Windows Memory Management (II)

Virtual Address Translation and Paging

Roadmap for This Lecture

- From virtual to physical addresses
- Address space layout
- Address translation
- Page directories, page tables
- Page faults, invalid page table entries
- Page frame number database

Virtual Memory - Concepts

- Application always references “virtual addresses”
- Hardware and software translates, or *maps,* virtual addresses to physical addresses
- Not all of an application’s virtual address space is in physical memory at one time...
  - ...But hardware and software fool the application into thinking that it is
  - The rest is kept on disk, and is brought into physical memory automatically as needed

Virtual Address Space

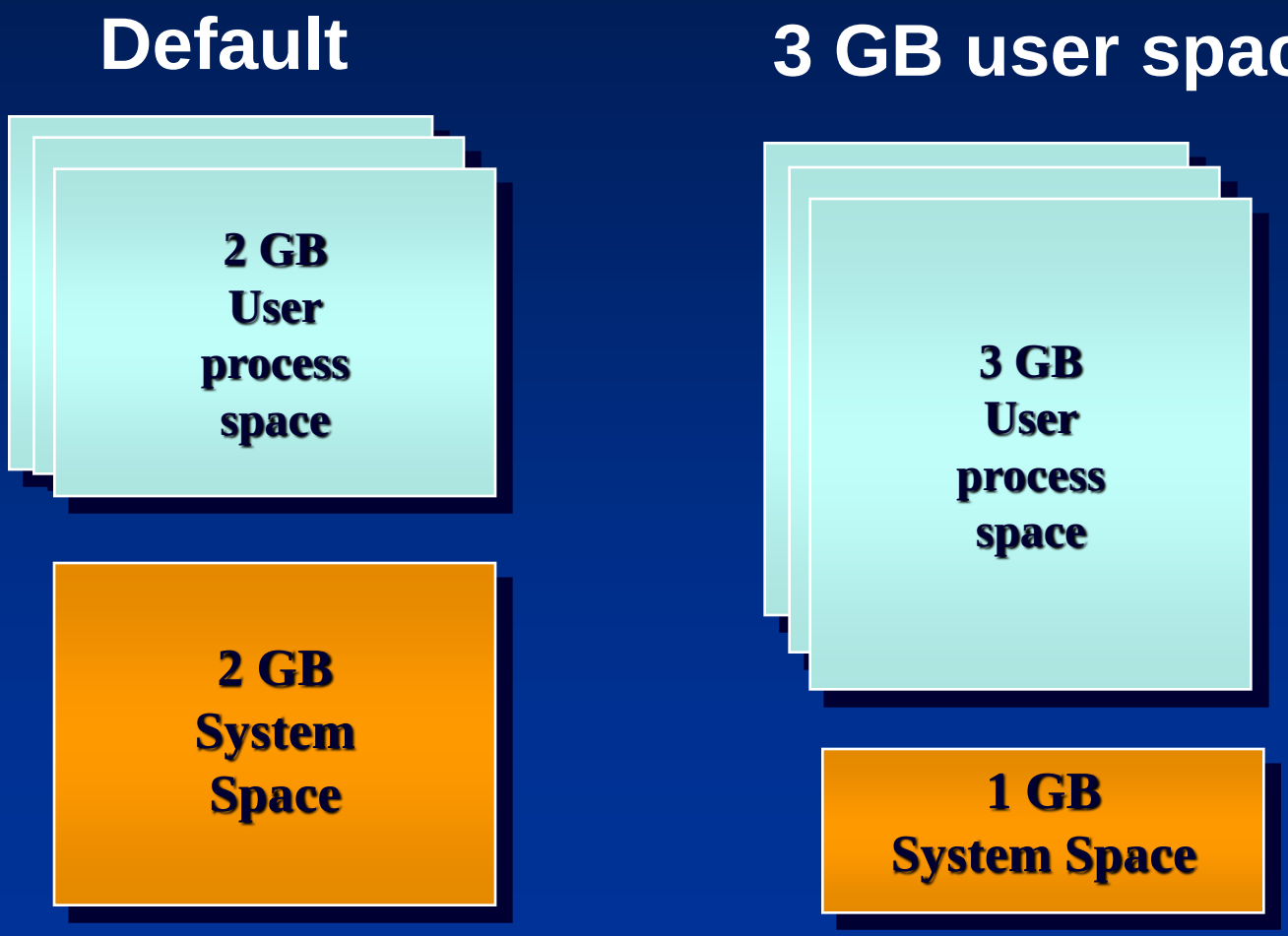
- Process private address space
  - Can’t access outside virtual addresses unless map to shared memory sections or use cross-process memory functions
  - Page table stored in system space
- Session space
  - All session-wide data structures
  - Session-specific paged pool
  - Copy of subsystem process (Csrss.exe) and logon process (Winlogon.exe)
- System space
  - Global OS code and data structures:
    - System code
    - System mapped views
    - Hyperspace
    - System working set list
    - System cache
    - Paged pools/non-paged pools
    - Page table

Virtual address descriptors (VADs)

- Memory manager uses demand paging algorithm
- Lazy evaluation is also used to construct page tables
  - Reserved vs. committed memory
  - Even for committed memory, page table are constructed *on demand*
- Memory manager maintains VAD structures to keep track of reserved virtual addresses
  - Self-balancing binary tree (AVL-tree)
- VAD stores:
  - range of addresses being reserved;
  - whether range will be shared or private;
  - Whether child process can inherit contents of the range
  - Page protection applied to pages within the address range

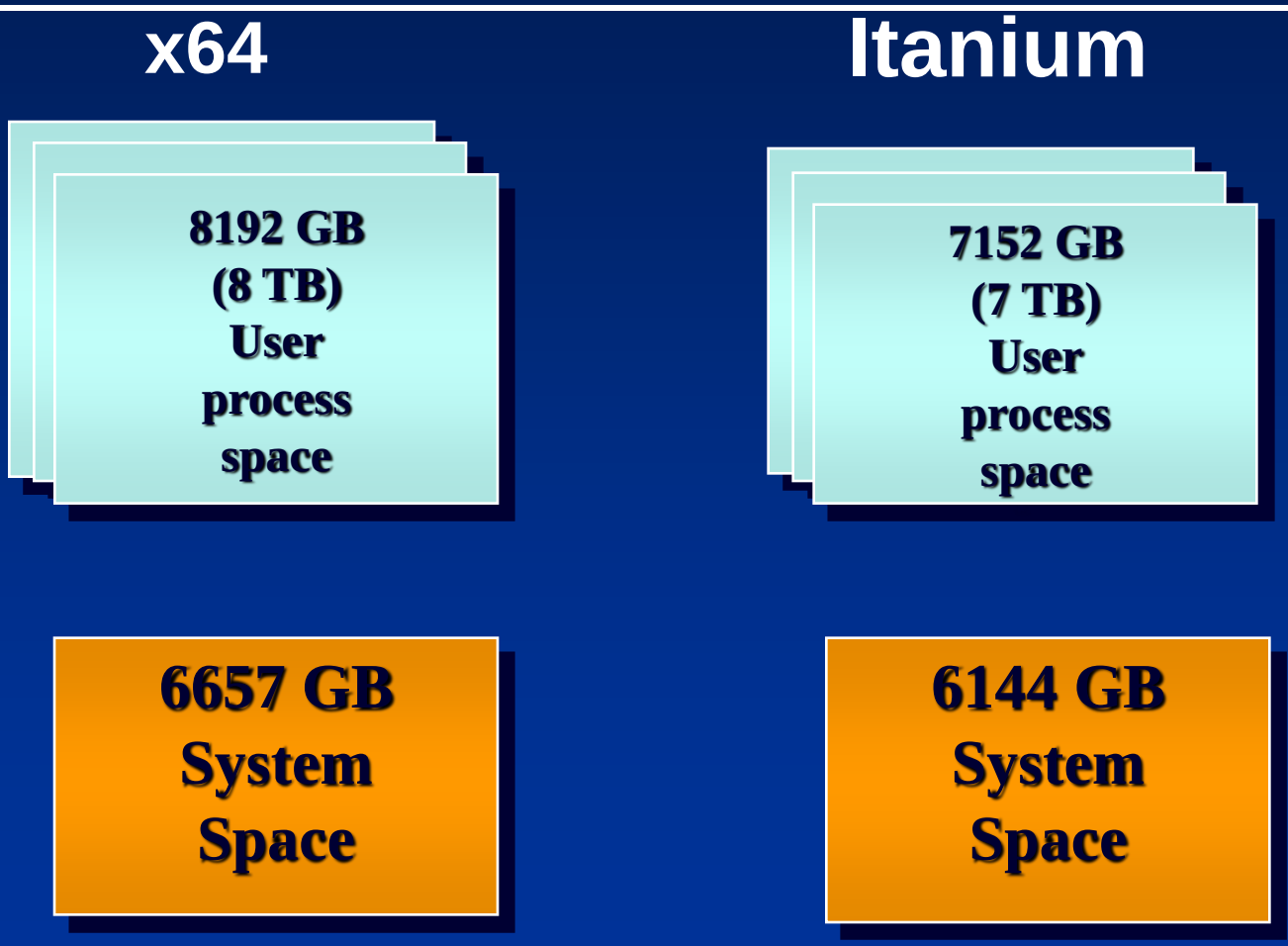
32-bit x86 Address Space

- 32-bits = 4 GB

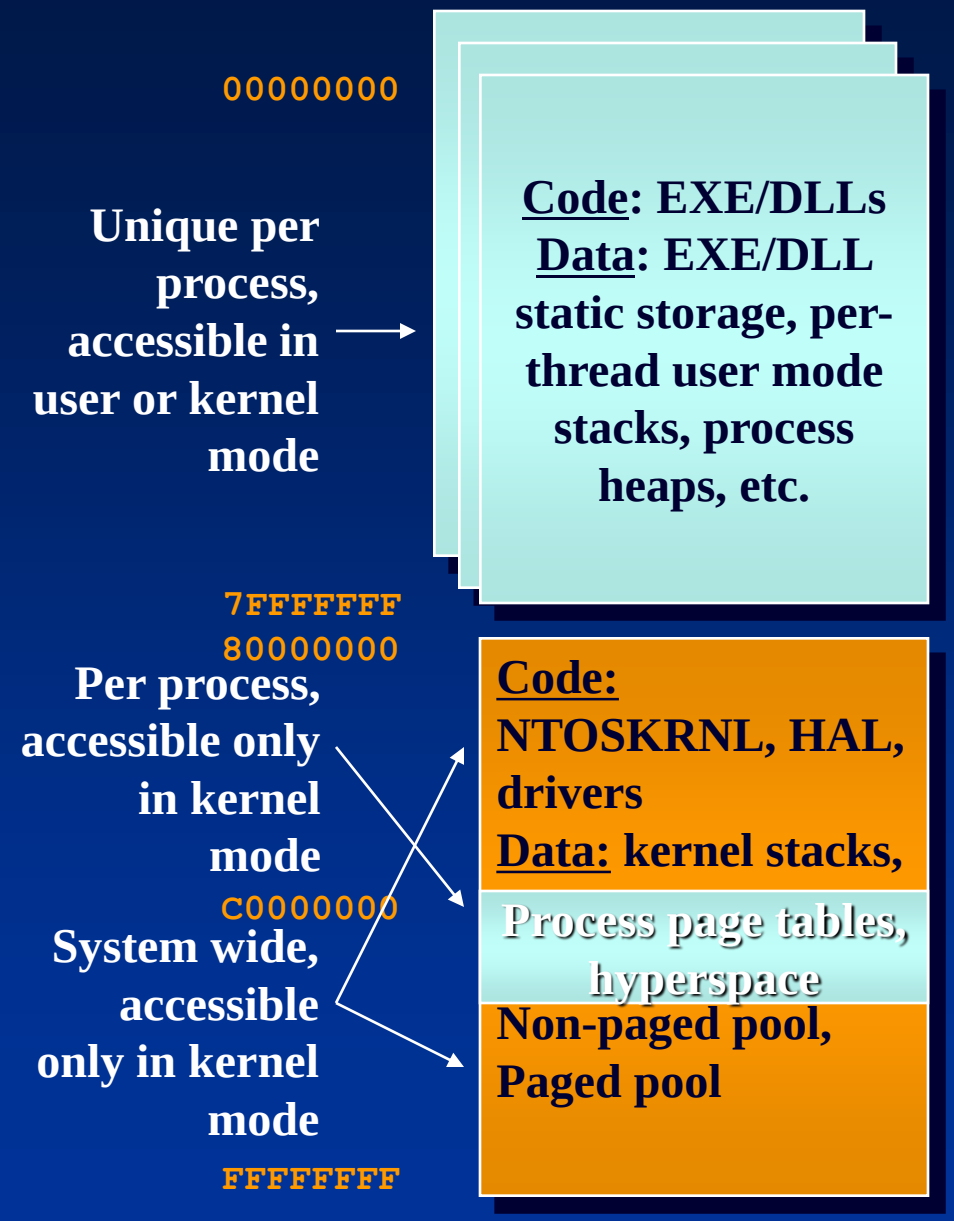


64-bit Address Spaces

- 64-bits = 17,179,869,184 GB
  - x64 today supports 48 bits virtual = 262,144 GB
  - IA-64 today support 50 bits virtual = 1,048,576 GB



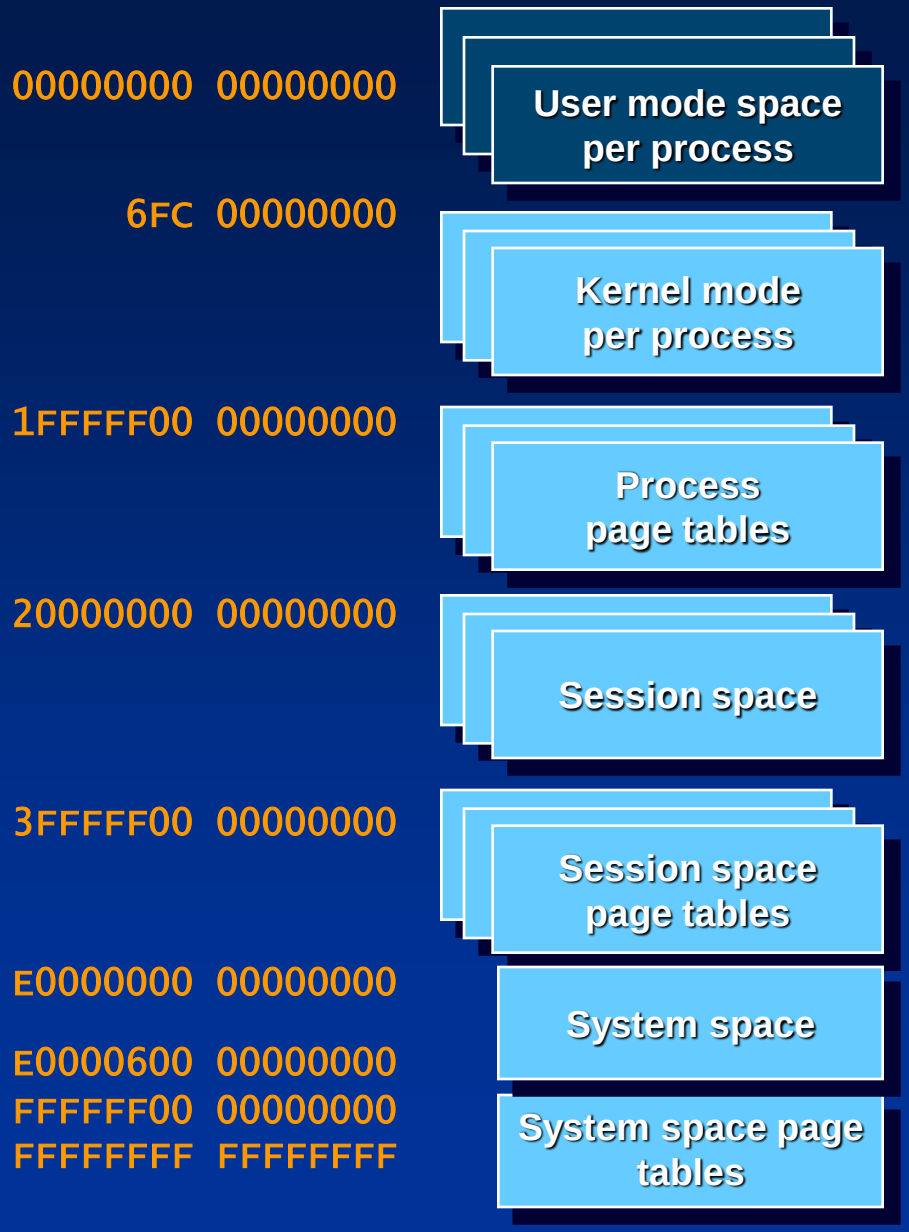
32-bit x86 Virtual Address Space



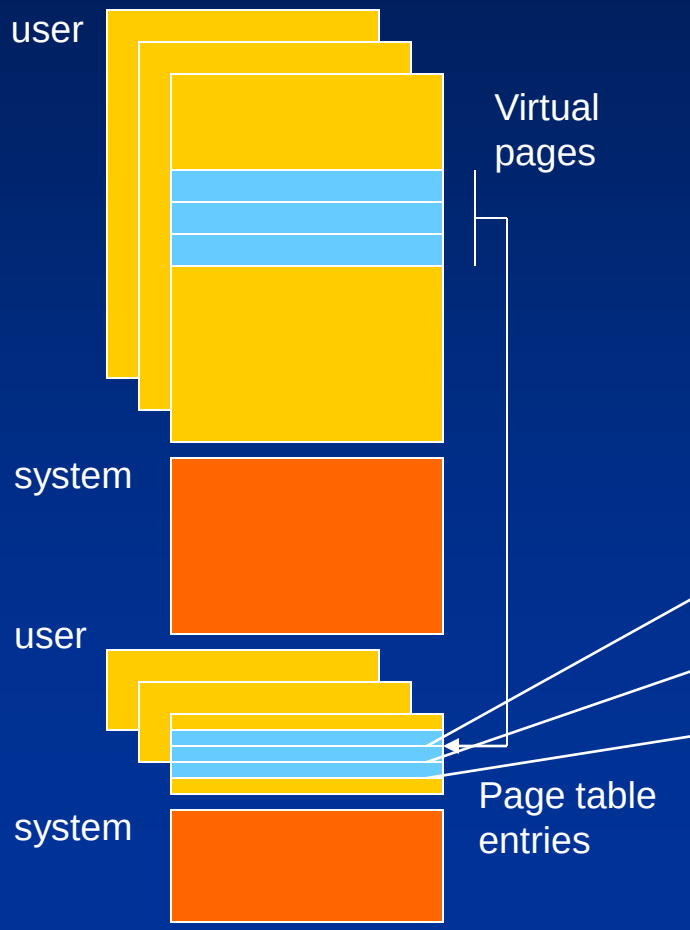
- 2 GB per-process
  - Address space of one process is not directly reachable from other processes
- 2 GB system-wide
  - The operating system is loaded here, and appears in every process's address space
  - The operating system is not a process (though there are processes that do things for the OS, more or less in "background")

64-bit ia64 (Itanium) Virtual Address Space

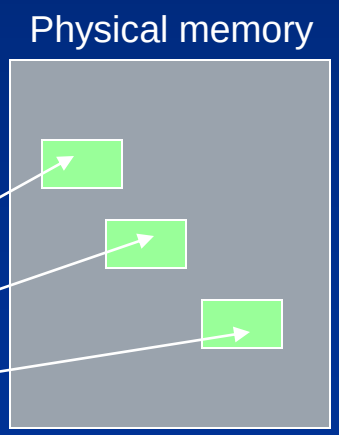
- 64 bits = 2^64 = 17 billion GB (16 exabytes) total
  - Diagram NOT to scale!
- 7152 GB default per-process
- Pages are 8 Kbytes
- All pointers are now 64 bits wide (and not the same size as a ULONG)



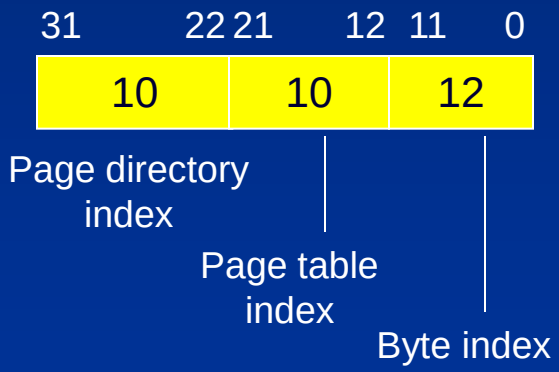
Address Translation - Mapping virtual addresses to physical memory



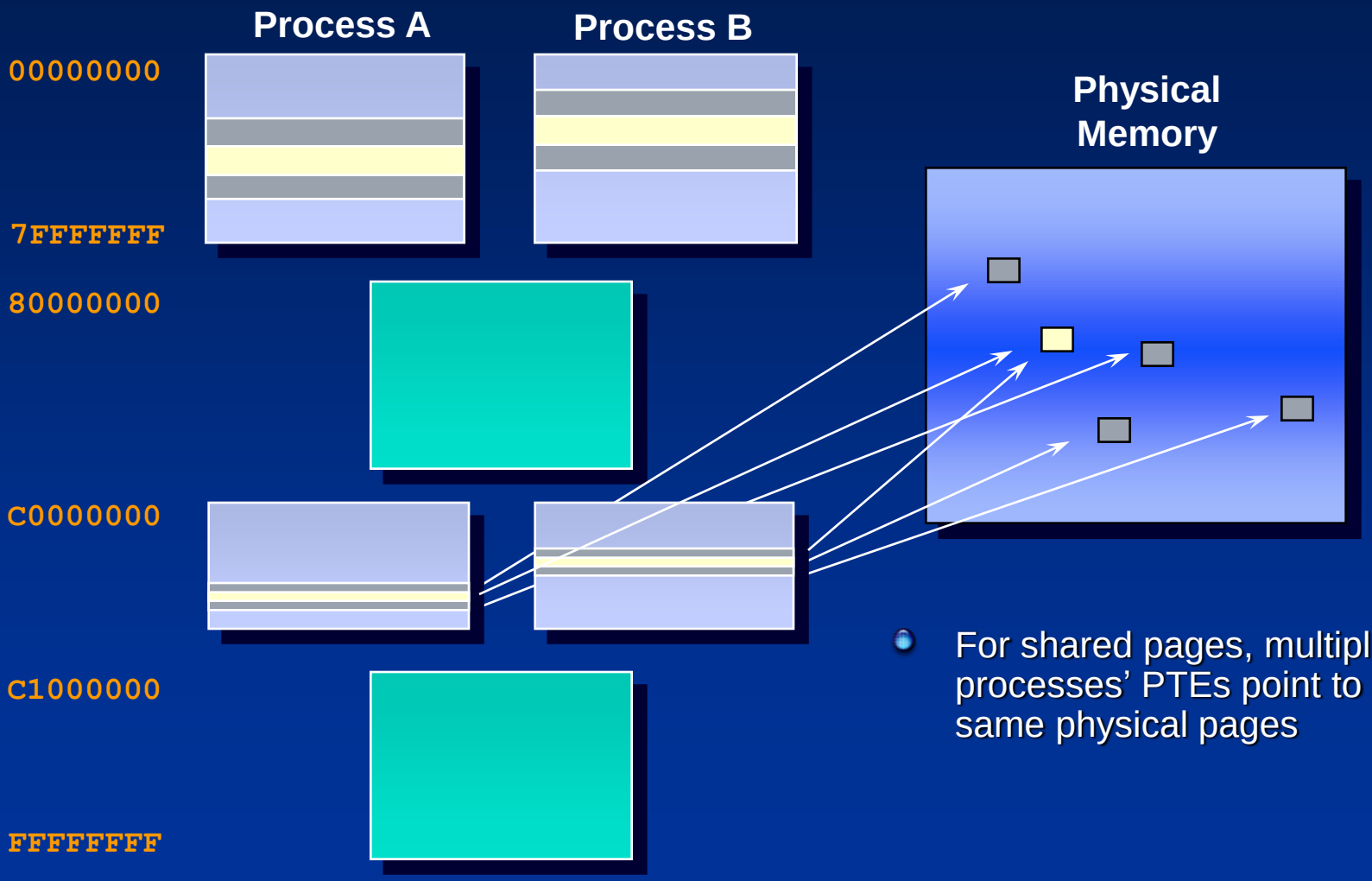
- Mapping via *page table entries*
- Indirect relationship between virtual pages and physical memory
- 12 bit = 4096 bytes



x86:



Shared and Private Pages



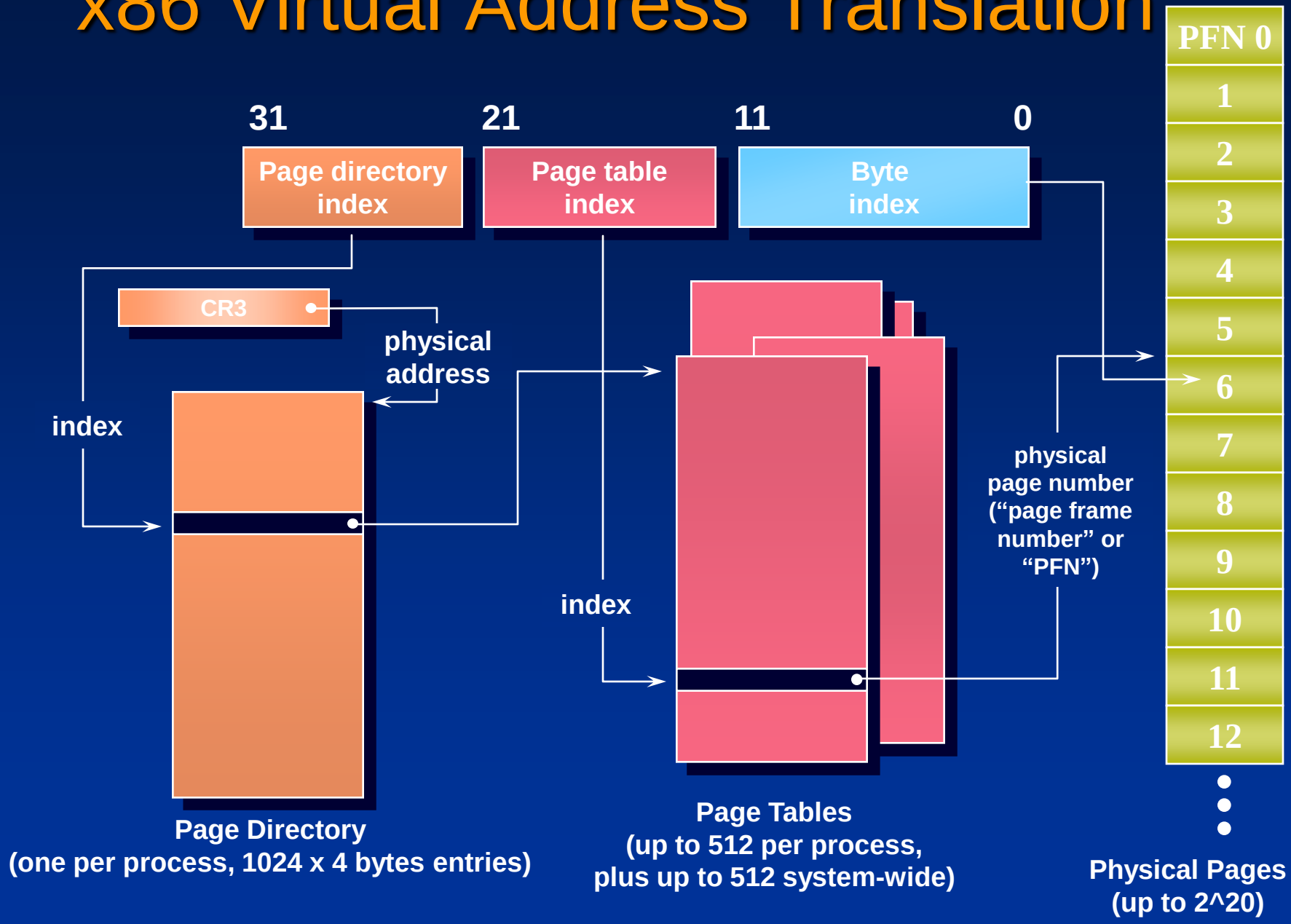
- For shared pages, multiple processes' PTEs point to same physical pages

Interpreting a Virtual Address

x86 32-bit

x64 64-bit (48-bit in today's processors)

x86 Virtual Address Translation



Translating a virtual address:

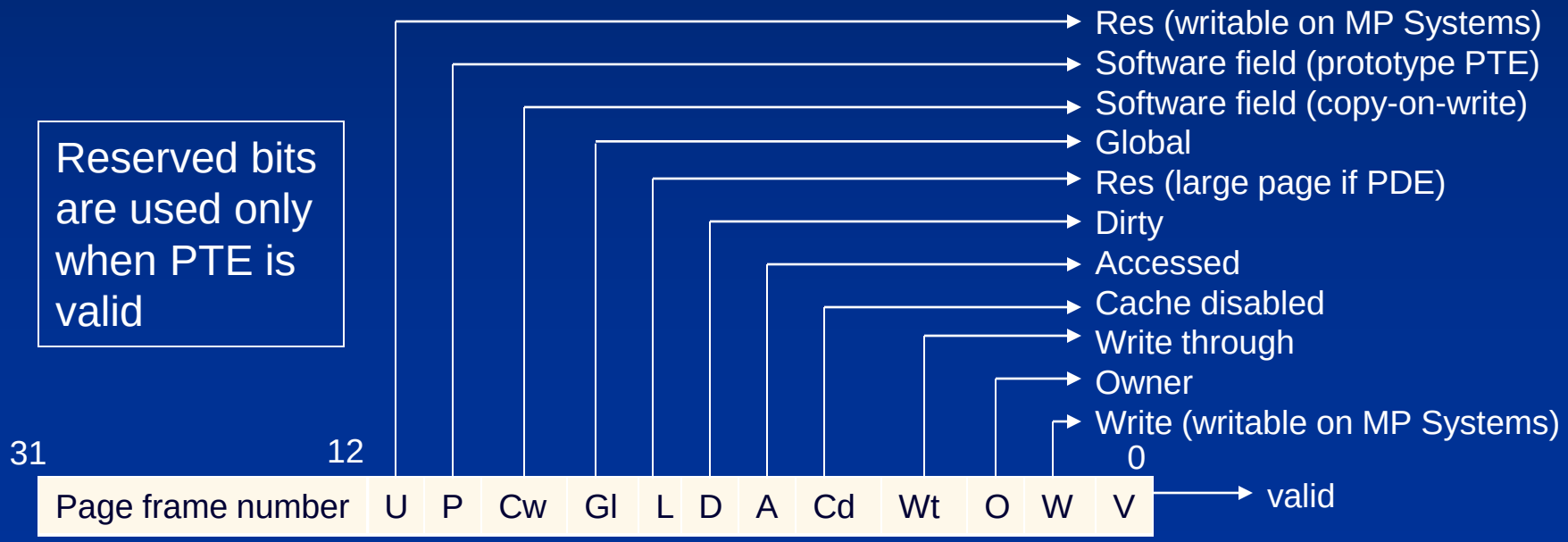
1. Memory management HW locates page directory for current process (CR3 register on Intel)
2. Page directory index directs to requested page table
3. Page table index directs to requested virtual page
4. If page is valid, PTE contains physical page number (PFN – page frame number) of the virtual page
   - Memory manager fault handler locates invalid pages and tries to make them valid
   - Access violation/bug check if page cannot be brought in (protection fault)
5. When PTE points to valid page, byte index is used to locate address of desired data

Page directories & Page tables

- Each process has a single page directory (phys. addr. in KPROCESS block, at 0xC0300000, in cr3 (x86))
  - CR3 is re-loaded on inter-process context switches
  - Page directory is composed of page directory entries (PDEs) which describe state/location of page tables for this process
    - Page tables are created on demand
  - x86: 1024 page tables describe 4GB
- Each process has a private set of page tables
- System has one set of page tables
  - System PTEs are a finite resource: computed at boot time
  - HKLM\System...\Control\SessionManager\SystemPages

Page Table Entries

- Page tables are array of Page Table Entries (PTEs)
- Valid PTEs have two fields:
  - Page Frame Number (PFN)
  - Flags describing state and protection of the page



X86 PTE Status and Protection Bits

Name of Bit	Meaning on x86
Accessed	Page has been read
Cache disabled	Disables caching for that page
Copy-on-write	Page is using copy-on-write
Dirty	Page has been written to
Global	Translation applies to all processes (a translation buffer flush won‘t affect this PTE)
Large page	Indicates that PDE maps a 4MB page (used to map kernel)
Owner	Indicates whether user-mode code can access the page of whether the page is limited to kernel mode access
Prototype	The PTE is a prototype PTE, used as a template to describe shared memory associated with section objects
Valid	Indicates whether translation maps to page in phys. Mem.
Write through	Disables caching of writes; immediate flush to disk
Write	Uniproc: Indicates whether page is read/write or read-only; Multiproc: ind. whether page is writeable/write bit in res. bit

Page Directory and Page Table Entries

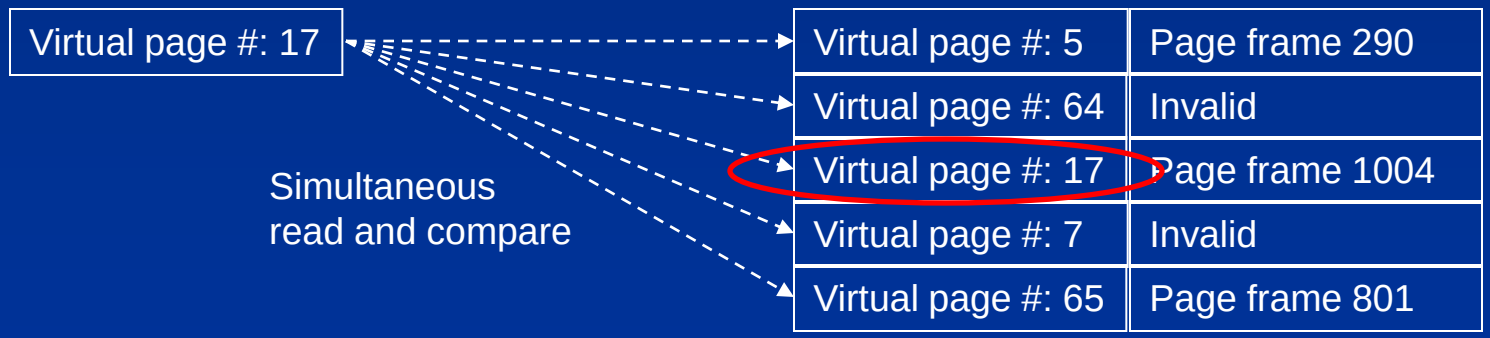
1. virtual address of PD Entry or PT Entry
2. contents of PDE or PTE
3. interpreted contents
4. Page Frame Number (== physical page number) of Page Table
5. Page Frame Number (== physical page number) for valid page
6. D = Dirty (modified since made valid)
7. A = Accessed (recently)
8. KW = Kernel mode writable
9. V = Valid bit
A. Where pager can find contents of an invalid page

```
KDx86> !pte fea80000
FEA80000  - PDE at C0300FE8 (1)   PTE at C03FAA00
          contains 0040C063 (2) contains 0002D063
          pfn 0040C DA--KWV (3) pfn 0002D DA--KWV
                (4)                   (5) (6)(7) (8)(9)

KDx86> !pte 10000
00010000  - PDE at C0300000      PTE at C0000040
          contains 002AF067     contains FFFFF480
          pfn 002AF DA--UWV  (A)  not valid
                                  Proto: VAD
                                  Protect:  4

KDx86> !pte 50000
00050000  - PDE at C0300000      PTE at C0000140
          contains 002AF067     contains 0011A080
          pfn 002AF DA--UWV       not valid
                             (A)  PageFile  0
                                  Offset 11a
                                  Protect:  4
```

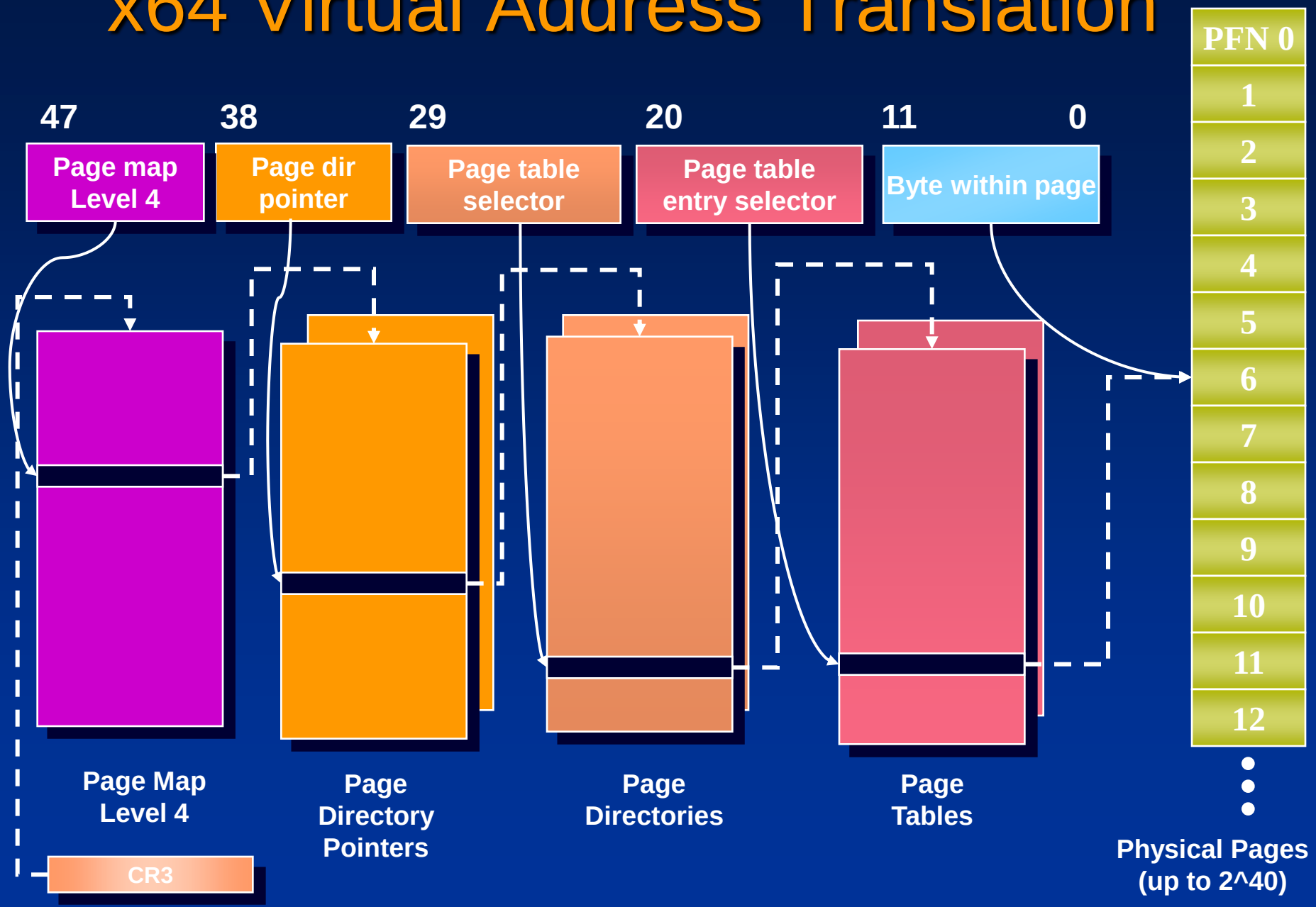
Screen snapshot from:
Kernel debugger !pte command on randomly-selected virtual addresses

Translation Look-Aside Buffer (TLB)

- Address translation requires two lookups:
  - Find right table in page directory
  - Find right entry in page table
- Most CPU *cache* address translations
  - Array of associative memory: translation look-aside buffer (TLB)
  - TLB: virtual-to-physical page mappings of most recently used pages

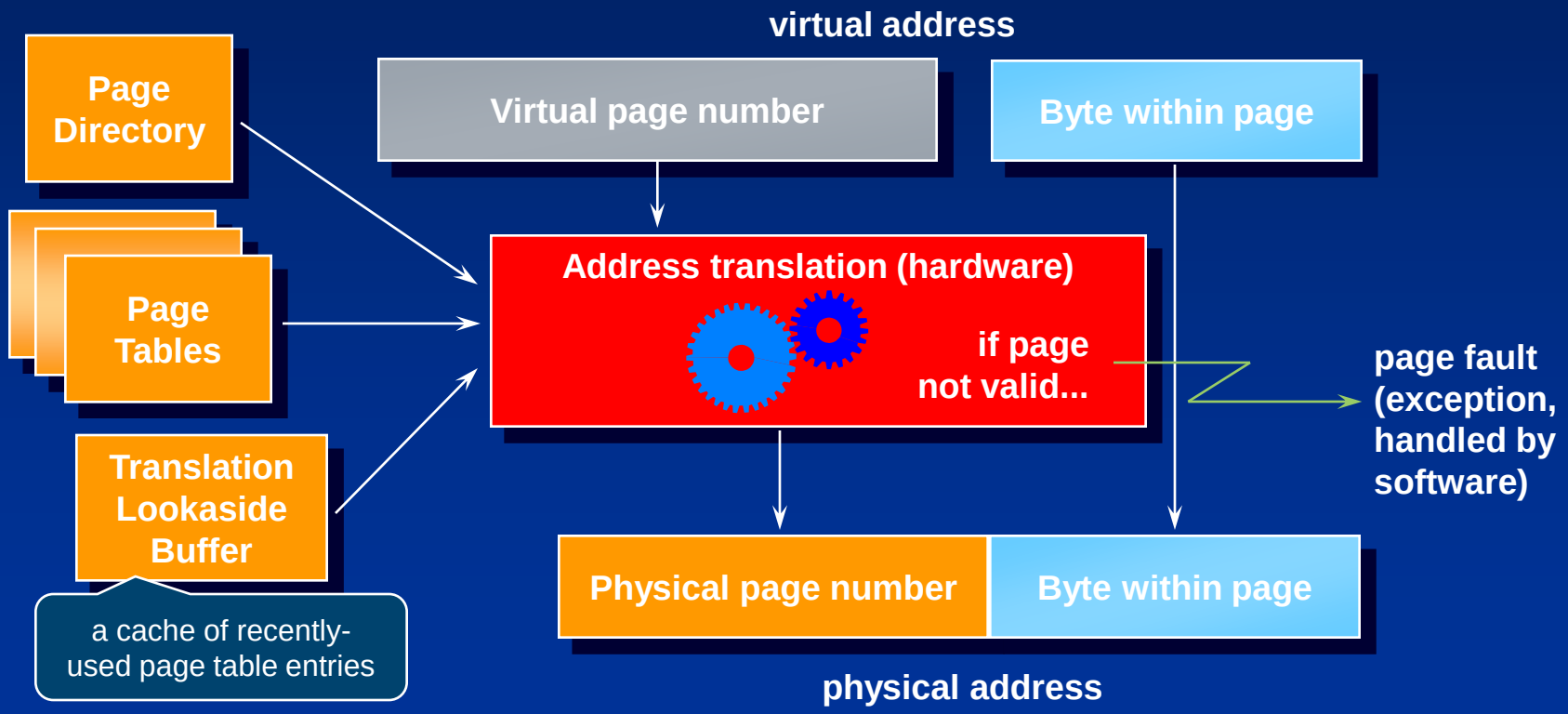


x64 Virtual Address Translation



Virtual Address Translation (Alternative view)

- The hardware converts each valid virtual address to a physical address



Page Fault Handling

- Reference to invalid page is called a page fault
- Kernel trap handler dispatches:
  - Memory manager fault handler (MmAccessFault) called
  - Runs in context of thread that incurred the fault
  - Attempts to resolve the fault or raises exception
- Page faults can be caused by variety of conditions

Reasons for access faults

- Accessing a page that's not in memory but on disk in a page file or a mapped file
  - Allocate a physical page, and read the desired page from disk to the relevant working set
- Accessing page that is on standby or modified list
  - Transfer the page to process or system working set
- Accessing page that has no committed storage
  - Access violation
- Accessing kernel page from user-mode
  - Access violation
- Writing to a read-only page
  - Access violation
- Accessing a demand-zero page
  - Add a zero-filled page to the relevant working set

Reasons for access faults (contd.)

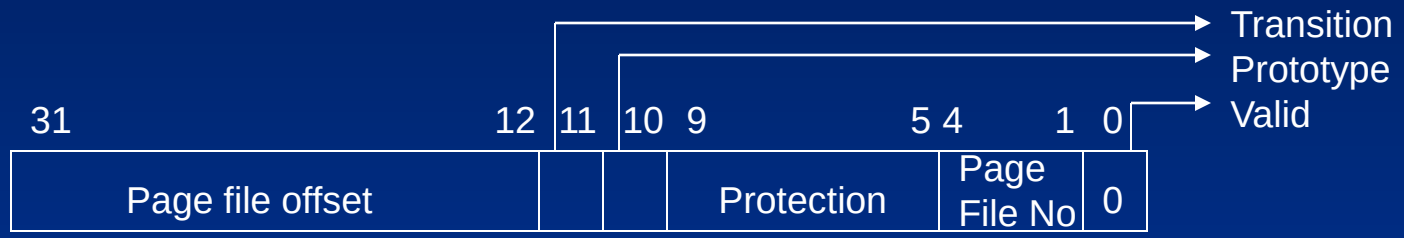
- Writing to a guard page
  - Guard page violation (if a reference to a user-mode stack, perform automatic stack expansion)
- Writing to a copy-on-write page
  - Make process-private copy of page and replace original in process or system working set
- Referencing a page in system space that is valid but not in the process page directory
  (if paged pool expanded after process directory was created)
  - Copy page directory entry from master system page directory structure and dismiss exception
- Writing to valid page that has not yet been written to
  - Set dirty bit in PTE

In-Paging I/O due to Access Faults

- Occurs when read operation must be issued to a file to satisfy page fault
  - Page tables are pageable → additional page faults possible
- In-page I/O is synchronous
  - Thread waits until I/O completes
  - Not interruptible by asynchronous procedure calls
- During in-page I/O: faulting thread does not own critical memory management synchronization objects
- Other threads in process may issue VM functions, but:
  - Another thread could have faulted same page: collided page fault
  - Page could have been deleted (remapped) from virtual address space
  - Protection on page may have changed
  - Fault could have been for prototype PTE and page that maps prototype PTE could have been out of working set

Invalid PTEs and their structure

- **Page file**: desired page resides in paging file
  in-page operation is initiated

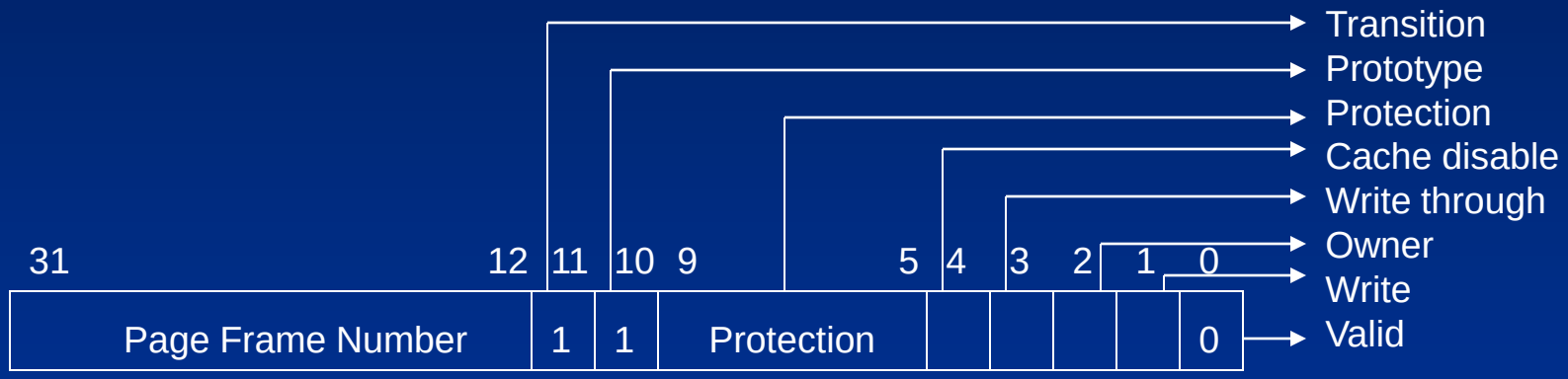


- **Demand Zero:** pager looks at zero page list;
  if list is empty, pager takes list from standby list and zeros it;

  PTE format as shown above, but page file number and offset are zeros

Invalid PTEs and their structure (contd.)

- **Transition**: the desired page is in memory on either the standby, modified, or modified-no-write list
  - Page is removed from the list and added to working set



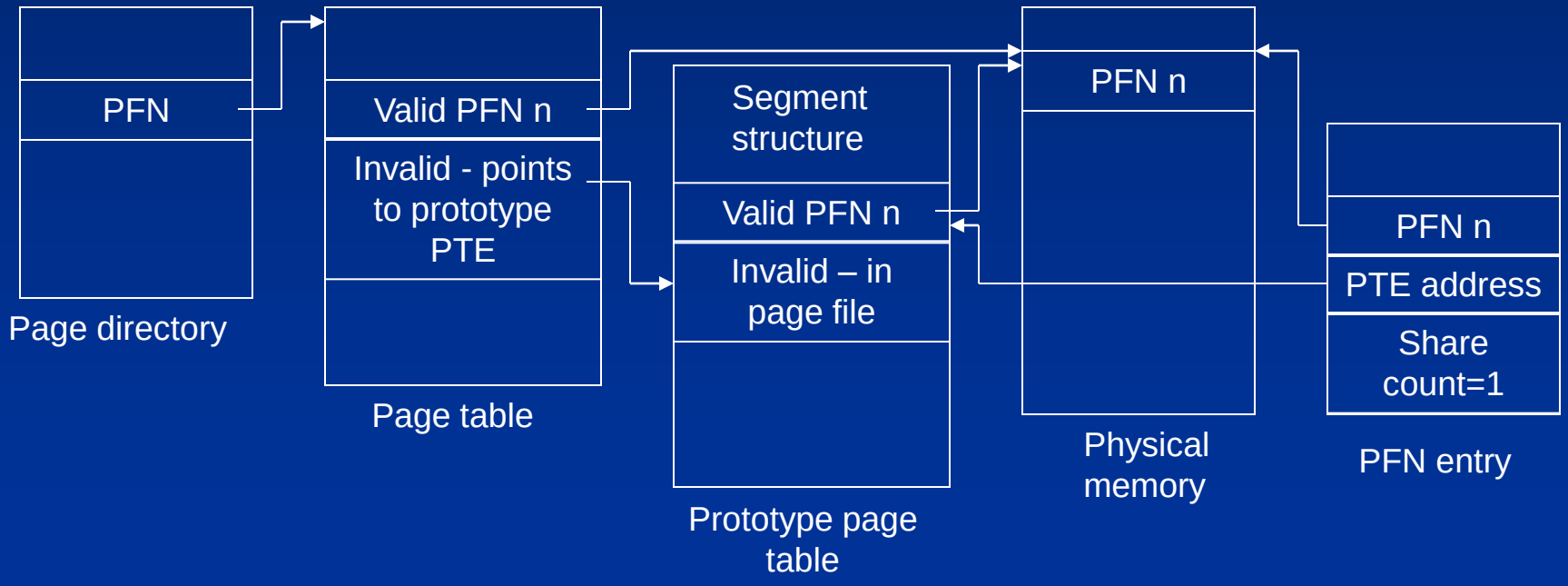
- **Unknown: the PTE is zero, or the page table does not yet exist**
  - examine virtual address space descriptors (VADs) to see whether this virtual address has been reserved
  - Build page tables to represent newly committed space

Prototype PTEs

- Software structure to manage potentially shared pages
  - Array of prototype PTEs is created as part of section object (part of *segment* structure)
  - First access of a page mapped to a view of a section object: memory manager uses prototype PTE to fill in real PTE used for address translation;
  - Reference count for shared pages in PFN database
- Shared page valid:
  - process & prototype PTE point to physical page
- Page invalidated:
  - process PTE points to prototype PTE
- Prototype PTE describes 5 states for shared page:
  - *Active/valid, Transition, Demand zero, Page file, Mapped file*
- Layer between page table and page frame database

Prototype PTEs for shared pages – the bigger picture

- Two virtual pages in a mapped view
- First page is valid; 2nd page is invalid and in page file
  - Prototype PTE contains exact location
  - Process PTE points to prototype PTE



Managing Physical Memory

- System keeps unassigned physical pages on one of several lists
  - Free page list
  - Modified page list
  - Standby page list
  - Zero page list
  - Bad page list - pages that failed memory test at system startup
- Lists are implemented by entries in the “PFN database”
  - Maintained as FIFO lists or queues

Paging Dynamics

- New pages are allocated to working sets from the top of the free or zero page list
- Pages released from the working set due to working set replacement go to the bottom of:
  - The modified page list (if they were modified while in the working set)
  - The standby page list (if not modified)
    - Decision made based on “D” (dirty = modified) bit in page table entry
  - Association between the process and the physical page is still maintained while the page is on either of these lists

Standby and Modified Page Lists

- Modified pages go to modified (dirty) list
  - Avoids writing pages back to disk too soon
- Unmodified pages go to standby (clean) list
- They form a system-wide cache of “pages likely to be needed again”
  - Pages can be faulted back into a process from the standby and modified page list
  - These are counted as page faults, but not page reads

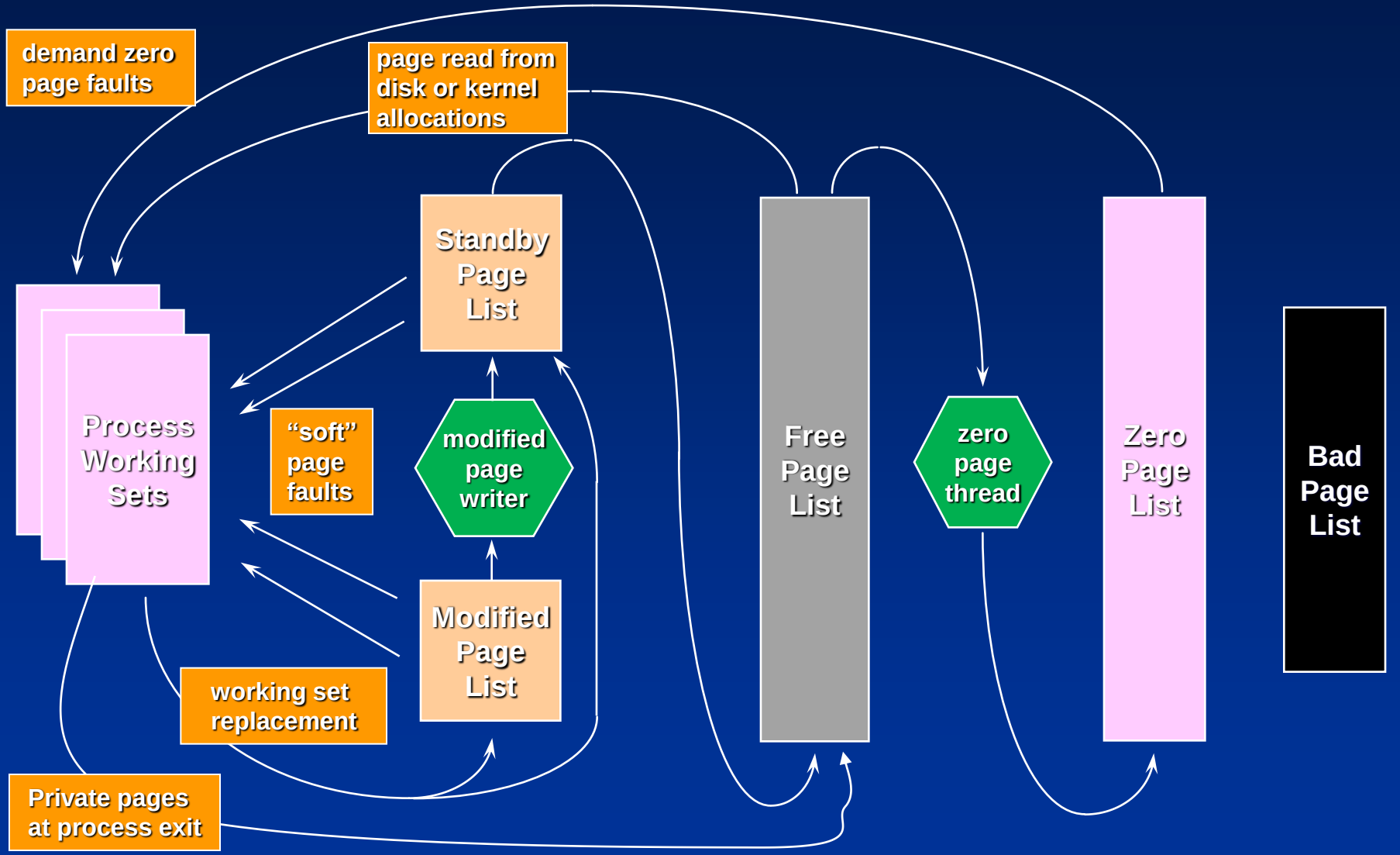
Modified Page Writer

- When modified list reaches certain size, modified page writer system thread is awoken to write pages out
  - See MmModifiedPageMaximum
  - Also triggered when memory is overcomitted (too few free pages)
  - Does not flush entire modified page list
- Two system threads
  - One for mapped files, one for the paging file
- At the same time, pages move from the modified list to the standby list
  - E.g. they can still be soft faulted into a working set

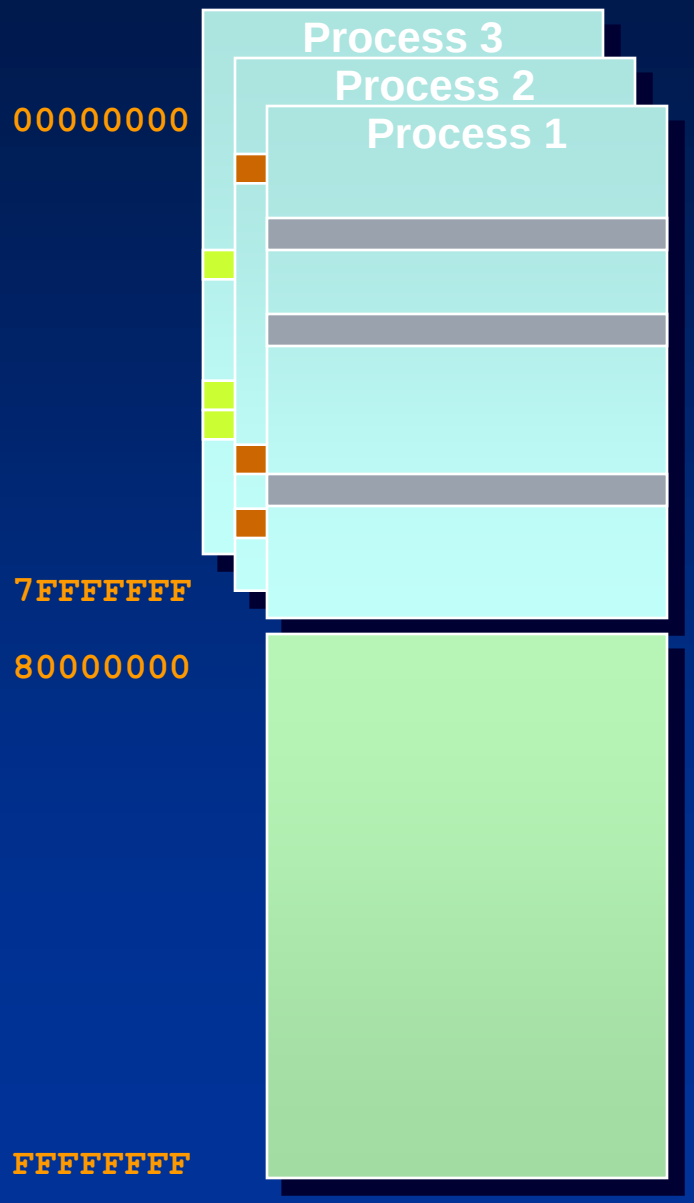
Free and Zero Page Lists

- Free Page List
  - Used for page reads
  - Private modified pages go here on process exit
  - Pages contain junk in them (e.g. not zeroed)
  - On most busy systems, this list is empty
- Zero Page List
  - Used to satisfy demand zero page faults
    - References to private pages that have not been created yet
  - When free page list has 8 or more pages, a priority zero thread is awoken to zero them
  - On most busy systems, this list is empty too

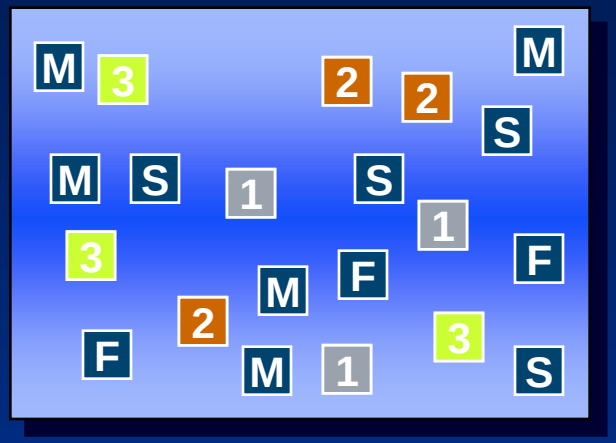
Paging Dynamics



Working Sets in Memory



Pages in Physical Memory



- As processes incur page faults, pages are removed from the free, modified, or standby lists and made part of the process working set
- A shared page may be resident in several processes' working sets at one time (this case not illustrated here)

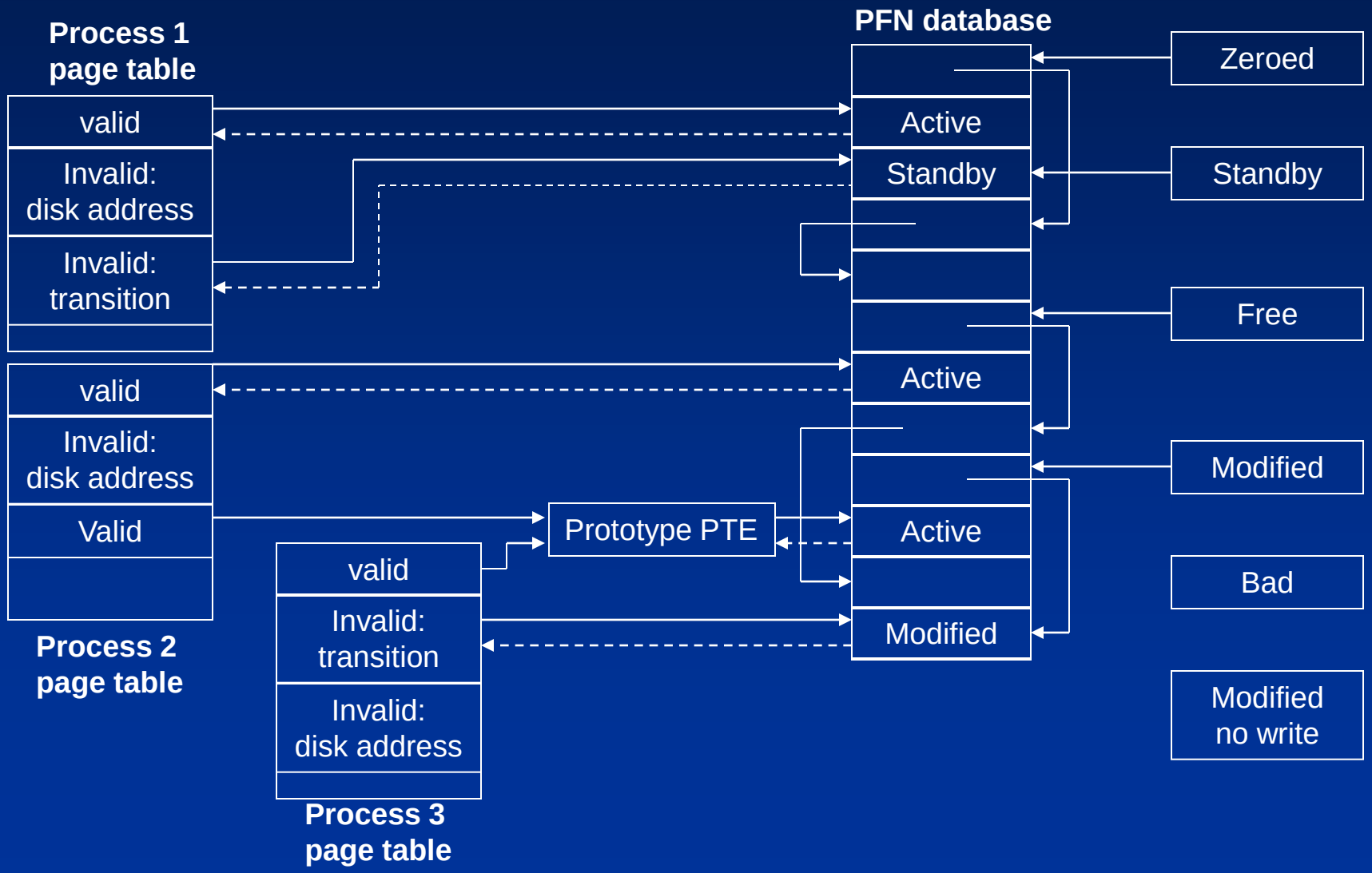
Page Frame Number Database

- One entry (24 bytes) for each physical page
  - Describes state of each page in physical memory
- Entries for active/valid and transition pages contain:
  - Original PTE value (to restore when paged out)
  - Original PTE virtual address and container PFN
  - Working set index hint (for the first process...)
- Entries for other pages are linked in:
  - Free, standby, modified, zeroed, bad lists (parity error will kill kernel)
- Share count (active/valid pages):
  - Number of PTEs which refer to that page; count → 0: candidate for free list
- Reference count:
  - INC when first added to working set or locked in memory for I/O; DEC when share count → 0 or unlocked
  - Share count = 0 & reference count = 1 is possible
  - Reference count → 0: page is inserted in free, standby or modified lists

Page Frame Database – states of pages in physical memory

Status	Description
Active/valid	Page is part of working set (sys/proc), valid PTE points to it
Transition	Page not owned by a working set, not on any paging list I/O is in progress on this page
Standby	Page belonged to a working set but was removed; not modified
Modified	Removed from working set, modified, not yet written to disk
Modified no write	Modified page, except modified page writer won't write to disk; e.g. used by NTFS for protected pages (explicit flushing)
Free	Page is free but has dirty data in it – cannot be given to user process without "zeroing" -- security requirement
Zeroed	Page is free and has been initialized by zero page thread
Bad	Page has generated parity or other hardware errors

Page tables and page frame database



Page Files

- What gets sent to the paging file?
  - Not code – only ***modified data*** (code can be re-read from image file anytime)
- When do pages get paged out?
  - Only when necessary
  - Page file space is only reserved at the time pages are written out
  - Once a page is written to the paging file, the space is occupied until the memory is deleted (e.g. at process exit), even if the page is read back from disk
- Page file maximums:
  - 16 page files per system
  - 32-bit x86: 4095MB
  - 32-bit PAE mode, 64-bit systems: 16 TB

Why Use Page File on Systems with Ample Free Memory?

- Because memory manager doesn't let process working sets grow arbitrarily
  - Processes are not allowed to expand to fill available memory (previously described)
    - Bias is to keep free pages for new or expanding processes
  - This will cause page file usage early in the system life even with ample memory free
- We talked about the standby list, but there is another list of modified pages recently removed from working sets
  - Modified private pages are held in memory in case the process asks for it back
  - When the list of modified pages reaches a certain threshold, the memory manager writes them to the paging file (or mapped file)
  - Pages are moved to the standby list, since they are still "valid" and could be requested again

Sizing the Page File

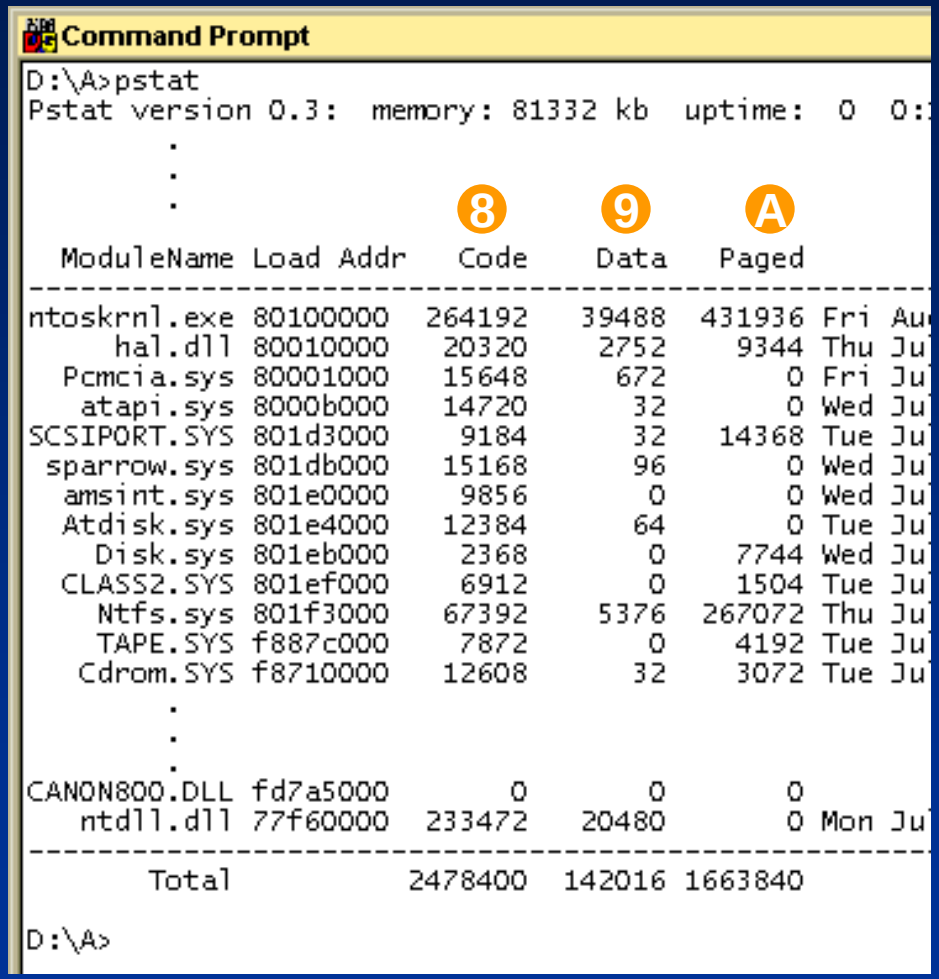
- Given understanding of page file usage, how big should the total paging file space be?
  - (Windows supports multiple paging files)
- Size should depend on total private virtual memory used by applications and drivers
  - Therefore, not related to RAM size (except for taking a full memory dump)
- Worst case: system has to page all private data out to make room for code pages
  - To handle, minimum size should be the maximum of VM usage ("Commit Charge Peak")
  - Hard disk space is cheap, so why not double this
  - Page file too large: doesn't change system performance
  - Page file too small: *"system run low on virtual memory!"*

When Page Files are Full

- When page file space runs low
  - 1. “System running low on virtual memory”
    - First time: Before pagefile expansion
    - Second time: When committed bytes reaching commit limit
  - 2. “System out of virtual memory”
    - Page files are full
- Look for who is consuming pagefile space:
  - Process memory leak: Check Task Manager, Processes tab, VM Size column
    - or Perfmon “private bytes”, same counter
  - Paged pool leak: Check paged pool size
    - Run poolmon to see what object(s) are filling pool
    - Could be a result of processes not closing handles - check process “handle count” in Task Manager

System Nonpaged Memory

- Nonpageable components:
  - Nonpageable parts of NtosKrnl.Exe, drivers
  - Nonpaged pool (see PerfMon, Memory object: Pool nonpaged bytes)
  - 8 non-paged code
  - 9 non-paged data
  - A pageable code+data (virtual size)
  - output of “drivers.exe” is similar
  - Win32K.Sys is paged, even though it shows up as nonpaged
  - Other drivers might do this too, so total nonpaged size is not really visible



Optimizing Applications
Minimizing Page Faults

- **Some page faults are unavoidable**
  - code is brought into physical memory (from .EXEs and .DLLs) via page faults
  - the file system cache reads data from cached files in response to page faults
- **First concern is to minimize number of “hard” page faults**
  - i.e. page faults to disk (page reads)
  - see Performance Monitor, “Memory” object, “page faults” vs. “page reads” (this is system-wide, not per process)
  - for an individual app, see Page Fault Monitor:

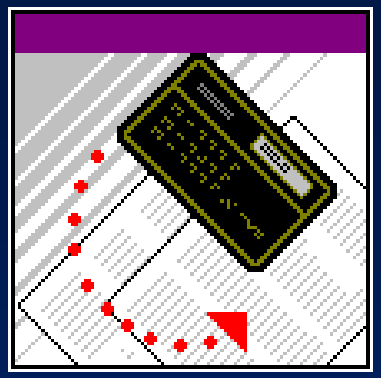
```
c:\> pfmon /?
c:\> pfmon program-to-be-monitored
```

  - note that these results are highly dependent on system load (physical memory usage by other apps)

Lab Demos

- View memory mapped files
- Determining the maximum pool sizes
- Viewing the system look-aside lists
- Checking large address aware
- Analyzing user virtual address space
- Translating addresses
- Viewing VADs, section objects, PFN databases/entries, working sets
- Modified page writer: Perfmon shows available bytes goes up once per second after apps are closed

Lab: Accounting for Physical Memory Usage



- Process working sets
  - Perfmon: Process / Working set
  - Note, shared resident pages are counted in the process working set of every process that's faulted them in
  - Hence, the total of all of these may be greater than physical memory
- Resident system code (NTOSKRNL + drivers, including win32k.sys & graphics drivers)
  - see total displayed by !drivers 1 command in kernel debugger
- Nonpageable pool
  - Perfmon: Memory / Pool nonpaged bytes
- Free, zero, and standby page lists
  - Perfmon: Memory / Available bytes
- Pageable, but currently-resident, system-space memory
  - Perfmon: Memory / Pool paged resident bytes
  - Perfmon: Memory / System cache resident bytes

  Memory | Cache bytes counter is really total of these four "resident" (physical) counters
- Modified, Bad page lists
  - can only see size of these with !memusage command in Kernel Debugger

Further Reading

- Mark E. Russinovich *et al*. Windows Internals, 5th Edition, Microsoft Press, 2009.
- Chapter 9 - Memory Management
  - Virtual address layout)from pp. 736)
  - Address Translation (from pp. 761)
  - Page fault handing (from pp. 774)
  - Page frame number database (from pp. 803)

Source Code References

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