

Unit OS5: Memory Management

5.2. Windows Memory Management Fundamentals

Windows Operating System Internals - by David A. Solomon and Mark E. Russinovich with Andreas Polze

Roadmap for Section 5.2.

- Memory Manager Features and Components
- Virtual Address Space Allocation
- Shared Memory and Memory-Mapped Files
- Physical Memory Limits
- Memory management APIs

3

Windows Memory Management Fundamentals

- Classical virtual memory management
 - Flat virtual address space per process
 - Private process address space
 - Global system address space
 - Per session address space
- Object based
 - Section object and object-based security (ACLs...)
- Demand paged virtual memory
 - Pages are read in on demand & written out when necessary (to make room for other memory needs)
- Provides flat virtual address space
 - 32-bit: 4 GB, 64-bit: 16 Exabytes (theoretical)

4

Windows Memory Management Fundamentals

- Lazy evaluation
 - Sharing – usage of prototype PTEs (page table entries)
 - Extensive usage of copy_on_write
 - ...whenever possible
- Shared memory with copy on write
- Mapped files (fundamental primitive)
 - Provides basic support for file system cache manager

5

Memory Manager Components

- System services for allocating, deallocating, and managing virtual memory
- A access fault trap handler for resolving hardware-detected memory management exceptions and making virtual pages resident on behalf of a process
- Six system threads
 - *Working set manager* (priority 16) – drives overall memory management policies, such as working set trimming, aging, and modified page writing
 - *Process/stack swapper* (priority 23) -- performs both process and kernel thread stack inswapping and outswapping
 - *Modified page writer* (priority 17) – writes dirty pages on the modified list back to the appropriate paging files
 - *Mapped page writer* (priority 17) – writes dirty pages from mapped files to disk
 - *Dereference segment thread* (priority 18) is responsible for cache and page file growth and shrinkage
 - *Zero page thread* (priority 0) – zeros out pages on the free list

6

MM: Process Support

- MmCreateProcessAddressSpace – 3 pages
 - The page directory
 - Points to itself
 - Map the page table of the hyperspace
 - Map system paged and nonpaged areas
 - Map system cache page table pages
 - The page table page for working set
 - The page for the working set list
- MmInitializeProcessAddressSpace
 - Initialize PFN for PD and hyperspace PDEs
 - MlInitializeWorkingSetList
 - Optional: MmMapViewOfSection for image file
- MmCleanProcessAddressSpace,
- MmDeleteProcess AddressSpace

7

MM: Process Swap Support

- MmOutSwapProcess / MmInSwapProcess
- MmCreateKernelStack
 - MiReserveSystemPtes for stack and no-access page
- MmDeleteKernelStack
 - MiReleaseSystemPtes
- MmGrowKernelStack
- MmOutPageKernelStack
 - Signature (thread_id) written on top of stack before write
 - The page goes to transition list
- MmInPageKernelStack
 - Check signature after stack page is read / bugcheck

8

MM: Working Sets

- Working Set:
 - The set of pages in memory at any time for a given process, or
 - All the pages the process can reference without incurring a page fault
 - Per process, private address space
 - WS limit: maximum amount of pages a process can own
 - Implemented as array of working set list entries (WSLE)
- Soft vs. Hard Page Faults:
 - Soft page faults resolved from memory (standby/modified page lists)
 - Hard page faults require disk access
- Working Set Dynamics:
 - Page replacement when WS limit is reached
 - NT 4.0: page replacement based on modified FIFO
 - Windows 2000: Least Recently Used algorithm (uniproc.)

9

MM: Working Set Management

- Modified Page Writer thread
 - Created at system initialization
 - Writing modified pages to backing file
 - Optimization: min. I/Os, contiguous pages on disk
 - Generally MPW is invoked before trimming
- Balance Set Manager thread
 - Created at system initialization
 - Wakes up every second
 - Executes MmWorkingSetManager
 - Trimming process WS when required: from current down to minimal WS for processes with lowest page fault rate
 - Aware of the system cache working set
 - Process can be out-swapped if all threads have pageable kernel stack

10

MM: I/O Support

- I/O Support operations:
 - Locking/Unlocking pages in memory
 - Mapping/Unmapping Locked Pages into current address space
 - Mapping/Unmapping I/O space
 - Get physical address of a locked page
 - Probe page for access
- Memory Descriptor List
 - Starting VAD
 - Size in Bytes
 - Array of elements to be filled with physical page numbers
- Physically contiguous vs. Virtually contiguous

11

MM: Cache Support

- System wide cache memory
 - Region of system paged area reserved at initialization time
 - Initial default: 512 MB (min. 64MB if /3GB, max 960 MB)
 - Managed as system wide working set
 - A valid cache page is valid in all address spaces
 - Lock the page in the cache to prevent WS removal
 - WS Manager trimming thread is aware of this special WS
 - Not accessible from user mode
 - Only views of mapped files may reside in the cache
- File Systems and Server interaction support
 - Map/Unmap view of section in system cache
 - Lock/Unlock pages in system cache
 - Read section file in system cache
 - Purge section

12

Memory Manager: Services

- Caller can manipulate own/remote memory
 - Parent process can allocate/deallocate, read/write memory of child process
 - Subsystems manage memory of their client processes this way
- Most services are exposed through Windows API
 - Page granularity virtual memory functions (Virtualxxx...)
 - Memory-mapped file functions (CreateFileMapping, MapViewOfFile)
 - Heap functions (Heapxxx, Localxxx (old), Globalxxx (old))
- Services for device drivers/kernel code (Mm...)

13

Protecting Memory

Attribute	Description
PAGE_NOACCESS	Read/write/execute causes access violation
PAGE_READONLY	Write/execute causes access violation; read permitted
PAGE_READWRITE	Read/write accesses permitted
PAGE_EXECUTE	Any read/write causes access violation; execution of code is permitted (relies on special processor support)
PAGE_EXECUTE_READ	Read/execute access permitted (relies on special processor support)
PAGE_EXECUTE_READWRITE	All accesses permitted (relies on special processor support)
PAGE_WRITECOPY	Write access causes the system to give process a private copy of this page; attempts to execute code cause access violation
PAGE_EXECUTE_WRITECOPY	Write access causes creation of private copy of pg.
PAGE_GUARD	Any read/write attempt raises EXCEPTION_GUARD_PAGE and turns off guard page status

14

Reserving & Committing Memory

- Optional 2-phase approach to memory allocation:
 1. Reserve address space (in multiples of page size)
 2. Commit storage in that address space
 - Can be combined in one call (VirtualAlloc, VirtualAllocEx)
- Reserved memory:
 - Range of virtual addresses reserved for future use (contiguous buffer)
 - Accessing reserved memory results in access violation
 - Fast, inexpensive
- Committed memory:
 - Has backing store (pagefile.sys, memory-mapped file)
 - Either private or mapped into a view of a section
 - Decommit via VirtualFree, VirtualFreeEx

A thread's user-mode stack is constructed using this 2-phase approach: initial reserved size is 1MB, only 2 pages are committed: stack & guard page

15

Features new to Windows 2000 Memory Management

- Support of 64 GB physical memory on Intel platform
 - PAE – physical address extension (36 bit, changes PDE/PTE structs)
 - New version of kernel (ntkrnlpa.exe, ntkrpamp.exe)
 - /PAE switch in boot.ini
- Integrated support for Terminal Server
 - HydraSpace : per session
 - In NT 4 Terminal Server had a specific kernel
- Driver Verifier: verifier.exe
 - Pool checking, IRQL checking
 - Low resources simulation, pool tracking, I/O verification

16

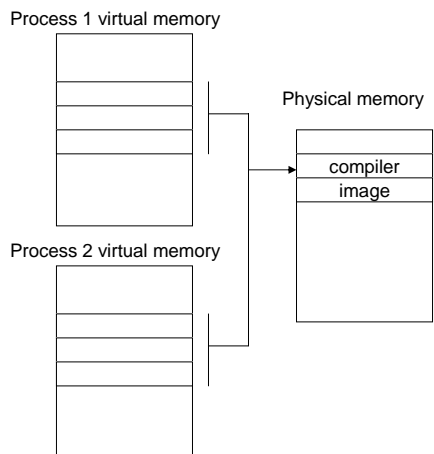
Features new to Windows XP/2003 Memory Management

- 64-bit support
- Up to 1024 Gbytes physical memory supported
- Support for Data Execution Prevention (DEP)
 - Memory manager supports HW no-execute protection
- Performance & Scalability enhancements

17

Shared Memory & Mapped Files

- Shared memory + copy-on-write per default
- Executables are mapped as read-only
- Memory manager uses section objects to implement shared memory (file mapping objects in Windows API)



18

Virtual Address Space Allocation

- Virtual address space is sparse
 - Address spaces contain reserved, committed, and unused regions
- Unit of protection and usage is one page
 - On x86, default page size is 4 KB (x86 supports 4KB or 4MB)
 - In PAE mode, large pages are 2 MB
 - On x64, default page size is 4 KB (large pages are 4 MB)
 - On Itanium, default page size is 8 KB (Itanium supports 4k, 8k, 16k, 64k, 256k, 1mb, 4mb, 16mb, 64mb, or 256mb) – large is 16MB

19

Large Pages

- Large pages allow a single page directory entry to map a larger region
 - x86, x64: 4 MB, IA64: 16 MB
 - Advantage: improves performance
 - Single TLB entry used to map larger area
- Large pages are used to map NTOSKRNL, HAL, nonpaged pool, and the PFN database if a “large memory system”
 - Windows 2000: more than 127 MB
 - Windows XP/2003: more than 255 MB
 - In other words, most systems...
- Disadvantage: disables kernel write protection
 - With small pages, OS/driver code pages are mapped as read only; with large pages, entire area must be mapped read/write
 - Drivers can then modify/corrupt system & driver code without immediately crashing system
 - Driver Verifier turns large pages off
 - Can also override by changing
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\LargePageMinimum to FFFFFFFF

20

Large Pages: Server 2003 Enhancements

- Can specify other drivers to map with large pages:
 - HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\Memory Management\LargePageDrivers (multi-string)
- Applications can use large pages for process memory
 - VirtualAlloc with MEM_LARGE_PAGE flag
 - Can query if system supports large pages with GetLargePageMinimum

21

Data Execution Prevention

- Windows XP SP2 and Windows Server 2003 SP1 support Data Execution Prevention (DEP)
 - Prevents code from executing in a memory page not specifically marked as executable
 - Stops exploits that rely on getting code executed in data areas
- Relies on hardware ability to mark pages as non executable
 - AMD calls it NX (“No Execute”)
 - Intel calls it XD (“Execute Disable”)
- Processor support:
 - Intel Itanium had this in 2001, but Windows didn’t support it until now
 - AMD64 was the next to support it
 - Then, AMD added Sempron (32-bit processor with NX support)
 - Intel added it first with their 64-bit extension chips (Xeon/Pentium 4s with EM64T)
 - More recently, Intel added it to their 32-bit processor line (anything ending in “J”)

22

Data Execution Prevention

- Attempts to execute code in a page marked no execute result in:
 - User mode: access violation exception
 - Kernel mode: `ATTEMPTED_EXECUTE_OF_NOEXECUTE_MEMORY` bugcheck (blue screen)
- Memory that needs to be executable must be marked as such using page protection bits on `VirtualAlloc` and `VirtualProtect` APIs:
 - `PAGE_EXECUTE`, `PAGE_EXECUTE_READ`, `PAGE_EXECUTE_READWRITE`, `PAGE_EXECUTE_WRITECOPY`

23

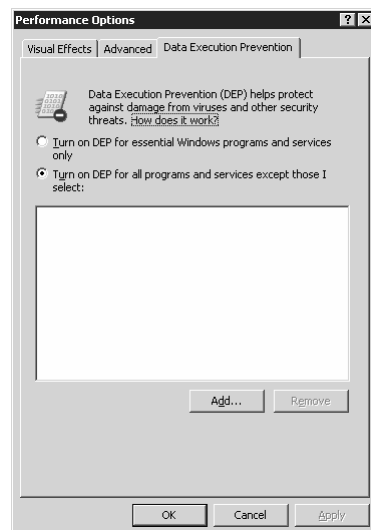
Controlling DEP

- New Boot.ini switch /NOEXECUTE
 - /NOEXECUTE=ALWAYSON – enables DEP for all applications
 - /NOEXECUTE=ALWAYSOFF – disables DEP
- Two qualifiers apply only to 32-bit applications:
 - /NOEXECUTE=OPTIN – enables DEP for core Windows programs
 - Default for Windows XP (32-bit and 64-bit editions)
 - /NOEXECUTE=OPTOUT – enables DEP for all applications except those excluded
 - Default for Windows Server 2003 (32-bit and 64-bit editions)

24

DEP on 64-bit Windows

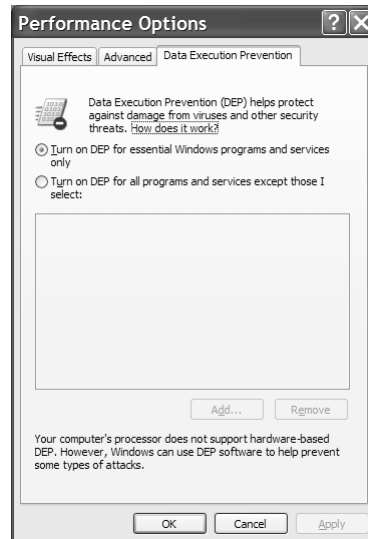
- Always applied to all 64-bit processes and device drivers
 - Protects user and kernel stacks, paged pool, session pool
- 32-bit processes depend on configuration settings



25

DEP on 32-bit Windows

- Hardware DEP used when running 32-bit Windows on systems that support it
- When enabled, system boots PAE kernel (Ntkrnlpa.exe)
- Kernel mode: applied to kernel stacks, but not paged/session pool
- User mode: depends on system configuration
- Even on processors without hardware DEP, some limited protection implemented for exception handlers



26

Mapped Files

- A way to take part of a file and map it to a range of virtual addresses (address space is 2 GB, but files can be much larger)
- Called “file mapping objects” in Windows API
- Bytes in the file then correspond one-for-one with bytes in the region of virtual address space
 - Read from the “memory” fetches data from the file
 - Pages are kept in physical memory as needed
 - Changes to the memory are eventually written back to the file (can request explicit flush)
- Initial mapped files in a process include:
 - The executable image (EXE)
 - One or more Dynamically Linked Libraries (DLLs)
- Processes can map additional files as desired (data files or additional DLLs)

27

i Section Objects (mapped files)

- Called “file mapping objects” in Windows API
- Files may be mapped into v.a.s.

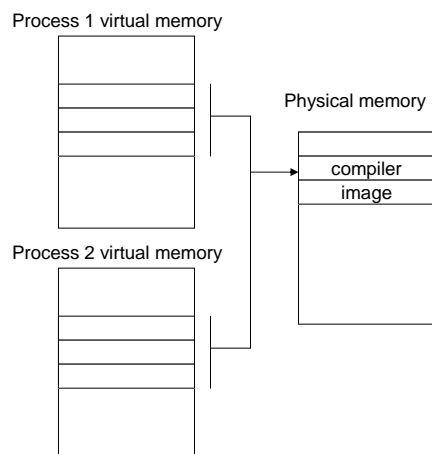
```
// first, do EITHER ...  
hMapObj = CreateFileMapping (hFile, security, protection, sizeHigh, sizeLow,  
    mapname);  
  
// ... OR ...  
hMapObj = OpenFileMapping (accessMode, inheritflag, mapname);  
  
// ... then, pass the resulting handle to a mapping object (section) to ...  
lpvoid = MapViewOfFile (hMapObj, accessMode,  
    offsetHigh, offsetLow, cbMap);
```

- Bytes in the file then correspond one-for-one with bytes in the region of virtual address space
 - Read from the “memory” fetches data from the file
 - Changes to the memory are written back to the file
 - Pages are kept in physical memory as needed
 - If desired, can map to only a part of the file at a time

28

Shared Memory

- Like most modern OS's, Windows provides a way for processes to share memory
 - High speed IPC (used by LPC, which is used by RPC)
 - Threads share address space, but applications may be divided into multiple processes for stability reasons
- It does this automatically for shareable pages
 - E.g. code pages in an EXE or DLL
- Processes can also create shared memory sections
 - Called page file backed file mapping objects
 - Full Windows security



29

Viewing DLLs & Memory Mapped Files

- Process Explorer lists memory mapped files

The screenshot shows Process Explorer with a list of processes. The 'POWERPNT.EXE' process is selected, and its memory-mapped files are listed in a table below.

Process	PID	CPU	De...	Owner	Sessi...	Han...	Window Title
LSASS.EXE	532	0	LSA...	NT AUTHORITY\SYST...	0	330	
CSRSS.EXE	996	0	Clie...	NT AUTHORITY\SYST...	1	168	
WINLOGON.EXE	1392	0	Wind...	NT AUTHORITY\SYST...	1	235	
wuauclt.exe	2040	0	Wind...	DANI\Admin	1	89	
EXPLORER.EXE	1560	0	Wind...	DANI\Daniel	0	252	
MSMSG.S.EXE	1660	0	Mes...	DANI\Daniel	0	45	
msmsgshrl.exe	1868	0	Mes...	DANI\Daniel	0	111	
EXPLORER.EXE	1924	0	Wind...	DANI\Admin	1	357	C:\david
POWERPNT.EXE	1200	2	Micr...	DANI\Admin	1	307	Microsoft PowerPoint - [f
OUTLOOK.EXE	1396	0	Micr...	DANI\Admin	1	251	Inbox - Microsoft Outool
MSMSG.S.EXE	2008	0	Mes...	DANI\Admin	1	45	
msmsgshrl.exe	156	0	Mes...	DANI\Admin	1	117	
cmd.exe	2080	0	Wind...	DANI\Admin	1	48	C:\WINDOWS\System32

Base	Size	MM	Description	Version	Time	Path
0x25B0000	0xC000	*			1/11/2003 1:58 PM	C:\Documents and Settings\Admin\Cook
0x25F0000	0x300000	*			1/11/2003 1:58 PM	C:\Documents and Settings\Admin\Loca
0x28F0000	0x5C000	*			1/11/2003 1:58 PM	C:\Documents and Settings\Admin\Loca
0x2D40000	0x1000	*			1/11/2003 1:58 PM	C:\Documents and Settings\Admin\Loca
0x2F00000	0x1000	*			1/11/2003 1:58 PM	C:\Documents and Settings\Admin\Loca
0x33E0000	0xE000	*			1/11/2003 2:10 PM	C:\david6-memmgmt.ppt
0x3000000	0x5B2000		Microsoft Po...	10.00.262...	2/26/2001 2:54 AM	C:\Program Files\Microsoft Office\Office
0x30B0000	0x988000		Microsoft Of...	10.00.331...	9/12/2001 8:29 PM	C:\Program Files\Common Files\Microso
0x317D0000	0x69000		Microsoft Po...	10.00.260...	2/13/2001 1:28 AM	C:\Program Files\Microsoft Office\Office

30

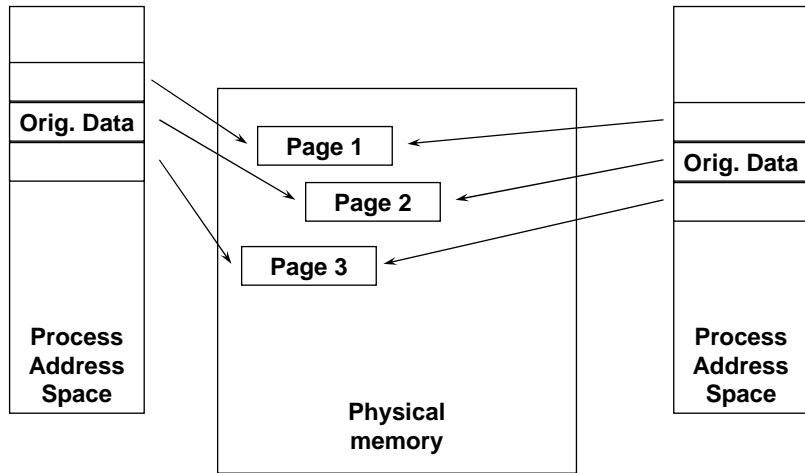
i Copy-On-Write Pages

- Used for sharing between process address spaces
- Pages are originally set up as shared, read-only, faulted from the common file
 - Access violation on write attempt alerts pager
 - pager makes a copy of the page and allocates it privately to the process doing the write, backed to the paging file
 - So, only need unique copies for the pages in the shared region that are actually written (example of "lazy evaluation")
 - Original values of data are still shared
 - e.g. writeable data initialized with C initializers

31



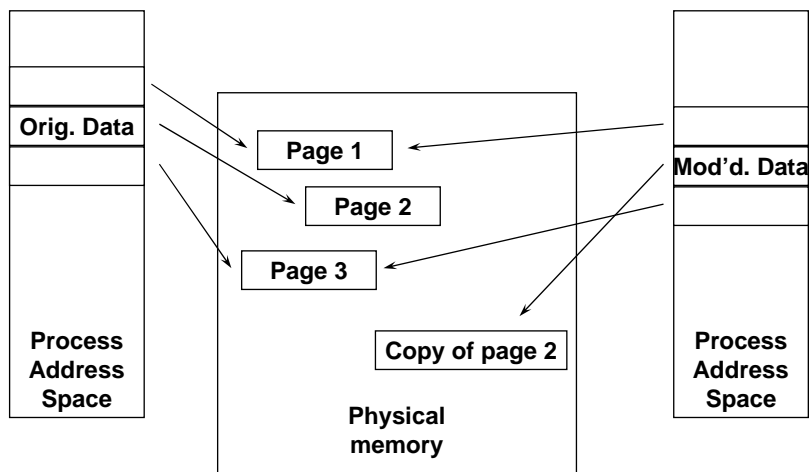
How Copy-On-Write Works Before



32

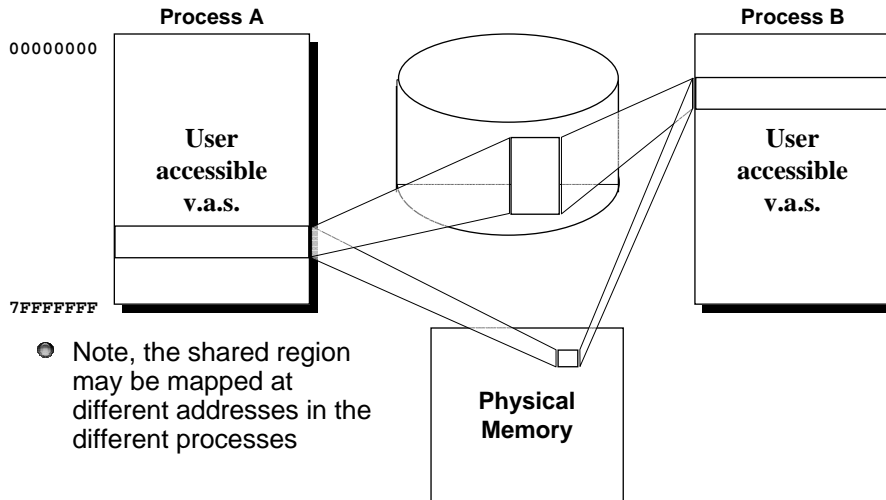


How Copy-On-Write Works After



33

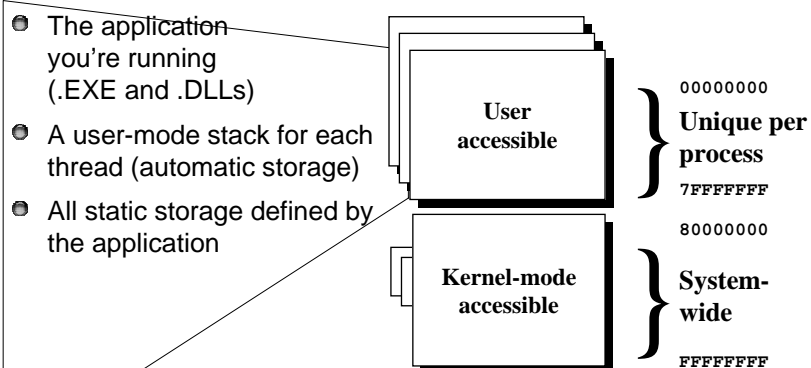
Shared Memory = File Mapped by Multiple Processes



34

Virtual Address Space (V.A.S.)

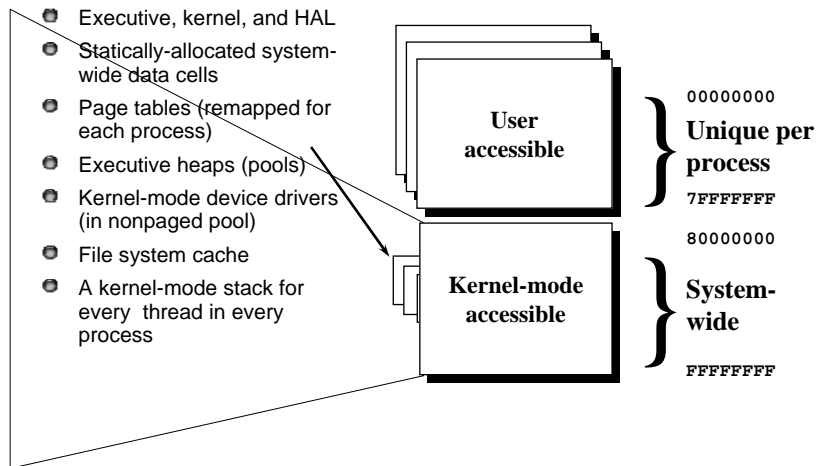
Process space contains:



35

Virtual Address Space (V.A.S.)

● System space contains:



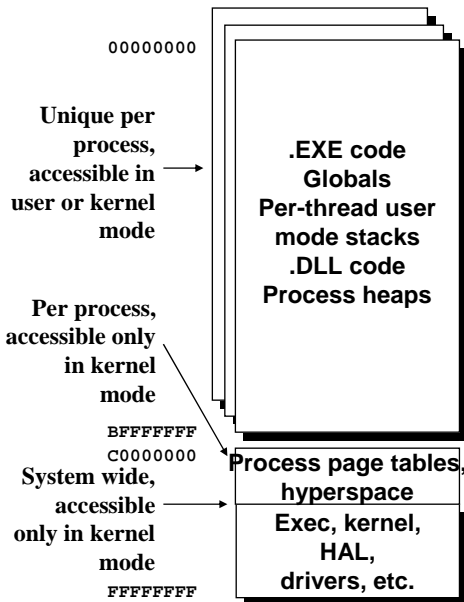
36

Windows User Process Address Space Layout

Range	Size	Function
0x0 – 0xFFFF	64 KB	No-access region to catch incorrect pointer ref.
0x10000 - 07FFFEFFFF	2 GB minus at least 192kb	The private process address space
0x7FFDE000 - 0x7FFDEFFF	4 KB	Thread Environment Block (TEB) for first thread, more TEBs are created at the page prior to that page
0x7FFDF000 - 0x7FFDFFFF	4 KB	Process Environment Block (PEB)
0x7FFE0000 - 0x7FFE0FFF	4 KB	Shared user data page – read-only, mapped to system space, contains system time, clock tick count, version number (avoid kernel-mode transition)
0x7FFE1000 – 0x7FFEFFFF	60 KB	No-access region
0x7FFF0000 – 0x7FFF7FFF	64 KB	No-access region to prevent threads from passing buffers that straddle user/system space boundary

37

3GB Process Space Option



- Only available on:
 - Windows 2003 Server, Enterprise Edition & Windows 2000 Advanced Server, XP SP2
 - Limits phys memory to 16 GB
 - /3GB option in BOOT.INI
 - Windows Server 2003 and XP SP2 supports variations from 2GB to 3GB (/USERVA=)
- Provides 3 GB per-process address space
 - Commonly used by database servers (for file mapping)
 - .EXE must have "large address space aware" flag in image header, or they're limited to 2 GB (specify at link time or with imagecfg.exe from ResKit)
 - Chief "loser" in system space is file system cache
 - Better solution: address windowing extensions
 - Even better: 64-bit Windows

38

Large Address Space Aware Images

- Images marked as "large address space aware":
 - Lsass.exe – Security Server
 - Inetinfo.exe—Internet Information Server
 - Chkdsk.exe – Check Disk utility
 - Dllhst3g.exe – special version of Dllhost.exe (for COM+ applications)
 - Esentutl.exe - jet database repair tool
- To see this type:

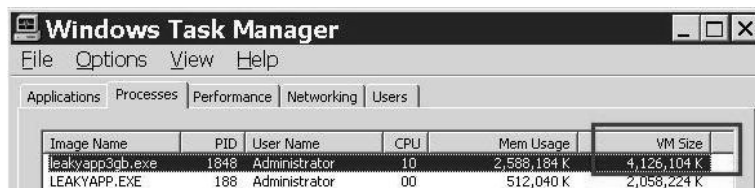

```
Imagecfg \windows\system32\*.exe > large_images.txt
```

 - Then search for "large" in large_images.txt

39

Large Address Space Aware on 64-bits

- Images marked large address space aware get a full 4 GB process virtual address space
 - OS isn't mapped there, so space is available for process



The screenshot shows the Windows Task Manager window with the 'Processes' tab selected. A table displays the following data:

Image Name	PID	User Name	CPU	Mem Usage	VM Size
leakyapp3gb.exe	1848	Administrator	10	2,588,184 K	4,126,104 K
LEAKYAPP.EXE	188	Administrator	00	512,040 K	2,058,224 K

40

Physical Memory

- Maximum on Windows NT 4.0 was 4 GB for x86 (8 GB for Alpha AXP)
 - This is fixed by page table entry (PTE) format
- What about x86 systems with > 4 GB?
 - Pentium Pro and Xeon systems can support up to 64 GB physical memory
 - Four more bits of physical address in PTEs = 36 bits = 64 GB
- NT4: Intel provides a driver that allows use of RAM beyond 4 GB as a RAM disk
- Windows 2000 added proper support for PAE
 - Requires booting /PAE to select the PAE kernel
- Actual physical memory usable varies by Windows package...

41

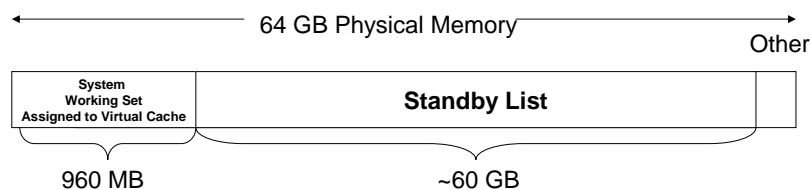
Physical Memory Limits (in GB)

	x86	x64 32-bit	x64 64-bit	IA-64 64-bit
XP Home	4	4	n/a	n/a
XP Professional	4	4	16	n/a
Server 2003 Web Edition	2	2	n/a	n/a
Server 2003 Standard	4	4	16	n/a
Server 2003 Enterprise	32	32	64	64
Server 2003 Datacenter	64	128	1024	1024

42

Physical Memory Usage on Systems in PAE Mode

- Virtual address space is still 4 GB, so how can you "use" > 4 GB of memory?
 - Although each process can only address 2 GB, many may be in memory at the same time (e.g. 5 * 2 GB processes = 10 GB)
 - Files in system cache remain in physical memory
 - Although file cache doesn't know it, memory manager keeps unmapped data in physical memory

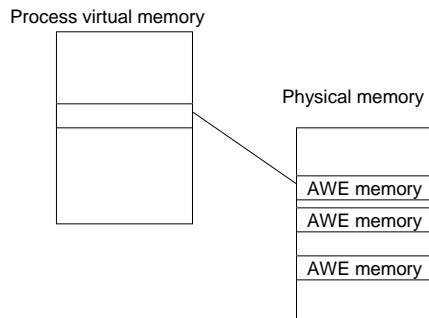


- New Address Windowing Extensions allow Windows processes to use more than 2 GB of memory

43

Address Windowing Extensions

- AWE functions allow Windows processes to allocate large amounts of physical memory and then map “windows” into that memory
- Applications: database servers can cache large databases
- Up to programmer to control
 - Like DOS enhanced memory (EMS) with more bits...
- 64-bits removes this need



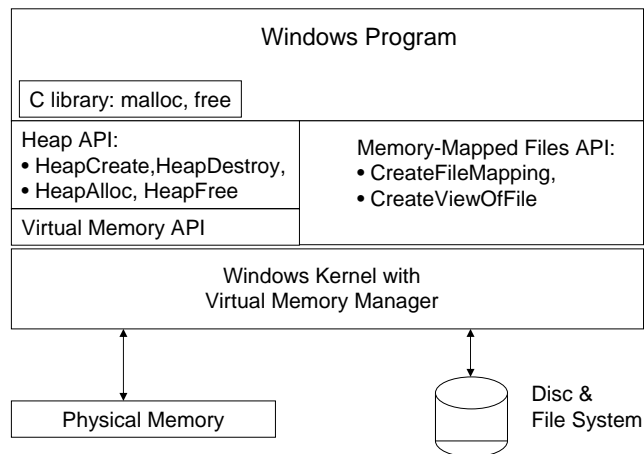
44

Windows Memory Allocation APIs

- HeapCreate, Alloc, etc. (process heap APIs)
 - Windows equivalent of malloc(), free(), etc.
- VirtualAlloc(MEM_RESERVE)
- VirtualAlloc(MEM_COMMIT)
- VirtualFree
- VirtualQuery

45

Windows API Memory Management Architecture



46

Windows Memory Management

- Windows maintains pools of memory in heaps
- A process can contain several heaps
 - C library functions manage default heap: malloc, free, calloc
- Heaps are Windows objects – have handle
 - Each process has own default heap
 - Return value of NULL indicates failure (instead of INVALID_HANDLE_VALUE)

```
HANDLE GetProcessHeap( VOID );
HANDLE HeapCreate (DWORD floptions,
                  DWORD dwInitialSize,
                  DWORD dwMaximumSize);
BOOL HeapDestroy( HANDLE hHeap );
```

47

Managing Heap Memory

```
LPVOID HeapAlloc( HANDLE hHeap,  
                 DWORD dwFlags,  
                 DWORD dwBytes );
```

- dwFlags:
 - HEAP_GENERATE_EXCEPTION,
 - raise SEH on memory allocation failure
 - STATUS_NO_MEMORY, STATUS_ACCESS_VIOLATION
 - HEAP_NO_SERIALIZE:
 - no serialization of concurrent (multithreaded) requests
 - HEAP_ZERO_MEMORY: initialize allocated memory to zero
- dwSize:
 - Block of memory to allocate
 - For non-growable heaps: 0x7FFF8 (0.5 MB)
- HeapFree(), HeapReAlloc(),
- HeapCompact(), HeapValidate()

```
HeapLock(), HeapUnlock():  
Manage concurrent accesses  
to heap
```

48

Excerpt: Sorting with Binary Search Tree

```
#define NODE_HEAP_SIZE 0x8000  
__try {  
    /* Open the input file. */  
    hIn = CreateFile (fname, GENERIC_READ, 0, NULL,  
                    OPEN_EXISTING, 0, NULL);  
    if (hIn == INVALID_HANDLE_VALUE)  
        fprintf(stderr, "Failed to open input file"), exit(1);  
    /* Allocate the two heaps. */  
    hNode = HeapCreate (  
        HEAP_GENERATE_EXCEPTIONS | HEAP_NO_SERIALIZE,  
        NODE_HEAP_SIZE, 0);  
    hData = HeapCreate (  
        HEAP_GENERATE_EXCEPTIONS | HEAP_NO_SERIALIZE,  
        DATA_HEAP_SIZE, 0);  
    /* Process the input file, creating the tree, actual search. */  
    pRoot = FillTree (hIn, hNode, hData);
```

49

Heap Management Example (contd.)

```
/* Display the tree in Key order. */
printf ("Sorted file: %s"), fname); Scan (pRoot);

/* Destroy the two heaps and data structures. */
HeapDestroy (hNode); hNode = NULL;
HeapDestroy (hData); hData = NULL;
CloseHandle (hIn);
} /* End of main file processing and try block. */

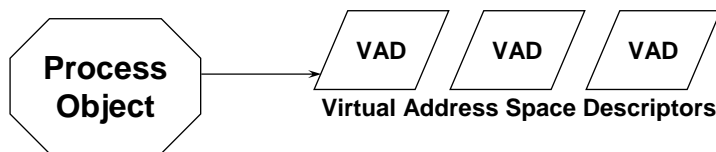
__except (EXCEPTION_EXECUTE_HANDLER) {
    if (hNode != NULL) HeapDestroy (hNode);
    if (hData != NULL) HeapDestroy (hData);
    if (hIn != INVALID_HANDLE_VALUE) CloseHandle (hIn);
}
return 0;
```

- UNIX C library uses only a single heap
- UNIX sbrk() can create a Process' address space – no general-purpose MM
- UNIX does not generate signals on memory alloc.

50



Virtual Address Space Descriptors (VADs)



- VADs describe layout of virtual address space
 - Not the page mappings
- Used by memory manager to interpret access faults
 - Assists in "lazy evaluation"

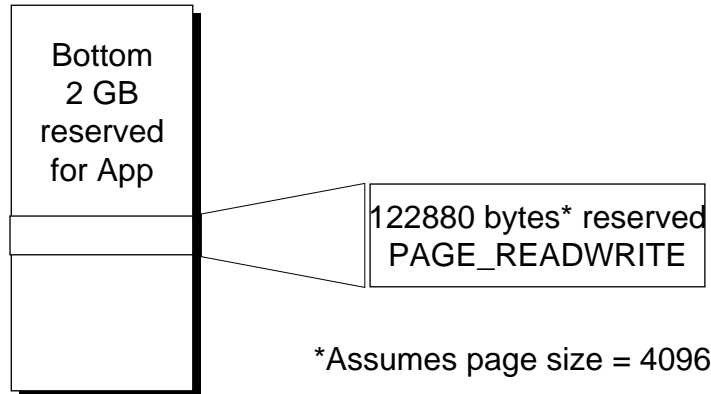
See kernel debugger
command:
!vad

51



Example: Reserving Address Space

```
LPVOID lpMem = VirtualAlloc(NULL, 120000, MEM_RESERVE,  
PAGE_READWRITE);
```

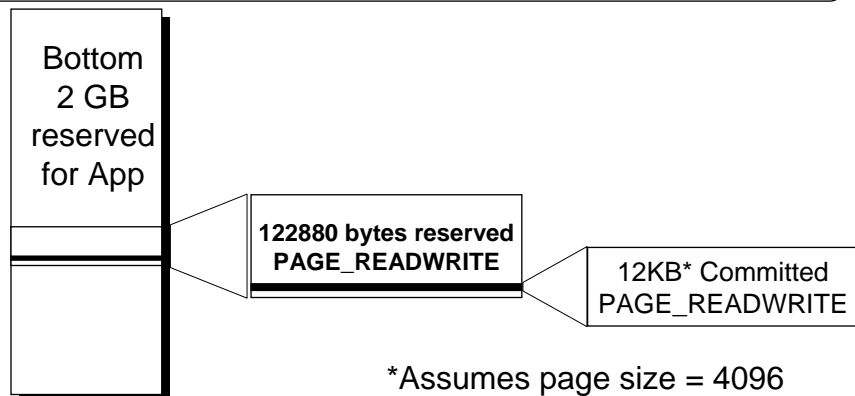


52



Example: Committing Address Space

```
VirtualAlloc(lpMem + 6 * 1024, 7 * 1024, MEM_COMMIT,  
PAGE_READWRITE);
```



53

Memory-Mapped Files

- No need to perform direct file I/O (read/write)
- Data structures will be saved – be careful with pointers
- Convenient & efficient in-memory algorithms:
 - Can process data much larger than physical memory
- Improved performance for file processing
- No need to manage buffers and file data
 - OS does the hard work: efficient & reliable
- Multiple processes can share memory
- No need to consume space in paging file

54

File Mapping Object

```
HANDLE CreateFileMapping (HANDLE hFile,  
LPSECURITY_ATTRIBUTES lpsa,  
DWORD fdwProtect,  
DWORD dwMaximumSizeHigh,  
DWORD dwMaximumSizeLow,  
LPCTSTR lpszMapName );
```

Parameters:

- hFile:
 - hFile: handle to open file with compatible access rights (fdwProtect)
 - hFile == 0xFFFFFFFF: paging file, no need to create separate file
- fdwProtect:
 - PAGE_READONLY, PAGE_READWRITE, PAGE_WRITECOPY
- dwMaximumSizeHigh, dwMaximumSizeLow:
 - Zero: current file size is used
- lpszMapName:
 - Name of mapping object for sharing between processes or NULL

55

Shared Memory

```
HANDLE OpenFileMapping (HANDLE hFile,  
    DWORD dwDesiredAccess,  
    BOOL blInheritHandle,  
    LPCTSTR lpName );
```

- Open an existing mapping object
 - Name comes from previous CreateFileMapping() call
 - First process creates mapping, subsequent processes open mapping
- dwDesiredAccess: same as fdwProtect
- lpName: name created with CreateFileMapping()
- CloseHandle() destroys mapping handles

56

Mapping Process Address Space to Mapping Objects

UNIX:
4.3BSD/SysV.4
have [mmap\(\)](#) call;

See also
[shmget\(\)](#),[shmctl\(\)](#),
[shmat\(\)](#),[shmdt\(\)](#)

```
LPVOID MapViewOfFile( HANDLE hMapObject,  
    DWORD fdwAccess, DWORD dwOffsetHigh,  
    DWORD dwOffsetLow, DWORD cbMap );  
BOOL UnmapViewOfFile ( LPVOID lpBaseAddress );
```

- Allocate virtual memory space and map it to a file through a mapping object
 - Similar to HeapAlloc – much coarser granularity
 - Pointer to allocated block is returned (file view)
- Parameters:
 - FILE_MAP_WRITE, FILE_MAP_READ, FILE_MAP_ALL_ACCESS flag bits for fdwAccess
 - cbMap: size; entire file if zero
- FlushViewOfFile(): create consistent view

Limitation:
2GB virtual
Address space

57

Example: File Conversion with Memory Mapping (Excerpt)

```
        /* Open the input file. */
hIn = CreateFile (fIn, GENERIC_READ, 0, NULL,
                OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
if (hIn == INVALID_HANDLE_VALUE) fprintf(stderr, "Failure opening input file."), exit(1);
        /* Create a file mapping object on the input file. Use the file size. */
hInMap = CreateFileMapping (hIn, NULL, PAGE_READONLY, 0, 0, NULL);
if (hInMap == INVALID_HANDLE_VALUE) fprintf(stderr, "Failure Creating input map."), exit(2);

pInFile = MapViewOfFile (hInMap, FILE_MAP_READ, 0, 0, 0);
if (pInFile == NULL) fprintf(stderr, "Failure Mapping input file."), exit(3);
        /* The output file MUST have Read/Write access for the mapping to succeed. */
hOut = CreateFile (fOut, GENERIC_READ | GENERIC_WRITE,
                0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
if (hOut == INVALID_HANDLE_VALUE) fprintf(stderr, "Failure Opening output file."), exit(4);
hOutMap = CreateFileMapping (hOut, NULL, PAGE_READWRITE, 0, 2 * FsLow, NULL);
if (hOutMap == INVALID_HANDLE_VALUE) fprintf(stderr, "Failure creating output map."), exit(5);

pOutFile = MapViewOfFile (hOutMap, FILE_MAP_WRITE, 0, 0, 2 * FsLow);
if (pOutFile == NULL) fprintf(stderr, "Failure mapping output file."), exit(6);
```

58

Example (contd.)

```
pIn = pInFile;                                /* actual file conversion */
pOut = pOutFile;
while (pIn < pInFile + FsLow) {
    *pOut = (WCHAR) *pIn; pIn++; pOut++;
}

/* Close all views and handles. */
UnmapViewOfFile (pOutFile); UnmapViewOfFile (pInFile);
CloseHandle (hOutMap); CloseHandle (hInMap);
CloseHandle (hIn); CloseHandle (hOut);
Complete = TRUE; return TRUE;
}

_except (EXCEPTION_EXECUTE_HANDLER) {
    /* Delete the output file if the operation did not complete successfully. */
    if (!Complete)
        DeleteFile (fOut);
    return FALSE;
}
```

59

Memory Management APIs

- Memory protection may be changed
 - per-page basis

```
status = VirtualProtect(baseAddress, size, newProtect, pOldprotect);
```

- Page protection choices:

```
PAGE_NOACCESS      PAGE_EXECUTE
PAGE_READONLY      PAGE_EXECUTE_READ
PAGE_READWRITE     PAGE_EXECUTE_READWRITE
PAGE_WRITECOPY     PAGE_EXECUTE_WRITECOPY

PAGE_GUARD
PAGE_NOCACHE
```

60

Memory Management Information

```
VOID GetSystemInfo(LPSYSTEM_INFO lpSystemInfo);
```

```
typedef struct _SYSTEM_INFO {
    DWORD          dwOemId;
    DWORD          dwPageSize;
    LPVOID lpMinimumApplicationAddress;
    LPVOID lpMaximumApplicationAddress;
    DWORD          dwActiveProcessorMask;
    DWORD          dwNumberOfProcessors;
    DWORD          dwProcessorType;
    DWORD          dwAllocationGranularity;
    DWORD          dwReserved;
} SYSTEM_INFO;
```

61



Querying Address Space

```
DWORD VirtualQuery(LPVOID lpAddress,  
PMEMORY_BASIC_INFORMATION lpBuffer, DWORD dwLength);
```

Returns:

```
typedef struct _MEMORY_BASIC_INFORMATION {  
    PVOID BaseAddress;           // Block base  
    PVOID AllocationBase;       // Region base  
    DWORD AllocationProtect;    // Region prot  
    DWORD RegionSize;           // # bytes in block  
    DWORD State;                // State of block:  
    // MEM_RESERVE, MEM_COMMIT, MEM_FREE  
    DWORD Protect;              // Pages prot  
    DWORD Type;                 // Type:  
    // MEM_IMAGE, MEM_MAPPED, MEM_PRIVATE  
} MEMORY_BASIC_INFORMATION;
```

62



Memory Management Information

```
VOID GlobalMemoryStatus(LPMEMORYSTATUS lpms);  
  
typedef struct _MEMORYSTATUS {  
    DWORD dwLength;              // sizeof(MEMORYSTATUS)  
    DWORD dwMemoryLoad;  
    DWORD dwTotalPhys;  
    DWORD dwAvailPhys;  
    DWORD dwTotalPageFile;  
    DWORD dwAvailPageFile;  
    DWORD dwTotalVirtual;       // Process specific  
    DWORD dwAvailVirtual;       // Process specific  
} MEMORYSTATUS, *LPMEMORYSTATUS;
```

Note: much more available via Registry Performance counters

63

Further Reading

- Mark E. Russinovich and David A. Solomon, Microsoft Windows Internals, 4th Edition, Microsoft Press, 2004.
 - Chapter 7 - Memory Management
 - Memory Manager (from pp.375)
 - Services the Memory Manager Provides (from pp. 382)
- Jeffrey Richter, Programming Applications for Microsoft Windows, 4th Edition, Microsoft Press, September 1999.
 - Chapter 5 - Windows API Memory Architecture
 - Chapter 7 - Using Virtual Memory
 - Chapter 8 - Memory-Mapped Files
 - Chapter 9 - Heaps

64

Source Code References

- Windows Research Kernel sources
 - `\base\ntos\mm` – Memory manager
 - `\base\ntos\inc\mm.h` – additional structure definitions
 - `\base\ntos\cache` – Cache manager

65