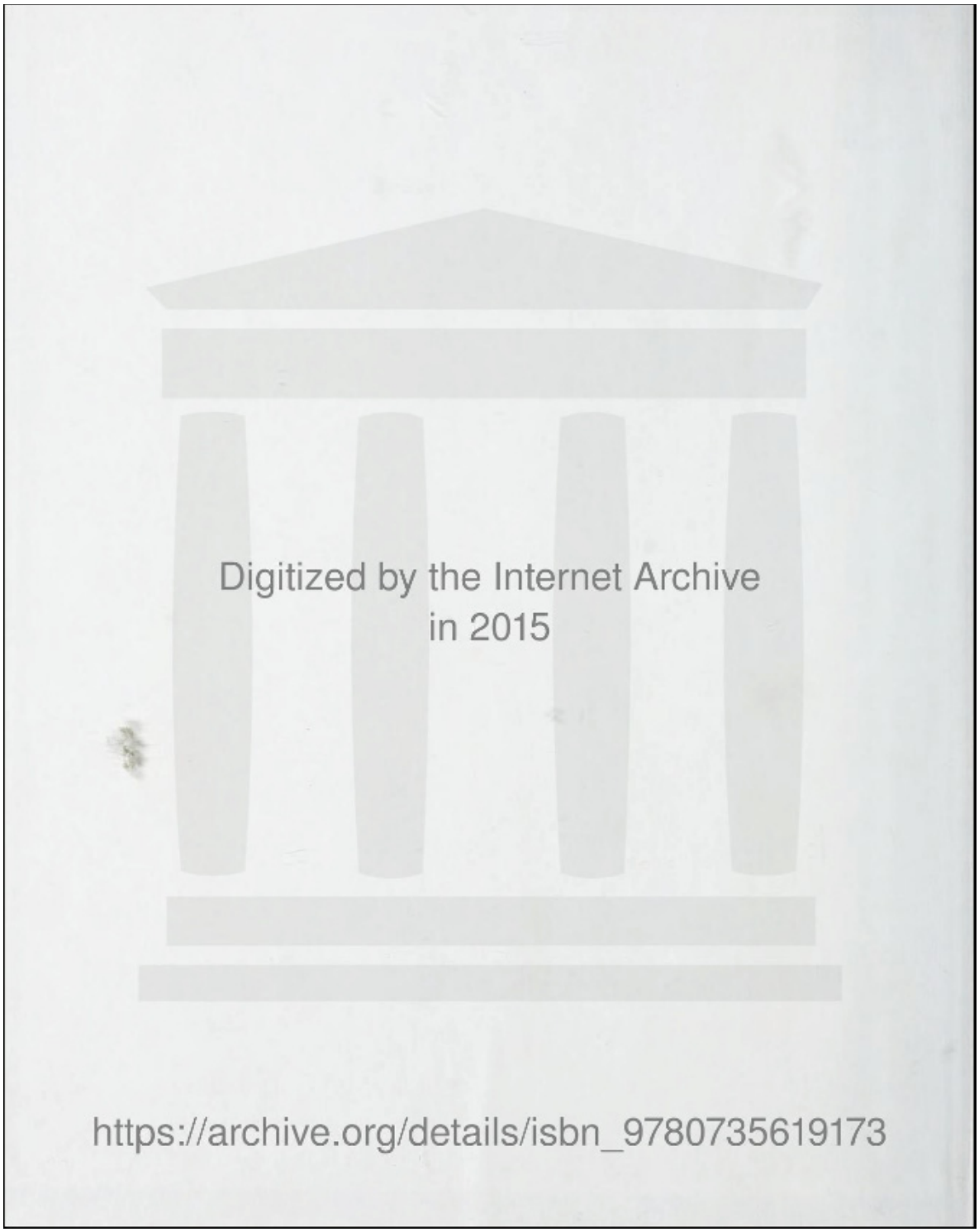
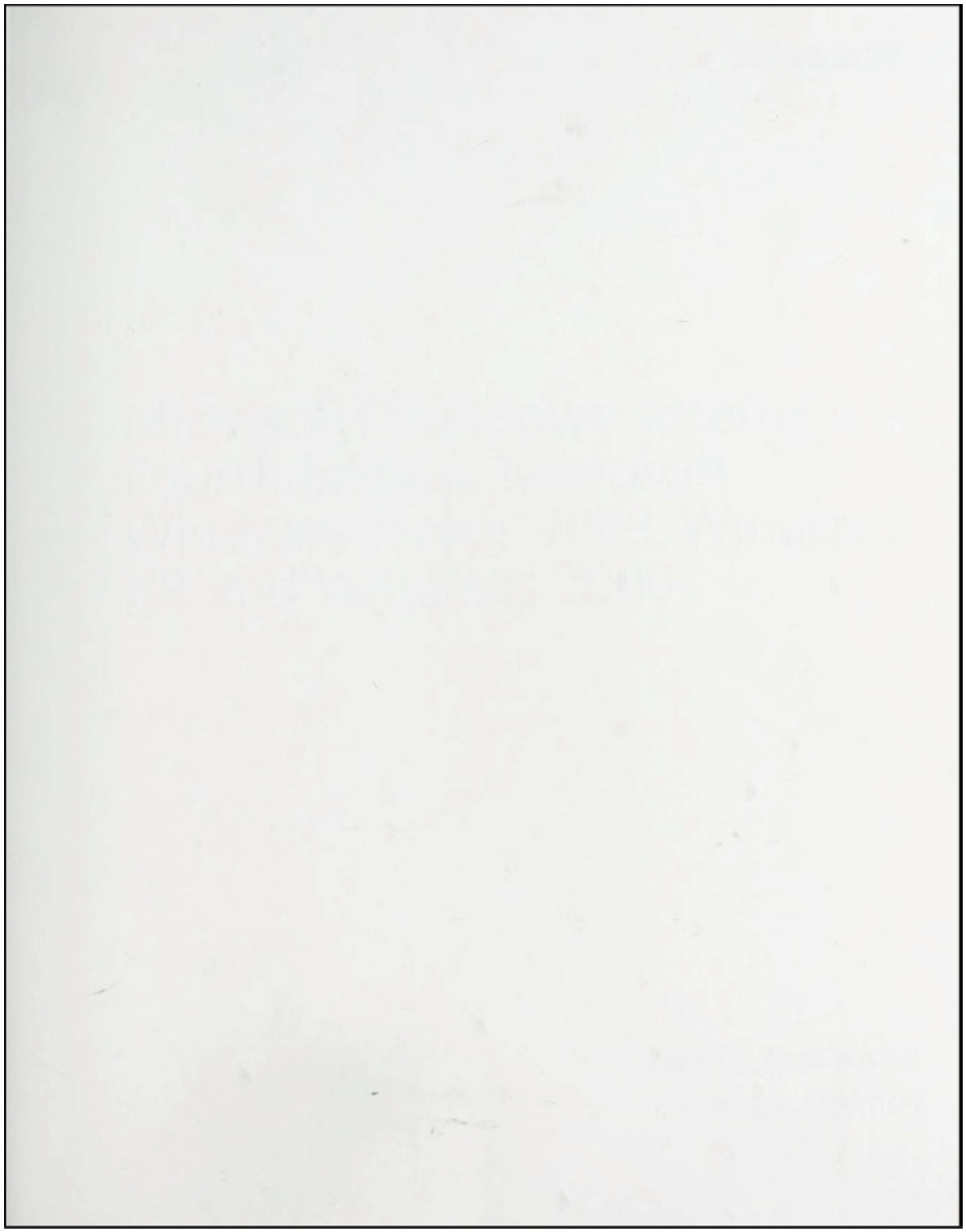


PAN EX1030
Palo Alto Networks v. Croga Innovations
IPR2024-01421


Digitized by the Internet Archive
in 2015

https://archive.org/details/isbn_9780735619173





Microsoft

Microsoft® Windows® Internals, Fourth Edition: Microsoft Windows Server™ 2003, Windows XP, and Windows 2000

Mark E. Russinovich
David A. Solomon

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Copyright © 2005 by David Solomon, Mark Russinovich

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Library of Congress Control Number 2004115221

Printed and bound in the United States of America.

4 5 6 7 8 9 QWT 9 8 7 6 5

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to mspinput@microsoft.com.

Microsoft, Active Desktop, Active Directory, ActiveX, DirectX, Microsoft Press, MSDN, MS-DOS, Outlook, PowerPoint, Visual Basic, Visual C++, Visual Studio, Win32, Windows, Windows NT, Windows Server, and WinFX are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other product and company names mentioned herein may be the trademarks of their respective owners.

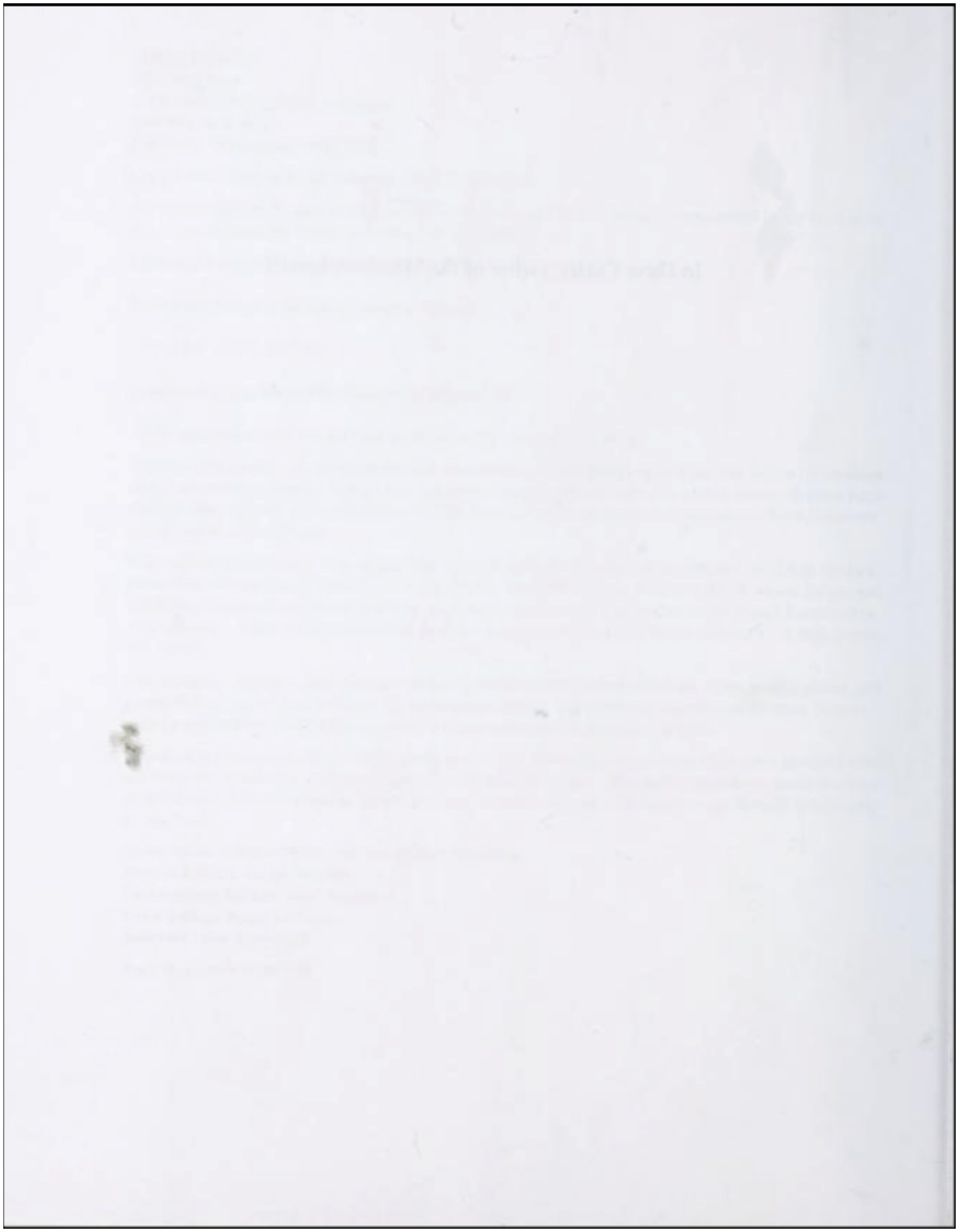
The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Acquisitions Editors: Robin Van Steenburgh, Ben Ryan
Project Editor: Valerie Woolley
Development Editor: Sally Stickney
Copy Editor: Roger LeBlanc
Indexer: Lynn Armstrong

Body Part No. X10-96109

To Dave Cutler, father of the Windows kernel



Contents at a Glance

1	Concepts and Tools	1
2	System Architecture	35
3	System Mechanisms	85
4	Management Mechanisms	183
5	Startup and Shutdown	251
6	Processes, Threads, and Jobs	289
7	Memory Management	375
8	Security	485
9	I/O System	537
10	Storage Management	615
11	Cache Manager	655
12	File Systems	689
13	Networking	787
14	Crash Dump Analysis	845

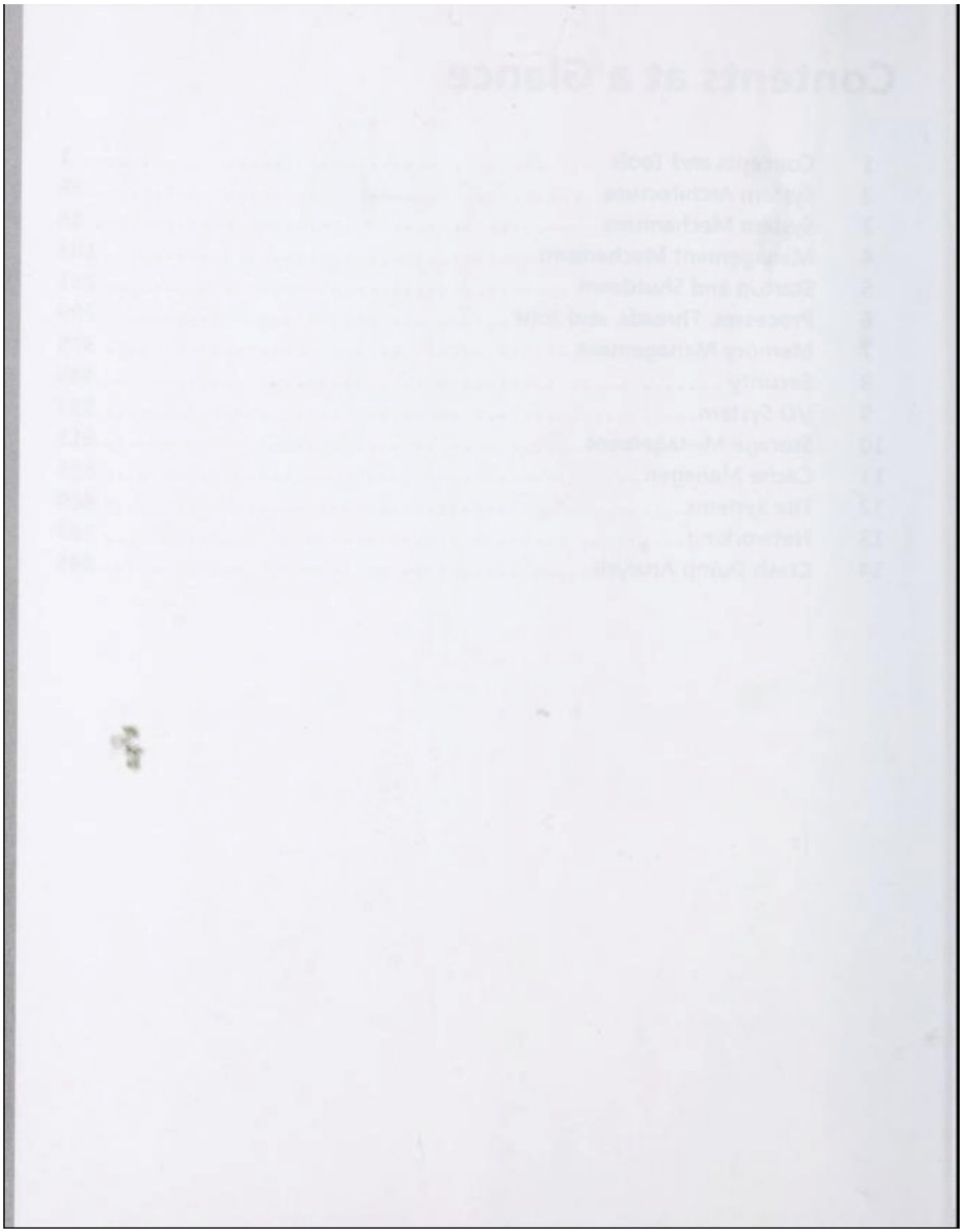


Table of Contents

Historical Perspective xix
Foreword xxiii
Acknowledgments xxv
Introduction xxvii

1 Concepts and Tools 1

Windows Operating System Versions 1
Foundation Concepts and Terms 3
Windows API 3
Services, Functions, and Routines 5
Processes, Threads, and Jobs 6
Virtual Memory 14
Kernel Mode vs. User Mode 16
Terminal Services and Multiple Sessions 21
Objects and Handles 22
Security 23
Registry 24
Unicode 25
Digging into Windows Internals 25
Performance Tool 27
Windows Support Tools 27
Windows Resource Kits 27
Kernel Debugging 28
Platform Software Development Kit (SDK) 33
Device Driver Kit (DDK) 34
Sysinternals Tools 34
Conclusion 34

2 System Architecture 35

Requirements and Design Goals 35
Operating System Model 36

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback about this publication so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit: *www.microsoft.com/learning/booksurvey/*

Architecture Overview . . . 37
Portability . . . 40
Symmetric Multiprocessing . . . 41
Scalability . . . 46
Differences Between Client and Server Versions . . . 47
Checked Build . . . 49
Key System Components . . . 51
Environment Subsystems and Subsystem DLLs . . . 53
Ntdll.dll . . . 63
Executive . . . 63
Kernel . . . 65
Hardware Abstraction Layer . . . 67
Device Drivers . . . 69
System Processes . . . 75
Conclusion . . . 84

3 System Mechanisms . . . 85

Trap Dispatching . . . 85
Interrupt Dispatching . . . 87
Exception Dispatching . . . 109
System Service Dispatching . . . 119
Object Manager . . . 124
Executive Objects . . . 126
Object Structure . . . 128
Synchronization . . . 149
High-IRQL Synchronization . . . 151
Low-IRQL Synchronization . . . 155
System Worker Threads . . . 166
Windows Global Flags . . . 168
Local Procedure Calls (LPCs) . . . 171
Kernel Event Tracing . . . 175
Wow64 . . . 178
Wow64 Process Address Space Layout . . . 179
System Calls . . . 179
Exception Dispatching . . . 179
User Callbacks . . . 179
File System Redirection . . . 180

Registry Redirection and Reflection 180
I/O Control Requests 181
16-bit Installer Applications 182
Printing 182
Restrictions 182
Conclusion 182

4 Management Mechanisms 183

The Registry 183
Viewing and Changing the Registry 183
Registry Usage 184
Registry Data Types 185
Registry Logical Structure 186
Troubleshooting Registry Problems 192
Registry Internals 197
Services 211
Service Applications 212
Service Accounts 217
The Service Control Manager 223
Service Startup 225
Startup Errors 229
Accepting the Boot and Last Known Good 230
Service Failures 231
Service Shutdown 232
Shared Service Processes 233
Service Control Programs 236
Windows Management Instrumentation 237
WMI Architecture 237
Providers 239
The Common Information Model and the Managed Object Format Language 240
The WMI Namespace 243
Class Association 244
WMI Implementation 247
WMI Security 248
Conclusion 249

5 Startup and Shutdown 251
Boot Process 251
x86 and x64 Preboot 251
The x86/x64 Boot Sector and Ntldr 255
The IA64 Boot Process 264
Initializing the Kernel and Executive Subsystems 266
Smss, Csrss, and Winlogon 269
Images that Start Automatically 273
Troubleshooting Boot and Startup Problems 274
Last Known Good 274
Safe Mode 274
Recovery Console 279
Solving Common Boot Problems 281
Shutdown 286
Conclusion 288

6 Processes, Threads, and Jobs 289
Process Internals 289
Data Structures 289
Kernel Variables 297
Performance Counters 297
Relevant Functions 298
Flow of CreateProcess 300
Stage 1: Opening the Image to Be Executed 302
Stage 2: Creating the Windows Executive Process Object 304
Stage 3: Creating the Initial Thread and Its Stack and Context 308
Stage 4 : Notifying the Windows Subsystem about the New Process 309
Stage 5: Starting Execution of the Initial Thread 310
Stage 6: Performing Process Initialization in the Context of the New Process 310
Thread Internals 313
Data Structures 313
Kernel Variables 320
Performance Counters 321
Relevant Functions 322
Birth of a Thread 322
Examining Thread Activity 323

Thread Scheduling . . . 325
Overview of Windows Scheduling . . . 326
Priority Levels . . . 327
Windows Scheduling APIs . . . 330
Relevant Tools . . . 331
Real-Time Priorities . . . 333
Thread States . . . 334
Dispatcher Database . . . 338
Quantum . . . 340
Scheduling Scenarios . . . 345
Context Switching . . . 347
Idle Thread . . . 348
Priority Boosts . . . 348
Multiprocessor Systems . . . 357
Multiprocessor Thread-Scheduling Algorithms . . . 366
Job Objects . . . 368
Conclusion . . . 373

7 Memory Management . . . 375

Introduction to the Memory Manager . . . 375
Memory Manager Components . . . 376
Internal Synchronization . . . 377
Configuring the Memory Manager . . . 378
Examining Memory Usage . . . 378
Services the Memory Manager Provides . . . 382
Large and Small Pages . . . 382
Reserving and Committing Pages . . . 384
Locking Memory . . . 385
Allocation Granularity . . . 385
Shared Memory and Mapped Files . . . 386
Protecting Memory . . . 388
No Execute Page Protection . . . 390
Copy-on-Write . . . 392
Heap Manager . . . 394
Address Windowing Extensions . . . 399
System Memory Pools . . . 401
Configuring Pool Sizes . . . 401
Monitoring Pool Usage . . . 404

Look-Aside Lists 408
Driver Verifier 409
Virtual Address Space Layouts 413
x86 User Address Space Layouts 415
x86 System Address Space Layout 417
x86 Session Space 418
System Page Table Entries 421
64-Bit Address Space Layouts 422
Address Translation 425
x86 Virtual Address Translation 425
Translation Look-Aside Buffer 434
Physical Address Extension (PAE) 435
IA-64 Virtual Address Translation 437
x64 Virtual Address Translation 438
Page Fault Handling 439
Invalid PTEs 440
Prototype PTEs 441
In-Paging I/O 443
Collided Page Faults 444
Page Files 444
Virtual Address Descriptors 448
Section Objects 450
Working Sets 457
Demand Paging 458
Logical Prefetcher 458
Placement Policy 462
Working Set Management 463
Balance Set Manager and Swapper 466
System Working Set 467
Page Frame Number Database 469
Page List Dynamics 472
Modified Page Writer 475
PFN Data Structures 476
Low and High Memory Notification 479
Conclusion 483

8 Security 485
Security System Components 488
Protecting Objects 492
Access Checks 493
Security Descriptors and Access Control 506
Account Rights and Privileges 516
Account Rights 517
Privileges 518
Super Privileges 523
Security Auditing 524
Logon 526
Winlogon Initialization 528
User Logon Steps 529
Software Restriction Policies 533
Conclusion 535

9 I/O System 537
I/O System Components 537
The I/O Manager 539
Typical I/O Processing 540
Device Drivers 541
Types of Device Drivers 541
Structure of a Driver 548
Driver Objects and Device Objects 550
Opening Devices 555
I/O Processing 561
Types of I/O 561
I/O Request Packets 564
I/O Request to a Single-Layered Driver 569
I/O Requests to Layered Drivers 577
I/O Completion Ports 585
Driver Verifier 589
The Plug and Play (PnP) Manager 590
Level of Plug and Play Support 591
Driver Support for Plug and Play 592
Driver Loading, Initialization, and Installation 594
Driver Installation 603

The Power Manager 607
Power Manager Operation 609
Driver Power Operation 610
Driver Control of Device Power 613
Conclusion 613

10 Storage Management 615
Storage Terminology 615
Disk Drivers 616
Ntldrx 616
Disk Class, Port, and Miniport Drivers 617
Disk Device Objects 620
Partition Manager 622
Volume Management 622
Basic Disks 624
Dynamic Disks 626
Multipartition Volume Management 632
The Volume Namespace 638
Volume I/O Operations 646
Virtual Disk Service 648
Volume Shadow Copy Service 649
Conclusion 654

11 Cache Manager 655
Key Features of the Cache Manager 655
Single, Centralized System Cache 656
The Memory Manager 656
Cache Coherency 656
Virtual Block Caching 658
Stream-Based Caching 658
Recoverable File System Support 658
Cache Virtual Memory Management 660
Cache Size 662
LargeSystemCache 662
Cache Virtual Size 663
Cache Working Set Size 665
Cache Physical Size 667

Cache Data Structures 668
Systemwide Cache Data Structures 669
Per-File Cache Data Structures 670
File System Interfaces 674
Copying to and from the Cache 676
Caching with the Mapping and Pinning Interfaces 677
Caching with the Direct Memory Access Interfaces 678
Fast I/O 679
Read Ahead and Write Behind 682
Intelligent Read-Ahead 682
Write-Back Caching and Lazy Writing 683
Write Throttling 686
System Threads 687
Conclusion 688

12 File Systems 689

Windows File System Formats 690
CDFS 690
UDF 691
FAT12, FAT16, and FAT32 691
NTFS 694
File System Driver Architecture 694
Local FSDs 695
Remote FSDs 696
File System Operation 700
File System Filter Drivers 705
Troubleshooting File System Problems 711
Filemon Basic vs. Advanced Modes 711
Filemon Troubleshooting Techniques 712
NTFS Design Goals and Features 717
High-End File System Requirements 717
Advanced Features of NTFS 719
NTFS File System Driver 729
NTFS On-Disk Structure 732
Volumes 732
Clusters 732
Master File Table 733

File Reference Numbers ... 739
File Records ... 740
Filenames ... 742
Resident and Nonresident Attributes ... 744
Data Compression and Sparse Files ... 747
The Change Journal File ... 752
Indexing ... 753
Object IDs ... 754
Quota Tracking ... 755
Consolidated Security ... 756
Reparse Points ... 758
NTFS Recovery Support ... 758
Evolution of File System Design ... 759
Logging ... 761
Recovery ... 767
NTFS Bad-Cluster Recovery ... 771
Encrypting File System Security ... 775
Encrypting a File for the First Time ... 778
The Decryption Process ... 783
Backing Up Encrypted Files ... 784
Conclusion ... 785

13 Networking ... 787

Windows Networking Architecture ... 787
The OSI Reference Model ... 787
Windows Networking Components ... 789
Networking APIs ... 791
Windows Sockets ... 791
Remote Procedure Call ... 798
Web Access APIs ... 803
Named Pipes and Mailslots ... 804
NetBIOS ... 811
Other Networking APIs ... 813
Multiple Redirector Support ... 815
Multiple Provider Router ... 816
Multiple UNC Provider ... 818

Name Resolution 820
Domain Name System 820
Windows Internet Name Service 820
Protocol Drivers 821
TCP/IP Extensions 824
NDIS Drivers 828
Variations on the NDIS Miniport 832
Connection-Oriented NDIS 832
Remote NDIS 835
QOS 836
Binding 838
Layered Network Services 839
Remote Access 839
Active Directory 840
Network Load Balancing 841
File Replication Service 843
Distributed File System 843
Conclusion 844

14 Crash Dump Analysis 845

Why Does Windows Crash? 845
The Blue Screen 846
Crash Dump Files 849
Crash Dump Generation 852
Windows Error Reporting 853
Online Crash Analysis 854
Basic Crash Dump Analysis 855
Notmyfault 855
Basic Crash Dump Analysis 856
Verbose Analysis 858
Using Crash Troubleshooting Tools 860
Buffer Overrun and Special Pool 861
Code Overwrite and System Code Write Protection 863
Advanced Crash Dump Analysis 864
Stack Trashes 865
Hung or Unresponsive Systems 866
When There Is No Crash Dump 869

Glossary 871

Index. 895

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback about this publication so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit: *www.microsoft.com/learning/booksurvey/*

Chapter 7

Memory Management

In this chapter, you'll learn how Microsoft Windows implements virtual memory and how it manages the subset of virtual memory kept in physical memory. We'll also describe the internal structure and components that make up the memory manager, including key data structures and algorithms. Before examining these mechanisms, let's review the basic services provided by the memory manager and key concepts such as reserved memory versus committed memory and shared memory.

Introduction to the Memory Manager

By default, the virtual size of a process on 32-bit Windows is 2 GB. If the image is marked specially as large address space aware, and the system is booted with a special switch (described later in this chapter), a 32-bit process can grow to 3 GB on 32-bit Windows and up to 4 GB on 64-bit Windows. The process virtual address space size on 64-bit Windows is 7152 GB on IA-64 systems and 8192 GB on x64 systems. (This value could be increased in future releases.)

As you saw in Chapter 2 (specifically in Table 2-4), the maximum amount of physical memory supported by Windows ranges from 2 GB to 1024 GB, depending on which version and edition of Windows you are running. Because the virtual address space might be larger or smaller than the physical memory on the machine, the memory manager has two primary tasks:

- Translating, or mapping, a process's virtual address space into physical memory so that when a thread running in the context of that process reads or writes to the virtual address space, the correct physical address is referenced. (The subset of a process's virtual address space that is physically resident is called the *working set*. Working sets are described in more detail later in this chapter.)
- Paging some of the contents of memory to disk when it becomes overcommitted—that is, when running threads or system code try to use more physical memory than is currently available—and bringing the contents back into physical memory when needed.

In addition to providing virtual memory management, the memory manager provides a core set of services on which the various Windows environment subsystems are built. These services include memory mapped files (internally called *section objects*), copy-on-write memory, and support for applications using large, sparse address spaces. In addition, the

memory manager provides a way for a process to allocate and use larger amounts of physical memory than can be mapped into the process virtual address space (for example, on 32-bit systems with more than 4 GB of physical memory). This is explained in the section "Address Windowing Extensions" later in this chapter.

Memory Manager Components

The memory manager is part of the Windows executive and therefore exists in the file Ntoskrnl.exe. No parts of the memory manager exist in the HAL. The memory manager consists of the following components:

- A set of executive system services for allocating, deallocating, and managing virtual memory, most of which are exposed through the Windows API or kernel-mode device driver interfaces
- A translation-not-valid and access fault trap handler for resolving hardware-detected memory management exceptions and making virtual pages resident on behalf of a process
- Several key components that run in the context of six different kernel-mode system threads:
  - The *working set manager* (priority 16), which the balance set manager (a system thread that the kernel creates) calls once per second as well as when free memory falls below a certain threshold, drives the overall memory management policies, such as working set trimming, aging, and modified page writing.
  - The *process/stack swapper* (priority 23) performs both process and kernel thread stack inswapping and outswapping. The balance set manager and the thread-scheduling code in the kernel awaken this thread when an inswap or outswap operation needs to take place.
  - The *modified page writer* (priority 17) writes dirty pages on the modified list back to the appropriate paging files. This thread is awakened when the size of the modified list needs to be reduced. (See the section "Modified Page Writer" to find out how you can change this default value.)
  - The *mapped page writer* (priority 17) writes dirty pages in mapped files to disk. It is awakened when the size of the modified list needs to be reduced or if pages for mapped files have been on the modified list for more than 5 minutes. This second modified page writer thread is necessary because it can generate page faults that result in requests for free pages. If there were no free pages and there was only one modified page writer thread, the system could deadlock waiting for free pages.

- The *dereference segment thread* (priority 18) is responsible for cache reduction as well as page file growth and shrinkage. (For example, if there is no virtual address space for paged pool growth, this thread trims the page cache so that the paged pool used to anchor it can be freed for reuse.)
- The *zero page thread* (priority 0) zeroes out pages on the free list so that a cache of zero pages is available to satisfy future demand-zero page faults. (Memory zeroing in some cases is done by a faster function called *MiZeroInParallel*. See the note in the section "Page List Dynamics.")

Each of these components is covered in more detail later in the chapter.

Internal Synchronization

Like all other components of the Windows executive, the memory manager is fully reentrant and supports simultaneous execution on multiprocessor systems—that is, it allows two threads to acquire resources in such a way that they don't corrupt each other's data. To accomplish the goal of being fully reentrant, the memory manager uses several different internal synchronization mechanisms to control access to its own internal data structures, such as spinlocks and executive resources. (Synchronization objects are discussed in Chapter 3.)

Systemwide resources to which the memory manager must synchronize access include the page frame number (PFN) database (controlled by a spinlock), section objects and the system working set (controlled by pushlocks), and page file creation (controlled by a mutex). In Windows XP and Windows Server 2003, a number of these locks have been either removed completely or optimized, resulting in much less contention. For example, in Windows 2000, spinlocks were used to synchronize changes to system address space and memory commitment; however to improve scalability, these spinlocks have been removed as of Windows XP. Per-process memory management data structures that require synchronization include the working set lock (held while changes are being made to the working set list) and the address space lock (held whenever the address space is being changed). Working set synchronization in Windows 2000 was implemented as a mutex; in Windows XP and later, a pushlock is used, improving parallelism and scalability, because pushlocks support both shared and exclusive access.

Other operations that no longer involve acquiring locks include charging nonpaged and paged pool quotas, charging commitment of pages, and allocating and mapping physical memory allocated through the address windowing extensions (AWE) functions. Also, the lock that synchronizes access to the structures that describe physical memory (the PFN database) is now acquired less and when acquired, held for less time. These changes translate into greater parallelism and scalability on multiprocessor systems because the number of times the memory manager might have to block while another CPU is making a change to a global structure has been reduced or eliminated.

Configuring the Memory Manager

Like most of Windows, the memory manager attempts to automatically provide optimal system performance for varying workloads on systems of varying sizes and types. While there are a limited number of registry values you can add and/or modify under the key HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management to override some of these default performance calculations, in general, the memory manager's default computations will be sufficient for the majority of workloads.

Many of the thresholds and limits that control memory manager policy decisions are computed at system boot time on the basis of memory size and product type. (Windows 2000 Professional and Windows XP Professional and Home editions are optimized for desktop interactive use, and Windows Server systems are optimized for running server applications.) These values are stored in various kernel variables and later used by the memory manager. To find some of these, search for global variables in Ntoskrnl.exe that have names beginning with *Mm* that contain the word "maximum" or "minimum." (See the experiment "Peering into Undocumented Interfaces" in Chapter 2.)



Warning Although you'll find references to some of these variables, you shouldn't change them. Windows has been tested to operate properly with the current possible permutations of these values that can be computed. Changing the value of these kernel variables on a running system can result in unpredictable system behavior, including system hangs or even crashes.

Examining Memory Usage

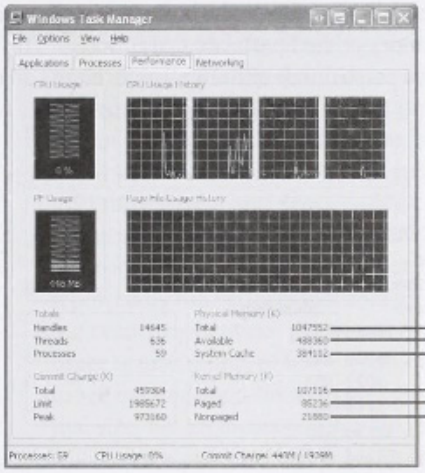
The Memory and Process performance counter objects provide access to most of the details about system and process memory utilization. Throughout the chapter, we'll include references to specific performance counters that contain information related to the component being described.

Besides the Performance tool, a number of tools in the Windows Support Tools and Windows resource kits display different subsets of memory usage information. We've included relevant examples and experiments throughout the chapter. One word of caution, however—different utilities use varying and sometimes inconsistent or confusing names when displaying memory information. The following experiment illustrates this point. (We'll explain the terms used in this example in subsequent sections.)



EXPERIMENT: Viewing System Memory Information

The Performance tab in the Windows Task Manager, shown in the following screen shot from Windows XP, displays basic system memory information. This information is a subset of the detailed memory information available through the performance counters.



Both Pmon.exe (in the Windows Support Tools) and Pstat.exe (in the Platform SDK) display system and process memory information. The annotations in the following output from Pstat explain the information reported. (For an explanation of the commit total and limit, see Table 7-15.)

Total physical memory

Total of standby, free, zero lists

Total of all process working sets (*not* the total process memory usage, which would double count shared pages)

Resident system code

Resident paged pool

```
Memory:  64692K Avail: 3676K TotalWs:  62836K  InRam Kernel: 3788K     P: 9604K
Commit: 111848K/85808K  Limit: 182400K Peak: 113732K  Pool N: 2344K    P:14448K
```

Nonpaged pool physical size

Paged pool virtual size

System working set size (*not* just the size of the file cache)

```
User Time    Kernel Time     Ws    Faults Commit Pri  Hnd  Thd Pid  Name
                          20032-  194378                            File Cache
0:00:00.000  8:03:55.734    16        1      0   0    0    2   0  Idle Process
0:00:00.000  0:01:24.421    32     1601     36   8  140   27   2  System
0:00:00.140  0:00:00.468   200     2008    164  11   35    6  20  smss.exe
0:00:00.640  0:00:02.843   716     8625   1588  13  145    9  30  csrss.exe
```

To see the specific usage of paged and nonpaged pool, use the Poolmon utility, described in the "Monitoring Pool Usage" section.

Finally, the *!vm* command in the kernel debugger shows the basic memory management information available through the memory-related performance counters. This command can be useful if you're looking at a crash dump or hung system. Here's an example of its output:

```
kd> !vm

*** Virtual Memory Usage ***
        Physical Memory:     32620   (  130480 Kb)
        Page File: \??\C:\pagefile.sys
           Current:    204800Kb Free Space:    101052Kb
           Minimum:    204800Kb Maximum:       204800Kb
        Available Pages:      3604   (   14416 Kb)
        ResAvail Pages:      24004   (   96016 Kb)
        Modified Pages:        768   (    3072 Kb)
        NonPagedPool Usage:   1436   (    5744 Kb)
        NonPagedPool Max:    12940   (   51760 Kb)
        PagedPool 0 Usage:    6817   (   27268 Kb)
        PagedPool 1 Usage:     982   (    3928 Kb)
        PagedPool 2 Usage:     984   (    3936 Kb)
        PagedPool Usage:      8783   (   35132 Kb)
        PagedPool Maximum:   26624   (  106496 Kb)
        Shared Commit:        1361   (    5444 Kb)
        Special Pool:            0   (       0 Kb)
        Free System PTEs:   189291   (  757164 Kb)
        Shared Process:       3165   (   12660 Kb)
        PagedPool Commit:     8783   (   35132 Kb)
        Driver Commit:        1098   (    4392 Kb)
        Committed pages:     45113   (  180452 Kb)
        Commit limit:        79556   (  318224 Kb)

        Total Private:       30536   (  122144 Kb)
            IEXPLORE.EXE      3028   (   12112 Kb)
            svchost.exe       2128   (    8512 Kb)
            WINWORD.EXE       1971   (    7884 Kb)
            POWERPNT.EXE      1905   (    7620 Kb)
            Acrobat.exe       1761   (    7044 Kb)
            winlogon.exe      1361   (    5444 Kb)
            explorer.exe      1300   (    5200 Kb)
            livekd.exe        1015   (    4060 Kb)
            hh.exe             960   (    3840 Kb)
```



EXPERIMENT: Accounting for Physical Memory Use

By combining information available from performance counters with output from kernel debugger commands, you can come close to accounting for physical memory usage on a machine running Windows. To examine the memory usage information available through performance counters, run the Performance tool and add the counters to view the following information. (You'll see the results more easily if you change the vertical scale maximum on the graph to 1000.)

- **Total process working set size** To view this information, select the Process performance object and the Working Set counter for the _Total process instance. This number will be larger than the actual total process memory utilization because shared pages are counted in each process working set. To get a more accurate picture of process memory utilization, subtract free memory (available bytes), operating system memory used (nonpaged pool, resident paged pool, and resident operating system and driver code), and the size of the modified list from the total physical memory on the machine. What you're left with is the memory being used by processes. Comparing this value against the total process working set size as reported by the Performance tool gives you some indication of the amount of sharing occurring between processes. Although examining process physical memory usage is interesting, of more concern is the private committed virtual memory usage by processes, because memory leaks show up as an increasing private virtual size, not an increasing working set size. At some point, the memory manager will stop the process from growing in physical size, though it can continue to grow in its virtual size until the systemwide commit limit—the maximum amount of private committed virtual memory available on the system—is reached (or less if the process is a member of a job with a job-wide or process virtual memory limit). For more information, see the section "Page Files."
- **Total system working set size** To view this information, select the Memory processor object and the Cache Bytes counter. As explained in the section "System Working Set," the total system working set size includes more than just the cache size—it includes the subset of paged pool, pageable operating system code, and pageable driver code that is resident and in the system working set.
- **Size of nonpaged pool** View this information by adding the Memory: Pool Nonpaged Bytes counter.
- **Size of the free, zero, and standby lists** View the sizes of these lists by adding the Memory: Available Bytes counter. (Use the *!memusage* kernel debugger command to get the size of each of the lists separately.)

Your graph or report now contains a representation of all of physical memory except for two components that you can't obtain from a performance counter:

- Nonpaged operating system and driver code
- The modified and modified-no-write paging lists

Although you can easily obtain the size of both the modified and modified-no-write lists by using the kernel debugger *!memusage* command, there's no easy way to get the size of the nonpaged operating system and driver code.

Services the Memory Manager Provides

The memory manager provides a set of system services to allocate and free virtual memory, share memory between processes, map files into memory, flush virtual pages to disk, retrieve information about a range of virtual pages, change the protection of virtual pages, and lock the virtual pages into memory.

Like other Windows executive services, the memory management services allow their caller to supply a process handle, indicating the particular process whose virtual memory is to be manipulated. The caller can thus manipulate either its own memory or (with the proper permissions) the memory of another process. For example, if a process creates a child process, by default it has the right to manipulate the child process's virtual memory. Thereafter, the parent process can allocate, deallocate, read, and write memory on behalf of the child process by calling virtual memory services and passing a handle to the child process as an argument. This feature is used by subsystems to manage the memory of their client processes, and it is also key for implementing debuggers because debuggers must be able to read and write to the memory of the process being debugged.

Most of these services are exposed through the Windows API. The Windows API has three groups of functions for managing memory in applications: page granularity virtual memory functions (*Virtualxxx*), memory-mapped file functions (*CreateFileMapping*, *MapViewOfFile*), and heap functions (*Heapxxx* and the older interfaces *Localxxx* and *Globalxxx*). (We'll describe the heap manager later in this section.)

The memory manager also provides a number of services, such as allocating and deallocating physical memory and locking pages in physical memory for direct memory access (DMA) transfers, to other kernel-mode components inside the executive as well as to device drivers. These functions begin with the prefix *Mm*. In addition, though not strictly part of the memory manager, executive support routines that begin with *Ex* are used to allocate and deallocate from the system heaps (paged and nonpaged pool) as well as to manipulate look-aside lists. We'll touch on these topics later in this chapter, in the section "System Memory Pools."

Although we'll be referring to Windows functions and kernel-mode memory management and memory allocation routines provided for device drivers, we won't cover the interface and programming details but rather the internal operations of these functions. Refer to the Platform Software Development Kit (SDK) and Device Driver Kit (DDK) documentation on MSDN for a complete description of the available functions and their interfaces.

Large and Small Pages

The virtual address space is divided into units called pages. That is because the hardware memory management unit translates virtual to physical addresses at the granularity of a page. Hence, a page is the smallest unit of protection at the hardware level. (The various page protection options are described in the section "Protecting Memory.") There are two page sizes: small and large. The actual sizes vary based on hardware architecture, and they are listed in Table 7-1.

Table 7-1 Page Sizes

Architecture	Small Page Size	Large Page Size
x86	4 KB	4 MB (2 MB on PAE systems)
x64	4 KB	2 MB
IA64	8 KB	16 MB

The advantage of large pages is speed of address translation for references to other data within the large page. This advantage exists because the first reference to any byte within a large page will cause the hardware's translation look-aside buffer (or TLB, which is described in the section "Translation Look-Aside Buffer") to have in its cache the information necessary to translate references to any other byte within the large page. If small pages are used, more TLB entries are needed for the same range of virtual addresses, thus increasing recycling of entries as new virtual addresses require translation. This, in turn, means having to go back to the page table structures when references are made to virtual addresses outside the scope of a small page whose translation has been cached. The TLB is a very small cache, and thus large pages make better use of this limited resource.

To take advantage of large pages, on systems considered to have enough memory (see Table 7-2 for the minimums), Windows maps with large pages the core operating system images (Ntoskrnl.exe and Hal.dll) as well as core operating system data (such as the initial part of nonpaged pool and the data structures that describe the state of each physical memory page). Windows also automatically maps I/O space requests (calls by device drivers to *MmMapIoSpace*) with large pages if the request is of satisfactory large page length and alignment. Lastly, Windows also allows applications to map their images, private memory and pagefile-backed sections with large pages. (See the MEM_LARGE_PAGE flag on the *VirtualAlloc* function.) You can also specify other device drivers to be mapped with large pages by adding a multistring registry value HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\LargePageDrivers and specifying the names of the drivers as separately null terminated strings.

Table 7-2 Large Page Minimums

Operating System	Minimum Memory to Use Large Pages
Windows 2000	>127 MB
Windows XP, Windows Server 2003	>255 MB

One side-effect of large pages is that because each large page must be mapped with a single protection (because hardware memory protection is on a per-page basis), if a large page contains both read-only code and read/write data, the page must be marked as read/write, which means that the code will be writable. This means device drivers or other kernel mode code could, as a result of a bug, modify what is supposed to be read-only operating system or driver code without causing a memory access violation. However, if small pages are used to map the kernel, the read-only portions of NTOSKRNL.EXE and HAL.DLL will be mapped as read-only pages. Although this reduces efficiency of address translation, if a device driver (or other ker-

nel mode code) attempts to modify a read-only part of the operating system, the system will crash immediately with the finger pointing at the offending instruction, as opposed to allowing the corruption to occur and later the system crashing (in a harder-to-diagnose way) when some other component trips over their corrupted data. If you suspect you are experiencing kernel code corruptions, enable Driver Verifier (described later in this chapter)—this will disable the use of large pages.

Reserving and Committing Pages

Pages in a process address space are free, reserved, or committed. Applications can first *reserve* address space and then *commit* pages in that address space. Or they can reserve and commit in the same function call. These services are exposed through the Windows *VirtualAlloc* and *VirtualAllocEx* functions.

Reserved address space is simply a way for a thread to reserve a range of virtual addresses for future use. Attempting to access reserved memory results in an access violation because the page isn't mapped to any storage that can resolve the reference.

Committed pages are pages that, when accessed, ultimately translate to valid pages in physical memory. Committed pages are either private and not shareable or mapped to a view of a section (which might or might not be mapped by other processes). Sections are described in two upcoming sections: "Shared Memory and Mapped Files" and "Section Objects."

If the pages are private to the process and have never been accessed before, they are created at the time of first access as zero-initialized pages (or *demand zero*). Private committed pages can later be automatically written to the paging file by the operating system if memory demands dictate. Committed pages that are private are inaccessible to any other process unless they're accessed using cross-process memory functions, such as *ReadProcessMemory* or *WriteProcessMemory*. If committed pages are mapped to a portion of a mapped file, they might need to be brought in from disk when accessed unless they've already been read earlier, either by the process accessing the page or by another process that had the same file mapped and had previously accessed the page. (See the section "Shared Memory and Mapped Files" later in this chapter.)

Pages are written to disk through normal modified page writing as pages are moved from the process working set to the modified list and ultimately to disk. (Working sets and the modified list are explained later in this chapter.) Mapped file pages can also be written back to disk as a result of an explicit call to *FlushViewOfFile*.

You can decommit pages and/or release address space with the *VirtualFree* or *VirtualFreeEx* function. The difference between decommittal and release is similar to the difference between reservation and committal—decommitted memory is still reserved, but released memory is neither committed nor reserved. (It's freed.)

Using the two-step process of reserving and committing memory can reduce memory usage by deferring committing pages until needed but keeping the convenience of virtual contiguity.

Reserving memory is a relatively fast and inexpensive operation under Windows because it doesn't consume any committed pages (a precious system resource) or process page file quota (a limit on the number of committed pages a process can consume—not necessarily page file space). All that needs to be updated or constructed is the relatively small internal data structures that represent the state of the process address space. (We'll explain these data structures, called *virtual address descriptors*, or VADs, later in the chapter.)

Reserving and then committing memory is useful for applications that need a potentially large virtually contiguous memory buffer; rather than committing pages for the entire region, the address space can be reserved and then committed later when needed. A utilization of this technique in the operating system is the user-mode stack for each thread. When a thread is created, a stack is reserved. (1 MB is the default; you can override this size with the *CreateThread* function call or on an imagewide basis by using the /STACK linker flag.) By default, the initial page in the stack is committed and the next page is marked as a guard page, which isn't committed, that traps references beyond the end of the committed portion of the stack and expands it.

Locking Memory

In general, it's better to let the memory manager decide which pages remain in physical memory. However, there might be special circumstances where this is necessary. Pages can be locked in memory in two ways:

- Windows applications can call the *VirtualLock* function to lock pages in their process working set. The number of pages a process can lock can't exceed its minimum working set size minus eight pages. Therefore, if a process needs to lock more pages, it can increase its working set minimum with the *SetProcessWorkingSetSize* function (referred to in the section "Working Set Management").
- Device drivers can call the kernel-mode functions *MmProbeAndLockPages*, *MmLockPagableCodeSection*, *MmLockPagableDataSection*, or *MmLockPagableSectionByHandle*. Pages locked using this mechanism remain in memory until explicitly unlocked. Although no quota is imposed on the number of pages a driver can lock in memory, a driver can't lock more pages than the resident available page count will allow.

Allocation Granularity

Windows aligns each region of reserved process address space to begin on an integral boundary defined by the value of the system *allocation granularity*, which can be retrieved from the Windows *GetSystemInfo* function. Currently, this value is 64 KB. This size was chosen so that if support were added for future processors with large page sizes (for example, up to 64 KB) or virtually indexed caches that require systemwide physical-to-virtual page alignment, the risk of requiring changes to applications that made assumptions about allocation alignment would be reduced. (Windows kernel-mode code isn't subject to the same restrictions; it can reserve memory on a single-page granularity.)

Finally, when a region of address space is reserved, Windows ensures that the size and base of the region is a multiple of the system page size, whatever that might be. For example, because x86 systems use 4-KB pages, if you tried to reserve a region of memory 18 KB in size, the actual amount reserved on an x86 system would be 20 KB. If you specified a base address of 3 KB for an 18-KB region, the actual amount reserved would be 24 KB.

Shared Memory and Mapped Files

As is true with most modern operating systems, Windows provides a mechanism to share memory among processes and the operating system. *Shared memory* can be defined as memory that is visible to more than one process or that is present in more than one process virtual address space. For example, if two processes use the same DLL, it would make sense to load the referenced code pages for that DLL into physical memory only once and share those pages between all processes that map the DLL, as illustrated in Figure 7-1.

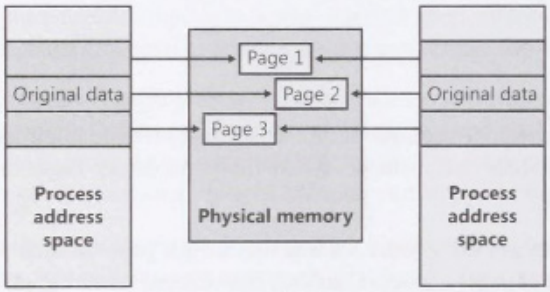


Figure 7-1 Sharing memory between processes

Each process would still maintain its private memory areas in which to store private data, but the program instructions and unmodified data pages could be shared without harm. As we'll explain later, this kind of sharing happens automatically because the code pages in executable images are mapped as execute-only and writable pages are mapped copy-on-write. (See the section "Copy-on-Write" for more information.)

The underlying primitives in the memory manager used to implement shared memory are called *section objects*, which are called *file mapping objects* in the Windows API. The internal structure and implementation of section objects are described in the section "Section Objects" later in this chapter.

This fundamental primitive in the memory manager is used to map virtual addresses, whether in main memory, in the page file, or in some other file that an application wants to access as if it were in memory. A section can be opened by one process or by many; in other words, section objects don't necessarily equate to shared memory.

A section object can be connected to an open file on disk (called a mapped file) or to committed memory (to provide shared memory). Sections mapped to committed memory are called

page file backed sections because the pages are written to the paging file if memory demands dictate. (Because Windows can run with no paging file, page file backed sections might in fact be "backed" only by physical memory.) As with any other page that is made visible to user mode (such as private committed pages), shared committed pages are always zero-filled when they are first accessed.

To create a section object, call the Windows *CreateFileMapping* function, specifying the file handle to map it to (or INVALID_HANDLE_VALUE for a page file backed section), and optionally a name and security descriptor. If the section has a name, other processes can open it with *OpenFileMapping*. Or you can grant access to section objects through handle inheritance (by specifying that the handle be inheritable when opening or creating the handle) or handle duplication (by using *DuplicateHandle*). Device drivers can also manipulate section objects with the *ZwOpenSection*, *ZwMapViewOfSection*, and *ZwUnmapViewOfSection* functions.

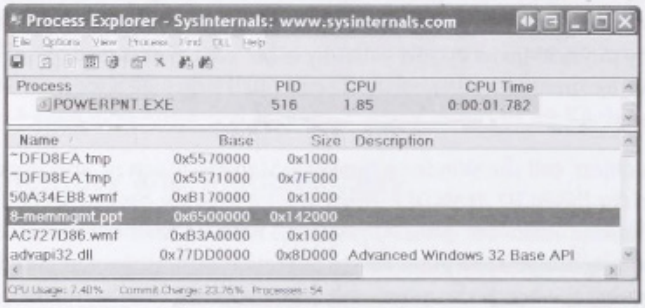
A section object can refer to files that are much larger than can fit in the address space of a process. (If the paging file backs a section object, sufficient space must exist in the paging file to contain it.) To access a very large section object, a process can map only the portion of the section object that it requires (called a *view* of the section) by calling the *MapViewOfFile* function and then specifying the range to map. Mapping views permits processes to conserve address space because only the views of the section object needed at the time must be mapped into memory.

Windows applications can use mapped files to conveniently perform I/O to files by simply making them appear in their address space. User applications aren't the only consumers of section objects: the image loader uses section objects to map executable images, DLLs, and device drivers into memory, and the cache manager uses them to access data in cached files. (For information on how the cache manager integrates with the memory manager, see Chapter 11.) How shared memory sections are implemented, both in terms of address translation and the internal data structures, is explained later in this chapter.



EXPERIMENT: Viewing Memory Mapped Files

You can list the memory mapped files in a process by using Process Explorer from Sysinternals. To view the memory mapped files by using Process Explorer, configure the lower pane to show the DLL view. (Click on View, Lower Pane View, DLLs.) Note that this is more than just a list of DLLs—it represents all memory mapped files in the process address space. Some of these are DLLs, one is the image file (EXE) being run, and additional entries might represent memory mapped data files. For example, the following display from Process Explorer shows a Microsoft PowerPoint process that has memory mapped the PowerPoint file being edited into its address space:



You can also search for memory mapped files by clicking on Find, DLL. This can be useful when trying to determine which process(es) are using a DLL that you are trying to replace.

Finally, comparing the list of DLLs loaded in a process with another instance of the same program running on another system might help point to DLL configuration issues, such as the wrong version of a DLL getting loaded in a process. This problem is known affectionately as "DLL hell."

Protecting Memory

As explained in Chapter 1, Windows provides memory protection so that no user process can inadvertently or deliberately corrupt the address space of another process or the operating system itself. Windows provides this protection in four primary ways.

First, all systemwide data structures and memory pools used by kernel-mode system components can be accessed only while in kernel mode—user-mode threads can't access these pages. If they attempt to do so, the hardware generates a fault, which in turn the memory manager reports to the thread as an access violation.



Note In contrast, Microsoft Windows 95, Microsoft Windows 98, and Microsoft Windows Millennium Edition have some pages in system address space that are writable from user mode, thus allowing an errant application to corrupt key system data structures and crash the system.

Second, each process has a separate, private address space, protected from being accessed by any thread belonging to another process. The only exceptions are if the process decides to share pages with other processes or if another process has virtual memory read or write access to the process object and thus can use the *ReadProcessMemory* or *WriteProcessMemory* functions. Each time a thread references an address, the virtual memory hardware, in concert with the memory manager, intervenes and translates the virtual address into a physical one. By controlling how virtual addresses are translated, Windows can ensure that threads running in one process don't inappropriately access a page belonging to another process.

Third, in addition to the implicit protection virtual-to-physical address translation offers, all processors supported by Windows provide some form of hardware-controlled memory protection (such as read/write, read-only, and so on); the exact details of such protection vary according to the processor. For example, code pages in the address space of a process are marked read-only and are thus protected from modification by user threads.

Table 7-3 lists the memory protection options defined in the Windows API. (See the *VirtualProtect*, *VirtualProtectEx*, *VirtualQuery*, and *VirtualQueryEx* functions.)

Table 7-3 Memory Protection Options Defined in the Windows API

Attribute	Description
PAGE_NOACCESS	Any attempt to read from, write to, or execute code in this region causes an access violation.
PAGE_READONLY	Any attempt to write to (and on processors with no execute support, execute code in) memory causes an access violation, but reads are permitted.
PAGE_READWRITE	The page is readable and writable, but not executable.
PAGE_EXECUTE*	Any attempt to write to code in memory in this region causes an access violation, but execution (and read on all existing processors) is permitted.
PAGE_EXECUTE_READ*	Any attempt to write to code in memory in this region causes an access violation, but executes and reads are permitted.
PAGE_EXECUTE_READWRITE*	The page is readable, writable, and executable—no action will cause an access violation.
PAGE_WRITECOPY	Any attempt to write to memory in this region causes the system to give the process a private copy of the page. On processors with no execute support, attempts to execute code in memory in this region cause an access violation.
PAGE_EXECUTE_WRITECOPY	Any attempt to write to memory in this region causes the system to give the process a private copy of the page. Reading and executing code in this region is permitted. (No copy is made in this case.)
PAGE_GUARD	Any attempt to read from or write to a guard page raises an EXCEPTION_GUARD_PAGE exception and turns off the guard page status. Guard pages thus act as a one-shot alarm. Note that this flag can be specified with any of the page protections listed in this table except PAGE_NOACCESS.
PAGE_NOCACHE	Use physical memory that is not cached. This is not recommended for general usage. It is useful for device drivers—for example, mapping a video frame buffer with no caching.
PAGE_WRITECOMBINE	Enables write-combined memory accesses. When enabled, the processor might cache memory write requests to optimize performance. For example, if multiple writes are made to the same address, only the most recent write might occur.

* No execute protection is supported by Windows XP Service Pack 2 and Windows Server 2003 Service Pack 1 and later on processors that have the necessary hardware support (for example, the x64, IA-64, and future x86 processors). On earlier versions of Windows and on processors that do not support no execute protection, all page permissions allow execution. The next section contains a more complete description of no execute protection.

And finally, shared memory section objects have standard Windows access-control lists (ACLs) that are checked when processes attempt to open them, thus limiting access of shared memory to those processes with the proper rights. Security also comes into play when a thread creates a section to contain a mapped file. To create the section, the thread must have at least read access to the underlying file object or the operation will fail.

Once a thread has successfully opened a handle to a section, its actions are still subject to the memory manager and the hardware-based page protections described earlier. A thread can change the page-level protection on virtual pages in a section if the change doesn't violate the permissions in the ACL for that section object. For example, the memory manager allows a thread to change the pages of a read-only section to have copy-on-write access but not to have read/write access. The copy-on-write access is permitted because it has no effect on other processes sharing the data.

These four primary memory protection mechanisms are part of the reason that Windows is a robust, reliable operating system that is impervious to and resilient to application errors.

No Execute Page Protection

Although the Windows memory management API has always had page protection bits defined in the programming interface that allow specification of whether or not pages can contain executable code, it is only as of Windows XP Service Pack 2 and Windows Server 2003 Service Pack 1 that this capability is supported on processors that have hardware "no execute" protection, including all AMD64 processors (including AMD Athlon64 and AMD Opteron) and certain exclusively 32-bit AMD processors (selected AMD Sempron processors—details are in AMD's product literature), Intel IA-64, and Intel Pentium 4 and Xeon processors with Intel Extended Memory 64 Technology (EM64T).

No execute page protection (also referred to as data execution prevention, or DEP) means an attempt to transfer control to an instruction in a page marked as "no execute" will generate an access fault. This can prevent certain types of viruses from exploiting bugs in the system that permit the execution of code placed in a data page. If an attempt is made in kernel mode to execute code in a page marked as no execute, the system will crash with the ATTEMPTED_EXECUTE_OF_NOEXECUTE_MEMORY bugcheck code. If this occurs in user mode, a STATUS_ACCESS_VIOLATION (0xc0000005) exception is delivered to the thread attempting the illegal reference. If a process allocates memory that needs to be executable, it must explicitly mark such pages by specifying the PAGE_EXECUTE, PAGE_EXECUTE_READ, PAGE_EXECUTE_READWRITE, or PAGE_EXECUTE_WRITECOPY flags on the page granularity memory allocation functions.

On 64-bit versions of Windows, execution protection is always applied to all 64-bit programs and device drivers and cannot be disabled. Execution protection for 32-bit programs depends on system configuration settings, described shortly. On 64-bit Windows, execution protection is applied to thread stacks (both user and kernel mode), user mode pages not specifically marked as executable, kernel paged pool, and kernel session pool (for a description of kernel memory pools, see the section "System Memory Pools"). However, on 32-bit Windows,

execution protection is only applied to thread stacks and user mode pages, not to paged pool and session pool. Also, when execution protection is enabled on 32-bit Windows, the system automatically boots in PAE mode (automatically selecting the PAE kernel, \Windows\System32\Ntkrnlpa.exe). For a description of PAE, see the section "Physical Address Extension (PAE)."

The application of execution protection for 32-bit programs depends on the Boot.ini /NOEXECUTE= switch. The settings can be changed by going to the Data Execution Prevention tab under My Computer, Properties, Advanced, Performance Settings. (See Figure 7-2.) When you configure no execute protection with the DEP settings dialog box, Boot.ini is modified to add the appropriate /NOEXECUTE Boot.ini switch. For a list of the variations of the switch and how they correspond to the DEP settings tab, see Table 7-4. 32-bit applications that are excluded from execution protection are listed as registry values under the key HKLM\Software\Microsoft\Windows NT\CurrentVersion\AppCompatFlags\Layers with the value name being the full path of the executable and the data set to "DisableNXShowUI".

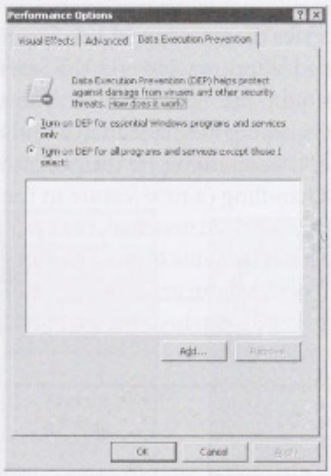


Figure 7-2 Data Execution Protection settings

On Windows XP (both 64-bit and 32-bit versions), execution protection for 32-bit applications is configured by default to apply only to core Windows operating system executables (/NOEXECUTE=OPTIN) so as not to break 32-bit applications that might rely on being able to execute code in pages not specifically marked as executable. On Windows Server 2003 systems, execution protection for 32-bit applications is configured by default to apply to all 32-bit programs (/NOEXECUTE=OPTOUT).



Note To obtain a complete list of which programs are protected, install the Windows Application Compatibility Toolkit (downloadable from *www.microsoft.com*) and run the Compatibility Administrator Tool. Click on System Database, Applications, and Windows Components and, on the right-hand pane, the list of protected executables will be shown.

Table 7-4 Boot.ini/NOEXECUTE Switch

Boot.ini Switch	Option in DEP Settings Dialog Box	Meaning
/NOEXECUTE=OPTIN	Turn on DEP for necessary Windows programs and services only.	Enables DEP for core Windows system images.
/NOEXECUTE=OPTOUT	Turn on DEP for all programs and services except those that I select.	Enables DEP for all executables except those specified.
/NOEXECUTE=ALWAYSON	(No GUI interface to select this option)	Enables DEP for all components with no ability to exclude certain applications.
/NOEXECUTE=ALWAYSOFF	(No GUI interface to select this option)	Disables DEP (not recommended).

Software Data Execution Prevention

Because most processors running Windows these days do not support hardware "no execute" protection, Windows XP Service Pack 2 and Windows Server 2003 Service Pack 1 and later support limited software data execution prevention (DEP). One aspect of software DEP reduces exploits of the exception handling mechanism in Windows. (See Chapter 3 for a description of structured exception handling.) If the program's image files are built with safe structured exception handling (a new feature in the Microsoft Visual C++ 2003 compiler), before an exception is dispatched, the system verifies that the exception handler is registered in the function table located within the image file. If the program's image files are not built with safe structured exception handling, software DEP ensures that before an exception is dispatched, the exception handler is located within a memory region marked as executable.

Copy-on-Write

Copy-on-write page protection is an optimization the memory manager uses to conserve physical memory. When a process maps a copy-on-write view of a section object that contains read/write pages, instead of making a process private copy at the time the view is mapped (as the Hewlett Packard OpenVMS operating system does), the memory manager defers making a copy of the pages until the page is written to. All modern UNIX systems use this technique as well. For example, as shown in Figure 7-3, two processes are sharing three pages, each marked copy-on-write, but neither of the two processes has attempted to modify any data on the pages.

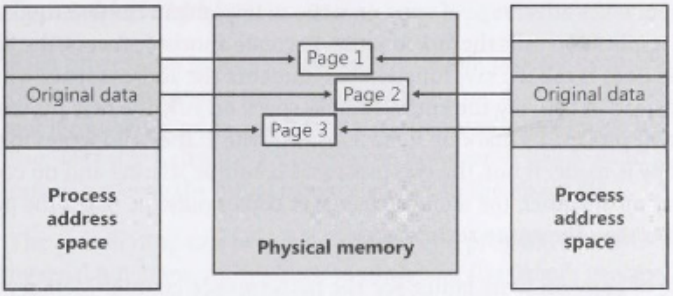


Figure 7-3 The "before" of copy-on-write

If a thread in either process writes to a page, a memory management fault is generated. The memory manager sees that the write is to a copy-on-write page, so instead of reporting the fault as an access violation, it allocates a new read/write page in physical memory, copies the contents of the original page to the new page, updates the corresponding page-mapping information (explained later in this chapter) in this process to point to the new location, and dismisses the exception, thus causing the instruction that generated the fault to be reexecuted. This time, the write operation succeeds, but as shown in Figure 7-4, the newly copied page is now private to the process that did the writing and isn't visible to the other processes still sharing the copy-on-write page. Each new process that writes to that same shared page will also get its own private copy.

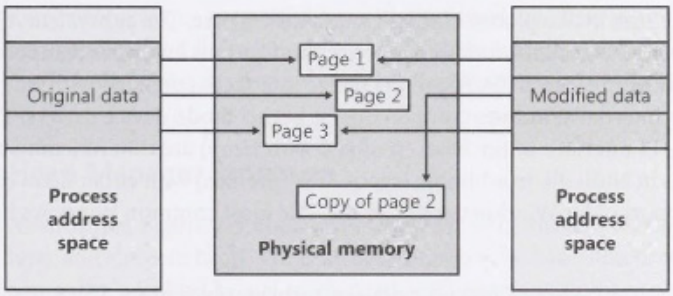


Figure 7-4 The "after" of copy-on-write

One application of copy-on-write is to implement breakpoint support in debuggers. For example, by default, code pages start out as execute-only. If a programmer sets a breakpoint while debugging a program, however, the debugger must add a breakpoint instruction to the code. It does this by first changing the protection on the page to PAGE_EXECUTE_READWRITE and then changing the instruction stream. Because the code page is part of a mapped section, the memory manager creates a private copy for the process with the breakpoint set, while other processes continue using the unmodified code page.

Copy-on-write is one example of an evaluation technique known as *lazy evaluation* that the memory manager uses as often as possible. Lazy-evaluation algorithms avoid performing an expensive operation until absolutely required—if the operation is never required, no time is wasted on it.

The POSIX subsystem takes advantage of copy-on-write to implement the *fork* function. Typically, when a UNIX application calls the *fork* function to create another process, the first thing that the new process does is call the *exec* function to reinitialize the address space with an executable program. Instead of copying the entire address space on *fork*, the new process shares the pages in the parent process by marking them copy-on-write. If the child writes to the data, a process private copy is made. If not, the two processes continue sharing and no copying takes place. One way or the other, the memory manager copies only the pages the process tries to write to rather than the entire address space.

To examine the rate of copy-on-write faults, see the performance counter Memory: Write Copies/Sec.

Heap Manager

Many applications allocate smaller blocks than the 64-KB minimum allocation granularity possible using page granularity functions such as *VirtualAlloc*. Allocating such a large area for relatively small allocations is not optimal from the memory usage and performance standpoint. To address this need, Windows provides a component called the *heap manager*, which manages allocations inside larger memory areas reserved using the page granularity memory allocation functions. The allocation granularity in the heap manager is relatively small: 8 bytes on 32-bit systems and 16 bytes on 64-bit systems. The heap manager has been designed to optimize memory usage and performance in the case of these smaller allocations.

The heap manager exists in two places: Ntdll.dll and Ntoskrnl.exe. The subsystem APIs (such as the Windows heap APIs) call the functions in Ntdll, and various executive components and device drivers call the functions in Ntoskrnl. Its native interfaces (prefixed with *Rtl*) are available only for use in internal Windows components or kernel mode device drivers. The documented Windows API interface to the heap (prefixed with *Heap*) are thin functions that call the native functions in Ntdll.dll. In addition, legacy APIs (prefixed with either *Local* or *Global*) are provided to support older Windows applications. The most common Windows heap functions are:

- *HeapCreate* or *HeapDestroy*—Creates or deletes, respectively, a heap. The initial reserved and committed size can be specified at creation.
- *HeapAlloc*—Allocates a heap block.
- *HeapFree*—Frees a block previously allocated with *HeapAlloc*.
- *HeapReAlloc*—Changes the size of an existing allocation (grows or shrinks an existing block).
- *HeapLock* or *HeapUnlock*—Controls mutual exclusion to the heap operations.
- *HeapWalk*—Enumerates the entries and regions in a heap.

Types of Heaps

Each process has at least one heap: the default process heap. The default heap is created at process startup and is never deleted during the process's lifetime. It defaults to 1 MB in size, but it can be bigger by specifying a starting size in the image file by using the /HEAP linker flag. This size is just the initial reserve, however—it will expand automatically as needed. (You can also specify the initial committed size in the image file.)

The default heap can be explicitly used by a program or implicitly used by some Windows internal functions. An application can query the default process heap by making a call to the Windows function *GetProcessHeap*. Processes can also create additional private heaps with the *HeapCreate* function. When a process no longer needs a private heap, it can recover the virtual address space by calling *HeapDestroy*. An array with all heaps is maintained in each process, and a thread can query them with the Windows function *GetProcessHeaps*.

A heap can manage allocations either in large memory regions reserved from the memory manager via *VirtualAlloc* or from memory mapped file objects mapped in the process address space. The latter approach is rarely used in practice, but it's suitable for scenarios where the content of the blocks need to be shared between two processes or between a kernel mode and a user mode component. If a heap is built on top of a memory mapped file region, certain constraints apply with respect to the component that can call heap functions. First, the internal heap structures use pointers, and therefore do not allow relocation to different addresses. Second, the synchronization across multiple processes or between a kernel component and a user process is not supported by the heap functions. Also, in the case of a shared heap between user and kernel mode, the user mode mapping should be read-only to prevent user mode code from corrupting the heap internal structures, which would result in a system crash.

Heap Manager Structure

As shown in Figure 7-5, the heap manager is structured in two layers: an optional front-end layer and the core heap. The core heap handles the basic functionality and is mostly common across the user and kernel mode heap implementations. The core functionality includes the management of blocks inside segments, the management of the segments, policies for extending the heap, committing and decommitting memory, and management of the large blocks.

For user mode heaps only, an optional front-end heap layer can exist on top of the existing core functionality. There are two types of front-end layers: look-aside lists and the Low Fragmentation Heap (or LFH, which is available in Windows XP and later), both of which are described later in this section. Only one front-end layer can be used for one heap at one time.

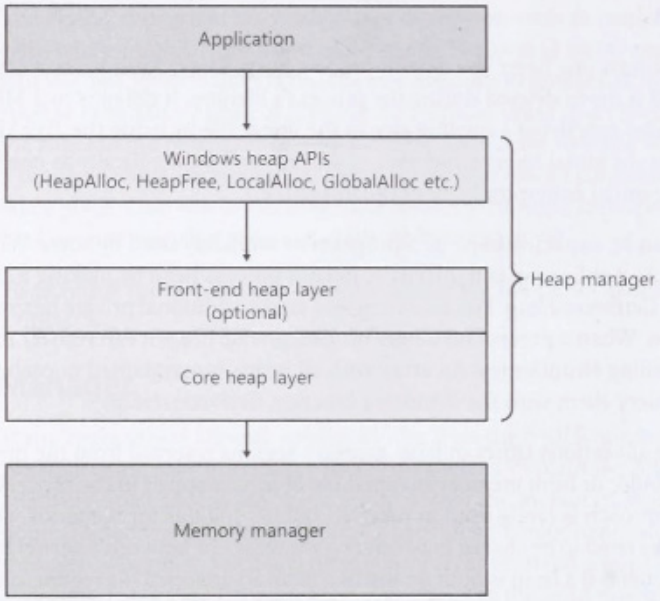


Figure 7-5 Heap manager layers

Heap Synchronization

The heap manager supports concurrent access from multiple threads by default. However, if a process is single threaded or uses an external mechanism for synchronization, it can tell the heap manager to avoid the overhead of synchronization by specifying HEAP_NO_SERIALIZE either at heap creation or on a per-allocation basis.

A process can also lock the entire heap and prevent other threads from performing heap operations for operations that would require consistent states across multiple heap calls. For instance, enumerating the heap blocks in a heap with the Windows function *HeapWalk* requires locking the heap if multiple threads can perform heap operations simultaneously.

If heap synchronization is enabled, there is one lock per heap that protects all internal heap structures. In heavily multithreaded applications (especially when running on multiprocessor systems), the heap lock might become a significant contention point. In that case, performance might be improved by enabling the front-end heap, described in an upcoming section.

Look-Aside Lists

Look-aside lists are single linked lists that allow elementary operations such as "push to the list" or "pop from the list" in a last in, first out (LIFO) order with nonblocking algorithms. A simplified version of these data structures is also available to Windows applications through the functions *InterlockedPopEntrySList* or *InterlockedPushEntrySList*. There are 128 look-aside

lists per heap, which handle allocations up to 1 KB on 32-bit platforms and up to 2 KB on 64-bit platforms.

Look-aside lists provide a significant performance improvement over normal heap allocations because multiple threads can concurrently perform allocation and deallocation operations without acquiring the heap global lock. Also, cache locality is optimized by using a LIFO ordering model and by accessing fewer internal data files in each heap operation.

The heap manager maintains the number of blocks in each look-aside list and some counters that help tune the usage of each list independently. If a thread allocates a block of a size that does not exist in the corresponding look-aside list, the heap manager will forward the call to the core heap manager to complete the operation. The heap manager will also update in this case an internal counter of misses at allocations, which will later be used in tuning decisions.

The heap manager creates look-aside lists automatically when a heap is created, as long as no debugging options are enabled and the heap is expandable. Some applications might have compatibility issues as a result of the heap manager's use of look-aside lists. In this case, these legacy applications can be made to run properly by specifying the DisableHeapLookaside flag in the image file execution options for that application. (Image file execution options can be specified using the Imagecfg.exe tool in the Windows 2000 Server Resource Kit, supplement 1.)

The Low Fragmentation Heap

Many applications running in Windows have relatively small heap memory usage (usually less than one megabyte). For this class of applications, the heap manager's best-fit policy helps keep a low memory footprint for each process. However, this strategy does not scale for large processes and multiprocessor machines. In these cases, memory available for heap usage might be reduced as a result of heap fragmentation. Performance can suffer in scenarios where only certain sizes are often used concurrently from different threads scheduled to run on different processors. This is because several processors need to modify the same memory location (for example, the head of the look-aside list for that particular size) at the same time, thus invalidating the corresponding cache line for the other processors.

The Low Fragmentation Heap (LFH) addresses these issues using the core heap manager and look-aside lists. Unlike the look-aside lists that are used as front-end heaps by default if other heap settings are allowing it, LFH is turned on only if an application calls the *HeapSetInformation* function. For large heaps, a significant percentage of allocations is generally grouped in a relatively small number of buckets of certain sizes. The allocation strategy used by LFH is to optimize the usage for these patterns by efficiently handling same-size blocks.

To address scalability, LFH expands the frequently accessed internal structures to a number of slots that is two times larger than the current number of processors on the machine. The assignment of threads to these slots is done by an LFH component called the *affinity manager*. Initially, LFH starts using the first slot for heap allocations; however, if a contention is detected at accessing some internal data, LFH switches the current thread to use a different slot. Fur-

ther contentions will spread threads on more slots. These slots are controlled for each size bucket, also to improve locality and minimize the overall memory consumption.

Heap Debugging Features

The heap manager includes several features to help detect bugs by using the following heap functions:

- **Enable tail checking** The end of each block carries a signature, which is checked when the block is released. If a buffer overrun destroyed the signature entirely or partially, the heap will report this error.
- **Enable free checking** A free block is filled with a pattern, which is checked at various points when the heap manager needs to access the block (such as at removal from the free list to allocate the block). If the process continued to write to the block after freeing it, the heap manager will detect changes in the pattern and the error will be reported.
- **Parameter checking** This function consists of extensive checking of the parameters passed to the heap functions.
- **Heap validation** The entire heap is validated at each heap call.
- **Heap tagging and stack traces support** This function supports specifying tags for allocation and/or captures user mode stack traces for the heap calls to help narrow the possible causes of a heap error.

The first three options are enabled by default if the loader detects that a process is started under the control of a debugger. (A debugger can override this behavior and turn off these features.) The heap debugging features can be specified for an executable image by setting various debugging flags in the image header using the gflags tool. (See the section "Windows Global Flags" in Chapter 3.) Or, heap debugging options can be enabled using the *!heap* command in the standard Windows debuggers. (See the debugger help for more information.)

Enabling heap debugger options affects all heaps in the process. Also, if any of the heap debug options are enabled, the front-end heap will be disabled automatically and the core heap will be used (with the required debugging options enabled). The front-end heaps are also not used for heaps that are not expandable (because of the extra overhead added to the existing heap structures) or for heaps that do not allow serialization.

Pageheap

Because the tail and free checking options described in the preceding sections might be discovering corruptions that occurred well before the problem was detected, an additional heap debugging tool, called *pageheap*, is provided that directs all or part of the heap calls to a different heap manager. Pageheap is part of the Windows Application Compatibility Toolkit, which can be downloaded from *www.microsoft.com*. The pageheap places allocations at the end of pages so that if a buffer overrun occurs, it will cause an access violation, making it easier to

detect the offending code. Optionally, pageheap allows placing the blocks at the beginning of the pages to detect buffer underrun problems. (This is a rare occurrence.) The pageheap also can protect freed pages against any access to detect references to heap blocks after they have been freed.

Note that using the pageheap can result in running out of address space because of the significant overhead added for small allocations. Also, performance can suffer as of result of the increase of references to demand zero pages, loss of locality, and additional overhead caused by frequent calls to validate heap structures. A process can reduce the impact by specifying that the pageheap be used only for blocks of certain sizes, address ranges, and/or originating DLLs.



Note For more information on pageheap, see article 286470 in the Microsoft Knowledge Base (*http://support.microsoft.com*).

Address Windowing Extensions

Although the 32-bit version of Windows can support up to 128 GB of physical memory (as shown in Table 2-4), each 32-bit user process has by default only a 2-GB virtual address space. (This can be configured up to 3 GB when using the /3GB and /USERVA Boot.ini switches, described in the upcoming section "x86 User Address Space Layouts.") To allow a 32-bit process to allocate and access more physical memory than can be represented in its limited address space, Windows provides a set of functions called *Address Windowing Extensions* (AWE). For example, on a Windows 2000 Advanced Server system with 8 GB of physical memory, a database server application could use AWE to allocate and use perhaps 6 GB of memory as a database cache.

Allocating and using memory via the AWE functions is done in three steps:

1. Allocating the physical memory to be used
2. Creating a region of virtual address space to act as a window to map views of the physical memory
3. Mapping views of the physical memory into the window

To allocate physical memory, an application calls the Windows function *AllocateUserPhysicalPages*. (This function requires the Lock Pages In Memory user right.) The application then uses the Windows *VirtualAlloc* function with the MEM_PHYSICAL flag to create a window in the private portion of the process's address space that is mapped to some or all of the physical memory previously allocated. The AWE-allocated memory can then be used with nearly all the Windows APIs. (For example, the Microsoft DirectX functions can't use AWE memory.)

If an application creates a 256-MB window in its address space and allocates 4 GB of physical memory (on a system with more than 4 GB of physical memory), the application can use the *MapUserPhysicalPages* or *MapUserPhysicalPagesScatter* Windows functions to access any

portion of the physical memory by mapping the memory into the 256-MB window. The size of the application's virtual address space window determines the amount of physical memory that the application can access with a given mapping. Figure 7-6 shows an AWE window in a server application address space mapped to a portion of physical memory previously allocated by *AllocateUserPhysicalPages*.

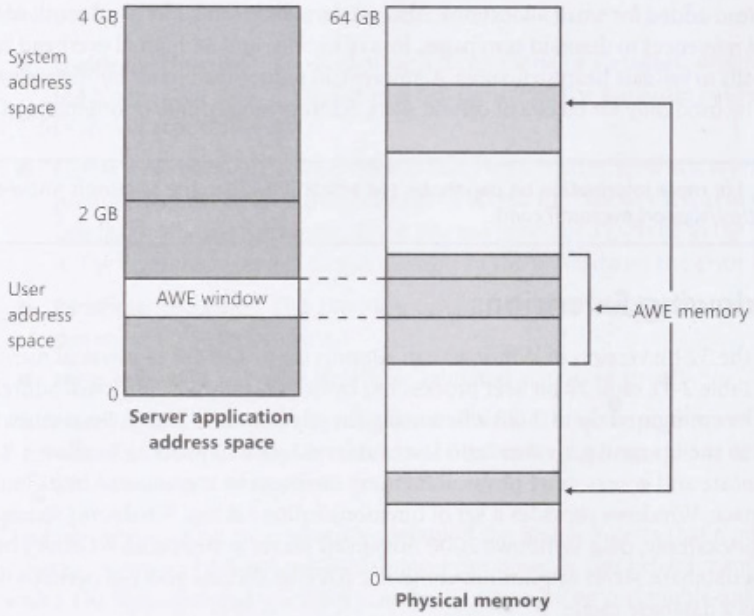


Figure 7-6 Using AWE to map physical memory

The AWE functions exist on all editions of Windows and are usable regardless of how much physical memory a system has. However, AWE is most useful on systems with more than 2 GB of physical memory because it's the only way for a 32-bit process to directly use more than 2 GB of memory. Another use is for security purposes: because AWE memory is never paged out, the data in AWE memory could never have a copy in the paging file that someone could examine by rebooting into an alternate operating system.

Finally, there are some restrictions on memory allocated and mapped by the AWE functions:

- Pages can't be shared between processes.
- The same physical page can't be mapped to more than one virtual address in the same process.
- On older versions of Windows, page protection is limited to read/write. In Windows Server 2003 Service Pack 1 and later, no access and read-only are supported.

For a description of the page table data structures used to map memory on systems with more than 4 GB of physical memory, see the section "Physical Address Extension (PAE)."

System Memory Pools

At system initialization, the memory manager creates two types of dynamically sized memory pools that the kernel-mode components use to allocate system memory:

- **Nonpaged pool** Consists of ranges of system virtual addresses that are guaranteed to reside in physical memory at all times and thus can be accessed at any time (from any IRQL level and from any process context) without incurring a page fault. One of the reasons nonpaged pool is required is because of the rule described in Chapter 2: page faults can't be satisfied at DPC/dispatch level or above.
- **Paged pool** A region of virtual memory in system space that can be paged in and out of the system. Device drivers that don't need to access the memory from DPC/dispatch level or above can use paged pool. It is accessible from any process context.

Both memory pools are located in the system part of the address space and are mapped in the virtual address space of every process. (In Table 7-8, you'll find out where in the system memory they start.) The executive provides routines to allocate and deallocate from these pools; for information on these routines, see the functions that start with *ExAllocatePool* in the Windows DDK documentation.

Uniprocessor systems have three paged pools; multiprocessor systems have five. Having more than one paged pool reduces the frequency of system code blocking on simultaneous calls to pool routines. Both nonpaged and paged pool start at an initial size based on the amount of physical memory on the system and then grow, if necessary, up to a maximum size computed at system boot time.



Note Future releases of Windows might support dynamic pool sizes, which means the maximum size will no longer apply. Therefore, applications and device drivers should not assume that the pool size maximum is a fixed value for each system.

Configuring Pool Sizes

You can override the initial size of these pools by changing the values NonPagedPoolSize and PagedPoolSize in the registry key HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management from 0 (which causes the system to compute the size) to the size desired in bytes. You can't, however, go beyond the maximum pool sizes listed in Table 7-5. A value of 0xFFFFFFFF for PagedPoolSize indicates that the largest possible size is selected, which means allowing a larger paged pool at the expense of system page table entries (PTEs).

Table 7-5 Maximum Pool Sizes

Pool Type	Maximum on 32-Bit Systems	Maximum on 64-Bit Systems
Nonpaged	256 MB (128 MB if booted /3GB)	128 GB
Paged	491.875 MB (Windows 2000 and Windows XP); 650 MB (Windows Server 2003)	128 GB

The computed sizes are stored in four kernel variables, three of which are exposed as performance counters. These variables and counters, as well as the two registry keys that can alter the sizes, are listed in Table 7-6.

Table 7-6 System Pool Size Variables and Performance Counters

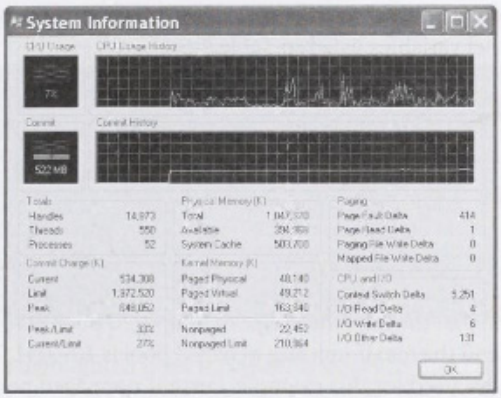
Kernel Variable	Performance Counter	Registry Key to Override	Description
MmSizeOfNonPaged-PoolInBytes	Memory: Pool Nonpaged Bytes	Not applicable	Current size of nonpaged pool
MmMaximum-NonPagedPoolInBytes	Not available	HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\NonPagedPoolSize	Maximum size of nonpaged pool
Not available	Memory: Pool Paged Bytes	Not applicable	Current virtual size of paged pool
MmPagedPoolPage (number of pages)	Memory: Pool Paged Resident Bytes	Not applicable	Current physical (resident) size of paged pool
MmSizeOfPagedPoolIn-Bytes	Not available	HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PagedPoolSize	Maximum (virtual) size of paged pool



EXPERIMENT: Determining the Maximum Pool Sizes

Because paged and nonpaged pool represent a critical system resource, it is important to know when you're nearing the maximum size computed for your system so that you can determine whether you need to override the default maximum with the appropriate registry values. The pool-size performance counters report only the current size, however, not the maximum size. So you don't know when you're reaching the limit until you've exhausted pool. (As noted earlier, future versions of Windows might support dynamically sized pools. Therefore, the need to check the pool maximums might no longer be necessary in the future.)

You can obtain the pool maximums by using either Process Explorer or live kernel debugging (explained in Chapter 1). To view pool maximums with Process Explorer, click on View, System Information. The pool maximums are displayed in the Kernel Memory section as shown below:



Note that for Process Explorer to retrieve this information, it must have access to the symbols for the kernel running on your system. (For a description of how to configure Process Explorer to use symbols, see the experiment "Viewing Process Details with Process Explorer" in Chapter 1.)

To view the same information by using the kernel debugger, you can use the *!vm* command as shown below:

```
lkd> !vm

*** Virtual Memory Usage ***
Physical Memory:    261980   ( 1047920 Kb)
Page File: \??\C:\pagefile.sys
   Current:   1024000Kb Free Space:    862236Kb
   Minimum:   1024000Kb Maximum:      1024000Kb
Available Pages:     96077   (  384308 Kb)
ResAvail Pages:     194722   (  778888 Kb)
Locked IO Pages:        89   (     356 Kb)
Free System PTEs:   175933   (  703732 Kb)
Free NP PTEs:        31832   (  127328 Kb)
Free Special NP:         0   (       0 Kb)
Modified Pages:        529   (    2116 Kb)
Modified PF Pages:     529   (    2116 Kb)
NonPagedPool Usage:   5617   (   22468 Kb)
NonPagedPool Max:    52741   (  210964 Kb)
PagedPool 0 Usage:    6582   (   26328 Kb)
PagedPool 1 Usage:    3009   (   12036 Kb)
PagedPool 2 Usage:    3044   (   12176 Kb)
PagedPool Usage:     12635   (   50540 Kb)
PagedPool Maximum:   40960   (  163840 Kb)
```

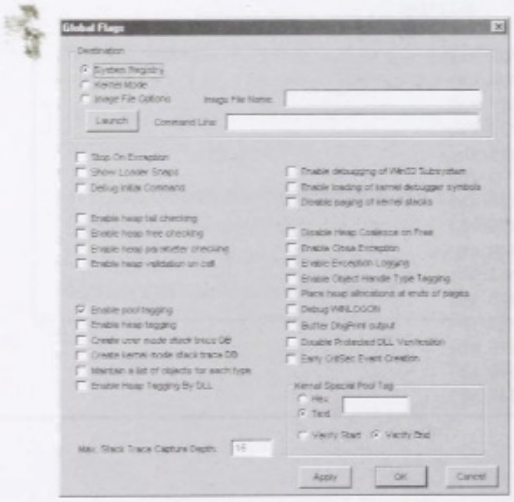
On this system, nonpaged and paged pool were far from their maximums. You can also examine the values of the kernel variables listed in Table 7-6:

```
kd> dd mmmaximumnonpagedpoolinbytes l1
8047f620  0328c000
kd> ? 328c000
Evaluate expression: 53002240 = 0328c000
kd> dd mmsizeofpagedpoolinbytes l1
80470a98  06800000
kd> ? 6800000
Evaluate expression: 109051904 = 06800000
```

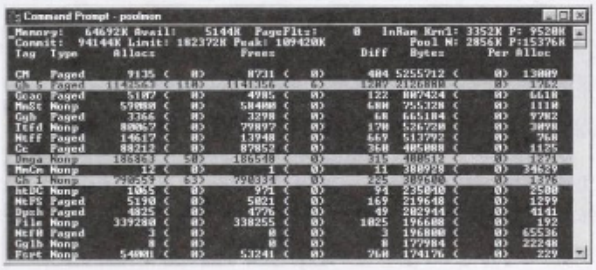
From this example, you can see that the maximum size of nonpaged pool is 53,002,240 bytes (approximately 50 MB) and the maximum size of paged pool is 109,051,904 bytes (104 MB). On the test system we used for this example, current nonpaged pool usage was 5.5 MB and paged pool usage was 34 MB, so both pools were far from full.

Monitoring Pool Usage

The Memory performance counter object has separate counters for the size of nonpaged pool and paged pool (both virtual and physical). In addition, the Poolmon utility (in the Windows Support Tools) allows you to monitor the detailed usage of nonpaged and paged pool. To do so, you must have the internal Enable Pool Tagging option enabled. Pool tagging is turned on by default as of Windows Server 2003 (in fact, it cannot be turned off) and in all checked builds of Windows. To enable pool tagging, run the Gflags utility in the Windows Support Tools, Platform SDK, or DDK, and select Enable Pool Tagging, as shown here:



Then click Apply, and reboot the system. After the system reboots, run Poolmon; you should see a display like this one:



The highlighted lines represent changes to the display. (You can disable the highlighting feature by typing *l* while running Poolmon. Type *l* again to reenable highlighting.) Type *?* while Poolmon is running to bring up its help screen. You can configure which pools you want to monitor (paged, nonpaged, or both) and the sort order. Also, the command-line options are shown, which allow you to monitor specific structures (or everything but one structure type). For example, the command *poolmon -iCM* will monitor only structures of type CM (the configuration manager, which manages the registry). The columns have the meanings shown in Table 7-7.

Table 7-7 Poolmon Columns

Column	Explanation
Tag	Four-byte tag given to the pool allocation
Type	Pool type (paged or nonpaged pool)
Allocs	Count of all allocations (The number in parentheses shows the difference in the Allocs column since the last update.)
Frees	Count of all Frees (The number in parentheses shows the difference in the Frees column since the last update.)
Diff	Count of Allocs minus Frees
Bytes	Total bytes consumed by this structure type (The number in parentheses shows the difference in the Bytes column since the last update.)
Per Alloc	Size in bytes of a single instance of this structure type

In this example, CM structures (the configuration manager, or registry) are taking up the most space in paged pool, and MmSt structures (a memory management–related structure used for mapped files) are taking up the most space in nonpaged pool.

For a description of the meaning of the pool tags used by Windows, see the file \Program Files\Debugging Tools for Windows\Triage\Pooltag.txt. (This file is installed as part of the Windows Debugging Tools described in Chapter 1.) Because third-party device driver pool tags are not listed in this file, you can use the "-c" switch on the version of Poolmon that comes with the Windows Server 2003 Device Driver Kit (DDK) to generate a local pool tag file

(Localtag.txt). This file will contain pool tags used by drivers found on your system, including third-party drivers. (Note that if a device driver binary has been deleted after it was loaded, its pool tags will not be recognized.)

Alternatively, you can search the device drivers on your system for a pool tag by using the Strings.exe tool from *www.sysinternals.com*. For example, the command

```
strings \windows\system32\drivers\*.sys > findstr /i "abcd"
```

will display drivers that contain the string "abcd". Note that device drivers do not necessarily have to be located in \Windows\System32\Drivers—they can be in any folder. To list the full path of all loaded drivers, click Start, click Run, and type **Msinfo32**. Then click on Software Environment and System Drivers. As already noted, if a device driver has been loaded and then deleted from the system, it will not be listed here.

An alternative to view pool usage by device driver is to enable the pool tracking feature of Driver Verifier, explained later in this chapter. While this makes the mapping from pool tag to device driver unnecessary, it does require a reboot (to enable Driver Verifier on the desired drivers). After rebooting with pool tracking enabled, you can either run the graphical Driver Verifier Manager (\Windows\System32\Verifier.exe) or use the Verifier /Log command to send the pool usage information to a file.

Finally, if you are looking at a crash dump, you can view pool usage with the kernel debugger *!poolused* command. The command *!poolused 2* shows nonpaged pool usage sorted by structure tag using the most amount of pool. The command *!poolused 4* lists paged pool usage, again sorted by structure tag using the most amount of pool. The following example shows the partial output from these two commands:

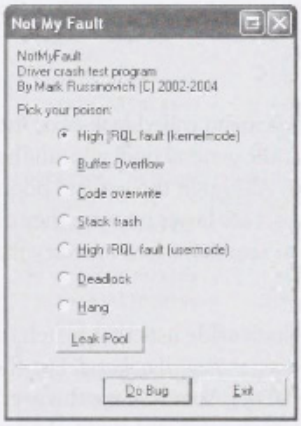
```
lkd> !poolused 2
  Sorting by  NonPaged Pool Consumed
 Pool Used:
           NonPaged            Paged
 Tag    Allocs     Used    Allocs     Used
 MmCm      386  2237440         0        0   Calls made to MmAllocateContiguousMemory , Bin
ary: nt!mm
 File    12544  2121232         0        0   File objects
 Devi      385  1487184         0        0   Device objects
 Ntfr    19163  1226888         0        0   ERESOURCE , Binary: ntfs.sys
kd> !poolused 4
   Sorting by Paged Pool Consumed

 Pool Used:
           NonPaged            Paged
 Tag    Allocs     Used    Allocs     Used
 CM         11      704      5146 22311904
 Gh 5        0        0       727  2523648
 MmSt        0        0       968  1975872
 Ntff       10     2240       790   682560
 Glal        1      128       383   600544
 Ttfd        0        0       265   545440
```



EXPERIMENT: Troubleshooting a Pool Leak

In this experiment, you will fix a real paged pool leak on your system so that you can put to use the techniques described in the previous section to track down the leak. The leak will be generated by the NotMyFault tool, which you can download from *www.sysinternals.com/windowsinternals.shtml*. (Note that this tool is not listed on the main Sysinternals tools list.) When you run NotMyFault.exe, it loads a device driver Myfault.sys and presents the following dialog box:



1. Click the Leak Pool button. This causes NotMyFault to begin sending requests to the Myfault device driver to allocate paged pool. (Do not click the Do Bug button or you will experience a system crash; this button is used in Chapter 14 to demonstrate various types of crashes.) NotMyFault will continue to do this until you click the Stop Leaking button. Note that the paged pool is not released even when you close the program; the pool is permanently leaked until you reboot the system. However, because you will leak pool only for a short period, this should not cause any problems on your system.
2. While the pool is leaking, first open Task Manager and click on the Performance tab. You should notice Paged Pool climbing. You can also check this with Process Explorer's System Information display. (Click on Show and then System Information.)
3. To determine the pool tag that is leaking, run Poolmon and press the "b" key to sort by the number of bytes. Press "p" twice so that Poolmon is showing only paged pool. You should notice the pool tag "Leak" climbing to the top of the list. (Poolmon shows changes to pool allocations by highlighting the lines that change.)

4. Now press the "Stop Leaking" button so that you don't exhaust paged pool on your system.
5. Using the technique described in the previous section, run Strings (from *www.sysinternals.com*) to look for driver binaries that contain the pool tag "Leak":

```
Strings \windows\system32\drivers\*.sys  |  findstr Leak
```

This should display a match on the file Myfault.sys, thus confirming it as the driver using the "Leak" pool tag.

Look-Aside Lists

Windows also provides a fast memory allocation mechanism called *look-aside lists*. The basic difference between pools and look-aside lists is that while general pool allocations can vary in size, a look-aside list contains only fixed-sized blocks. Although the general pools are more flexible in terms of what they can supply, look-aside lists are faster because they don't use any spinlocks and also because the system doesn't have to search for free memory that fits a varying size allocation.

Executive components and device drivers can create look-aside lists that match the size of frequently allocated data structures by using the *ExInitializeNPagedLookasideList* and *ExInitializePagedLookasideList* functions (documented in the DDK). To minimize the overhead of multiprocessor synchronization, several executive subsystems (such as the I/O manager, cache manager, and object manager) create separate look-aside lists for each processor for their frequently accessed data structures. The executive also creates a general per-processor paged and nonpaged look-aside list for small allocations (256 bytes or less).



If a look-aside list is empty (as it is when it's first created), the system must allocate from paged or nonpaged pool. But if it contains a freed structure, the allocation can be satisfied very quickly. (The list grows as structures are returned to it.) The pool allocation routines automatically tune the number of freed buffers that look-aside lists store according to how often a device driver or executive subsystem allocates from the list—the more frequent the allocations, the more buffers are stored on a list. Look-aside lists are automatically reduced in size if they aren't being allocated from. (This check happens once per second when the balance set manager system thread wakes up and calls the function *KiAdjustLookasideDepth*.)



EXPERIMENT: Viewing the System Look-Aside Lists

You can display the contents and sizes of the various system look-aside lists with the kernel debugger *!lookaside* command. The following excerpt is from the output of this command:

```
kd> !lookaside

Lookaside "nt!IopSmallIrpLookasideList" @ 804758a0 "Irps"
    Type      =     0000 NonPagedPool
    Current Depth  =        3   Max Depth  =        4
    Size           =      148   Max Alloc  =      592
    AllocateMisses =       32   FreeMisses =        9
    TotalAllocates =       52   TotalFrees =       32
    Hit Rate       =       38%  Hit Rate   =       71%
Lookaside "nt!IopLargeIrpLookasideList" @ 804756a0 "Irpl"
    Type      =     0000 NonPagedPool
    Current Depth  =        4   Max Depth  =        4
    Size           =      436   Max Alloc  =     1744
    AllocateMisses =     2623   FreeMisses =     2443
    TotalAllocates =     7039   TotalFrees =     6863
    Hit Rate       =       62%  Hit Rate   =       64%
"
Lookaside "nt!IopMdlLookasideList" @ 80475740 "Mdl "
    Type      =     0000 NonPagedPool
    Current Depth  =        3   Max Depth  =        4
    Size           =      120   Max Alloc  =      480
    AllocateMisses =     7017   FreeMisses =     1824
    TotalAllocates =    10901   TotalFrees =     5711
    Hit Rate       =       35%  Hit Rate   =       68%
§
Total NonPaged currently allocated for above lists = 4200
Total NonPaged potential for above lists           = 6144
Total Paged currently allocated for above lists    = 5136
Total Paged potential for above lists              = 12032
```

Driver Verifier

Driver Verifier is a mechanism that can be used to help find and isolate commonly found bugs in device driver or other kernel-mode system code. Microsoft uses Driver Verifier to check its own device drivers as well as all device drivers that vendors submit for Hardware Compatibility List (HCL) testing. Doing so ensures that the drivers on the HCL are compatible with Windows and free from common driver errors. (Although not described in this book, there is also a corresponding Application Verifier tool that has resulted in quality improvements for user-mode code in Windows.)

Driver Verifier consists of support in several system components: the memory manager, I/O manager, and the HAL all have driver verification options that can be enabled. This section describes the memory management–related verification options Driver Verifier provides. The options related to device drivers are described in Chapter 9.

Driver Verifier Configuration and Initialization

To configure Driver Verifier and view statistics about its operation, run the Driver Verifier Manager (\Windows\System32\Verifier.exe). (The Windows 2000 version is shown in Figure 7-7.) It displays several tabbed pages. You use the Settings tab to specify which device

drivers you want to verify (including an option to Verify All Drivers) and what types of verification you want performed.

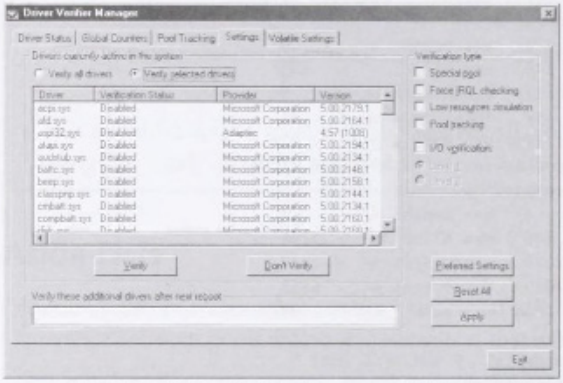


Figure 7-7 The Driver Verifier Manager in Windows 2000

In Windows XP and Windows Server 2003, Verifier.exe has a wizard-style interface, as shown in Figure 7-8.

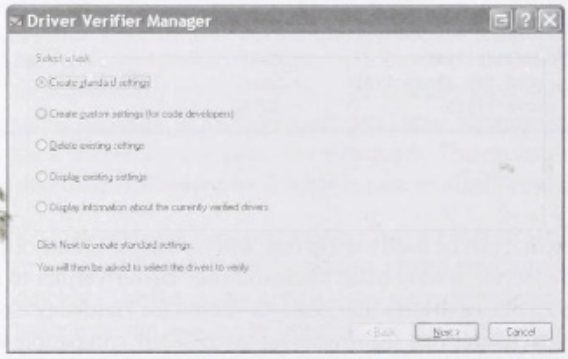


Figure 7-8 The Driver Verifier Manager in Windows XP and Windows Server 2003

You can also enable and disable Driver Verifier as well as display current settings by using its command line interface. Type "**verifier /?**" to see the switches.

The settings are stored in the registry under HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management. The value VerifyDriverLevel contains a bitmask that represents the verification types enabled. The VerifyDrivers value contains the names of the drivers to validate. (These values won't exist in the registry until you select drivers to verify in the Driver Verifier Manager.) If you choose to verify all drivers, VerifyDrivers is set to an asterisk (*) character. Depending on the settings you have made, you might need to reboot the system for the selected verification to occur.

Early in the boot process, the memory manager reads the Driver Verifier registry values to determine which drivers to verify and which Driver Verifier options you enabled. (Note that if you boot in safe mode, any Driver Verifier settings are ignored.) Subsequently, if you've selected at least one driver for verification, the kernel checks the name of every device driver it loads into memory against the list of drivers you've selected for verification. For every device driver that appears in both places, the kernel invokes the *MiApplyDriverVerifier* function, which replaces the driver's references to a number of kernel functions with references to Driver Verifier–equivalent versions of those functions. For example, *ExAllocatePool* is replaced with a call to *VerifierAllocatePool*. The windowing system driver also makes similar changes to use Driver Verifier–equivalent functions.

Now that we've reviewed how to set up Driver Verifier, let's examine the four memory-related verification options that can be applied to device drivers: Special Pool, Pool Tracking, Force IRQL Checking, and Low Resources Simulation.

Special Pool The Special Pool option causes the pool allocation routines to bracket pool allocations with an invalid page so that references before or after the allocation will result in a kernel-mode access violation, thus crashing the system with the finger pointed at the buggy driver. Special pool also causes some additional validation checks to be performed when a driver allocates or frees memory.

When special pool is enabled, the pool allocation routines allocate a region of kernel memory for Driver Verifier to use. Driver Verifier redirects memory allocation requests that drivers under verification make to the special pool area rather than to the standard kernel-mode memory pools. When a device driver allocates memory from special pool, Driver Verifier rounds up the allocation to an even-page boundary. Because Driver Verifier brackets the allocated page with invalid pages, if a device driver attempts to read or write past the end of the buffer, the driver will access an invalid page and the memory manager will raise a kernel-mode access violation.

Figure 7-9 shows an example of the special pool buffer that Driver Verifier allocates to a device driver when Driver Verifier checks for overrun errors.

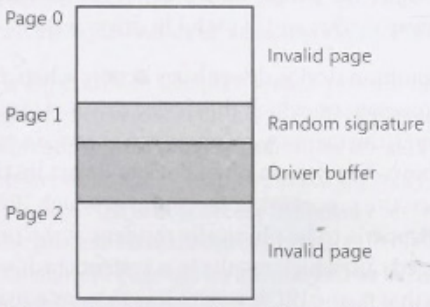


Figure 7-9 Layout of special pool allocations

By default, Driver Verifier performs *overrun* detection. It does this by placing the buffer that the device driver uses at the end of the allocated page and fills the beginning of the page with a random pattern. Although the Driver Verifier Manager doesn't let you specify underrun detection, you can set this type of detection manually by adding the DWORD registry value HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\MemoryManagement\PoolTagOverruns and setting it to 0 (or by running the Gflags utility and selecting the Verify Start option instead of the default option Verify End). When Windows enforces underrun detection, Driver Verifier allocates the driver's buffer at the beginning of the page rather than at the end.

The overrun-detection configuration includes some measure of underrun detection as well. When the driver frees its buffer to return the memory to Driver Verifier, Driver Verifier ensures that the pattern preceding the buffer hasn't changed. If the pattern is modified, the device driver has underrun the buffer and written to memory outside the buffer.

Special pool allocations also check to ensure that the processor IRQL at the time of an allocation and deallocation is legal. This check catches an error that some device drivers make: allocating pageable memory from an IRQL at DPC/dispatch level or above.

You can also configure special pool manually by adding the DWORD registry value HKLM\SYSTEMCurrentControlSet\Control\Session Manager\Memory Management\PoolTag, which represents the allocation tags the system uses for special pool. Thus, even if Driver Verifier isn't configured to verify a particular device driver, if the tag the driver associates with the memory it allocates matches what is specified in the PoolTag registry value, the pool allocation routines will allocate the memory from special pool. If you set the value of PoolTag to 0x0000002a or the wildcard (*), all memory that drivers allocate is from special pool, provided there's enough virtual and physical memory. (The drivers will revert to allocating from regular pool if there aren't enough free pages—bounding exists, but each allocation uses two pages.)

Pool Tracking If pool tracking is enabled, the memory manager checks at driver unload time whether the driver freed all the memory allocations it made. If it didn't, it crashes the system, indicating the buggy driver. Driver Verifier also shows general pool statistics on the Driver Verifier Manager's Pool Tracking tab. You can also use the *!verifier* kernel debugger command. This command shows more information than Driver Verifier and is useful to driver writers.

Force IRQL Checking One of the most common device driver bugs occurs when a driver accesses pageable data or code when the processor on which the device driver is executing is at an elevated IRQL. As explained in Chapter 3, the memory manager can't service a page fault when the IRQL is DPC/dispatch level or above. The system often doesn't detect instances of a device driver accessing pageable data when the processor is executing at a high IRQL level because the pageable data being accessed happens to be physically resident at the time. At other times, however, the data might be paged out, which results in a system crash with the stop code IRQL_NOT_LESS_OR_EQUAL (that is, the IRQL wasn't less than or equal to the level required for the operation attempted—in this case, accessing pageable memory).

Although testing device drivers for this kind of bug is usually difficult, Driver Verifier makes it easy. If you select the Force IRQL Checking option, Driver Verifier forces all kernel-mode pageable code and data out of the system working set whenever a device driver under verification raises the IRQL. The internal function that does this is *MmTrimAllSystemPagableMemory*. With this setting enabled, whenever a device driver under verification accesses pageable memory when the IRQL is elevated, the system instantly detects the violation and the resulting system crash identifies the faulty driver.

Low Resources Simulation Enabling Low Resources Simulation causes Driver Verifier to randomly fail memory allocations that verified device drivers perform. In the past, developers wrote many device drivers under the assumption that kernel memory would always be available and that if memory ran out, the device driver didn't have to worry about it because the system would crash anyway. However, because low-memory conditions can occur temporarily, it's important that device drivers properly handle allocation failures that indicate kernel memory is exhausted.

Beginning 7 minutes after the system boots—which is enough time to get past the critical initialization period in which a low-memory condition might prevent a device driver from loading—Driver Verifier starts randomly failing allocation calls for device drivers it is verifying. If a driver doesn't correctly handle allocation failures, this will likely show up as a system crash.

Driver Verifier is a valuable addition to the arsenal of verification and debugging tools available to device driver writers. Many device drivers that first ran with Driver Verifier had bugs that Driver Verifier was able to expose. Thus, Driver Verifier has resulted in an overall improvement in the quality of all kernel-mode code running in Windows.

Virtual Address Space Layouts

This section describes the components in the user and system address space, followed by the specific layouts on 32-bit and 64-bit systems. This information helps you to understand the limits on process and system virtual memory on both platforms.

Three main types of data are mapped into the virtual address space in Windows: per-process private code and data, sessionwide code and data, and systemwide code and data.

As explained in Chapter 1, each process has a private address space that cannot be accessed by other processes (unless they have permission to open the process for read or write access). Threads within the process can never access virtual addresses outside this private address space, unless they map to shared memory sections and/or use the cross-process memory functions that allow access to another process's address space. The information that describes the process virtual address space, called *page tables*, is described in the section on address translation. Page tables are marked as kernel-mode-only accessible pages so that user threads in a process cannot modify their own address space layout.

On systems with multiple sessions (Windows 2000 Server with Terminal Services installed, Windows XP, and Windows Server 2003), *session space* contains information global to each session. (For a description of sessions, see Chapter 2.) A session consists of the processes and other system objects (such as the window station, desktops, and windows) that represent a single user's workstation logon session. Each session has a session-specific paged pool area used by the kernel-mode portion of the Windows subsystem (Win32k.sys) to allocate session-private GUI data structures. In addition, each session has its own copy of the Windows subsystem process (Csrss.exe) and logon process (Winlogon.exe). The session manager process (Smss.exe) is responsible for creating new sessions, which includes loading a session-private copy of Win32k.sys, creating the session-private object manager namespace, and creating the session-specific instances of the Csrss and Winlogon processes.

To virtualize sessions, all sessionwide data structures are mapped into a region of system space called *session space*. When a process is created, this range of addresses is mapped to the pages appropriate to the session that the process belongs to. The size of the space for mapped views in session space can be configured by modifying registry values under HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management. (For 32-bit systems, these registry values are ignored if the system is booted /3GB.)

Finally, *system space* contains global operating system code and data structures visible to each process. System space consists of the following components:

- **System code** Contains the operating system image, HAL, and device drivers used to boot the system.
- **System mapped views** Used to map Win32k.sys, the loadable kernel-mode part of the Windows subsystem, as well as kernel-mode graphics drivers it uses. (See Chapter 2 for more information on Win32k.sys.)
- **Hyperspace** A special region used to map the process working set list and to temporarily map other physical pages for operations such as zeroing a page on the free list (when the zero list is empty and a zero page is needed), invalidating page table entries in other page tables (such as when a page is removed from the standby list), and process creation to set up a new process's address space.
- **System working set list** The working set list data structures that describe the system working set.*
- **System cache** Virtual address space used to map files open in the system cache. (See Chapter 11 for detailed information about the cache manager.)
- **Paged pool** Pageable system memory heap.

- **System page table entries (PTEs)** Pool of system PTEs used to map system pages such as I/O space, kernel stacks, and memory descriptor lists. You can see how many system PTEs are available by examining the value of the Memory: Free System Page Table Entries counter in the Performance tool.
- **Nonpaged pool** Nonpageable system memory heap, usually existing in two parts—one in the lower end of system space and one in the upper end.
- **Crash dump information** Reserved to record information about the state of a system crash.
- **HAL usage** System memory reserved for HAL-specific structures.



Note *Internally, the system working set is called the *system cache working set*. This term is misleading, however, because it includes not only the system cache but also the paged pool, pageable system code and data, and pageable driver code and data.

Now that we've described the basic components of the virtual address space in Windows, let's examine the specific layout on the x86, IA-64, and x64 platforms.

x86 User Address Space Layouts

By default, each user process on the 32-bit version of Windows can have up to a 2-GB private address space; the operating system takes the remaining 2 GB. Windows 2000 Advanced Server, Windows 2000 Datacenter Server, Windows XP Service Pack 2 and later, and Windows Server 2003 (all versions) support a boot-time option (the /3GB switch in Boot.ini) that allows 3-GB user address spaces. Windows XP and Windows Server 2003 support an additional switch (/USERVA), which allows specifying a user address space size between 2 GB and 3 GB. (The value is in number of megabytes.) These two possible address space layouts are shown in Figure 7-10.

The ability for a 32-bit process to grow beyond 2 GB was added as a short-term solution to accommodate the need for applications such as database servers to keep more data in memory than could be done with a 2-GB address space. The AWE functions described earlier in this chapter provide a much better solution to the need for accessing more data than can fit in the limited 2-GB (or 3-GB) process address space.

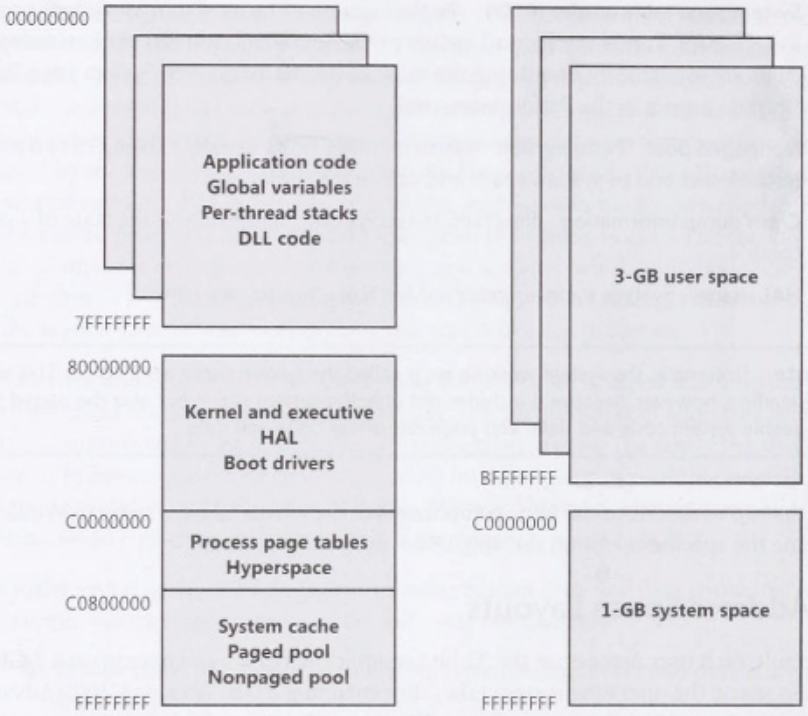


Figure 7-10 x86 virtual address space layouts

For a process to grow beyond 2 GB of address space, the image file must have the IMAGE_FILE_LARGE_ADDRESS_AWARE flag set in the image header. Otherwise, Windows reserves the additional address space for that process so that the application won't see virtual addresses greater than 0x7FFFFFFF. (This is to avoid breaking applications that might break with such addresses.) You set this flag by specifying the linker flag /LARGEADDRESSAWARE when building the executable. This flag has no effect when running the application on a system with a 2-GB user address space. (If you boot with /3GB in versions of Windows Server that do not support the /3GB switch, system space is reduced to 1 GB, but user processes are still limited to 2 GB, even if the large-address-space aware flag is set on an image that is run.)

Several system images are marked as large address space aware so that they can take advantage of systems running with large process address spaces. These include:

- Lsass.exe—the Local Security Authority Subsystem
- Inetinfo.exe—Internet Information Server
- Chkdsk.exe—the Check Disk utility
- Dllhst3g.exe—a special version of Dllhost.exe (for COM+ applications)

Finally, because by default, memory allocations using VirtualAlloc start with low virtual addresses and grow higher, unless a process allocates a lot of virtual memory or it has a very fragmented virtual address space, it will never get back very high virtual addresses. Therefore, for testing purposes, you can force memory allocations to start from high addresses by adding a DWORD registry value, HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management\AllocationPreference, and setting it to 0x100000.

x86 System Address Space Layout

This section describes the detailed layout and contents of system space in 32-bit Windows. Figure 7-11 shows the overall structure on x86 systems with a 2-GB system space.

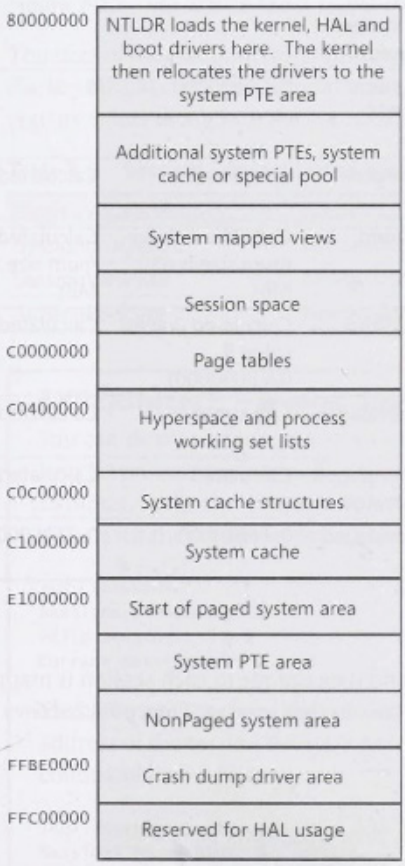


Figure 7-11 x86 system space layout (not proportional)

Table 7-8 lists the kernel variables that contain start and end addresses of various system space regions. Some of these regions are fixed; some are computed at system boot time on the basis of memory size and whether the system is running a client or server edition of Windows.

Table 7-8 System Variables That Describe System Space Regions

System Variables	Description	x86 2-GB System Space (non-PAE)	x86 1-GB System Space (non-PAE)
MmSystemRangeStart	Start address of system space	0x80000000	0xC0000000
MmSystemCacheWorkingSetList	System working set list	0xC0C00000	Calculated
MmSystemCacheStart	Start of system cache	Calculated	Calculated
MmSystemCacheEnd	End of system cache	Calculated	Calculated
MiSystemCacheStartExtra	Start of system cache or system PTE extension	Calculated	0
MiSystemCacheEndExtra	End of system cache or PTE extension	0xC0000000	0
MmPagedPoolStart	Start of paged pool	Calculated	Calculated
MmPagedPoolEnd	End of paged pool	Calculated (maximum size is 650 MB)	Calculated (minimum size is 160 MB)
MmNonPagedSystemStart	Start of system PTEs	Calculated (lowest value is 0xEB000000)	Calculated
MmNonPagedPoolStart	Start of nonpaged pool	Calculated	Calculated
MmNonPagedPoolExpansionStart	Start of nonpaged pool expansion	Calculated	Calculated
MmNonPagedPoolEnd	End of nonpaged pool	0xFFBE0000	0xFFBE0000

x86 Session Space

For systems with multiple sessions, the code and data unique to each session is mapped into system address space but shared by the processes in that session. Figure 7-12 shows the general layout of session space.

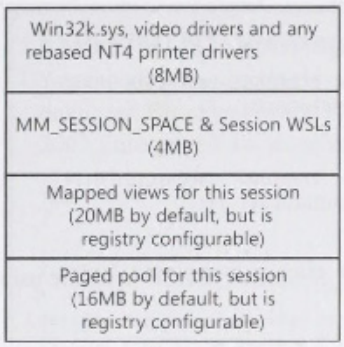


Figure 7-12 x86 session space layout (not proportional)

The sizes of the components of session space can be configured by adding registry values under the key HKLM\System\CurrentControlSet\Session Manager\Memory Management. These registry values along with the kernel variables that contain the actual sizes are listed in Table 7-9.

Table 7-9 Session Space Configuration Values

Registry Value Under	Kernel Variable
SessionImageSize	MmSessionImageSize
SessionViewSize	MmSessionViewSize
SessionPoolSize	MmSessionPoolSize



EXPERIMENT: Viewing Sessions

You can display which processes are members of which sessions by examining the session ID process performance counter. This can be viewed with Task Manager, Process Explorer, or the Performance Tool. Using the kernel debugger, you can list the active sessions with the *!session* command as follows:

```
lkd> !session
Sessions on machine: 3
Valid Sessions: 0 1 2
Current Session 0
```

Then you can set the active session using the *!session* –s command and display the address of the session data structures and the processes in that session with the *!sprocess* command:

```
lkd> !session -s 2
Sessions on machine: 3
Implicit process is now 8144f3a0
Using session 2
lkd> !sprocess
Dumping Session 2
```

```
_MM_SESSION_SPACE f9d5a000
_MMSESSION        f9d5ad80
PROCESS 8144f3a0  SessionId: 2  Cid: 0d18    Peb: 7ffdf000  ParentCid: 0134
    DirBase: 07ef7000  ObjectTable: e1855bb0  HandleCount:  71.
    Image: csrss.exe

PROCESS 813f6500  SessionId: 2  Cid: 0d04    Peb: 7ffdf000  ParentCid: 0134
    DirBase: 0f3fd000  ObjectTable: e18f3008  HandleCount: 131.
    Image: winlogon.exe
```

To view the details of the session, dump the MM_SESSION_SPACE structure using the *dt* command as follows:

```
lkd> dt nt!_mm_session_space f9d4a000
nt!_MM_SESSION_SPACE
   +0x000 GlobalVirtualAddress : 0xf9d4a000
   +0x004 ReferenceCount    : 0x19
   +0x008 u                 : __unnamed
   +0x00c SessionId         : 0
   +0x010 ProcessList       : _LIST_ENTRY [ 0x817d92cc - 0x812de88c ]
   +0x018 LastProcessSwappedOutTime : _LARGE_INTEGER 0x0
   +0x020 SessionPageDirectoryIndex : 0x6a29
   +0x024 NonPagablePages   : 0xf
   +0x028 CommittedPages    : 0x3f
   +0x02c PagedPoolStart    : 0xbc000000
   +0x030 PagedPoolEnd      : 0xbc3fffff
   +0x034 PagedPoolBasePde  : 0xc0300bc0
   +0x038 Color             : 0xd6
   +0x03c ProcessOutSwapCount : 5
   +0x040 SessionPoolAllocationFailures : [4] 0
   +0x050 ImageList         : _LIST_ENTRY [ 0x81801fa8 - 0x816933a0 ]
   +0x058 LocaleId          : 0x409
   +0x05c AttachCount       : 0
   +0x060 AttachEvent       : _KEVENT
   +0x070 LastProcess       : (null)
   +0x074 ProcessReferenceToSession : 25
   +0x078 WsListEntry       : _LIST_ENTRY [ 0x8056d058 - 0x8056d058 ]
   +0x080 Lookaside         : [26] _GENERAL_LOOKASIDE
   +0xd80 Session           : _MMSESSION
   +0xdc0 PagedPoolMutex    : _KGUARDED_MUTEX
   +0xde0 PagedPoolInfo     : _MM_PAGED_POOL_INFO
   +0xe00 Vm                : _MMSUPPORT
   +0xe60 Wsle              : 0xbf400038
   +0xe64 Win32KDriverUnload : 0xbf9169ab     +ffffffffbf9169ab
   +0xe68 PagedPool         : _POOL_DESCRIPTOR
   +0x1e98 PageTables       : 0x816e7008
   +0x1e9c ImageLoadingCount : 0
```



EXPERIMENT: Viewing Session Space Utilization

You can view session space memory utilization with the *!vm 4* command in the kernel debugger. For example, the following output was taken from a 32-bit Windows Server 2003 Enterprise Edition system with two active sessions:

```
lkd> !vm 4
.
.
Terminal Server Memory Usage By Session:

Session Paged Pool Maximum is 4096K
Session View Space Maximum is 49152K

Session ID 0 @ f9d4a000:
Paged Pool Usage:           0K
Commit Usage:             252K

Session ID 1 @ f9d4c000:
Paged Pool Usage:           0K
Commit Usage:             172K
```

Executing the same command on an x64 Windows Server 2003 Enterprise Edition system with two active sessions shows the much larger session space regions:

```
Terminal Server Memory Usage By Session:
Session Paged Pool Maximum is 65536K
Session View Space Maximum is 106496K

Session ID 0 @ fffffadfe4bb9000:
Paged Pool Usage:           0K
Commit Usage:            4608K
Session ID 1 @ fffffadfe4c5f000:
Paged Pool Usage:           0K
Commit Usage:             584K
```

System Page Table Entries

System page table entries are used to dynamically map system pages such as I/O space, kernel stacks, and memory descriptor lists. System PTEs aren't an infinite resource. For example, Windows 2000 can describe only 660 MB of variable system virtual address space (440 MB of which could be contiguous). With the 32-bit edition of Windows XP and Windows Server 2003, the number of available system page table entries (PTEs) has increased such that the system can describe 1.3 GB of system virtual address space, 960 MB of which can be contiguous. On 64-bit Windows, system PTEs can describe up to 128 GB of contiguous virtual address space.

You can see how many system PTEs are available by examining the value of the Memory: Free System Page Table Entries counter in the Performance tool. By default, Windows calculates at boot time how many system PTEs to allocate based on the memory size. You can override this calculation by setting the registry value HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\SystemPages to the number of PTEs you want. (This might be needed to support, for example, a device that requires a large number of system PTEs, such as a video card with 512 MB of video memory that has to be mapped all at one time.) If the value is 0xFFFFFFFF, the maximum number of system PTEs will be reserved.

64-Bit Address Space Layouts

The theoretical 64-bit virtual address space is 16 exabytes (18,446,744,073,709,551,616 bytes, or approximately 17.2 billion gigabytes). Unlike on the x86 where the default address space is divided in two parts (half for a process and half for the system), the 64-bit address is divided into a number of different sized regions whose components match conceptually the portions of user, system, and session space. The various sizes of these regions, listed in Table 7-10, represent current implementation limits that could easily be extended in future releases. (The sizes on the 32-bit x86 platform are included for comparison purposes.) Clearly, 64-bits provides a tremendous leap in terms of address space sizes.

Table 7-10 The 64-Bit Address Space Sizes

Region	IA-64	x64	x86
Process Address Space	7152 GB	8192 GB	2 to 3 GB
System PTE Space	128 GB	128 GB	1.2 GB
System Cache	128 GB	128 GB	960 MB
Paged Pool	128 GB	128 GB	470 to 650 MB
Nonpaged Pool	128GB	128 GB	256 MB

The detailed IA-64 and x64 address space layouts vary slightly. The IA-64 address space layout is shown in Figure 7-13, and the x64 address space layout is shown in Figure 7-14.

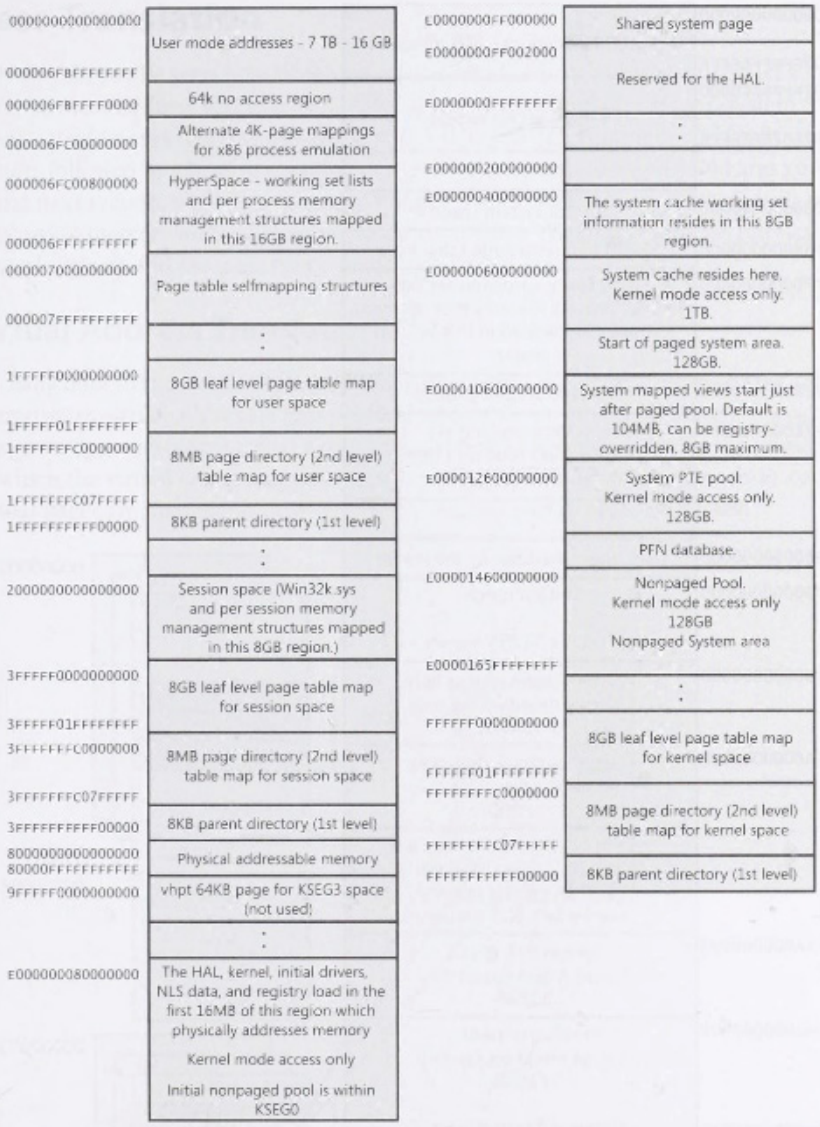


Figure 7-13 IA-64 address space layout

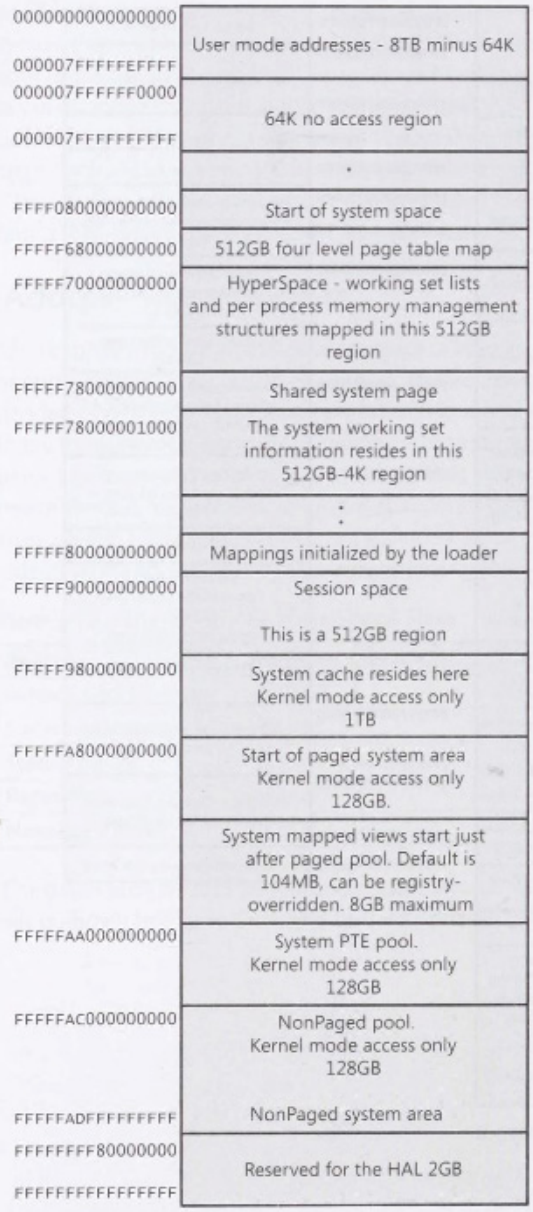


Figure 7-14 x64 address space layout

Address Translation

Now that you've seen how Windows structures the virtual address space, let's look at how it maps these address spaces to real physical pages. User applications and system code reference virtual addresses. This section starts with a detailed description of 32-bit x86 address translation, followed by a brief description of the differences on the 64-bit IA-64 and x64 platform. In the next section, we'll describe what happens when such a translation doesn't resolve to a physical memory address (paging) and explain how Windows manages physical memory via working sets and the page frame database.

x86 Virtual Address Translation

Using data structures the memory manager creates and maintains called *page tables*, the CPU translates virtual addresses into physical addresses. Each virtual address is associated with a system-space structure called a *page table entry* (PTE), which contains the physical address to which the virtual one is mapped. For example, Figure 7-15 shows how three consecutive virtual pages are mapped to three physically discontiguous pages on an x86.

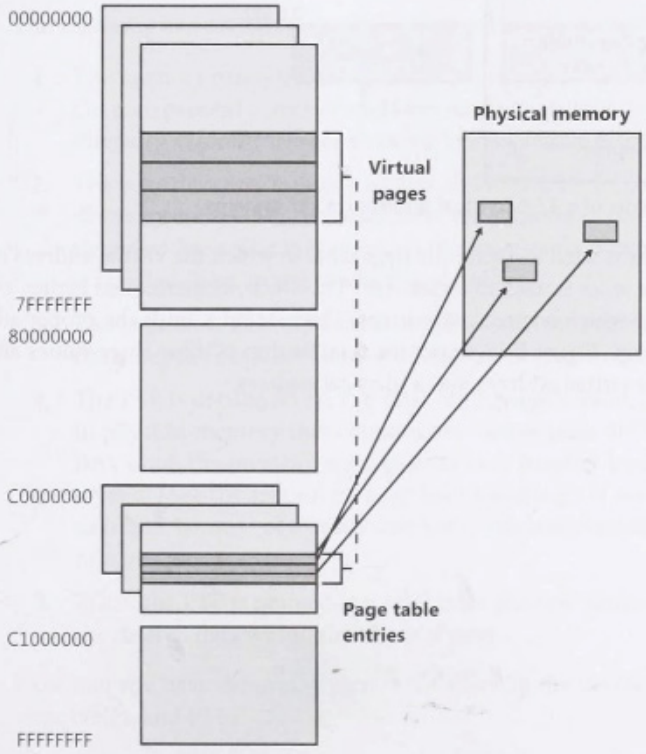


Figure 7-15 Mapping virtual addresses to physical memory (x86)

The dashed line connecting the virtual pages to the PTEs in Figure 7-15 represents the indirect relationship between virtual pages and physical memory.



Note Kernel-mode code (such as device drivers) can reference physical memory addresses by mapping them to virtual addresses. For more information, see the memory descriptor list (MDL) support routines described in the DDK documentation.

By default, Windows on an x86 system uses a two-level page table structure to translate virtual to physical addresses. (x86 systems running the PAE kernel use a three-level page table—this section assumes non-PAE systems.) A 32-bit virtual address is interpreted as three separate components—the page directory index, the page table index, and the byte index—that are used as indexes into the structures that describe page mappings, as illustrated in Figure 7-16. The page size and the PTE width dictate the width of the page directory and page table index fields. For example, on x86 systems, the byte index is 12 bits because pages are 4096 bytes (212 = 4096).

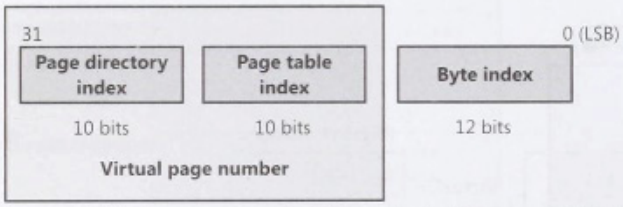


Figure 7-16 Components of a 32-bit virtual address on x86 systems

The *page directory index* is used to locate the page table in which the virtual address's PTE is located. The *page table index* is used to locate the PTE, which, as mentioned earlier, contains the physical address to which a virtual page maps. The *byte index* finds the proper address within that physical page. Figure 7-17 shows the relationship of these three values and how they are used to map a virtual address into a physical address.

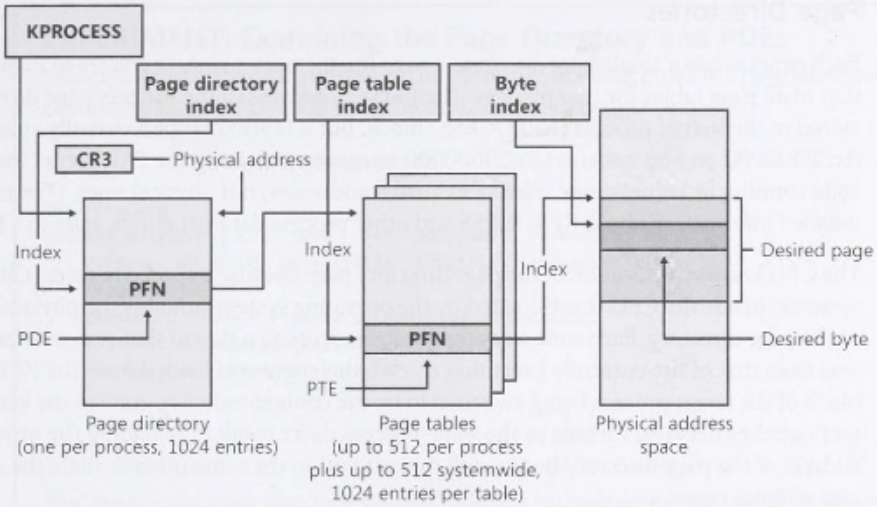


Figure 7-17 Translating a valid virtual address (x86-specific)

The following basic steps are involved in translating a virtual address:

1. The memory management hardware locates the page directory for the current process. On each process context switch, the hardware is told the address of a new process page directory, typically by the operating system setting a special CPU register.
2. The page directory index is used as an index into the page directory to locate the page directory entry (PDE) that describes the location of the page table needed to map the virtual address. The PDE contains the page frame number (PFN) of the page table (if it is resident—page tables can be paged out).
3. The page table index is used as an index into the page table to locate the PTE that describes the physical location of the virtual page in question.
4. The PTE is used to locate the page. If the page is valid, it contains the PFN of the page in physical memory that contains the virtual page. If the PTE indicates that the page isn't valid, the memory management fault handler locates the page and tries to make it valid. (See the section on page fault handling.) If the page can't be made valid (for example, because of a protection fault), the fault handler generates an access violation or a bug check.
5. When the PTE is pointed to a valid page, the byte index is used to locate the address of the desired data within the physical page.

Now that you have the overall picture, let's look at the detailed structure of page directories, page tables, and PTEs.

Page Directories

Each process has a single *page directory*, a page the memory manager creates to map the location of all page tables for that process. The physical address of the process page directory is stored in the kernel process (KPROCESS) block, but it is also mapped virtually at address 0xC0300000 on x86 systems (0xC0600000 on systems running the PAE kernel image). All code running in kernel mode references virtual addresses, not physical ones. (For more detailed information about KPROCESS and other process data structures, refer to Chapter 6.)

The CPU knows the location of the page directory page because a special register (CR3 on x86 systems) inside the CPU that is loaded by the operating system contains the physical address of the page directory. Each time a context switch occurs to a thread that is in a different process than that of the currently executing thread, this register is loaded from the KPROCESS block of the target process being switched to by the context-switch routine in the kernel. Context switches between threads in the same process don't result in reloading the physical address of the page directory because all threads within the same process share the same process address space.

The page directory is composed of *page directory entries* (PDEs), each of which is 4 bytes long (8 bytes on systems running the PAE kernel image) and describes the state and location of all the possible page tables for that process. (As described later in the chapter, page tables are created on demand, so the page directory for most processes points only to a small set of page tables.) The format of a PDE isn't repeated here because it's mostly the same as a hardware PTE.

On x86 systems running in non-PAE mode, 1024 page tables are required to describe the full 4-GB virtual address space. The process page directory that maps these page tables contains 1024 PDEs. Therefore, the page directory index needs to be 10 bits wide ($2^{10} = 1024$). On x86 systems running in PAE mode, there are 512 entries in a page table (because the page directory index is 9 bits wide). Because there are 4 page directories, the result is a maximum of 2048 page tables.



EXPERIMENT: Examining the Page Directory and PDEs

You can see the physical address of the currently running process's page directory by examining the DirBase field in the *!process* kernel debugger output:

```
                    Physical address of PDE
                    |
kd> !process        |
PROCESS 80145880 |  Cid: 0000   Peb: 00000000   ParentCid: 0000
    DirBase: 00030000  ObjectTable: 80695668  TableSize: 126.
    Image: Idle
    VadRoot 0 Clone 0 Private 0. Modified 0. Locked 0.
    80145A3C MutantState Locked OwningThread 0
    Process Lock Owned by Thread 0
        .
        .
        .
```

You can see the page directory's virtual address by examining the kernel debugger output for the PTE of a particular virtual address, as shown here:

```
                          Virtual address of PDE
                                  Contents of PDE
                          |       |
kd> !pte 50001            |       |
00050001  -  PDE at C0300000      |      PTE at C0000140
          contains 00700067 ------+    contains 00E63047
          pfn 00700  --DA--UW        pfn 00E63  --D---UWV
            |          |
            PFN of page  State and
            table page   protection of
                         page table
                         page
```

The PTE part of the kernel debugger output is defined in the section "Page Table Entries."

Because Windows provides a private address space for each process, each process has its own set of process page tables to map that process's private address space. However, the page tables that describe system space are shared among all processes (and session space is shared among processes in a session). To avoid having multiple page tables describing the same virtual memory, when a process is created, the page directory entries that describe system space are initialized to point to the existing system page tables. If the process is part of a session, session space page tables are also shared by pointing the session space page directory entries to the existing session page tables.

But as shown in Figure 7-18, not all processes have the same view of system space. For example, if paged pool expansion requires the allocation of a new system page table, the memory manager doesn't go back and update all the process page directories to point to the new system page table. Instead, it updates the process page directories when the processes reference the new virtual address.

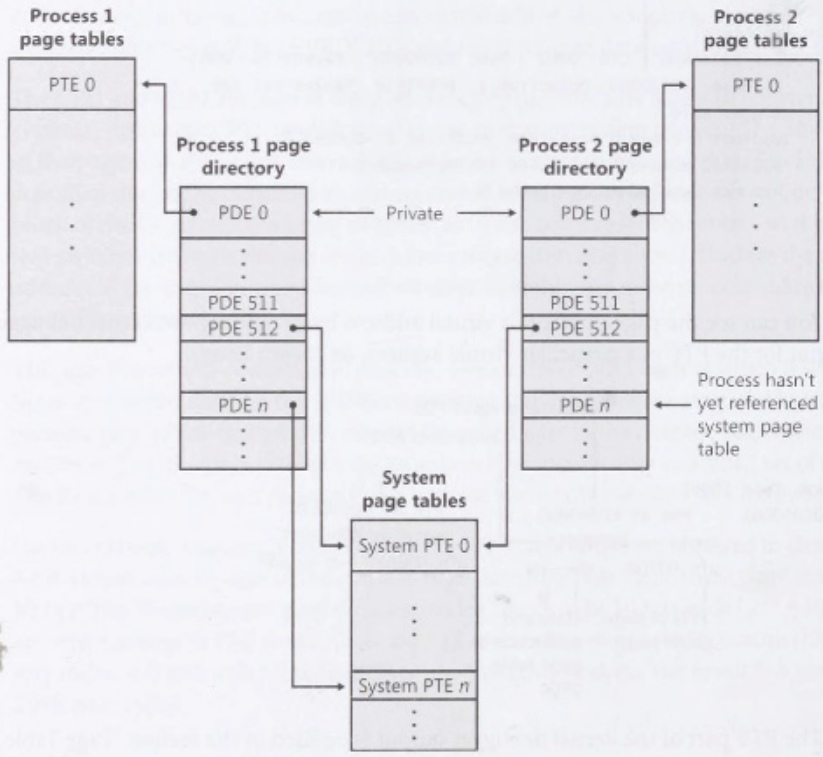


Figure 7-18 System and process-private page tables

Thus, a process can take a page fault when referencing paged pool that is in fact physically resident because its process page directory doesn't yet point to the new system page table that describes the new area of pool. Page faults don't occur when accessing nonpaged pool, even though it too can be expanded, because Windows builds enough system page tables to describe the maximum size during system initialization.

Page Tables and Page Table Entries

The process page directory entries point to individual page tables. Page tables are composed of an array of PTEs. The virtual address's page table index field (as shown in Figure 7-17) indicates which PTE within the page table maps the data page in question. On x86 systems, the page table index is 10 bits wide (9 on PAE), allowing you to reference up to 1024 4-byte PTEs

(512 8-byte PTEs on PAE systems). However, because 32-bit Windows provides a 4-GB private virtual address space, more than one page table is needed to map the entire address space. To calculate the number of page tables required to map the entire 4-GB process virtual address space, divide 4 GB by the virtual memory mapped by a single page table. Recall that each page table on an x86 system maps 4 MB (2 MB on PAE) of data pages. Thus, 1024 page tables (4 GB/4 MB)—or 2048 page tables (4 GB/2 MB) for PAE—are required to map the full 4-GB address space.

You can use the *!pte* command in the kernel debugger to examine PTEs. (See the experiment "Translating Addresses.") We'll discuss valid PTEs here and invalid PTEs in a later section. Valid PTEs have two main fields: the page frame number (PFN) of the physical page containing the data or of the physical address of a page in memory, and some flags that describe the state and protection of the page, as shown in Figure 7-19.

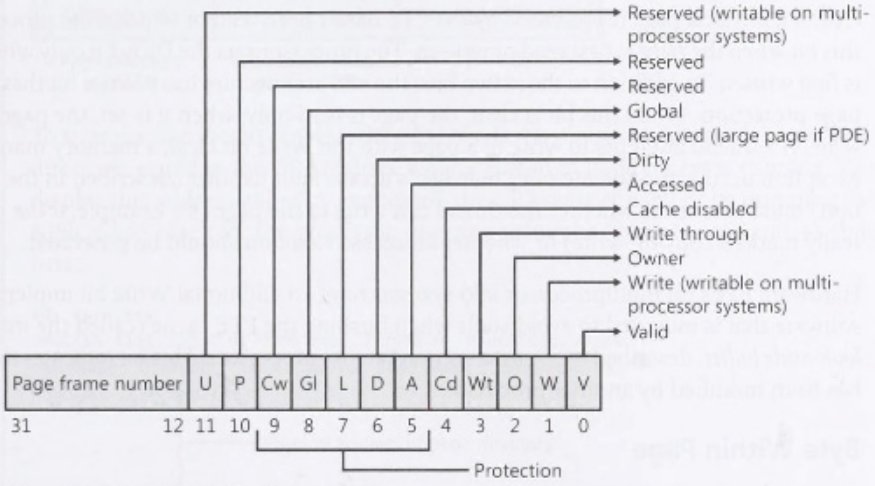


Figure 7-19 Valid x86 hardware PTEs

As you'll see later, the bits labeled Reserved in Figure 7-19 are used only when the PTE isn't valid. (The bits are interpreted by software.) Table 7-11 briefly describes the hardware-defined bits in a valid PTE.

Table 7-11 PTE Status and Protection Bits

Name of Bit	Meaning
Accessed	Page has been read.
Cache disabled	Disables caching for that page.
Dirty	Page has been written to.
Global	Translation applies to all processes. (For example, a translation buffer flush won't affect this PTE.)

Table 7-11 PTE Status and Protection Bits

Name of Bit	Meaning
Large page	Indicates that the PDE maps a 4-MB page (or 2 MB on PAE systems). See the section "Large and Small Pages" earlier in the chapter.
Owner	Indicates whether user-mode code can access the page or whether the page is limited to kernel-mode access.
Valid	Indicates whether the translation maps to a page in physical memory.
Write through	Disables caching of writes to this page so that changes are immediately flushed to disk.
Write	On uniprocessor systems, indicates whether the page is read/write or read-only; on multiprocessor systems, indicates whether the page is writable. (The Write bit is stored in a reserved bit in the PTE.)

On x86 systems, a hardware PTE contains a Dirty bit and an Accessed bit. The Accessed bit is clear if a physical page represented by the PTE hasn't been read or written; the processor sets this bit when the page is first read or written. The processor sets the Dirty bit only when a page is first written. In addition to those two bits, the x86 architecture has a Write bit that provides page protection. When this bit is clear, the page is read-only; when it is set, the page is read/write. If a thread attempts to write to a page with the Write bit clear, a memory management exception occurs and the memory manager's access fault handler (described in the next section) must determine whether the thread can write to the page (for example, if the page was really marked copy-on-write) or whether an access violation should be generated.

Hardware PTEs on multiprocessor x86 systems have an additional Write bit implemented in software that is intended to avoid stalls when flushing the PTE cache (called the *translation look-aside buffer*, described in the next section) across processors. This bit indicates that a page has been modified by another processor.

Byte Within Page

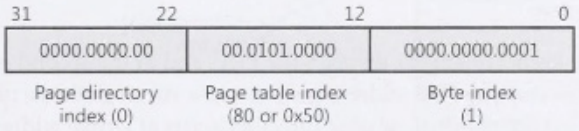
Once the memory manager has found the physical page in question, it must find the requested data within that page. This is where the byte index field comes in. The byte index field tells the CPU which byte of data in the page you want to reference. On x86 systems, the byte index is 12 bits wide, allowing you to reference up to 4096 bytes of data (the size of a page). So, adding the byte offset to the physical page number retrieved from the PTE completes the translation of a virtual address to a physical address.



EXPERIMENT: Translating Addresses

To clarify how address translation works, let's go through a real example of translating a virtual address on an x86 non-PAE system, using the available tools in the kernel debugger to examine page directories, page tables, and PTEs. In this example, we'll use a process that has virtual address 0x50001 currently mapped to a valid physical address. In later examples, you'll see how to follow address translation for invalid addresses with the kernel debugger.

First let's convert 0x50001 to binary and break it into the three fields that are used to translate an address. In binary, 0x50001 is 101.0000.0000.0000.0001. Breaking it into the component fields yields the following:



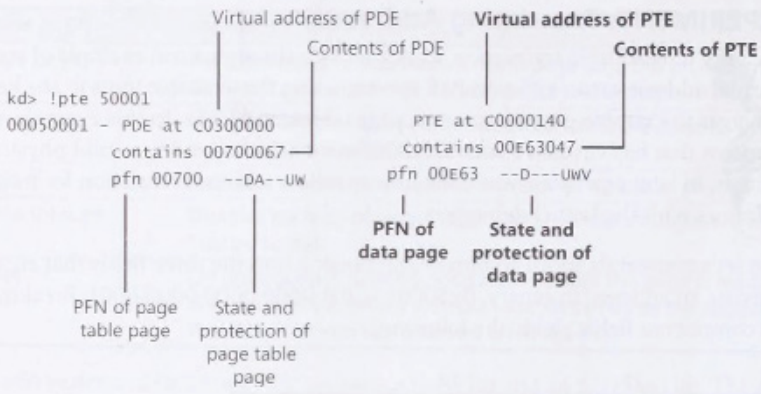
To start the translation process, the CPU needs the physical address of the process page directory, stored in the CR3 register while a thread in that process is running. You can display this address either by examining the CR3 register itself or by dumping the KPROCESS block for the process in question using the *!process* command, as shown here:

```
kd> !process
PROCESS 81555020 Cid: 0099 Peb: 7ffdf000  ParentCid: 0094
 DirBase: 012f0000  ObjectTable: 80695ba8  TableSize: 46.
 Image: IEXPLORE.EXE
      .
      .
      .
kd> r cr3
Cr3: 012F0000
```

Physical address of page directory

In this case, the page directory is stored at physical address 0x12F0000. As shown in the preceding illustration, the page directory index field in this example is 0. Therefore, the PDE is at physical address 0x12F0000.

The kernel debugger *!pte* command displays the PDE and PTE that describe a virtual address, as shown here:



In the first column the kernel debugger displays the PDE, and in the second column it displays the PTE. Notice that the PDE address is shown as a virtual address, not a physical address—as noted earlier, the process page directory starts at virtual address 0xC0300000 on x86 systems. Because we're looking at the first PDE in the page directory, the PDE address is the same as the page directory address.

The PTE is at virtual address 0xC0000140. You can compute this address by multiplying the page table index (0x50 in this example) by the size of a PTE: 0x50 multiplied by 4 equals 0x140. Because the memory manager maps page tables starting at 0xC0000000, adding 140 yields the virtual address shown in the kernel debugger output: 0xC0000140. The page table page is at PFN 0x700, and the data page is at PFN 0xe63.

The PTE flags are displayed to the right of the PFN number. For example, the PTE that describes the page being referenced has flags of D—UWV. *D* here stands for dirty (the page has been modified), *U* for user-mode page (as opposed to a kernel-mode page), *W* for writable page (rather than read-only), and *V* for valid. (The PTE represents a valid page in physical memory.)

Translation Look-Aside Buffer

As we've learned so far, each address translation requires two lookups: one to find the right page table in the page directory and one to find the right entry *in* the page table. Because doing two additional memory lookups for every reference to a virtual address would result in unacceptable system performance, most CPUs cache address translations so that repeated accesses to the same addresses don't have to be retranslated. The processor provides such a cache in the form of an array of associative memory called the *translation look-aside buffer*, or TLB. Associative memory, such as the TLB, is a vector whose cells can be read simultaneously and compared to a target value. In the case of the TLB, the vector contains the virtual-to-physical page mappings of the most recently used pages, as shown in Figure 7-20, and the type of

page protection applied to each page. Each entry in the TLB is like a cache entry, whose tag holds portions of the virtual address and whose data portion holds a physical page number, protection field, valid bit, and usually a dirty bit indicating the condition of the page to which the cached PTE corresponds. If a PTE's global bit is set (used for system space pages that are globally visible to all processes), the TLB entry isn't invalidated on process context switches.

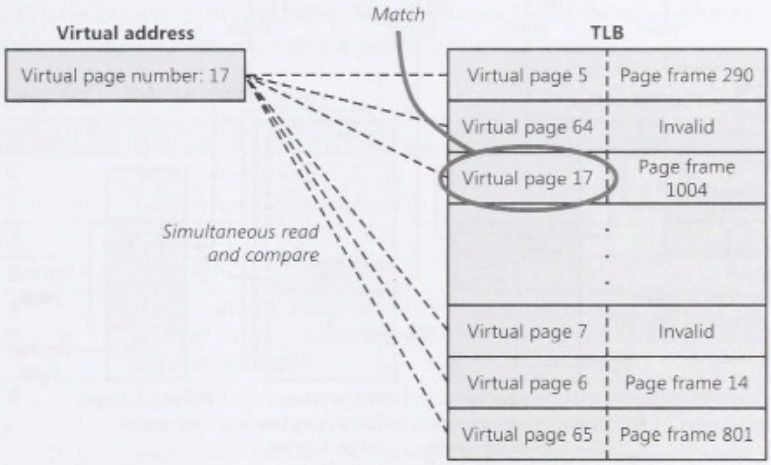


Figure 7-20 Accessing the translation look-aside buffer

Virtual addresses that are used frequently are likely to have entries in the TLB, which provides extremely fast virtual-to-physical address translation and, therefore, fast memory access. If a virtual address isn't in the TLB, it might still be in memory, but multiple memory accesses are needed to find it, which makes the access time slightly slower. If a virtual page has been paged out of memory or if the memory manager changes the PTE, the memory manager explicitly invalidates the TLB entry. If a process accesses it again, a page fault occurs and the memory manager brings the page back into memory and re-creates an entry for it in the TLB.

To maximize the amount of common code, the memory manager treats all PTEs the same whenever possible, whether they are maintained by hardware or by software. For example, the memory manager calls a kernel routine when a PTE changes from invalid to valid. The job of this routine is to load this new PTE into the TLB in whatever hardware-specific manner the architecture requires. On x86 systems, the code is a NOP because the processor loads the TLB without any intervention from the software.

Physical Address Extension (PAE)

The Intel x86 Pentium Pro processor introduced a memory-mapping mode called *Physical Address Extension* (PAE). With the proper chipset, the PAE mode allows access to up to 64 GB of physical memory on current Intel x86 processors and 1024 GB of physical memory on x64 processors (though Windows currently limits this to 128 GB due to the size of the PFN data-

base required to map so much memory). When the processor executes in PAE mode, the memory management unit (MMU) divides virtual addresses into four fields, as shown in Figure 7-21.

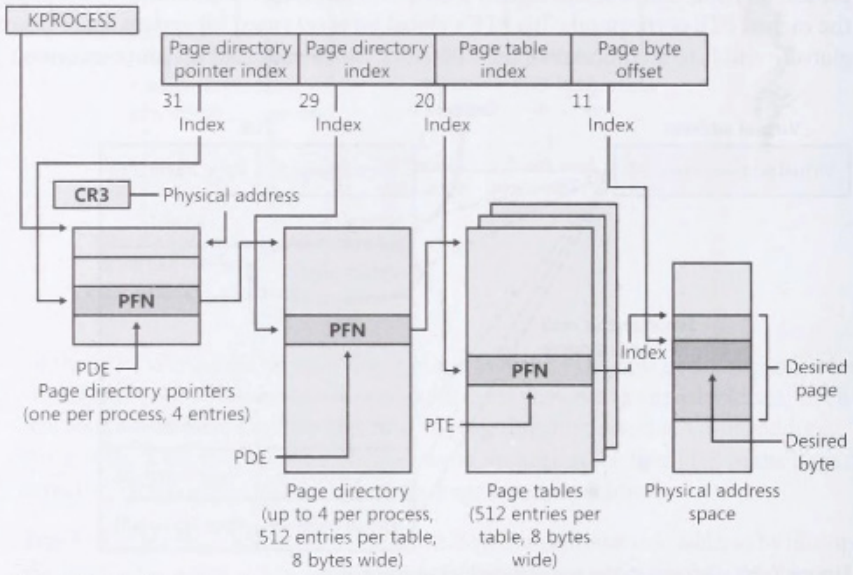


Figure 7-21 Page mappings with PAE

The MMU still implements page directories and page tables, but a third level, the page directory pointer table, exists above them. PAE mode can address more memory than the standard translation mode not because of the extra level of translation but because PDEs and PTEs are 64 bits wide rather than 32 bits. The system represents physical addresses internally with 25 bits, which gives the ability to support a maximum of 2^{25+12} bytes, or 128 GB, of memory. One way in which 32-bit applications can take advantage of such large memory configurations is described in the earlier section "Address Windowing Extensions." However, even if applications are not using such functions, the memory manager will use all available physical memory for file cache data through the use of the system cache, standby, and modified lists (described in the section "Page Frame Number Database").

As explained in Chapter 2, there is a special version of the 32-bit Windows kernel with support for PAE called Ntkrnlpa.exe. To select this PAE-enabled kernel, you must boot with the /PAE switch in Boot.ini. Note that this special version of the kernel image is installed on all 32-bit Windows systems, even Windows 2000 Professional or Windows XP systems with small memory. The reason for this is to facilitate device driver testing. Because the PAE kernel presents 64-bit addresses to device drivers and other system code, booting /PAE even on a small memory system allows device driver developers to test parts of their drivers with large addresses. The other relevant Boot.ini switch is /NOLOWMEM, which discards memory below 4 GB (assuming you have at least 5 GB of physical memory) and relocates device driv-

ers above this range. This guarantees that drivers will be presented with physical addresses greater than 32 bits, which makes any possible driver sign extension bugs easier to find.

IA-64 Virtual Address Translation

The virtual address space for IA-64 is divided into eight regions by the hardware. Each region can have its own set of page tables. Windows uses five of the regions, three of which have page tables. Table 7-12 lists the regions and how they are used.

Table 7-12 The IA-64 Regions

Region	Use
0	User code and data
1	Session space code and data
2	Unused
3	Unused
4	Kseg3, which is a cached 1-to-1 mapping of physical memory. No page tables are needed for this region because the necessary TLB inserts are done directly by the memory manager.
5	Kseg4, which is a noncached 1-to-1 mapping for physical memory. This is used only in a few places for accessing I/O locations such as the I/O port range. There are no page tables needed for this region.
6	Unused
7	Kernel code and data

Address translation by 64-bit Windows on the IA-64 platform uses a three-level page table scheme. Each process has a page directory pointer structure that contains 1024 pointers to page directories. Each page directory contains 1024 pointers to page tables, which in turn point to physical pages. Figure 7-22 shows the format of an IA-64 hardware PTE.

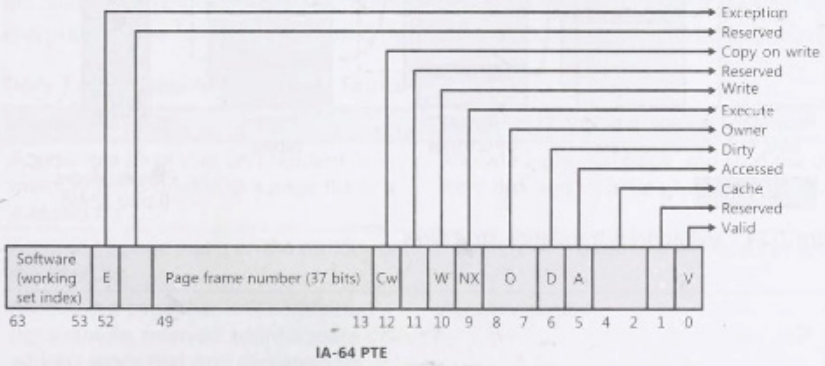


Figure 7-22 IA-64 page table entry

x64 Virtual Address Translation

64-bit Windows on the x64 architecture uses a four-level page table scheme. Each process has a top level extended page directory (called the page map level 4) that contains 512 pointers to a third-level structure called a page parent directory. Each page parent directory contains 512 pointers to second-level page directories, each of which contain 512 pointers to the individual page tables. Finally, the page tables (each of which contain 512 page table entries) point to pages in memory. Current implementations of the x64 architecture limit virtual addresses to 48 bits. The components that make up this 48-bit virtual address are shown in Figure 7-23. The connections between these structures are shown in Figure 7-24. Finally, the format of an x64 hardware page table entry is shown in Figure 7-25.

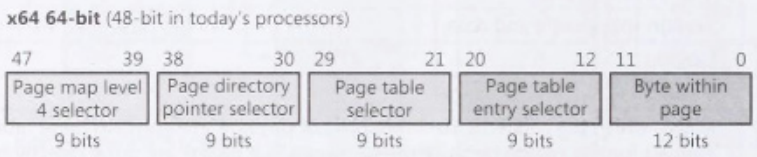


Figure 7-23 x64 virtual address

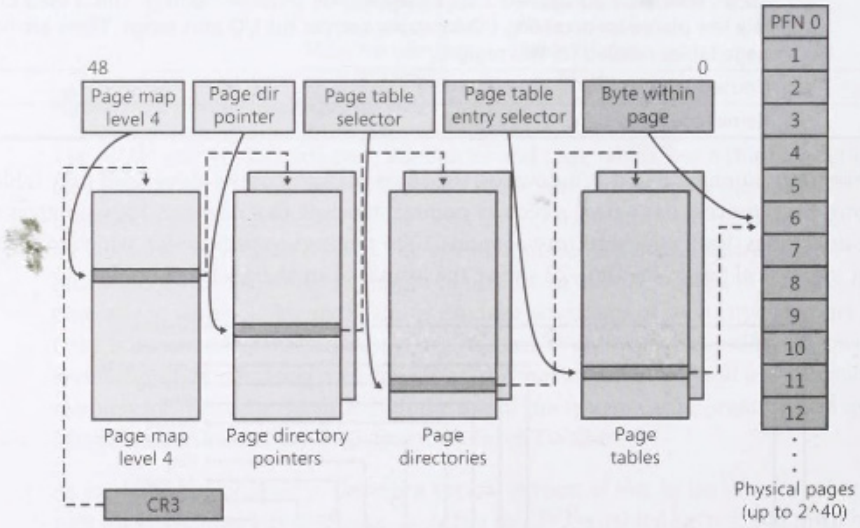


Figure 7-24 x64 address translation structures

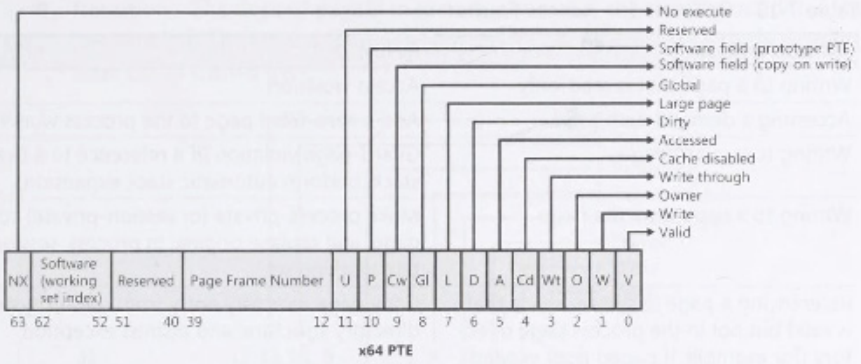


Figure 7-25 x64 hardware page table entry

Page Fault Handling

Earlier, you saw how address translations are resolved when the PTE is valid. When the PTE valid bit is clear, this indicates that the desired page is for some reason not (currently) accessible to the process. This section describes the types of invalid PTEs and how references to them are resolved.



Note Only the 32-bit x86 PTE formats are detailed in this book. PTEs for 64-bit systems contain similar information, but their detailed layout is not presented.

A reference to an invalid page is called a *page fault*. The kernel trap handler (introduced in the section "Trap Dispatching" in Chapter 3) dispatches this kind of fault to the memory manager fault handler (*MmAccessFault*) to resolve. This routine runs in the context of the thread that incurred the fault and is responsible for attempting to resolve the fault (if possible) or raise an appropriate exception. These faults can be caused by a variety of conditions, as listed in Table 7-13.

Table 7-13 Reasons for Access Faults

Reason for Fault	Result
Accessing a page that isn't resident in memory but is on disk in a page file or a mapped file	Allocate a physical page, and read the desired page from disk and into the working set
Accessing a page that is on the standby or modified list	Transition the page to the process or system working set
Accessing a page that isn't committed (for example, reserved address space or address space that isn't allocated)	Access violation
Accessing a page from user mode that can be accessed only in kernel mode	Access violation

Table 7-13 Reasons for Access Faults

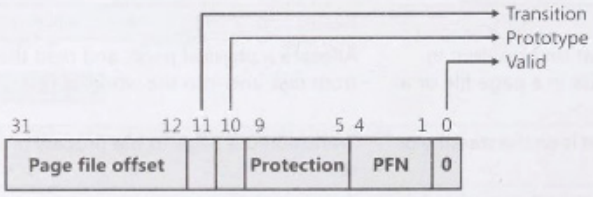
Reason for Fault	Result
Writing to a page that is read-only	Access violation
Accessing a demand-zero page	Add a zero-filled page to the process working set
Writing to a guard page	Guard-page violation (if a reference to a user-mode stack, perform automatic stack expansion)
Writing to a copy-on-write page	Make process-private (or session-private) copy of page, and replace original in process, session, or system working set
Referencing a page in system space that is valid but not in the process page directory (for example, if paged pool expanded after the process page directory was created)	Copy page directory entry from master system page directory structure, and dismiss exception
On a multiprocessor system, writing to a page that is valid but hasn't yet been written to	Set dirty bit in PTE
Executing code in a page that is marked as no execute	Access violation (supported only on hardware platforms that support no execute protection running Windows XP Service Pack 2 or Windows Server 2003 Service Pack 1 and later)

The following section describes the four basic kinds of invalid PTEs that are processed by the access fault handler. Following that is an explanation of a special case of invalid PTEs, prototype PTEs, which are used to implement shareable pages.

Invalid PTEs

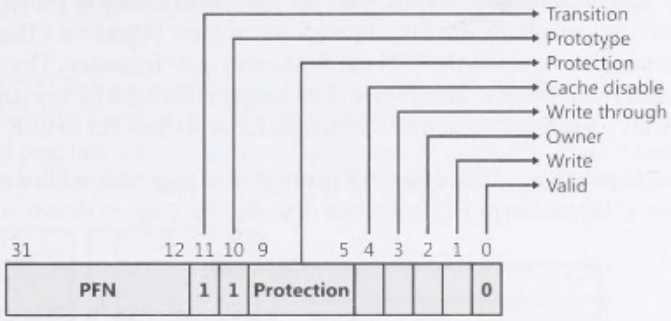
The following list details the four kinds of invalid PTEs and their structure. Some of the flags are the same as those for a hardware PTE as described in Table 7-11.

- **Page file** The desired page resides within a paging file. An in-page operation is initiated, as illustrated here:



- **Demand zero** The desired page must be satisfied with a page of zeros. The pager looks at the zero page list. If the list is empty, the pager takes a page from the free list and zeroes it. If that list is empty, it takes a page from the standby list and zeroes it. The PTE format is the same as the page file PTE shown in the previous entry, but the page file number and offset are zeros.

- **Transition** The desired page is in memory on either the standby, modified, or modified-no-write list. The page will be removed from the list and added to the working set if referenced, as shown here:



- **Unknown** The PTE is zero, or the page table doesn't yet exist. In both cases, this flag means that you should examine the virtual address descriptors (VADs) to determine whether this virtual address has been committed. If so, page tables are built to represent the newly committed address space. (See the discussion of VADs later in the chapter.)

Prototype PTEs

If a page can be shared between two processes, the memory manager relies on a software structure called *prototype page table entries* (prototype PTEs) to map these potentially shared pages. For page file backed sections, an array of prototype PTEs is created when a section object is first created; for mapped files, portions of the array are created on demand as each view is mapped. (See the following note.) These prototype PTEs are part of the *segment* structure, described at the end of this chapter.



Note In Windows 2000 and Windows 2000 Service Pack 1, the memory manager allocates all the prototype page table entries needed to map the entire file, even though applications might only map views to small parts of the file(s) at any one time. Because these structures are allocated from a finite resource (paged pool), attempting to map large files can exhaust this resource. As a result, this limits the total amount of mapped file(s) in use at any time to approximately 200 GB.

This limit has been removed in Windows 2000 Service Pack 2 and beyond by having the memory manager allocate these structures only when mapped views into the file are created. A result of this change is that it is now possible for file backup of any size file on any size system. Previously, you couldn't back up a 500-GB file on a 32-MB system because there wasn't enough paged pool to create the prototype page table entries for the entire section. Now that these structures are allocated for active views only, large file backups are supported on systems with small physical memory configurations.

When a process first references a page mapped to a view of a section object (recall that the VADs are created only when the view is mapped), the memory manager uses the information in the prototype PTE to fill in the real PTE used for address translation in the process page table. When a shared page is made valid, both the process PTE and the prototype PTE point to the physical page containing the data. To track the number of process PTEs that reference a valid shared page, a counter in the PFN database entry is incremented. Thus, the memory manager can determine when a shared page is no longer referenced by any page table and thus can be made invalid and moved to a transition list or written out to disk.

When a shareable page is invalidated, the PTE in the process page table is filled in with a special PTE that points to the prototype PTE entry that describes the page, as shown in Figure 7-26.

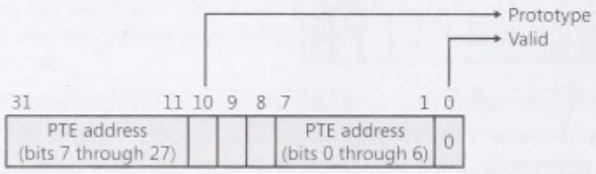


Figure 7-26 Structure of an invalid PTE that points to the prototype PTE

Thus, when the page is later accessed, the memory manager can locate the prototype PTE using the information encoded in this PTE, which in turn describes the page being referenced. A shared page can be in one of six different states as described by the prototype PTE entry:

- **Active/valid** The page is in physical memory as a result of another process that accessed it.
- **Transition** The desired page is in memory on the standby or modified list.
- **Modified-no-write** The desired page is in memory and on the modified-no-write list. (See Table 7-20.)
- **Demand zero** The desired page should be satisfied with a page of zeros.
- **Page file** The desired page resides within a page file.
- **Mapped file** The desired page resides within a mapped file.

Although the format of these prototype PTE entries is the same as that of the real PTE entries described earlier, these prototype PTEs aren't used for address translation—they are a layer between the page table and the page frame number database and never appear directly in page tables.

By having all the accessors of a potentially shared page point to a prototype PTE to resolve faults, the memory manager can manage shared pages without needing to update the page tables of each process sharing the page. For example, a shared code or data page might be

paged out to disk at some point. When the memory manager retrieves the page from disk, it needs only to update the prototype PTE to point to the page's new physical location—the PTEs in each of the processes sharing the page remain the same (with the valid bit clear and still pointing to the prototype PTE). Later, as processes reference the page, the real PTE will get updated.

Figure 7-27 illustrates two virtual pages in a mapped view. One is valid, and the other is invalid. As shown, the first page is valid and is pointed to by the process PTE and the prototype PTE. The second page is in the paging file—the prototype PTE contains its exact location. The process PTE (and any other processes with that page mapped) points to this prototype PTE.

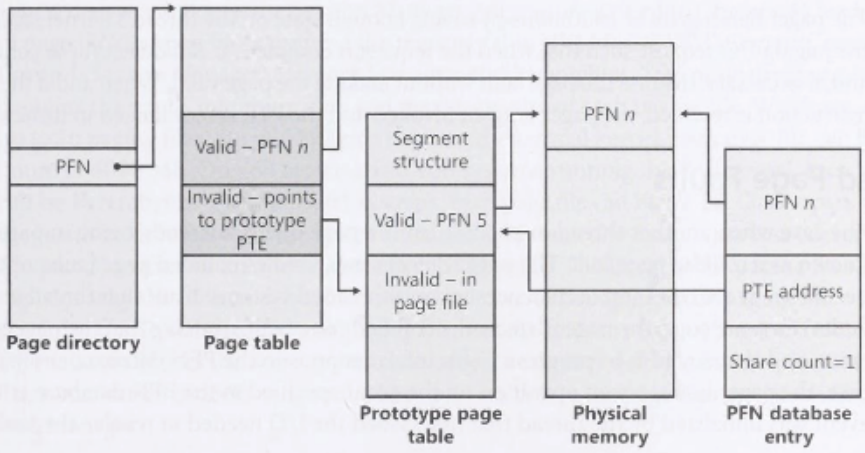


Figure 7-27 Prototype page table entries

In-Paging I/O

In-paging I/O occurs when a read operation must be issued to a file (paging or mapped) to satisfy a page fault. Also, because page tables are pageable, the processing of a page fault can incur additional page faults when the system is loading the page table page that contains the PTE or the prototype PTE that describes the original page being referenced.

The in-page I/O operation is synchronous—that is, the thread waits on an event until the I/O completes—and isn't interruptible by asynchronous procedure call (APC) delivery. The pager uses a special modifier in the I/O request function to indicate paging I/O. Upon completion of paging I/O, the I/O system triggers an event, which wakes up the pager and allows it to continue in-page processing.

While the paging I/O operation is in progress, the faulting thread doesn't own any critical memory management synchronization objects. Other threads within the process are allowed to issue virtual memory functions and handle page faults while the paging I/O takes place.

But a number of interesting conditions that the pager must recognize when the I/O completes are exposed:

- Another thread in the same process or a different process could have faulted the same page (called a *collided page fault* and described in the next section).
- The page could have been deleted (and remapped) from the virtual address space.
- The protection on the page could have changed.
- The fault could have been for a prototype PTE, and the page that maps the prototype PTE could be out of the working set.

The pager handles these conditions by saving enough state on the thread's kernel stack before the paging I/O request such that when the request is complete, it can detect these conditions and, if necessary, dismiss the page fault without making the page valid. When and if the faulting instruction is reissued, the pager is again invoked and the PTE is reevaluated in its new state.

Collided Page Faults

The case when another thread or process faults a page that is currently being in-paged is known as a *collided page fault*. The pager detects and handles collided page faults optimally because they are common occurrences in multithreaded systems. If another thread or process faults the same page, the pager detects the collided page fault, noticing that the page is in transition and that a read is in progress. (This information is in the PFN database entry.) In this case, the pager issues a wait operation on the event specified in the PFN database entry. This event was initialized by the thread that first issued the I/O needed to resolve the fault.



When the I/O operation completes, all threads waiting on the event have their wait satisfied. The first thread to acquire the PFN database lock is responsible for performing the in-page completion operations. These operations consist of checking I/O status to ensure the I/O operation completed successfully, clearing the read-in-progress bit in the PFN database, and updating the PTE.

When subsequent threads acquire the PFN database lock to complete the collided page fault, the pager recognizes that the initial updating has been performed as the read-in-progress bit is clear and checks the in-page error flag in the PFN database element to ensure that the in-page I/O completed successfully. If the in-page error flag is set, the PTE isn't updated and an in-page error exception is raised in the faulting thread.

Page Files

Page files are used to store modified pages that are still in use by some process but have had to be written to disk (because of modified page writing). Page file space is reserved when the pages are initially committed, but the actual page file locations are not chosen until pages are written out to disk. The important point is that the system commit limit is charged for private pages as they are created. Thus, the Process: Page File Bytes performance counter is actually

the total process private committed memory, of which none, some, or all may be in the paging file. (In fact, it's the same as the Process: Private Bytes performance counter.)

The memory manager keeps track of private committed memory usage on a global basis, termed *commitment*, and on a per-process basis as *page file quota*. (Again, this memory usage doesn't represent page file usage—it represents private committed memory usage.) Commitment and page file quota are charged whenever virtual addresses that require new private physical pages are committed. Once the global commit limit has been reached (physical memory and the page files are full), allocating virtual memory will fail until processes free committed memory (for example, when a process exits).

When the system boots, the Session Manager process (described in Chapter 4) reads the list of page files to open by examining the registry value HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PagingFiles. This multistring registry value contains the name, minimum size, and maximum size of each paging file. Windows supports up to 16 paging files. On x86 systems running the normal kernel, each page file can be a maximum of 4095 MB. On x64 systems and x86 systems running the PAE kernel, each page file can be 16 terabytes (TB). On IA-64 systems, each page file can be 32 TB. Once open, the page files can't be deleted while the system is running because the System process (described in Chapter 2) maintains an open handle to each page file. The fact that the paging files are open explains why the built-in defragmentation tool cannot defragment the paging file while the system is up. To defragment your paging file, use the freeware Pagedefrag tool from *www.sysinternals.com*. It uses the same approach as other third-party defragmentation tools—it runs its defragmentation process early in the boot process before the page files are opened by the Session Manager.

Because the page file contains parts of process and kernel virtual memory, for security reasons the system can be configured to clear the page file at system shutdown. To enable this, set the registry value HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\ClearPageFileAtShutdown to 1. Otherwise, after shutdown, the page file will contain whatever data happened to have been paged out while the system was up. This data could then be accessed by someone who gained physical access to the machine.

If no paging files are specified, Windows 2000 creates a default 20-MB page file on the boot partition. Windows XP and Windows Server 2003 do not create this temporary paging file, which means the system virtual memory commit limit is based on available memory. In Windows XP and Windows Server 2003, if the minimum and maximum paging file sizes are both zero, this indicates a system managed paging file, which causes the system to choose the page file size as shown in Table 7-14.

Table 7-14 Default Page File Sizes

System Memory Size	Minimum Page File Size	Maximum Page File Size
< 1 GB	1.5 * RAM	3 * RAM
>= 1 GB	1 * RAM	3 * RAM



EXPERIMENT: Viewing System Page Files

To view the list of page files, look in the registry at HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\PagingFiles. This contains the paging file configuration settings modified through the System utility in Control Panel. In Windows 2000, click the Performance Options button on the Advanced tab, and then click the Change button. In Windows XP and Windows Server 2003, click the Advanced tab, click the Settings button in the Performance section, click the Advanced tab, and finally, click the Change button in the Virtual Memory section.

To add a new page file, Control Panel uses the (internal only) *NtCreatePagingFile* system service defined in Ntdll.dll. Page files are always created as noncompressed files, even if the directory they are in is compressed. To keep new page files from being deleted, a handle is duplicated into the System process so that when the creating process closes the handle to the new page file, another process can still open the page file.

The performance counters listed in Table 7-15 allow you to examine private committed memory usage on a systemwide or per-page-file basis. There's no way to determine how much of a process's private committed memory is resident and how much is paged out to paging files.

Table 7-15 Committed Memory and Page File Performance Counters

Performance Counter	Description
Memory: Committed Bytes	Number of bytes of virtual (not reserved) memory that has been committed. This number doesn't necessarily represent page file usage because it includes private committed pages in physical memory that have never been paged out. Rather, it represents the amount of page file space that would be used if the process was completely made nonresident.
Memory: Commit Limit	Number of bytes of virtual memory that can be committed without having to extend the paging files; if the paging files can be extended, this limit is soft.
Paging File: % Usage	Percentage of the paging file committed.
Paging File: % Usage Peak	Highest percentage of the paging file committed.

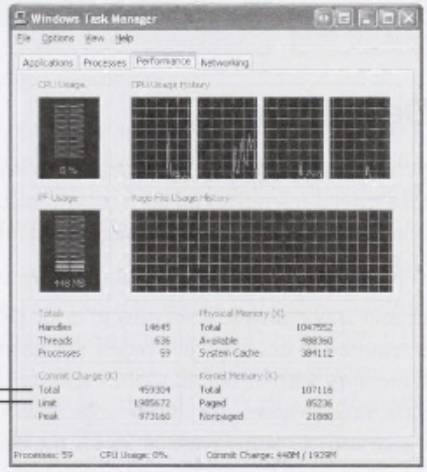
Note that these counters can assist you in choosing a page file size. Although most do it, basing page file size as a function of RAM makes no sense because the more memory you have, the less likely you are to need to page data out. To determine how much page file space your system really needs based on the mix of applications that have run since the system booted, examine the peak commit charge (displayed in the Commit Charge section of Task Manager's performance tab and also in Process Explorer's System Information display). This number represents the peak amount of page file space since the system booted that would have been needed if the system had to page out all private committed virtual memory (which rarely happens).

If the page file on your system is too big, the system will not use it any more or less—in other words, increasing the size of the page file does not change system performance, it simply means the system can have more nonshareable committed virtual memory. If the page file is too small for the mix of applications you are running, you might get the "system running low on virtual memory" error message. In this case, first check to see whether a process has a memory leak by examining the process private bytes count (found in the "VM Size" column in Task Manager's Processes tab). If no process appears to have a leak, check the system paged pool size—if a device driver is leaking paged pool, this might also explain the error. (See the "Troubleshooting a Pool Leak" experiment in the "System Memory Pools" section for how to troubleshoot a pool leak.)



EXPERIMENT: Viewing Page File Usage with Task Manager

You can also view committed memory usage with Task Manager by clicking its Performance tab. You'll see the following counters related to page files:

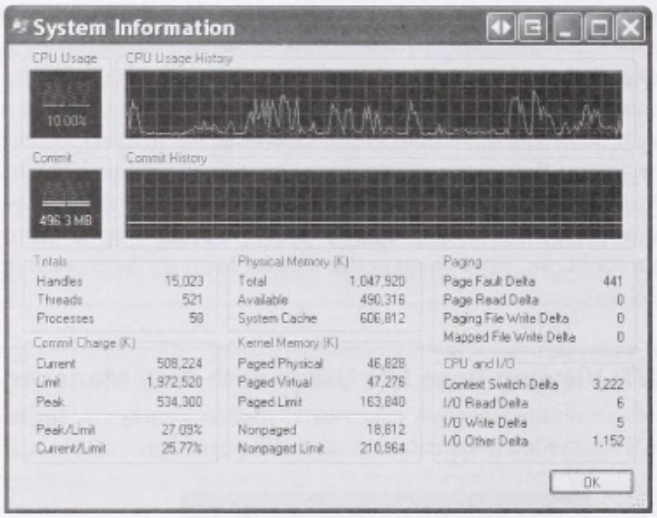


Total virtual memory that has committed storage

Maximum virtual memory without extending page file

Note that the Mem Usage bar (called PF Usage in Windows XP and Windows Server 2003) is actually the system commit total. This number represents *potential* page file usage, not actual page file usage. It is how much page file space would be used if all the private committed virtual memory in the system had to be paged out all at once.

Process Explorer's System Information display shows an additional piece of information about system commit usage, namely the percentage of the peak as compared to the limit and the current usage as compared to the limit:



Virtual Address Descriptors

The memory manager uses a demand-paging algorithm to know when to load pages into memory, waiting until a thread references an address and incurs a page fault before retrieving the page from disk. Like copy-on-write, demand paging is a form of *lazy evaluation*—waiting to perform a task until it is required.



The memory manager uses lazy evaluation not only to bring pages into memory but also to construct the page tables required to describe new pages. For example, when a thread commits a large region of virtual memory with *VirtualAlloc*, the memory manager could immediately construct the page tables required to access the entire range of allocated memory. But what if some of that range is never accessed? Creating page tables for the entire range would be a wasted effort. Instead, the memory manager waits to create a page table until a thread incurs a page fault, and then it creates a page table for that page. This method significantly improves performance for processes that reserve and/or commit a lot of memory but access it sparsely.

With the lazy-evaluation algorithm, allocating even large blocks of memory is a fast operation. When a thread allocates memory, the memory manager must respond with a range of addresses for the thread to use. To do this, the memory manager maintains another set of data structures to keep track of which virtual addresses have been reserved in the process's address space and which have not. These data structures are known as *virtual address descriptors* (VADs). For each process, the memory manager maintains a set of VADs that describes the status of the process's address space. VADs are structured as a self-balancing binary tree to make lookups efficient. Windows Server 2003 implements an AVL tree algorithm (named after their inventors, Adelson-Velskii and Landis) that better balances the VAD tree, resulting in, on aver-

age, fewer comparisons when searching for a VAD corresponding with a virtual address. A diagram of a VAD tree is shown in Figure 7-28.

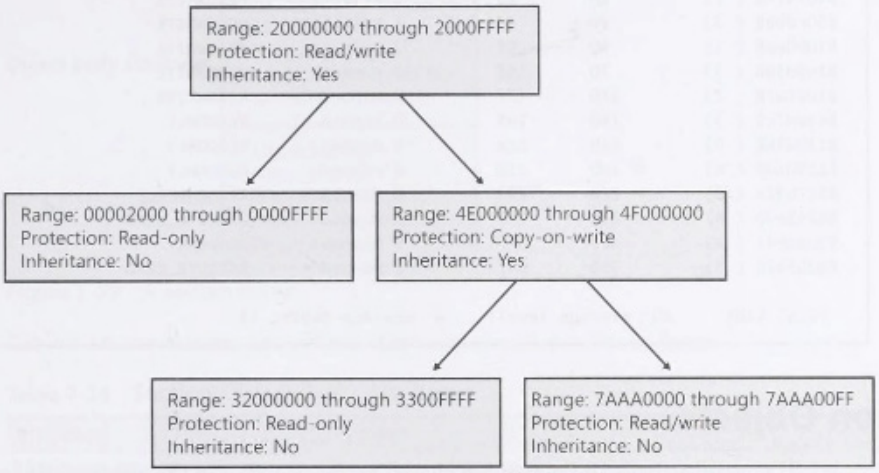


Figure 7-28 Virtual address descriptors

When a process reserves address space or maps a view of a section, the memory manager creates a VAD to store any information supplied by the allocation request, such as the range of addresses being reserved, whether the range will be shared or private, whether a child process can inherit the contents of the range, and the page protection applied to pages in the range.

When a thread first accesses an address, the memory manager must create a PTE for the page containing the address. To do so, it finds the VAD whose address range contains the accessed address and uses the information it finds to fill in the PTE. If the address falls outside the range covered by the VAD or in a range of addresses that are reserved but not committed, the memory manager knows that the thread didn't allocate the memory before attempting to use it and therefore generates an access violation.



EXPERIMENT: Viewing Virtual Address Descriptors

You can use the kernel debugger's *!vad* command to view the VADs for a given process. First find the address of the root of the VAD tree with the *!process* command. Then specify that address to the *!vad* command, as shown in the following example of the VAD tree for a process running Notepad.exe:

```
kd> !process 2a0 1
Searching for Process with Cid == 2a0
PROCESS 8614d030 SessionId: 0 Cid: 02a0 Peb: 7ffdf000 ParentCid: 0554
    DirBase: 00d93000  ObjectTable: 81bc47c8  TableSize:  41.
    Image: notepad.exe
    VadRoot 8118d868 Clone 0 Private 252. Modified 0. Locked 0.
```

```
kd> !vad 8118d868
VAD     level      start      end    commit
84df4148 ( 2)         10       10         1 Private      READWRITE
850cdbe8 ( 3)         20       20         1 Private      READWRITE
810b0ee8 ( 1)         30       6f         7 Private      READWRITE
8109d308 ( 3)         70      16f        32 Private      READWRITE
810e9a28 ( 2)        170      17f         0 Mapped       READWRITE
84aedfc8 ( 3)        180      195         0 Mapped       READONLY
8118d868 ( 0)        1a0      1ce         0 Mapped       READONLY
81190a08 ( 4)        1d0      210         0 Mapped       READONLY
85c7b928 ( 3)        220      223         0 Mapped       READONLY
86253a08 ( 4)        230      2f7         0 Mapped       EXECUTE_READ
810aab48 ( 2)        300      342         0 Mapped       READONLY
80db5448 ( 5)        350      64f         0 Mapped       EXECUTE_READ

 Total VADs:    49  average level:    6  maximum depth: 13
```

Section Objects

As you'll remember from the section on shared memory earlier in the chapter, the *section object*, which the Windows subsystem calls a *file mapping object*, represents a block of memory that two or more processes can share. A section object can be mapped to the paging file or to another file on disk.

The executive uses sections to load executable images into memory, and the cache manager uses them to access data in a cached file. (See Chapter 11 for more information on how the cache manager uses section objects.) You can also use section objects to map a file into a process address space. The file can then be accessed as a large array by mapping different views of the section object and reading or writing to memory rather than to the file (an activity called *mapped file I/O*). When the program accesses an invalid page (one not in physical memory), a page fault occurs and the memory manager automatically brings the page into memory from the mapped file. If the application modifies the page, the memory manager writes the changes back to the file during its normal paging operations (or the application can flush a view by using the Windows *FlushViewOfFile* function).

Section objects, like other objects, are allocated and deallocated by the object manager. The object manager creates and initializes an object header, which it uses to manage the objects; the memory manager defines the body of the section object. The memory manager also implements services that user-mode threads can call to retrieve and change the attributes stored in the body of section objects. The structure of a section object is shown in Figure 7-29.

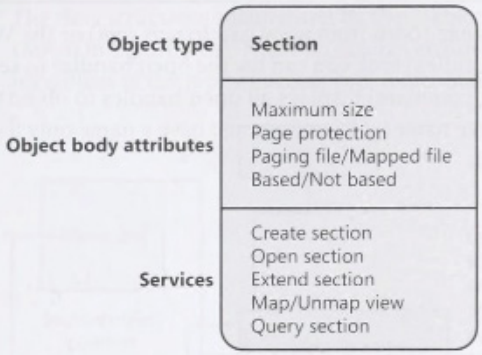


Figure 7-29 A section object

Table 7-16 summarizes the unique attributes stored in section objects.

Table 7-16 Section Object Body Attributes

Attribute	Purpose
Maximum size	The largest size to which the section can grow in bytes; if mapping a file, the maximum size is the size of the file.
Page protection	Page-based memory protection assigned to all pages in the section when it is created.
Paging file/Mapped file	Indicates whether the section is created empty (backed by the paging file—as explained earlier, page file backed sections use page-file resources only when the pages need to be written out to disk) or loaded with a file (backed by the mapped file).
Based/Not based	Indicates whether a section is a based section, which must appear at the same virtual address for all processes sharing it, or a non-based section, which can appear at different virtual addresses for different processes.



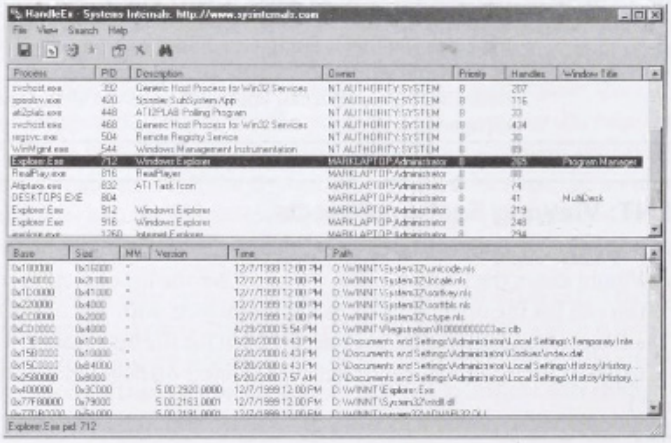
EXPERIMENT: Viewing Section Objects

With the Object Viewer (Winobj.exe from *www.sysinternals.com* from the book's companion CD or Winobj.exe in the Platform SDK), you can see the list of sections that have global names. You can list the open handles to section objects with any of the tools described in the Object Manager section in Chapter 3 that list the open handle table. (As explained in Chapter 3, these names are stored in the object manager directory \BaseNamedObjects.)

Using Process Explorer or Handles.exe (both from *www.sysinternals.com*) or the Windows resource kit Oh.exe (Open Handles) tool, you can list the open handles to section objects. For example, the following command displays all open handles to objects of type *section*, whether or not they have names. (A section must have a name only if other processes need to open it by name.)

```
c:\> oh -t section -a
00000008 System          Section  0070
0000008C smss.exe        Section  0004
000000A4 csrss.exe       Section  0004
000000A4 csrss.exe       Section  0024
000000A4 csrss.exe       Section  0038
000000A4 csrss.exe       Section  0040 \NLS\NlsSectionUnicode
000000A4 csrss.exe       Section  0044 \NLS\NlsSectionLocale
000000A4 csrss.exe       Section  0048 \NLS\NlsSectionCType
000000A4 csrss.exe       Section  004c \NLS\NlsSectionSortkey
000000A4 csrss.exe       Section  0050 \NLS\NlsSectionSortTbls
000000A0 winlogon.exe    Section  0004
000000A0 winlogon.exe    Section  0034
000000A0 winlogon.exe    Section  0168 \BaseNamedObjects\mmGlobalPnpInfo
§
```

You can also use Process Explorer from *www.sysinternals.com* to view mapped files. Select DLLs from the Lower Pane View entry of the View menu. Files marked with an asterisk in the MM column are mapped files (rather than DLLs and other files the image loader loads as modules). Here's an example:



The data structures maintained by the memory manager that describe mapped sections are shown in Figure 7-30. These structures ensure that data read from mapped files is consistent, regardless of the type of access (open file, mapped file, and so on).

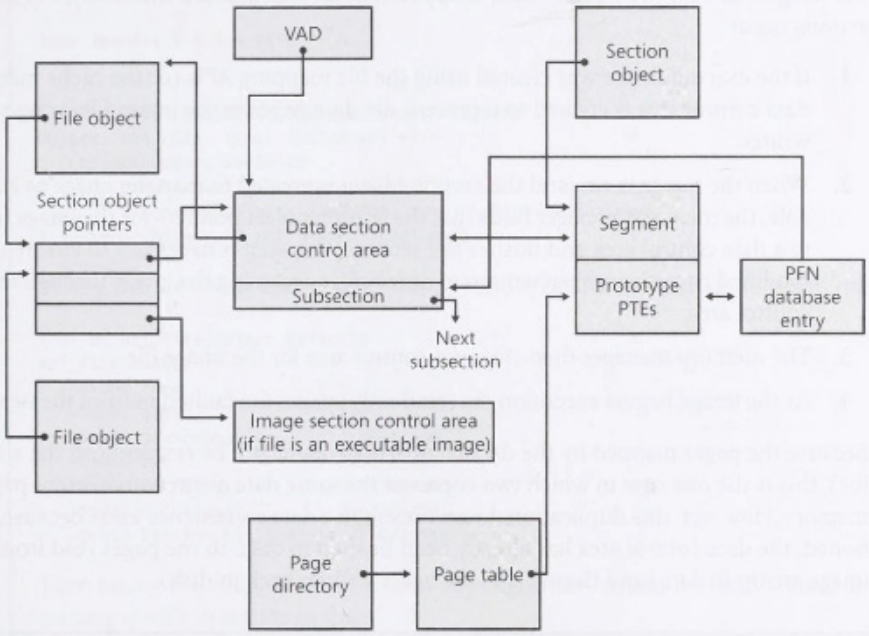


Figure 7-30 Internal section structures

For each open file (represented by a file object), there is a single *section object pointers* structure. This structure is the key to maintaining data consistency for all types of file access as well as to providing caching for files. The section object pointers structure points to one or two *control areas*. One control area is used to map the file when it is accessed as a data file, and one is used to map the file when it is run as an executable image.

A control area in turn points to *subsection* structures that describe the mapping information for each section of the file (read-only, read-write, copy-on-write, and so on). The control area also points to a *segment* structure allocated in paged pool, which in turn points to the prototype PTEs used to map to the actual pages mapped by the section object. As described earlier in the chapter, process page tables point to these prototype PTEs, which in turn map the pages being referenced.

Although Windows ensures that any process that accesses (reads or writes) a file will always see the same, consistent data, there is one case in which two copies of pages of a file can reside in physical memory (but even in this case, all accessors get the latest copy and data consistency

is maintained). This duplication can happen when an image file has been accessed as a data file (having been read or written) and then run as an executable image (for example, when an image is linked and then run—the linker had the file open for data access, and then when the image was run, the image loader mapped it as an executable). Internally, the following actions occur:

1. If the executable file was created using the file mapping APIs (or the cache manager), a data control area is created to represent the data pages in the image file being read or written.
2. When the image is run and the section object is created to map the image as an executable, the memory manager finds that the section object pointers for the image file point to a data control area and flushes the section. This step is necessary to ensure that any modified pages have been written to disk before accessing the image through the image control area.
3. The memory manager then creates a control area for the image file.
4. As the image begins execution, its (read-only) pages are faulted in from the image file.

Because the pages mapped by the data control area might still be resident (on the standby list), this is the one case in which two copies of the same data are in two different pages in memory. However, this duplication doesn't result in a data consistency issue because, as mentioned, the data control area has already been flushed to disk, so the pages read from the image are up to date (and these pages are never written back to disk).



EXPERIMENT: Viewing Control Areas

To find the address of the control area structures for a file, you must first get the address of the file object in question. You can obtain this address through the kernel debugger by dumping the process handle table with the *!handle* command and noting the object address of a file object. Although the kernel debugger *!file* command displays the basic information in a file object, it doesn't display the pointer to the section object pointers structure. Then, using the *dt* command, format the file object to get the address of the section object pointers structure. This structure consists of three pointers: a pointer to the data control area, a pointer to the shared cache map (explained in Chapter 11), and a pointer to the image control area. From the section object pointers structure, you can obtain the address of a control area for the file (if one exists) and feed that address into the *!ca* command.

For example, if you open a PowerPoint file and display the handle table for that process using *!handle*, you will find an open handle to the PowerPoint file as shown below. (For information on using *!handle*, see the "Object Manager" section in Chapter 3.)

```
lkd> !handle 1 f 5c4 file
.
.
0284: Object: 8645c038  GrantedAccess: 00120089
Object: 8645c038  Type: (867ddca0) File
    ObjectHeader: 8645c020
        HandleCount: 1  PointerCount: 1
        Directory Object: 00000000  Name: \slides\ntint\new\
3-systemarchitecture.ppt {HarddiskVolume1}
```

Taking the file object address (8645c038) and formatting it with *dt* results in this:

```
lkd> dt nt!_file_object 8645c038
nt!_FILE_OBJECT
   +0x000 Type             : 5
   +0x002 Size             : 112
   +0x004 DeviceObject     : 0x86742e30
   +0x008 Vpb              : 0x867a3090
   +0x00c FsContext        : 0xe2786530
   +0x010 FsContext2       : 0xe309a378
   +0x014 SectionObjectPointer : 0x85512fec
```

Then taking the address of the section object pointers structure (0x85512fec) and formatting it with *dt* results in this:

```
lkd> dt nt!_section_object_pointers 0x85512fec
nt!_SECTION_OBJECT_POINTERS
   +0x000 DataSectionObject : 0x8564f8a0
   +0x004 SharedCacheMap    : (null)
   +0x008 ImageSectionObject : (null)
```

Finally, use *!ca* to display the control area using the address:

```
lkd> !ca 0x8564f8a0
ControlArea @8564f8a0
  Segment:    e3817528    Flink              0   Blink              0
  Section Ref        1    Pfn Ref          25c   Mapped Views       1
  User Ref           2    WaitForDel         0   Flush Count        0
  File Object 85774580    ModWriteCount      0   System Views       0
  Flags (9008080) File WasPurged HadUserReference Accessed
  File: \slides\ntint\new\3-systemarchitecture.ppt
```

Another technique is to display the list of all control areas with the *!memusage* command. The following excerpt is from the output of this command:

```
kd> !memusage
 loading PFN database
loading (99% complete)
             Zeroed:      9 (    36 kb)
               Free:      0 (     0 kb)
            Standby:   2103 (  8412 kb)
           Modified:    300 (  1200 kb)
    ModifiedNowrite:      1 (     4 kb)
       Active/Valid:  30318 (121272 kb)
         Transition:      3 (    12 kb)
            Unknown:      0 (     0 kb)
              TOTAL:  32734 (130936 kb)
  Building kernel map
  Finished building kernel map
  Usage Summary (in Kb):
Control Valid Standby Dirty Shared Locked PageTables  name
8119b608  3000   528     0      0      0      0  mapped_file( WINWORD.EXE )
849e7c68    96     0     0      0      0      0    No Name for File
8109c388    24     0     0      0      0      0  mapped_file( DEVDTG.PKG )
81402488   236     0     0      0      0      0    No Name for File
80fba0a8   268     0     0      0      0      0  mapped_file( kernel32.pdb )
810ab168  1504   380     0      0      0      0  mapped_file( OUTLLIB.DLL )
81126a08     0   276     0      0      0      0  mapped_file( H )
81112d28   656     0     0      0      0      0    No Name for File
```

The Control column points to the control area structure that describes the mapped file. You can display control areas, segments, and subsections with the kernel debugger *!ca* command. For example, to dump the control area for the mapped file Winword.exe in this example, type the *!ca* command followed by the Control number, as shown here:

```
kd> !ca 8119b608

ControlArea @8119b608
  Segment:    e2c28000  Flink             0  Blink:            0
  Section Ref        1  Pfn Ref         372  Mapped Views:     1
  User Ref           2  Subsections       6  Flush Count:      0
  File Object 8213fb98  ModWriteCount     0  System Views:     0
  WaitForDel         0  Paged Usage    3000  NonPaged Usage  100
  Flags (90000a0) Image File HadUserReference Accessed

   File: \Program Files\Microsoft Office\Office\WINWORD.EXE

Segment @ e2c28000:
   Base address        0  Total Ptes       86a  NonExtendPtes:      86a
   Image commit       d7  ControlArea 8119b608  SizeOfSegment: 86a000
   Image Base          0  Committed          0  PTE Template:  919b6c38
   Based Addr   30000000  ProtoPtes   e2c28038  Image Info:    e2c2a1e4

Subsection 1. @ 8119b640
   ControlArea: 8119b608  Starting Sector 0 Number Of Sectors 8
```

```
    Base Pte     e2c28038  Ptes In subsect         1 Unused Ptes          0
    Flags              15  Sector Offset           0 Protection           1
     ReadOnly CopyOnWrite

Subsection 2. @ 8119b660
    ControlArea: 8119b608  Starting Sector 10 Number Of Sectors 3c00
    Base Pte     e2c2803c  Ptes In subsect       780 Unused Ptes          0
    Flags              35  Sector Offset           0 Protection           3
     ReadOnly CopyOnWrite

Subsection 3. @ 8119b680
    ControlArea: 8119b608  Starting Sector 3c10 Number Of Sectors 5d8
    Base Pte     e2c29e3c  Ptes In subsect        c1 Unused Ptes          0
    Flags              55  Sector Offset           0 Protection           5
     ReadOnly CopyOnWrite

Subsection 4. @ 8119b6a0
    ControlArea: 8119b608  Starting Sector 41e8 Number Of Sectors a8
    Base Pte     e2c2a140  Ptes In subsect        15 Unused Ptes          0
    Flags              55  Sector Offset           0 Protection           5
     ReadOnly CopyOnWrite

Subsection 5. @ 8119b6c0
    ControlArea: 8119b608  Starting Sector 0 Number Of Sectors 0
    Base Pte     e2c2a194  Ptes In subsect         1 Unused Ptes          0
    Flags              55  Sector Offset           0 Protection           5
     ReadOnly CopyOnWrite
 Subsection 6. @ 8119b6e0
    ControlArea: 8119b608  Starting Sector 4290 Number Of Sectors 90
    Base Pte     e2c2a198  Ptes In subsect        12 Unused Ptes          0
    Flags              15  Sector Offset           0 Protection           1
     ReadOnly CopyOnWrite
```

Working Sets

In the last several sections, we've concentrated on the virtual view of a Windows process—page tables, PTEs, and VADs. In the remainder of this chapter, we'll explain how Windows keeps a subset of virtual addresses in physical memory.

As you'll recall, the term used to describe a subset of virtual pages resident in physical memory is called a *working set*. There are three kinds of working sets:

- Process working sets contain the pages referenced by threads within a single process.
- The system working set contains the resident subset of the pageable system code (for example, Ntoskrnl.exe and drivers), paged pool, and the system cache.
- On systems with terminal services enabled, each session has a working set that contains the resident subset of the kernel-mode session-specific data structures allocated by the kernel-mode part of the Windows subsystem (Win32k.sys), session paged pool, session mapped views, and other session-space device drivers.

Before examining the details of each type of working set, let's look at the overall policy for deciding which pages are brought into physical memory and how long they remain. After that, we'll explore the various types of working sets.

Demand Paging

The Windows memory manager uses a demand-paging algorithm with clustering to load pages into memory. When a thread receives a page fault, the memory manager loads into memory the faulted page plus a small number of pages preceding and/or following it. This strategy attempts to minimize the number of paging I/Os a thread will incur. Because programs, especially large ones, tend to execute in small regions of their address space at any given time, loading clusters of virtual pages reduces the number of disk reads. For page faults that reference data pages in images, the cluster size is 3 pages. For all other page faults, the cluster size is 7 pages.

However, a demand paging policy can result in a process incurring many page faults when its threads first begin executing or when they resume execution at a later point. To optimize the startup of a process (and the system), Windows XP and Windows Server 2003 have an intelligent prefetch engine called the *logical prefetcher*, described in the next section.

Logical Prefetcher

During a typical system boot or application startup, the order of faults is such that some pages are brought in from one part of a file, then perhaps from a distant part of the same file, then from a different file, perhaps from a directory, and then again from the first file. This jumping around slows down each access considerably and, thus, analysis shows that disk seek times are a dominant factor in slowing boot and application startup times. By prefetching batches of pages all at once, a more sensible ordering of access, without excessive backtracking, can be achieved, thus improving the overall time for system and application startup. The pages that are needed can be known in advance because of the high correlation in accesses across boots or application starts.

The prefetcher, introduced in Windows XP, tries to speed the boot process and application startup by monitoring the data and code accessed by boot and application startups and using that information at the beginning of a subsequent boot or application startup to read in the code and data. When the prefetcher is active, the memory manager notifies the prefetcher code in the kernel of page faults, both those that require that data be read from disk (hard faults) and those that simply require data already in memory be added to a process's working set (soft faults). The prefetcher monitors the first 10 seconds of application startup. For boot, the prefetcher by default traces from system start through the 30 seconds following the start of the user's shell (typically Explorer) or, failing that, up through 60 seconds following Windows service initialization or through 120 seconds, whichever comes first.

The trace assembled in the kernel notes faults taken on the NTFS Master File Table (MFT) metadata file (if the application accesses files or directories on NTFS volumes), on referenced files, and on referenced directories. With the trace assembled, the kernel prefetcher code notifies the prefetcher component of the Task Scheduler service (\Windows\System32\Schedsvc.dll), running in a copy of Svchost, by signaling an event object named *PrefetchTracesReady*.



Note You can enable or disable prefetching of the boot or application startups by editing the DWORD registry value HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management\PrefetchParameters\EnablePrefetcher. Set it to 0 to disable prefetching altogether, 1 to enable prefetching of only applications, 2 for prefetching of boot only, and 3 for both boot and applications.

When the *PrefetchTracesReady* event is signaled, the Task Scheduler performs a call to the internal *NtQuerySystemInformation* system call requesting the trace data. The Task Scheduler post processes the trace data, combining it with previously collected data, and writes it out to a file in the \Windows\Prefetch folder, which is shown in Figure 7-31. The file's name is the name of the application to which the trace applies followed by a dash and the hexadecimal representation of a hash of the file's path. The file has a ".pf" extension, so an example would be NOTEPAD.EXE-AF43252301.PF.

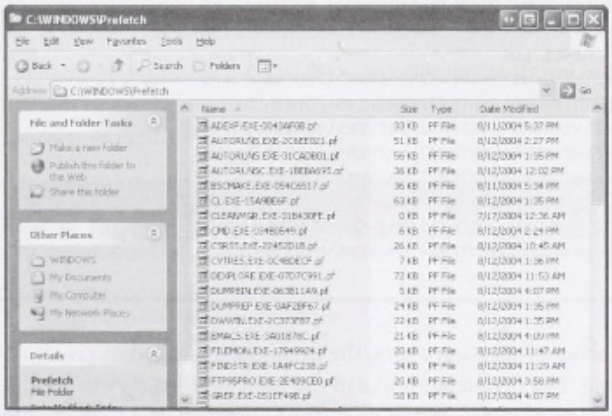


Figure 7-31 Prefetch folder

There are two exceptions to the file name rule. The first is for images that host other components, including the Microsoft Management Console (\Windows\System32\Mmc.exe) and Dllhost (\Windows\System32\Dllhost.exe). Because add-on components are specified on the command line for these applications, the prefetcher includes the command line in the generated hash. Thus, invocations of these applications with different components on the command line will result in different traces. The prefetcher reads the list of executables that it should treat this

way from the *HostingAppList* value in its parameters registry key, HKLM\System\CurrentControlSet\Control\Session Manager\Memory Management\PrefetchParameters.

The other exception to the file name rule is the file that stores the boot's trace, which is always named NTOSBOOT-B00DFAAD.PF. (If read as a word, "boodfaad" sounds similar to the English words *boot fast*.) Only after the prefetcher has finished the boot trace (the time of which was defined earlier) does it collect page fault information for specific applications.



EXPERIMENT: Looking Inside a Prefetch File

A prefetch file's contents serve as a record of files and directories accessed during the boot or an application startup, and you can use the Strings utility from *www.sysinternals.com* to see the record. The following command lists all the files and directories referenced during the last boot:

```
C:\Windows\Prefetch>Strings ntosboot-boodfaad.pf

Strings v2.1
Copyright (C) 1999-2003 Mark Russinovich
Systems Internals - www.sysinternals.com

NTOSBOOT
SCCA
\DEVICE\HARDDISKVOLUME2\$MFT
\DEVICE\HARDDISKVOLUME2\WINDOWS\PREFETCH\NTOSBOOT-B00DFAAD.PF
\DEVICE\HARDDISKVOLUME2\SYSTEM VOLUME INFORMATION\_RESTORE{987E0331-0F01-427C-A5
8A-7A2E4AABF84D}\RP24\CHANGE.LOG
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\PROCESSR.SYS
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\FGLRYM.SYS
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\VIDEOPRT.SYS
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\E1000325.SYS
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\USBUHCI.SYS
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\USBPORT.SYS
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\USBEHCI.SYS
\DEVICE\HARDDISKVOLUME2\WINDOWS\SYSTEM32\DRIVERS\NIC1394.SYS
_
```

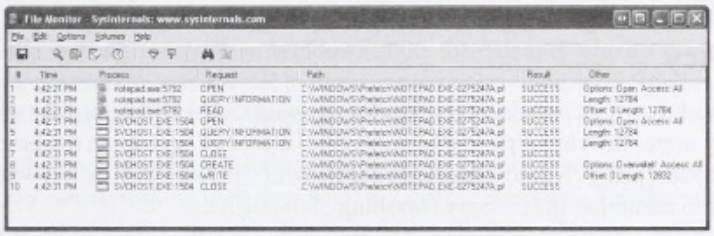


When the system boots or an application starts, the prefetcher is called to give it an opportunity to perform prefetching. The prefetcher looks in the prefetch directory to see if a trace file exists for the prefetch scenario in question. If it does, the prefetcher calls NTFS to prefetch any MFT metadata file references, reads in the contents of each of the directories referenced, and finally opens each file referenced. It then calls the memory manager function *MmPrefetchPages* to read in any data and code specified in the trace that's not already in memory. The memory manager initiates all the reads asynchronously and then waits for them to complete before letting an application's startup continue.



EXPERIMENT: Watching Prefetch File Reads and Writes

If you capture a trace of application startup with Filemon from *www.sysinternals.com* in Windows XP, you can see the prefetcher check for and read the application's prefetch file (if it exists), and roughly ten seconds after the application started, see the prefetcher write out a new copy of the file. Below is a capture of Notepad startup with an Include filter set to "prefetch" so that Filemon shows only accesses to the \Windows\Prefetch directory:



Lines 1 through 3 show the Notepad prefetch file being read in the context of the Notepad process during its startup. Lines 4 through 10, which have time stamps 10 seconds later than the first 3 lines, show the Task Scheduler, which is running in the context of a Svchost process, write out the updated prefetch file.

To minimize seeking even further, every three days or so, during system idle periods, the Task Scheduler organizes a list of files and directories in the order that they are referenced during a boot or application start and stores the list in a file named Windows\Prefech\Layout.ini, shown in Figure 7-32.

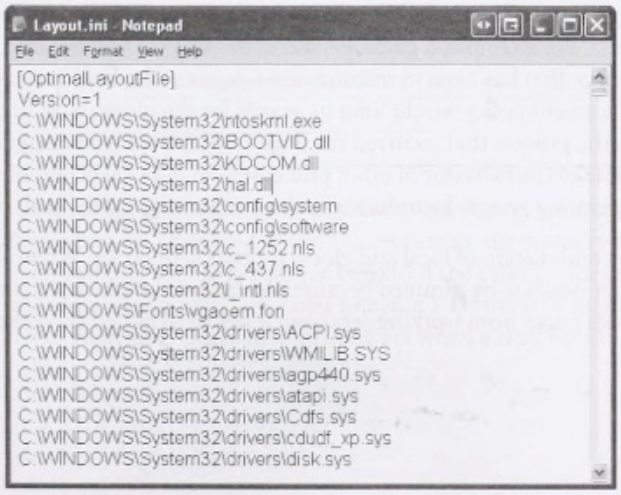


Figure 7-32 Prefetch defragmentation layout file

Then it launches the system defragmenter with a command-line option that tells the defragmenter to defragment based on the contents of the file instead of performing a full defrag. The defragmenter finds a contiguous area on each volume large enough to hold all the listed files and directories that reside on that volume and then moves them in their entirety into the area so that they are stored one after the other. Thus, future prefetch operations will even be more efficient because all the data read in is now stored physically on the disk in the order it will be read. Because the files defragmented for prefetching usually number only in the hundreds, this defragmentation is much faster than full volume defragmentations. (See Chapter 12 for more information on defragmentation.)

Placement Policy

When a thread receives a page fault, the memory manager must also determine where in physical memory to put the virtual page. The set of rules it uses to determine the best position is called a *placement policy*. Windows considers the size of CPU memory caches when choosing page frames to minimize unnecessary thrashing of the cache.

If physical memory is full when a page fault occurs, a *replacement policy* is used to determine which virtual page must be removed from memory to make room for the new page. Common replacement policies include *least recently used* (LRU) and *first in, first out* (FIFO). The LRU algorithm (also known as the *clock algorithm*, as implemented in most versions of UNIX) requires the virtual memory system to track when a page in memory is used. When a new page frame is required, the page that hasn't been used for the greatest amount of time is removed from the working set. The FIFO algorithm is somewhat simpler; it removes the page that has been in physical memory for the greatest amount of time, regardless of how often it's been used.



Replacement policies can be further characterized as either global or local. A global replacement policy allows a page fault to be satisfied by any page frame, whether or not that frame is owned by another process. For example, a global replacement policy using the FIFO algorithm would locate the page that has been in memory the longest and would free it to satisfy a page fault; a local replacement policy would limit its search for the oldest page to the set of pages already owned by the process that incurred the page fault. Global replacement policies make processes vulnerable to the behavior of other processes—an ill-behaved application can undermine the entire operating system by inducing excessive paging activity in all processes.

Windows implements a combination of local and global replacement policy. When a working set reaches its limit and/or needs to be trimmed because of demands for physical memory, the memory manager removes pages from working sets until it has determined there are enough free pages.

Working Set Management

Every process starts with a default working set minimum of 50 pages and a working set maximum of 345 pages. Although it has little effect, you can change the process working set limits with the Windows *SetProcessWorkingSetSize* function, though you must have the "increase scheduling priority" user right to do this. However, unless you have configured the process to use hard working set limits (new in Windows Server 2003), these limits are ignored, in that the memory manager will permit a process to grow beyond its maximum if it is paging heavily and there is ample memory (and conversely, the memory manager will shrink a process below its working set minimum if it is not paging and there is a high demand for physical memory on the system). Although in Windows 2000, the extent to which a process was over its working set maximum was a factor in its priority to be trimmed, as of Windows XP, this decision is based solely on how many pages have been accessed.

In Windows Server 2003, hard working set limits can be set using the *SetProcessWorkingSetSizeEx* function along with the QUOTA_LIMITS_HARDWS_ENABLE flag. An sample consumer of this function is the Windows System Resource Manager (WSRM), described in Chapter 6.

The maximum working set size can't exceed the systemwide maximum calculated at system initialization time and stored in the kernel variable *MmMaximumWorkingSetSize*. This value is set to be the number of available pages (the size of the zero, free, and standby list) at the time the computation is made minus 512 pages. However, there are hard upper limits for working set sizes—these are listed in Table 7-17.

Table 7-17 Upper Limit for Working Set Maximums

Windows Version	Working Set Maximum
x86 versions of Windows 2000, Windows XP, Windows XP SP1, Windows Server 2003	1984 MB
x86 versions of Windows XP SP2, Windows Server 2003 SP1	2047.9 MB
x86 versions of Windows booted /3GB	3008 MB
IA-64	7152 GB
x64	8192 GB

When a page fault occurs, the process's working set limits and the amount of free memory on the system are examined. If conditions permit, the memory manager allows a process to grow to its working set maximum (or beyond if the process does not have a hard working set limit and there are enough free pages available). However, if memory is tight, Windows replaces rather than adds pages in a working set when a fault occurs.

Although Windows attempts to keep memory available by writing modified pages to disk, when modified pages are being generated at a very high rate, more memory is required in order to meet memory demands. Therefore, when physical memory runs low, the *working set manager*, a routine that runs in the context of the balance set manager system thread (described in the next section), initiates automatic working set trimming to increase the amount of free memory available in the system. (With the Windows *SetProcessWorkingSetSize* function mentioned earlier, you can also initiate working set trimming of your own process—for example, after process initialization.)

The working set manager examines available memory and decides which, if any, working sets need to be trimmed. If there is ample memory, the working set manager calculates how many pages could be removed from working sets if needed. If trimming is needed, it looks at working sets that are above their minimum setting. It also dynamically adjusts the rate at which it examines working sets as well as arranges the list of processes that are candidates to be trimmed into an optimal order. For example, processes with many pages that have not been accessed recently are examined first; larger processes that have been idle longer are considered before smaller processes that are running more often; the process running the foreground application is considered last; and so on.

When it finds processes using more than their minimums, the working set manager looks for pages to remove from their working sets, making the pages available for other uses. If the amount of free memory is still too low, the working set manager continues removing pages from processes' working sets until it achieves a minimum number of free pages on the system.

On a single-processor Windows 2000 system and all systems running Windows XP and Windows Server 2003, the working set manager tries to remove pages that haven't been accessed recently. It does this by checking the accessed bit in the hardware PTE to see whether the page has been accessed. If the bit is clear, the page is *aged*, that is, a count is incremented indicating that the page hasn't been referenced since the last working set trim scan. Later, the age of pages is used to locate candidate pages to remove from the working set.

> **Note** On a Windows 2000 multiprocessor system, the working set manager mistakenly didn't check the access bit, resulting in pages being removed from the working set without regard to the state of the accessed bit.

If the hardware PTE accessed bit is set, the working set manager clears it and goes on to examine the next page in the working set. In this way, if the accessed bit is clear the next time the working set manager examines the page, it knows that the page hasn't been accessed since the last time it was examined. This scan for pages to remove continues through the working set list until either the number of desired pages has been removed or the scan has returned to the starting point. (The next time the working set is trimmed, the scan picks up where it left off last.)



EXPERIMENT: Viewing Process Working Set Sizes

You can use the Performance tool to examine process working set sizes by looking at the following performance counters:

Counter	Description
Process: Working Set	Current size of the selected process's working set in bytes
Process: Working Set Peak	Peak size of the selected process's working set in bytes
Process: Page Faults/Sec	Number of page faults for the process that occur each second

Several other process viewer utilities (such as Task Manager, Pview, and Pviewer) also display the process working set size.

You can also get the total of all the process working sets by selecting the _Total process in the instance box in the Performance tool. This process isn't real—it's simply a total of the process-specific counters for all processes currently running on the system. The total you see is misleading, however, because the size of each process working set includes pages being shared by other processes. Thus, if two or more processes share a page, the page is counted in each process's working set.



EXPERIMENT: Viewing the Working Set List

You can view the individual entries in the working set by using the kernel debugger *!wsle* command. The following example shows a partial output of the working set list of LiveKd. (This command was run on the LiveKd process.)

```
kd> !wsle 7

Working Set @ c0502000
    Quota:      9f  FirstFree:  40  FirstDynamic:      3
    LastEntry  1fe  NextSlot:    3  LastInitialized  257
    NonDirect   5c  HashTable:   0  HashTableSize:     0

Virtual Address  Age  Locked  ReferenceCount
c0300203           0       1       1
c0301203           0       1       1
c0502203           0       1       1
c01df201           0       0       1
c01ff201           0       0       1
c0005201           0       0       1
c0001201           0       0       1
c0002201           0       0       1
c0000201           0       0       1
c0006201           0       0       1
77e87119           0       0       1
```

```
00402319        0       0       1
77e01201        0       0       1
7ffdf201        0       0       1
00130201        0       0       1
77e9e119        0       0       1
78033201        0       0       1
00230221        0       0       1
00131201        0       0       1
77d50119        0       0       1
00132201        0       0       1
c01e0201        0       0       1
00411309        0       0       1
0040d201        0       0       1
77edf201        0       0       1
77ee0201        0       0       1
77fcd201        0       0       1
0040e201        0       0       1
7ffc1009        0       0       1
00401319        0       0       1
```

Notice that some entries in the working set list are page table pages (the ones with addresses greater than 0xC0000000), some are from system DLLs (the ones in the 0x7nnnnnnn range), and some are from the code of LiveKd.exe itself (those in the 0x004nnnnn range).

Balance Set Manager and Swapper



Working set expansion and trimming take place in the context of a system thread called the *balance set manager* (routine *KeBalanceSetManager*). The balance set manager is created during system initialization. Although the balance set manager is technically part of the kernel, it calls the memory manager's working set manager to perform working set analysis and adjustment.

The balance set manager waits for two different event objects: an event that is signaled when a periodic timer set to fire once per second expires and an internal working set manager event that the memory manager signals at various points when it determines that working sets need to be adjusted. For example, if the system is experiencing a high page fault rate or the free list is too small, the memory manager wakes up the balance set manager so that it will call the working set manager to begin trimming working sets. When memory is more plentiful, the working set manager will permit faulting processes to gradually increase the size of their working sets by faulting pages back into memory, but the working sets will grow only as needed.

When the balance set manager wakes up as the result of its 1-second timer expiring, it takes the following four steps:

1. Every fourth time the balance set manager wakes up because its 1-second timer has expired, it signals an event that wakes up another system thread called the swapper (routine *KeSwapProcessOrStack*).

2. The balance set manager then checks the look-aside lists and adjusts their depths if necessary (to improve access time and to reduce pool usage and pool fragmentation).
3. It looks for threads that might warrant having their priority boosted because they are CPU starved. (See the section "Priority Boosts for CPU Starvation" in Chapter 6.)
4. It calls the memory manager's working set manager. (The working set manager has its own internal counters that regulate when to perform working set trimming and how aggressively to trim.)

The swapper is also awakened by the scheduling code in the kernel if a thread that needs to run has its kernel stack swapped out or if the process has been swapped out. The swapper looks for threads that have been in a wait state for 7 seconds in Windows 2000 and 15 seconds in Windows XP and Windows Server 2003. If it finds one, it puts the thread's kernel stack in transition (moving the pages to the modified or standby lists) so as to reclaim its physical memory, operating on the principle that if a thread's been waiting that long, it's going to be waiting even longer. When the last thread in a process has its kernel stack removed from memory, the process is marked to be entirely outswapped. That's why, for example, processes that have been idle for a long time (such as Winlogon is after you log on) can have a zero working set size.

System Working Set

Just as processes have working sets, the pageable code and data in the operating system are managed by a single *system working set*. Five different kinds of pages can reside in the system working set:

- System cache pages
- Paged pool
- Pageable code and data in Ntoskrnl.exe
- Pageable code and data in device drivers
- System mapped views

You can examine the size of the system working set or the size of the five components that contribute to it with the performance counters or system variables shown in Table 7-18. Keep in mind that the performance counter values are in bytes whereas the system variables are measured in terms of pages.

You can also examine the paging activity in the system working set by examining the Memory: Cache Faults/Sec performance counter, which describes page faults that occur in the system working set (both hard and soft).

Table 7-18 System Working Set Performance Counters

Performance Counter (in Bytes)	System Variable (in Pages)	Description
Memory: Cache Bytes*	*MmSystemCacheWs.WorkingSetSize*	Total size of system working set (including the cache, paged pool, pageable Ntoskrnl and driver code, and system mapped views); this is *not* the size of the system cache alone, even though the name implies that it is.
Memory: Cache Bytes Peak	*MmSystemCacheWs.Peak*	Peak system working set size.
Memory: System Cache Resident Bytes	*MmSystemCachePage*	Physical memory consumed by the system cache.
Memory: System Code Resident Bytes	*MmSystemCodePage*	Physical memory consumed by pageable code in Ntoskrnl.exe.
Memory: System Driver Resident Bytes	*MmSystemDriverPage*	Physical memory consumed by pageable device driver code.
Memory: Pool Paged Resident Bytes	*MmPagedPoolPage*	Physical memory consumed by paged pool.

* Internally, this working set is called the system *cache* working set, even though the system cache is just one of five different components in it. Thus, several utilities think they are displaying the size of the file cache when they are displaying the total size of the system working set.

The system variable that contains the value for this counter is *MmSystemCacheWs.PageFaultCount*.

The minimum and maximum system working set size is computed at system initialization time based on the amount of physical memory on the machine and whether the system is running a client or server edition of Windows. The calculated working set minimum and maximum are stored in the kernel variables shown in Table 7-19. These variables aren't available through any performance counter, but you can examine them with the kernel debugger.

Table 7-19 System Variables That Store Working Set Minimums or Maximums

Variable	Type	Description
MmSystemCacheWs.MinimumWorkingSetSize	ULONG	Minimum working set size
MmSystemCacheWs.MaximumWorkingSetSize	ULONG	Maximum working set size

You can configure the system to give priority to the system working set (as opposed to process working sets) by changing the value of HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\LargeSystemCache. In Windows 2000 Server systems, you can adjust this value indirectly by setting the properties of the file server service; in Windows XP and Windows Server 2003, you can adjust this by going to My Computer, choosing Properties, clicking the Advanced tab, pressing the Settings button in the Performance section, and finally clicking the Advanced tab. (See Chapter 11 for details.)

Page Frame Number Database

Whereas working sets describe the resident pages owned by a process or the system, the *page frame number (PFN) database* describes the state of each page in physical memory. The page states are listed in Table 7-20.

Table 7-20 Page States

Status	Description
Active (also called Valid)	The page is part of a working set (either a process working set or the system working set) or it's not in any working set (for example, nonpaged kernel page), and a valid PTE points to it.
Transition	A temporary state for a page that isn't owned by a working set and isn't on any paging list. A page is in this state when an I/O to the page is in progress. The PTE is encoded so that collided page faults can be recognized and handled properly. (Note that this use of the term "transition" differs from the use of the word in the section on Invalid PTEs; an invalid transition PTE refers to a page on the standby or modified list.)
Standby	The page previously belonged to a working set but was removed. The page wasn't modified since it was last written to disk. The PTE still refers to the physical page but is marked invalid and in transition.
Modified	The page previously belonged to a working set but was removed. However, the page was modified while it was in use and its current contents haven't yet been written to disk. The PTE still refers to the physical page but is marked invalid and in transition. It must be written to disk before the physical page can be reused.
Modified no-write	Same as a modified page, except that it has been marked so that the memory manager's modified page writer won't write it to disk. The cache manager marks pages as modified no-write at the request of file system drivers. For example, NTFS uses this state for pages containing file system metadata so that it can first ensure that transaction log entries are flushed to disk before the pages they are protecting are written to disk. (NTFS transaction logging is explained in Chapter 12.)
Free	The page is free but has unspecified dirty data in it. (These pages can't be given as a user page to a user process without being initialized with zeros, for security reasons.)
Zeroed	The page is free and has been initialized with zeros by the zero page thread.
Rom	The page has been faulted in from read-only memory (new as of Windows XP).
Bad	The page has generated parity or other hardware errors and can't be used.

The PFN database consists of an array of structures that represent each physical page of memory on the system. The PFN database and its relationship to page tables are shown in

Figure 7-33. As this figure shows, valid PTEs point to entries in the PFN database, and the PFN database entries (for nonprototype PFNs) point back to the page table that is using them. For prototype PFNs, they point back to the prototype PTE.

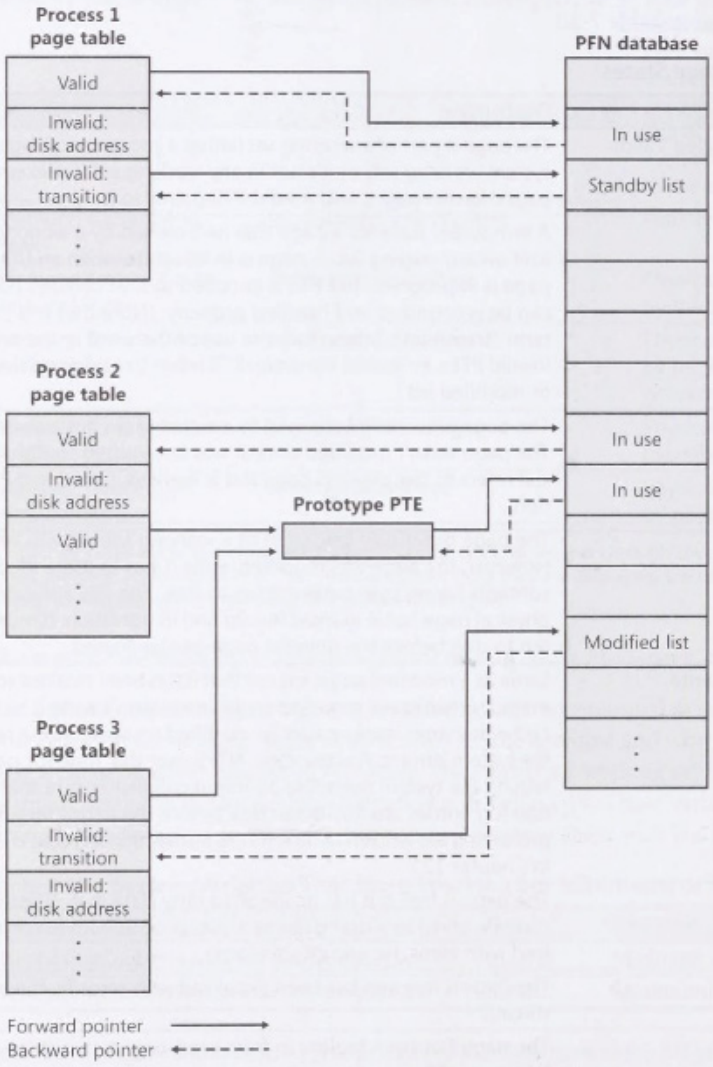


Figure 7-33 Page tables and the page frame number database

Of the page states listed in Table 7-20, six are organized into linked lists so that the memory manager can quickly locate pages of a specific type. (Active/valid pages and transition pages aren't in any systemwide page list.) Figure 7-34 shows an example of how these entries are linked together.

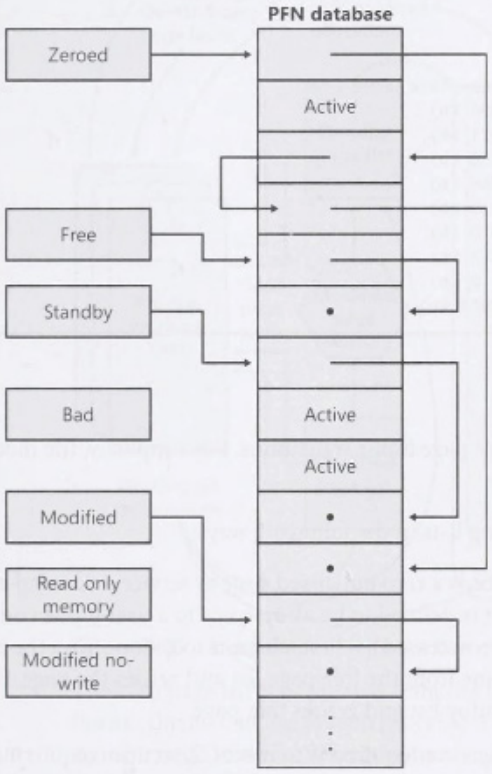


Figure 7-34 Page lists in the PFN database

In the next section, you'll find out how these linked lists are used to satisfy page faults and how pages move to and from the various lists.



EXPERIMENT: Viewing the PFN Database

Using the kernel debugger *!memusage* command, you can dump the size of the various paging lists. The following is the output from this command:

```
lkd> !memusage
 loading PFN database
loading (100% complete)
Compiling memory usage data (99% Complete).
              Zeroed:   8474 ( 33896 kb)
                Free:    256 (  1024 kb)
             Standby:  50790 (203160 kb)
            Modified:    496 (  1984 kb)
       ModifiedNoWrite:    0 (     0 kb)
          Active/Valid: 201980 (807920 kb)
          Transition:      1 (     4 kb)
             Unknown:      0 (     0 kb)
               TOTAL: 261997 (1047988 kb)
```

Page List Dynamics

Figure 7-35 shows a state diagram for page frame transitions. For simplicity, the modified-no-write list isn't shown.

Page frames move between the paging lists in the following ways:

- When the memory manager needs a zero-initialized page to service a demand-zero page fault (a reference to a page that is defined to be all zeros or to a user-mode committed private page that has never been accessed), it first attempts to get one from the zero page list. If the list is empty, it gets one from the free page list and zeroes the page. If the free list is empty, it goes to the standby list and zeroes that page.

  One reason zero-initialized pages are required is to meet C2 security requirements. C2 specifies that user-mode processes must be given initialized page frames to prevent them from reading a previous process's memory contents. Therefore, the memory manager gives user-mode processes zeroed page frames unless the page is being read in from a mapped file. If that's the case, the memory manager prefers to use nonzeroed page frames, initializing them with the data off the disk.

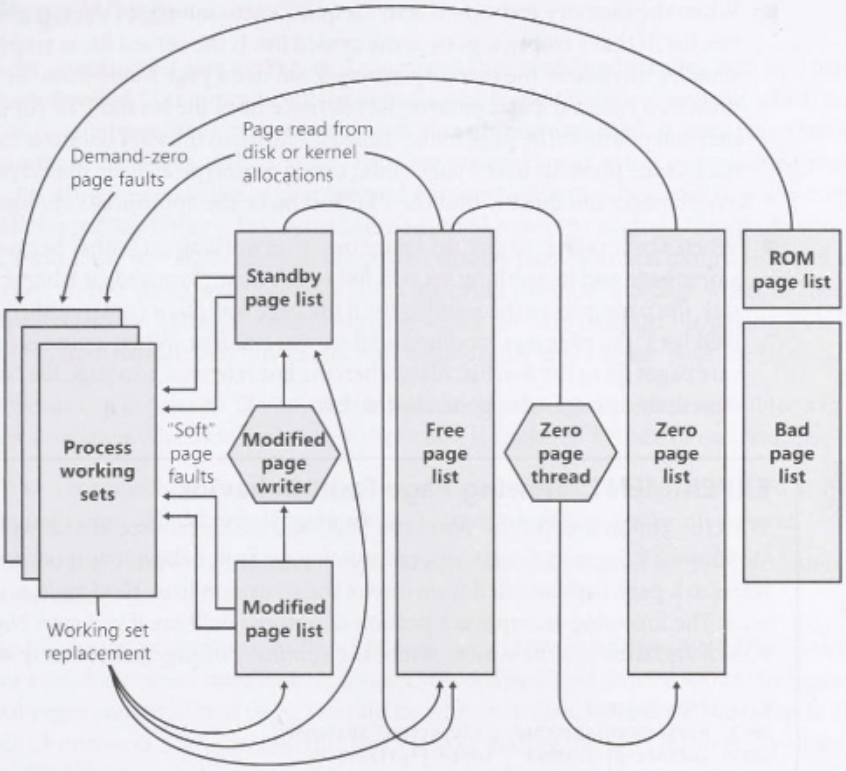


Figure 7-35 State diagram for page frames

The zero page list is populated from the free list by a system thread called the *zero page thread* (thread 0 in the System process). The zero page thread waits on an event object to signal it to go to work. When the free list has eight or more pages, this event is signaled. However, the zero page thread will run only if no other threads are running, because the zero page thread runs at priority 0 and the lowest priority that a user thread can be set to is 1.



Note In Windows Server 2003 and later, when memory needs to be zeroed as a result of a physical page allocation by a driver that calls *MmAllocatePagesForMdl* or a Windows application that calls *AllocateUserPhysicalPages*, or when an application allocates large pages, the memory manager zeroes the memory using a higher performing function called *MiZeroInParallel*, which maps larger regions than the zero page thread, which only zeroes a page at a time. In addition, on multiprocessor systems, it creates additional system threads to perform the zeroing in parallel (and in a NUMA-optimized fashion on NUMA platforms).

- When the memory manager doesn't require a zero-initialized page, it goes first to the free list. If that's empty, it goes to the zeroed list. If the zeroed list is empty, it goes to the standby list. Before the memory manager can use a page frame from the standby list, it must first backtrack and remove the reference from the invalid PTE (or prototype PTE) that still points to the page frame. Because entries in the PFN database contain pointers back to the previous user's page table (or to a prototype PTE for shared pages), the memory manager can quickly find the PTE and make the appropriate change.
- When a process has to give up a page out of its working set (either because it referenced a new page and its working set was full or the memory manager trimmed its working set), the page goes to the standby list if the page was clean (not modified) or to the modified list if the page was modified while it was resident. When a process exits, all the private pages go to the free list. Also, when the last reference to a page file backed section is closed, these pages also go to the free list.



EXPERIMENT: Viewing Page Fault Behavior

With the Pfmon tool (in the Windows 2000 and 2003 resource kits, as well as in the Windows XP Support Tools), you can watch page fault behavior as it occurs. A *soft fault* refers to a page fault satisfied from one of the transition lists. *Hard faults* refer to a disk-read. The following example is a portion of output you'll see if you start Notepad with Pfmon and then exit. Be sure to notice the summary of page fault activity at the end.



```
C:\> pfmon notepad
SOFT: KiUserApcDispatcher : KiUserApcDispatcher
SOFT: LdrInitializeThunk : LdrInitializeThunk
SOFT: 0x77f61016 :  : 0x77f61016
SOFT: 0x77f6105b :  : fltused+0xe00
HARD: 0x77f6105b :  : fltused+0xe00
SOFT: LdrQueryImageFileExecutionOptions :
                        LdrQueryImageFileExecutionOptions
SOFT: RtlAppendUnicodeToString : RtlAppendUnicodeToString
SOFT: RtlInitUnicodeString : RtlInitUnicodeString

  notepad Caused   8 faults had   9 Soft  5 Hard faulted VA's
    ntdll Caused  94 faults had  42 Soft  8 Hard faulted VA's
 comdlg32 Caused   3 faults had   0 Soft  3 Hard faulted VA's
  shlwapi Caused   2 faults had   2 Soft  2 Hard faulted VA's
    gdi32 Caused  18 faults had  10 Soft  2 Hard faulted VA's
 kernel32 Caused  48 faults had  36 Soft  3 Hard faulted VA's
   user32 Caused  38 faults had  26 Soft  6 Hard faulted VA's
 advapi32 Caused   7 faults had   6 Soft  3 Hard faulted VA's
   rpcrt4 Caused   6 faults had   4 Soft  2 Hard faulted VA's
 comctl32 Caused   6 faults had   5 Soft  2 Hard faulted VA's
  shell32 Caused   6 faults had   5 Soft  2 Hard faulted VA's
          Caused  10 faults had   9 Soft  5 Hard faulted VA's
winspool Caused   4 faults had   2 Soft  2 Hard faulted VA's

PFMON: Total Faults 250
      (KM 74 UM 250 Soft 204, Hard 46, Code 121, Data 129)
```

Modified Page Writer

When the modified list gets too big, or if the size of the zeroed and standby lists falls below a minimum threshold (as indicated by the kernel variable *MmMinimumFreePages*, which is computed at system boot time), one of two system threads are awakened to write pages back to disk and move the pages to the standby list. One system thread writes out modified pages (*MiModifiedPageWriter*) to the paging file, and a second one writes modified pages to mapped files (*MiMappedPageWriter*). Two threads are required to avoid creating a deadlock, which would occur if the writing of mapped file pages caused a page fault that in turn required a free page when no free pages were available (thus requiring the modified page writer to create more free pages). By having the modified page writer perform mapped file paging I/Os from a second system thread, that thread can wait without blocking regular page file I/O.

Both threads run at priority 17 and, after initialization, wait for separate event objects to trigger their operation. The modified page writer event is triggered for one of two reasons:

- When the number of modified pages exceeds the maximum value computed at system initialization (*MmModifiedPageMaximum*)—currently 800 pages for all systems
- When the number of available pages (*MmAvailablePages*) goes below *MmMinimumFreePages*

The modified page writer waits for an additional event (*MiMappedPagesTooOldEvent*) that is set after a predetermined number of seconds (*MmModifiedPageLifeInSeconds*) to indicate that mapped pages (not modified pages) should be written to disk. By default, this value is 300 seconds (5 minutes). (You can override this value by adding the DWORD registry value HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\ ModifiedPageLife). The reason for this additional event is to reduce data loss in the case of a system crash or power failure by eventually writing out modified mapped pages even if the modified list hasn't reached its threshold of 800 pages.

When invoked, the mapped page writer attempts to write as many pages as possible to disk with a single I/O request. It accomplishes this by examining the original PTE field of the PFN database elements for pages on the modified page list to locate pages in contiguous locations on the disk. Once a list is created, the pages are removed from the modified list, an I/O request is issued, and at successful completion of the I/O request, the pages are placed at the tail of the standby list.

Pages that are in the process of being written can be referenced by another thread. When this happens, the reference count and the share count in the PFN entry that represents the physical page are incremented to indicate that another process is using the page. When the I/O operation completes, the modified page writer notices that the share count is no longer 0 and doesn't place the page on the standby list.

PFN Data Structures

Although PFN database entries are of fixed length, they can be in several different states, depending on the state of the page. Thus, individual fields have different meanings depending on the state. The states of a PFN entry are shown in Figure 7-36.

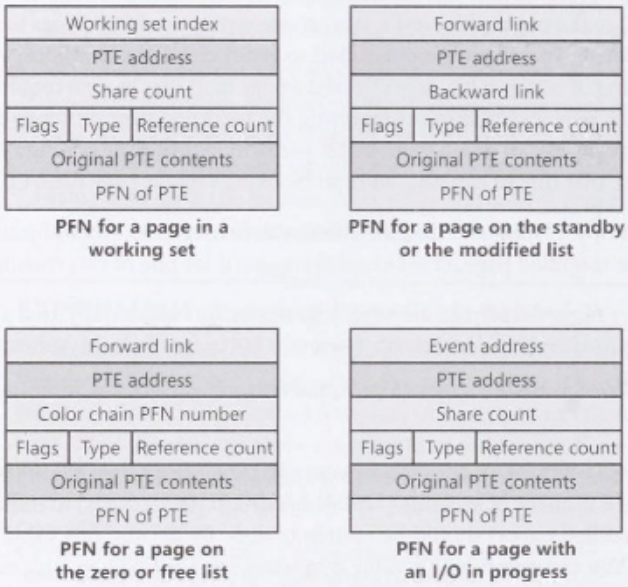


Figure 7-36 States of PFN database entries

Several fields are the same for several PFN types, but others are specific to a given type of PFN. The following fields appear in more than one PFN type:

- **PTE address** Virtual address of the PTE that points to this page.
- **Reference count** The number of references to this page. The reference count is incremented when a page is first added to a working set and/or when the page is locked in memory for I/O (for example, by a device driver). The reference count is decremented when the share count becomes 0 or when pages are unlocked from memory. When the share count becomes 0, the page is no longer owned by a working set. Then, if the reference count is also zero, the PFN database entry that describes the page is updated to add the page to the free, standby, or modified list.
- **Type** The type of page represented by this PFN. (Types include active/valid, transition, standby, modified, modified-no-write, free, zeroed, bad, and transition.)
- **Flags** The information contained in the flags field is shown in Table 7-21.

- **Original PTE contents** All PFN database entries contain the original contents of the PTE that pointed to the page (which could be a prototype PTE). Saving the contents of the PTE allows it to be restored when the physical page is no longer resident.
- **PFN of PTE** Physical page number of the page table page containing the PTE that points to this page.

Table 7-21 Flags Within PFN Database Entries

Flag	Meaning
Modified state	Indicates whether the page was modified. (If the page is modified, its contents must be saved to disk before removing it from memory.)
Prototype PTE	Indicates that the PTE referenced by the PFN entry is a prototype PTE. (For example, this page is shareable.)
Parity error	Indicates that the physical page contains parity or error correction control errors.
Read in progress	Indicates that an in-page operation is in progress for the page. The first DWORD contains the address of the event object that will be signaled when the I/O is complete; also used to indicate the first PFN for nonpaged pool allocations.
Write in progress	Indicates that a page write operation is in progress. The first DWORD contains the address of the event object that will be signaled when the I/O is complete; also used to indicate the last PFN for nonpaged pool allocations.
Start of nonpaged pool	For nonpaged pool pages, indicates that this is the first PFN for a given nonpaged pool allocation.
End of nonpaged pool	For nonpaged pool pages, indicates that this is the last PFN for a given nonpaged pool allocation.
In-page error	Indicates that an I/O error occurred during the in-page operation on this page. (In this case, the first field in the PFN contains the error code.)

The remaining fields are specific to the type of PFN. For example, the first PFN in Figure 7-36 represents a page that is active and part of a working set. The share count field represents the number of PTEs that refer to this page. (Pages marked read-only, copy-on-write, or shared read/write can be shared by multiple processes.) For page table pages, this field is the number of valid PTEs in the page table. As long as the share count is greater than 0, the page isn't eligible for removal from memory.

The working set index field is an index into the process working set list (or the system or session working set list, or zero if not in any working set) where the virtual address that maps this physical page resides. If the page is a private page, the working set index field refers directly to the entry in the working set list because the page is mapped only at a single virtual address. In the case of a shared page, the working set index is a hint that is guaranteed to be correct only for the first process that made the page valid. (Other processes will try to use the same index

where possible.) The process that initially sets this field is guaranteed to refer to the proper index and doesn't need to add a working set list hash entry referenced by the virtual address into its working set hash tree. This guarantee reduces the size of the working set hash tree and makes searches faster for these particular direct entries.

The second PFN in Figure 7-36 is for a page on either the standby or the modified list. In this case, the forward and backward link fields link the elements of the list together within the list. This linking allows pages to be easily manipulated to satisfy page faults. When a page is on one of the lists, the share count is by definition 0 (because no working set is using the page) and therefore can be overlaid with the backward link. The reference count is also 0 if the page is on one of the lists. If it is nonzero (because an I/O could be in progress for this page—for example, when the page is being written to disk), it is first removed from the list.

The third PFN in Figure 7-36 is for a page on the free or zeroed list. Besides being linked together within the two lists, these PFN database entries use an additional field to link physical pages by "color," their location in the processor CPU memory cache. Windows attempts to minimize unnecessary thrashing of CPU memory caches by using different physical pages in the CPU cache. It achieves this optimization by avoiding using the same cache entry for two different pages wherever possible. For systems with direct mapped caches, optimally using the hardware's capabilities can result in a significant performance advantage.

The fourth PFN in Figure 7-36 is for a page that has an I/O in progress (for example, a page read). While the I/O is in progress, the first field points to an event object that will be signaled when the I/O completes. If an in-page error occurs, this field contains the Windows error status code representing the I/O error. This PFN type is used to resolve collided page faults.



EXPERIMENT: Viewing PFN Entries

You can examine individual PFN entries with the kernel debugger *!pfn* command. You first need to supply the PFN as an argument. (For example, *!pfn 0* shows the first entry, *!pfn 1* shows the second, and so on.) In the following example, the PTE for virtual address 0x50000 is displayed, followed by the PFN that contains the page directory, and then the actual page:

```
kd> !pte 50000
00050000  - PDE at C0300000          PTE at C0000140
          contains 00700067        contains 00DAA047
        pfn 00700 --DA--UWV      pfn 00DAA --D---UWV
kd> !pfn 700
  PFN 00000700 at address 827CD800
  flink  00000004 blink / share count 00000010 pteaddress C0300000
  reference count 0001                                  color 0
  restore pte 00000080  containing page        00030  Active     M
  Modified

kd> !pfn daa
  PFN 00000DAA at address 827D77F0
```

```
flink  00000077 blink / share count 00000001 pteaddress c0000140
reference count 0001                                  color 0
restore pte 00000080  containing page        00700  Active     M
Modified
```

In addition to the PFN database, the system variables in Table 7-22 describe the overall state of physical memory.

Table 7-22 System Variables That Describe Physical Memory

Variable	Description
MmNumberOfPhysicalPages	Total number of physical pages available on the system
MmAvailablePages	Total number of available pages on the system—the sum of the pages on the zeroed, free, and standby lists
MmResidentAvailablePages	Total number of physical pages that would be available if every process were at its minimum working set size

Low and High Memory Notification

Windows XP and Windows Server 2003 provide a way for user mode processes to be notified when physical memory is low and/or plentiful. This information can be used to determine memory usage as appropriate. For example, if available memory is low, the application can reduce memory consumption. If available memory is high, the application can allocate more memory.

To be notified of low or high memory conditions, call the *CreateMemoryResourceNotification* function, specifying whether low or high memory notification is desired. A handle can be provided to any of the wait functions. When memory is low (or high), the wait completes, thus notifying the thread of the condition. Alternatively, the *QueryMemoryResourceNotification* can be used to query the system memory condition at any time.

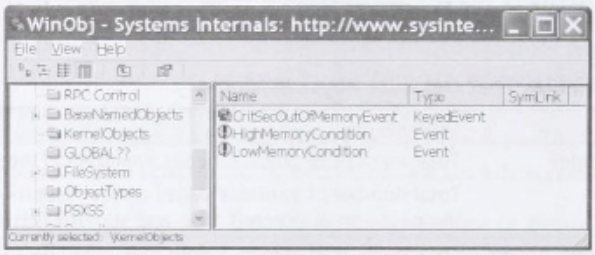
Notification is implemented by the memory manager signaling a globally named event object *LowMemoryCondition* or *HighMemoryCondition*. These named objects are not in the normal \BaseNamedObjects object manager directory, but in a special directory called \KernelObjects. When low (or high) memory condition is detected, the appropriate event is signaled, thus waking up any waiting threads.

The default level of available memory that signals a low-memory-resource notification event is approximately 32 MB per 4 GB, to a maximum of 64 MB. The default level that signals a high-memory-resource notification event is three times the default low-memory value. These values can be overridden by adding a DWORD registry value *LowMemoryThreshold* or *HighMemoryThreshold* under HKEY_LOCAL_MACHINE\System\CurrentControlSet\Session Manager\Memory Management that specifies the number of megabytes to use as the low or high threshold.



EXPERIMENT: Viewing the Memory Resource Notification Events

To see the memory resource notification events, run Winobj from *www.sysinternals.com* and click on the KernelObjects folder. You will see both the low and high memory condition events shown on the right pane:



If you double-click either event, you can see how many handles and/or references have been made to the objects.

To see whether any processes in the system have requested memory resource notification, search the handle table for references to "LowMemoryCondition" or "HighMemoryCondition". You can do this by using Process Explorer's Find menu and choosing the Handle capability or by using the Oh.exe tool from the Resource Kits. (For a description of the handle table, see the section on the "Object Manager" in Chapter 3.)

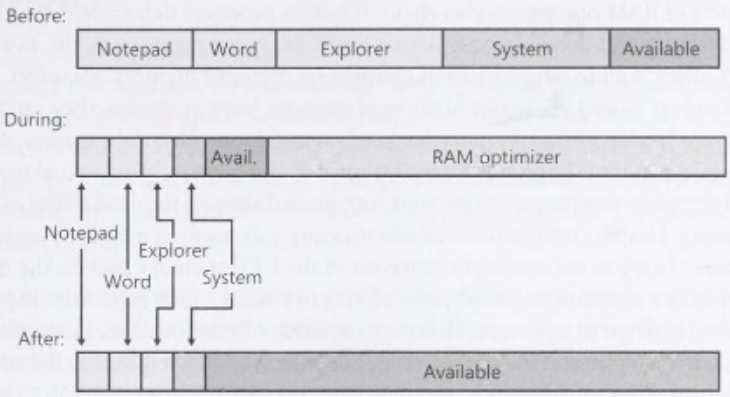


RAM Optimizers: Fact or Fiction?

As you've surfed the Web, you've probably seen browser pop-ups such as "Defragment your memory and improve performance" and "Minimize application and system failures and free unused memory." The links lead you to utilities that promise to do all that and more. Do they really work?

Memory optimizers typically present a UI that shows a graph labeled "available memory" and a line representing a threshold below which the product will take action. Another line typically shows the amount of memory that the optimizer will try to free when it runs. You can usually configure one or both levels, as well as trigger manual memory optimization or schedule optimizations. Some tools also display the processes running on the system. When a scheduled optimization job runs, the utility's available memory counter often goes up, sometimes dramatically, which implies that the tool is actually freeing up memory for your applications to use. But what it's really doing is causing useful memory to be zeroed to artificially increase the available memory.

RAM optimizers work by allocating and then freeing large amounts of virtual memory. The figure below shows the effect a RAM optimizer has on a system:



The "before" bar depicts the working sets and available memory before optimization. The "during" bar shows that the RAM optimizer creates a high memory demand, which it does by incurring many page faults in a short time. In response, the memory manager increases the RAM optimizer's working set. This working-set expansion occurs at the expense of available memory and—when available memory becomes low—at the expense of other process working sets. The "after" bar illustrates how, after the RAM optimizer frees its memory, the memory manager moves all the pages that were assigned to the RAM optimizer to the free page list (which ultimately get zeroed by the zero page thread and moved to the zero page list), thus contributing to the available memory value. Most optimizers hide the rapid decline in available memory that occurs during the first step, but if you run Task Manager during an optimization, you can often see the decline as it takes place.

While gaining more available memory might seem like a good thing, it isn't. As RAM optimizers force the available memory counter up, they force other processes' data and code out of memory. If you're running Word, for example, the text of open documents and the program code that was part of Word's working set before the optimization (and was therefore present in physical memory) must be reread from disk as you continue to edit your document. The performance degradation can be severe on servers, on which file data that was cached in the standby list and in the system working set (as well as the code and data used by any running server applications) would be discarded.

Some vendors make additional claims for their RAM-optimizer products. One claim you might see is that a product frees memory that's needlessly consumed by unused processes, such as those that run in the Taskbar tray. That claim could be true only if those processes had sizable working sets at the time of optimization. However, because Windows automatically trims idle processes' working sets, all such claims are untrue. The memory manager handles all necessary memory optimization.

Developers of RAM optimizers also claim that their products defragment memory. The act of allocating and then freeing a large amount of virtual memory might, as a conceivable side-effect, lead to large blocks of contiguous available memory. However, because virtual memory masks the layout of physical memory from processes, they can't directly benefit from having virtual memory backed by contiguous physical memory. As processes execute and undergo working-set trimming and growth, their virtual-memory-to-physical-memory mappings will become fragmented despite the availability of contiguous memory. Having contiguous available memory can result in increased performance in one case: To try to maximize the behavior of the CPU memory caches, the memory manager uses a mechanism called *page coloring* to decide which page from the free or zeroed list to assign to a process. However, any minor benefit that might result from making available physical memory contiguous is heavily outweighed by the negative effect of discarding valuable code and data from memory.

Finally, vendors often claim that RAM optimizers regain memory lost to leaks. This is perhaps the most patently false assertion of all. The memory manager knows at all times what physical and virtual memory belongs to a process. However, if a process allocates memory and then doesn't free it because of a bug (an occurrence known as a leak), the Memory Manager can't recognize that the allocated memory won't be accessed again at some point and must wait until the process exits to reclaim the memory. Even for leaking processes that do not exit, as a result of working-set trimming, the memory manager eventually will steal from the process's working set any physical pages that are assigned to leaked virtual memory. That process sends the leaked pages to the paging file and lets physical memory be used for other purposes. Thus, a memory leak has only a limited effect on available physical memory. The real effect is on virtual memory consumption, which Task Manager calls both PF Usage and Commit Charge. No utility can do anything about virtual memory consumption other than kill processes that are consuming memory.

In summary, common sense suggests that if RAM optimization were possible (and could be implemented by so many small-time upstarts), Microsoft developers would have long since integrated the technology into the kernel.

Conclusion

In this chapter, we've examined how the Windows memory manager implements virtual memory management. As with most modern operating systems, each process is given access to a private address space, protecting one process's memory from another's but allowing processes to share memory efficiently and securely. Advanced capabilities, such as the inclusion of mapped files and the ability to sparsely allocate memory, are also available. The Windows environment subsystem makes most of the memory manager's capabilities available to applications through the Windows API.

The memory manager's implementation relies on lazy-evaluation techniques whenever possible to avoid performing time-consuming and unnecessary operations unless they are required. It is also self-tuning, adapting to both large multiprocessor servers as well as uniprocessor desktop workstations.

One aspect of the memory manager that we didn't describe in this chapter is its tight integration with the cache manager, which we'll cover in Chapter 11. But before we get to that, let's take a closer look at the Windows security mechanisms.



MICROSOFT®

WINDOWS INTERNALS

MICROSOFT WINDOWS SERVER™ 2003, WINDOWS XP, AND WINDOWS 2000

Fourth Edition



The definitive guide—fully updated

Get the architectural perspectives and inside details you need to understand the Microsoft Windows kernel!

Delve inside the Windows kernel with noted internals experts Mark Russinovich and David Solomon, in collaboration with the Microsoft Windows product development team. This classic guide—fully updated for Windows Server 2003, Windows XP, and Windows 2000, including 64-bit extensions—describes the architecture and internals of the Windows operating system. You'll find hands-on experiments you can use to experience Windows internal behavior firsthand, along with advanced troubleshooting information to help keep your systems running smoothly and efficiently. Whether you're a developer or a system administrator, you'll find critical architectural insights that you can quickly apply for better design, debugging, performance, and support.

Get in-depth, inside knowledge of the Windows operating system:

- Understand the key mechanisms that configure and control Windows, including dispatching, startup and shutdown, and the registry
- Explore the Windows security model, including access, privileges, and auditing
- Investigate internal system architecture using the kernel debugger and other tools
- Examine the data structures and algorithms that deal with processes, threads, and jobs
- Observe how Windows manages virtual and physical memory
- Understand the operation and format of NTFS, and troubleshoot file system access problems
- View the Windows networking stack from top to bottom, including mapping, APIs, name resolution, and protocol drivers
- Troubleshoot boot problems and perform crash analysis

About the Authors:

Mark E. Russinovich is chief software architect and cofounder of Winternals Software, a company that specializes in advanced systems software for Windows. He authored the Sysinternals tools featured in this book. Mark has a Ph.D. in computer engineering from Carnegie Mellon University. He is also a senior contributing editor for *Windows IT Pro* magazine.

David A. Solomon is the president of David Solomon Expert Seminars and teaches classes on Windows internals to corporations worldwide, including Microsoft. He is also a regular speaker at Microsoft technical conferences. David was previously a lead developer on the VMS operating system at Digital Equipment Corporation.

Mark and David coauthored the previous edition of this book, *Inside Microsoft Windows 2000* from Microsoft Press.

To see more developer learning resources, visit: **microsoft.com/mspress**

Part No. X10-96110

ISBN 0-7356-1917-4

U.S.A. $59.99
Canada $86.99
[Recommended]

Programming/ Microsoft Windows

Microsoft