Windows Kernel Internals
Virtual Memory Manager

David B. Probert, Ph.D.
Windows Kernel Development
Microsoft Corporation

Virtual Memory Manager
Features

- Provides 4 GB flat virtual address space (IA32)
- Manages process address space
- Handles pagefaults
- Manages process working sets
- Manages physical memory
- Provides memory-mapped files
- Allows pages shared between processes
- Facilities for I/O subsystem and device drivers
- Supports file system cache manager

Virtual Memory Manager

Features

- Provide session space for Win32 GUI applications
- Address Windowing Extensions (physical overlays)
- Address space cloning (posix/fork() support)
- Kernel-mode memory heap allocator (pool)
  - Paged Pool, Non-paged pool, Special pool/verifier

Virtual Memory Manager

Windows Server 2003 enhancements

- Support for Large (4MB) page mappings
- Improved TB performance, remove ContextSwap lock
- On-demand proto-PTE allocation for mapped files
- Other performance & scalability improvements
- **Support for IA64 and Amd64 processors**

Virtual Memory Manager

NT Internal APIs

NtCreatePagingFile

NtAllocateVirtualMemory (Proc, Addr, Size, Type, Prot)

- Process: handle to a process
- Protection: NOACCESS, EXECUTE, READONLY, READWRITE, NOCACHE
- Flags: COMMIT, RESERVE, PHYSICAL, TOP_DOWN, RESET, LARGE_PAGES, WRITE_WATCH

NtFreeVirtualMemory(Process, Address, Size, FreeType)

- FreeType: DECOMMIT or RELEASE

NtQueryVirtualMemory, NtProtectVirtualMemory

Virtual Memory Manager

NT Internal APIs

Pagefault

NtLockVirtualMemory, NtUnlockVirtualMemory

- locks a region of pages within the working set list
- requires PROCESS_VM_OPERATION on target process and SeLockMemoryPrivilege

NtReadVirtualMemory, NtWriteVirtualMemory (

Proc, Addr, Buffer, Size)

NtFlushVirtualMemory

Virtual Memory Manager

NT Internal APIs

NtCreateSection

- **creates a section but does not map it**

NtOpenSection

- opens an existing section

NtQuerySection

- query attributes for section

NtExtendSection

NtMapViewOfSection (Sect, Proc, Addr, Size, …)

NtUnmapViewOfSection

Virtual Memory Manager

NT Internal APIs

APIs to support AWE (Address Windowing Extensions)

- Private memory only
- Map only in current process
- Requires LOCK_VM privilege

NtAllocateUserPhysicalPages (Proc, NPages, &PFNs[])

NtMapUserPhysicalPages (Addr, NPages, PFNs[])

NtMapUserPhysicalPagesScatter

NtFreeUserPhysicalPages (Proc, &NPages, PFNs[])

NtResetWriteWatch

NtGetWriteWatch

Read out dirty bits for a section of memory since last reset

NtAllocateVm flags

MEM_RESERVE – Only virtual address alloc

MEM_COMMIT – Physical alloc too

MEM_TOP_DOWN – Alloc VA at highest available (subject to addr mask)

MEM_RESET – Discard pagefile space

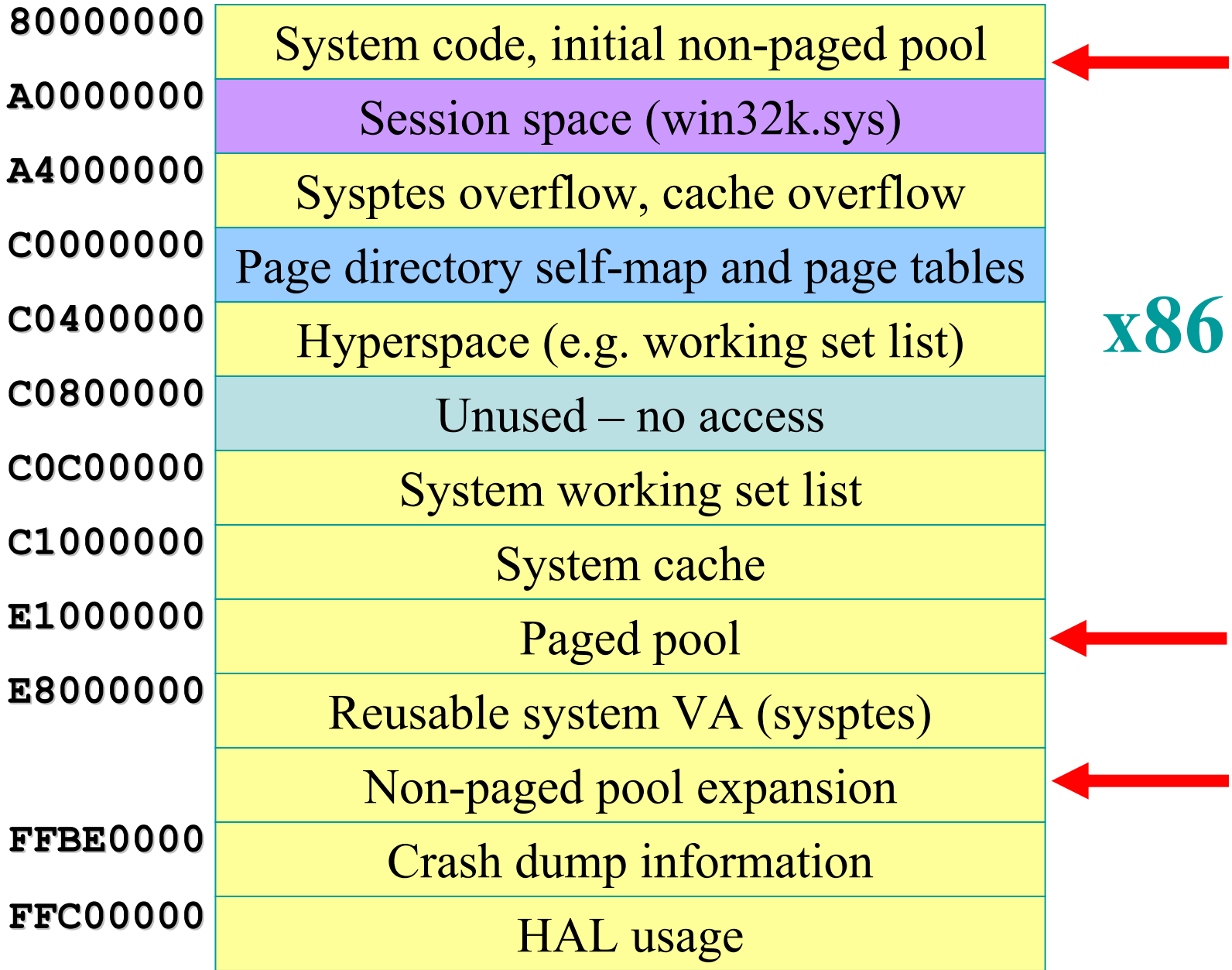
MEM_PHYSICAL – For AWE

MEM_LARGE_PAGES – 4MB Pages

MEM_WRITE_WATCH – Used for Write-Watch

Allocating kernel memory (pool)

- Tightest x86 system resource is KVA
  - Kernel Virtual Address space
- Pool allocates in small chunks:
  - < 4KB: 8B granulariy
  - >= 4KB: page granularity
- Paged and Non-paged pool
  - Paged pool backed by pagefile
- Special pool used to find corruptors
- Lots of support for debugging/diagnosis

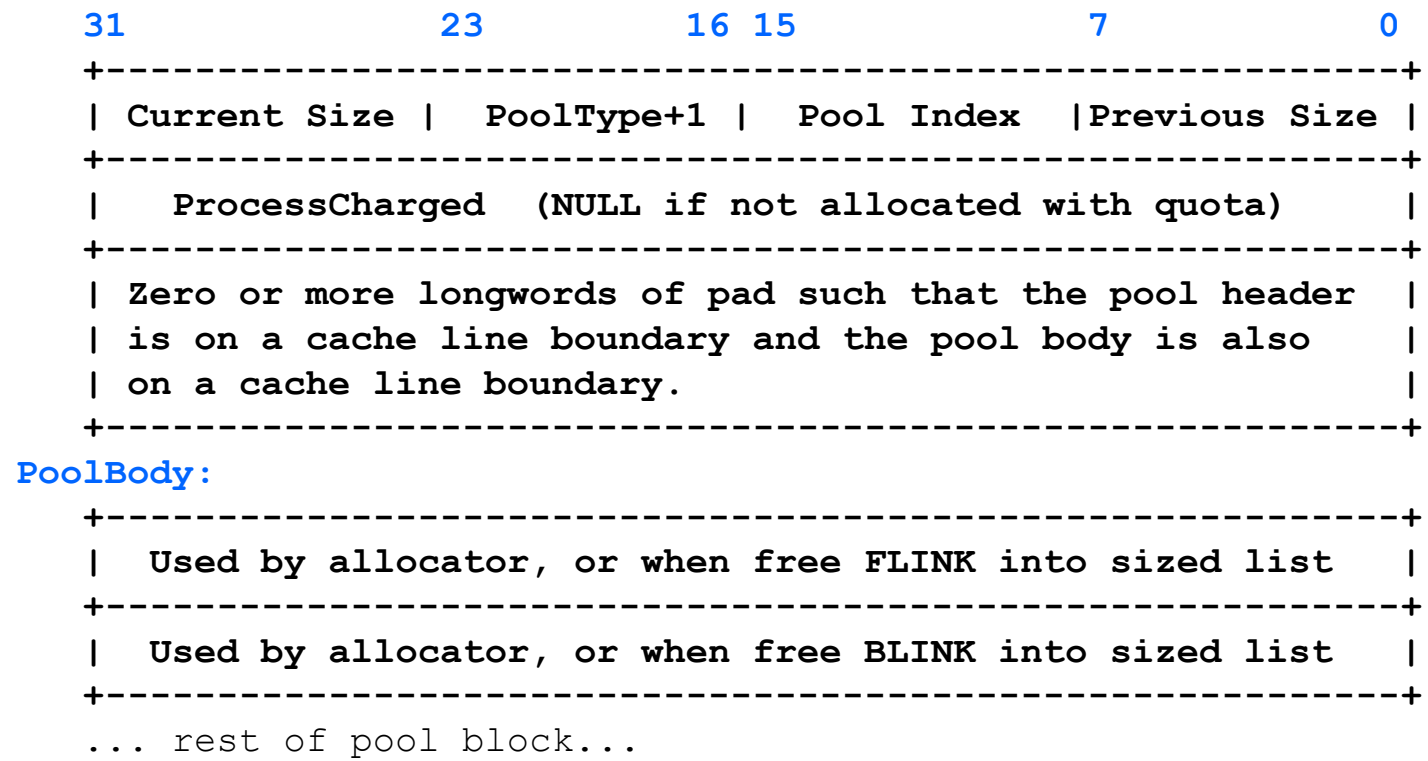


Looking at a pool page

```
kd> !pool e1001050
 e1001000 size: 40 prev size:  0  (Allocated)  MmDT
 e1001040 size: 10 prev size: 40  (Free)       Mm
*e1001050 size: 10 prev size: 10  (Allocated) *ObDi
 e1001060 size: 10 prev size: 10  (Allocated)  ObDi
 e1001070 size: 10 prev size: 10  (Allocated)  Symt
 e1001080 size: 40 prev size: 10  (Allocated)  ObDm
 e10010c0 size: 10 prev size: 40  (Allocated)  ObDi
```

MmDT - nt!mm - Mm debug
Mm - nt!mm - general Mm Allocations
ObDi - nt!ob - object directory
Symt - <unknown> - Symbolic link target strings
ObDm - nt!ob - object device map

Layout of pool headers



Size fields of pool headers expressed in units of smallest pool block size.

Managing memory for I/O

Memory Descriptor Lists (MDL)

- Describes pages in a buffer in terms of physical pages

```
typedef struct _MDL {
    struct _MDL *Next;
    CSHORT Size;
    CSHORT MdlFlags;
    struct _EPROCESS *Process;
    PVOID MappedSystemVa;
    PVOID StartVa;
    ULONG ByteCount;
    ULONG ByteOffset;
} MDL, *PMDL;
```

MDL flags

```
MDL_MAPPED_TO_SYSTEM_VA      0x0001
MDL_PAGES_LOCKED             0x0002
MDL_SOURCE_IS_NONPAGED_POOL  0x0004
MDL_ALLOCATED_FIXED_SIZE     0x0008
MDL_PARTIAL                  0x0010
MDL_PARTIAL_HAS_BEEN_MAPPED  0x0020
MDL_IO_PAGE_READ             0x0040
MDL_WRITE_OPERATION          0x0080
MDL_PARENT_MAPPED_SYSTEM_VA  0x0100
MDL_FREE_EXTRA_PTES          0x0200
MDL_DESCRIBES_AWE            0x0400
MDL_IO_SPACE                 0x0800
MDL_NETWORK_HEADER           0x1000
MDL_MAPPING_CAN_FAIL         0x2000
MDL_ALLOCATED_MUST_SUCCEED   0x4000
```

some MDL Operations

MmAllocatePagesForMdl

- Search the PFN database for free, zeroed or standby pages
- Allocates pages and puts in MDL
- Does not map pages (caller responsibility)
- Designed to be used by an AGP driver

MmProbeAndLockPages

- Probes the specified pages
- Makes the pages resident
- Locks the physical pages
- MDL list updated to describe the physical pages

MmUnlockPages

- Unlocks the physical pages

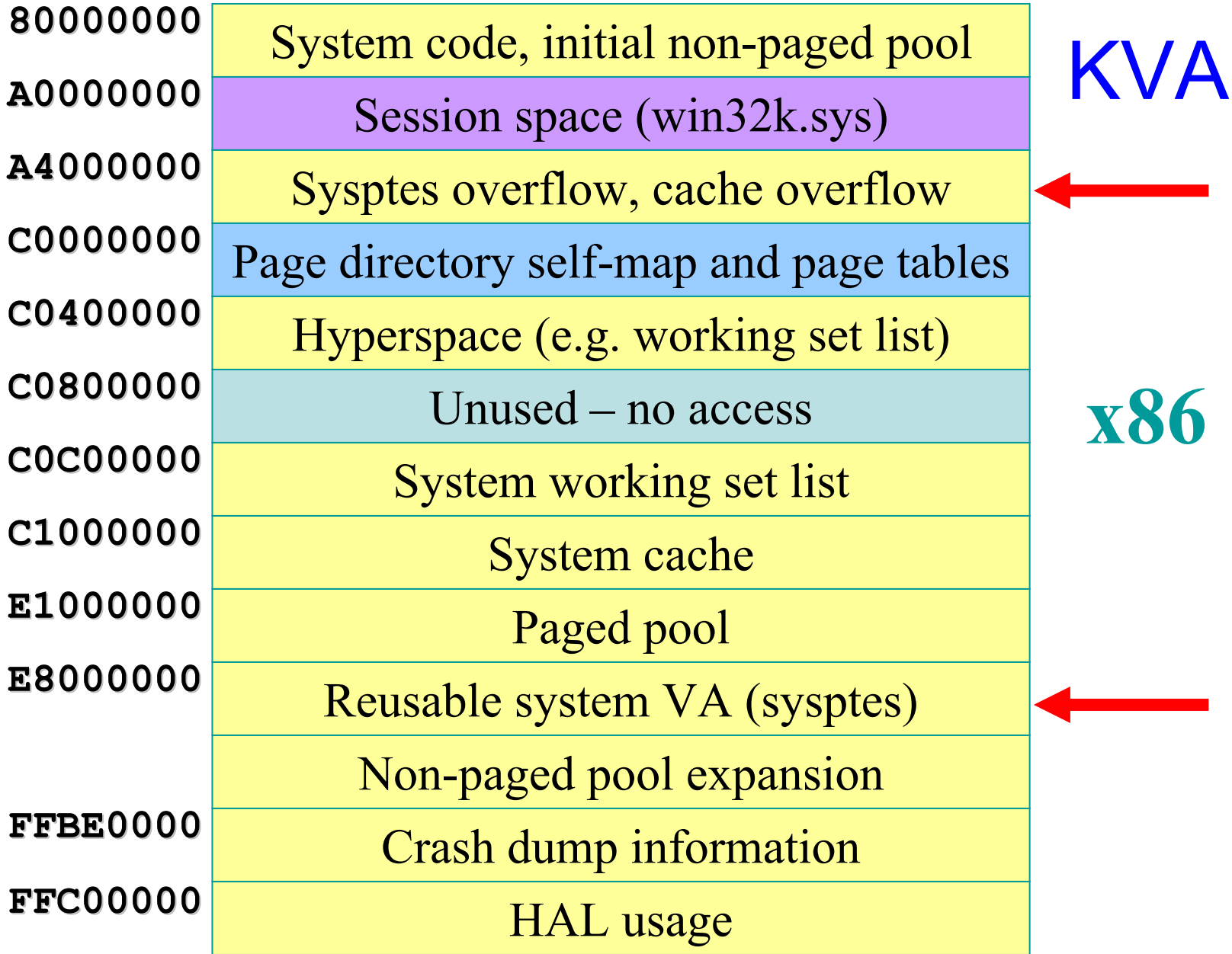
some MDL Operations

MmBuildMdlForNonPagedPool

- Like MmProbeAndLockPages, but…
- No lock (and corresponding unlock)

MmMapLockedPages

- Maps physical pages into system or user virtual addresses



Sysptes

Used to manage random use of kernel virtual memory, e.g. by device drivers.

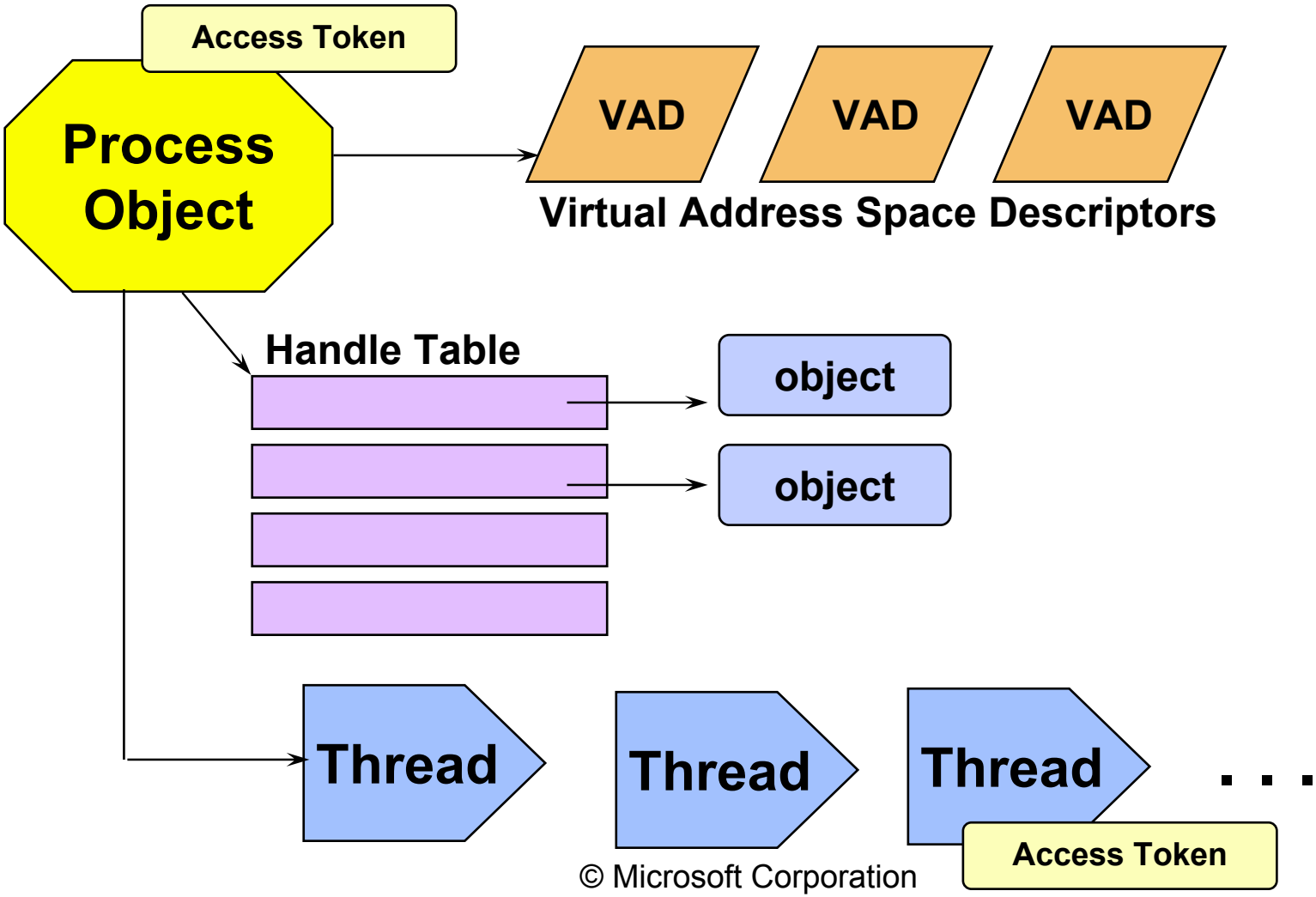
Kernel implements functions like:

- MiReserveSystemPtes (n, type)
- MiMapLockedPagesInUserSpace

  (mdl, virtaddr, cachetype,basevirtaddr)

Often a critical resource!

Processes & Threads



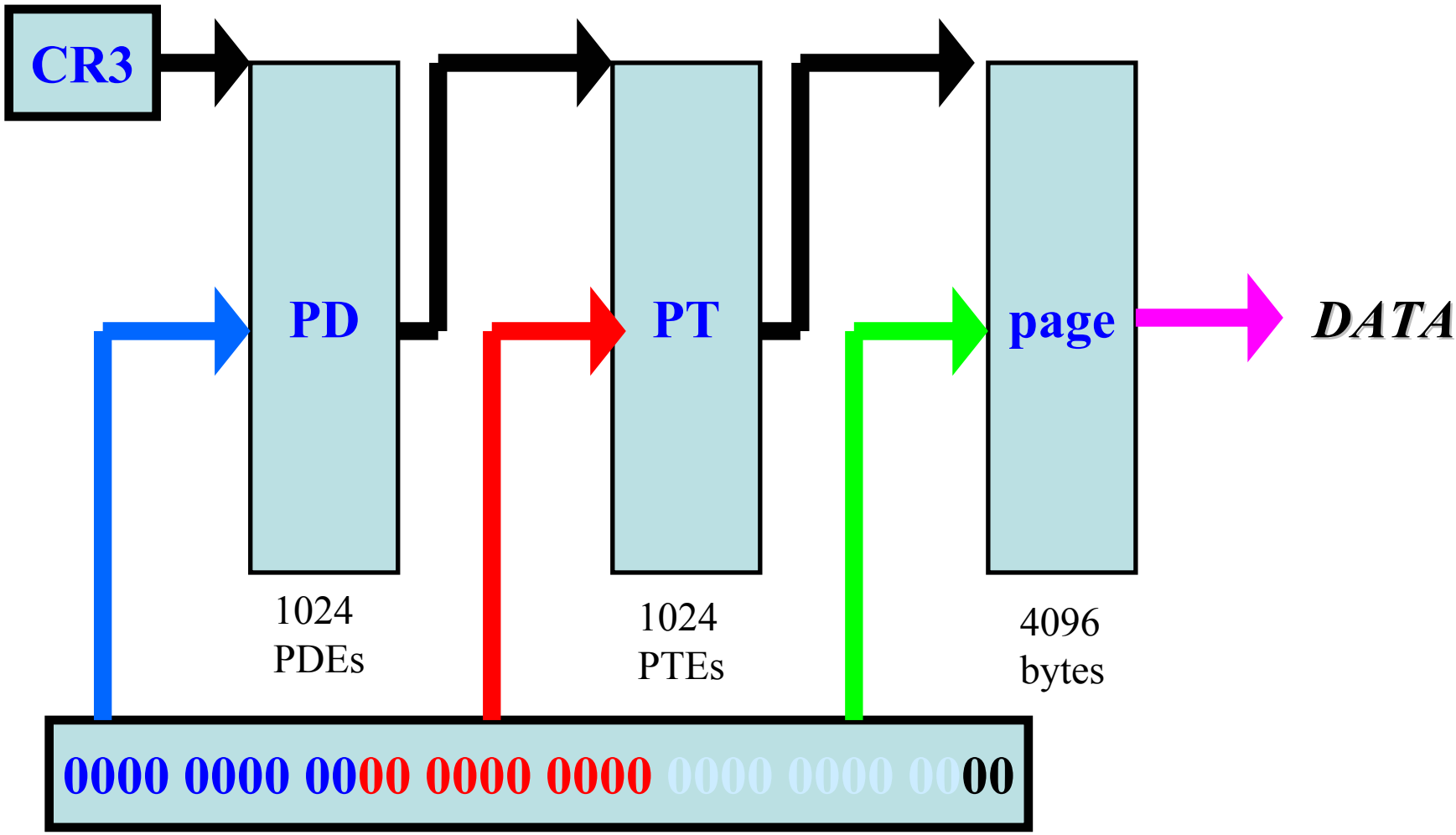
Each process has its own…

- Virtual address space (including program global storage, heap storage, threads' stacks)
  - processes cannot corrupt each other's address space by mistake
- Working set (physical memory "owned" by the process)
- Access token (includes security identifiers)
- Handle table for Win32 kernel objects
- These are common to all threads in the process, but separate and protected between processes

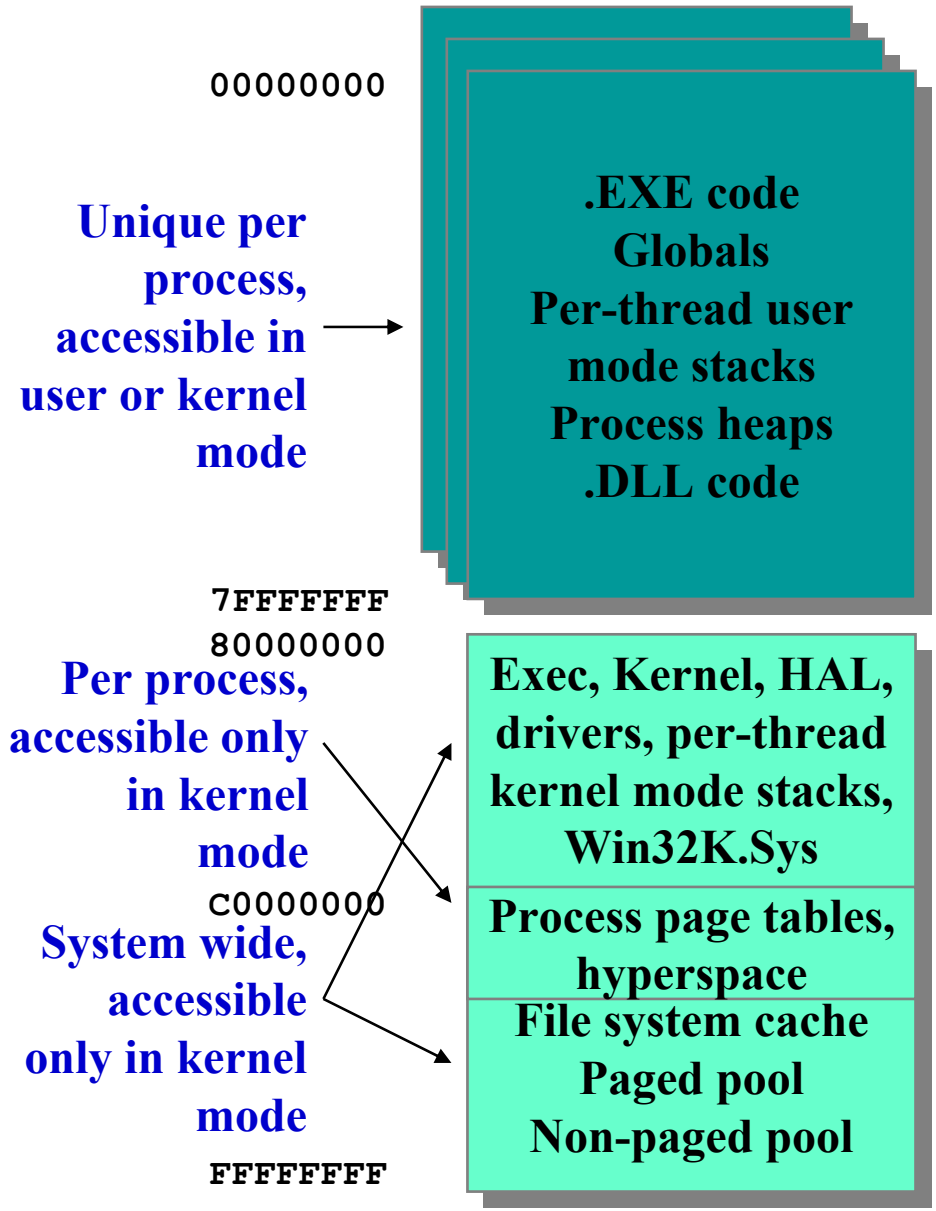
Each thread has its own…

- Stack (automatic storage, call frames, etc.)
- Instance of a top-level function
- Scheduling state (Wait, Ready, Running, etc.) and priority
- Current access mode (user mode or kernel mode)
- Saved CPU state if it isn't Running
- Access token (optional -- overrides process's if present)

Virtual Address Translation

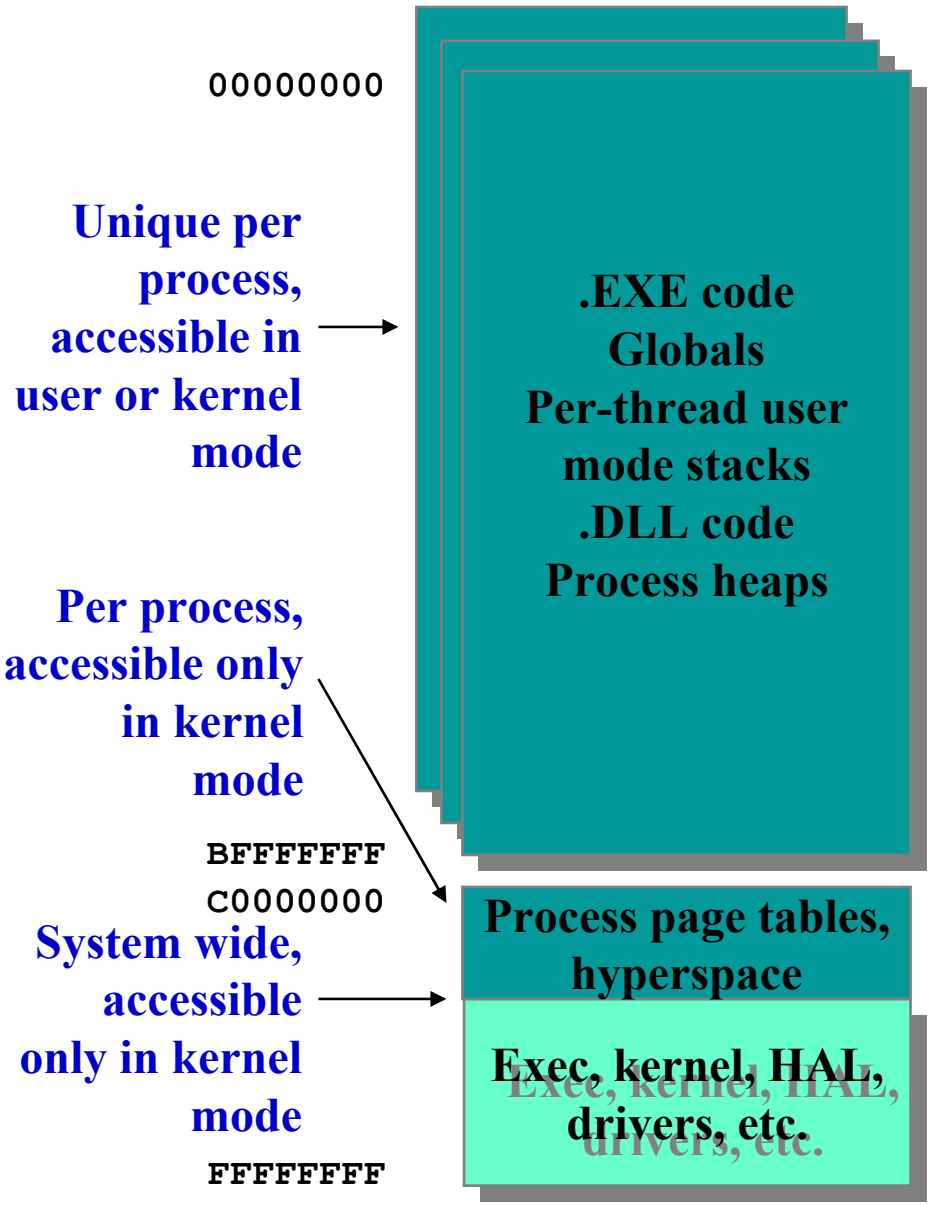


32-bit Virtual Address Space



- 2 GB per-process
  - Address space of one process is not directly reachable from other processes
- 2 GB systemwide
  - The operating system is loaded here, and appears in every process's address space
  - There is no process for "the operating system" (though there are processes that do things for the OS, more or less in "background")

/3GB Option

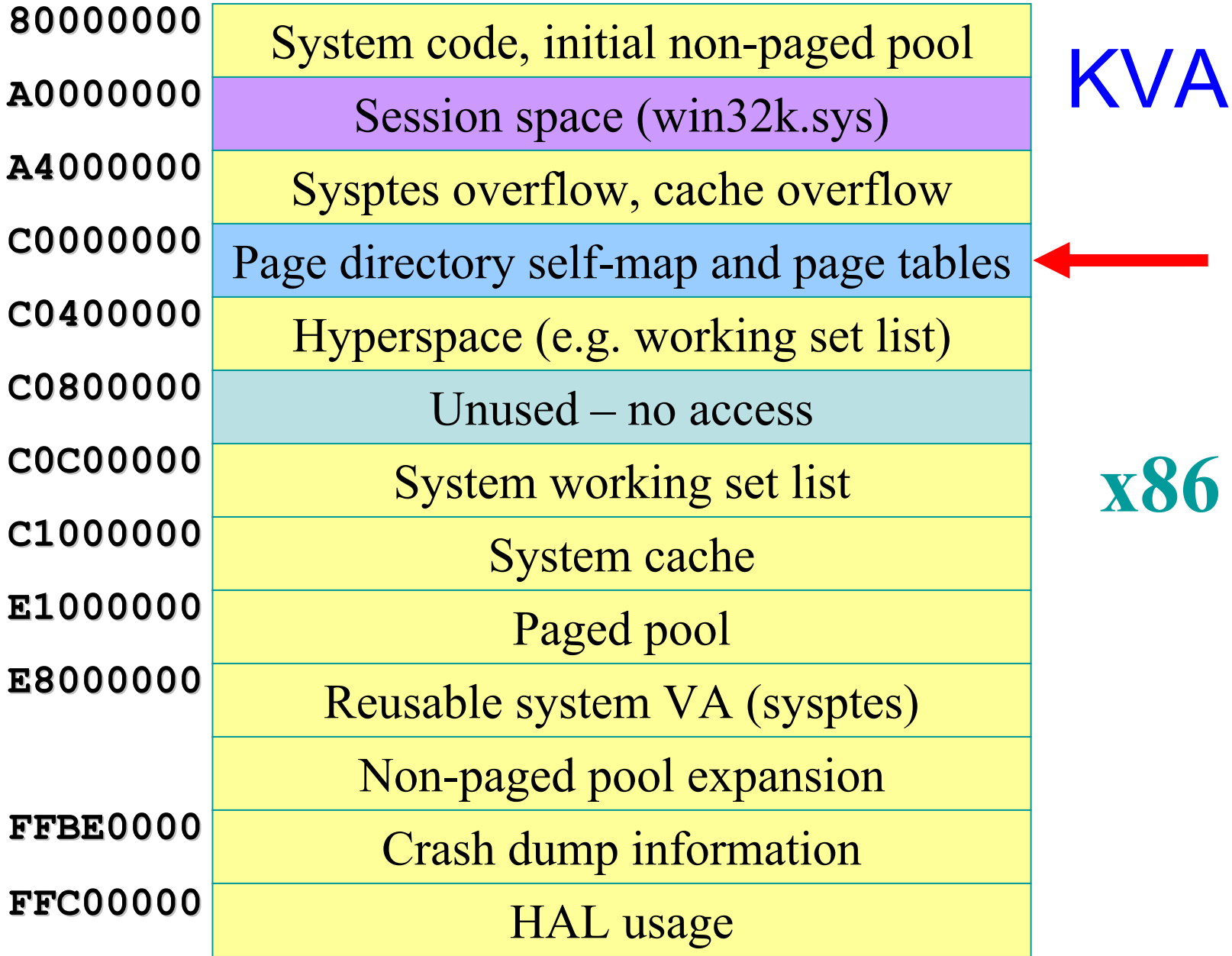


- Available on Advanced and Enterprise Servers
- Provides 3 GB per-process address space
  - Commonly used by database servers
  - .EXE must have "large address space aware" flag in image header, or they're limited to 2 GB (specify at link time or with ¥support¥debug¥imagecfg.exe)
  - Chief "loser" in system space is file system cache
  - Better solution: Address Window Extensions (AWE)
  - Even better solution: IA64

Self-mapping page tables

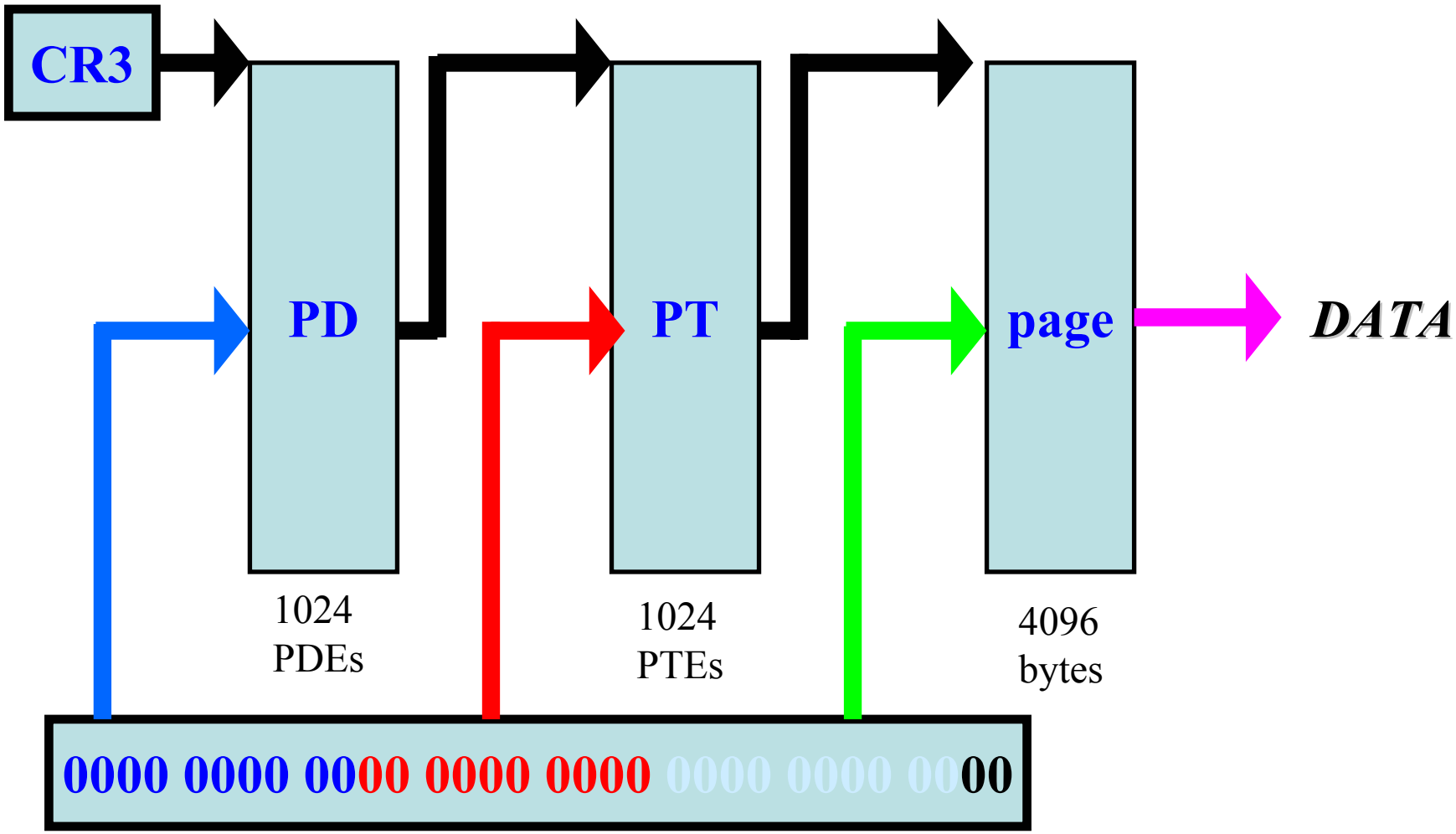
- Page Table Entries (PTEs) and Page Directory Entries (PDEs) contain **Physical Frame Numbers (PFNs)**
  - But Kernel runs with **Virtual Addresses**
- To access PDE/PTE from kernel use the self-map for the current process:

  PageDirectory[0x300] uses PageDirectory as PageTable
  - GetPdeAddress(va): 0xc0300000[va>>20]
  - GetPteAddress(va): 0xc0000000[va>>10]
- PDE/PTE formats are compatible!
- Access another process VA via thread ‘attach’



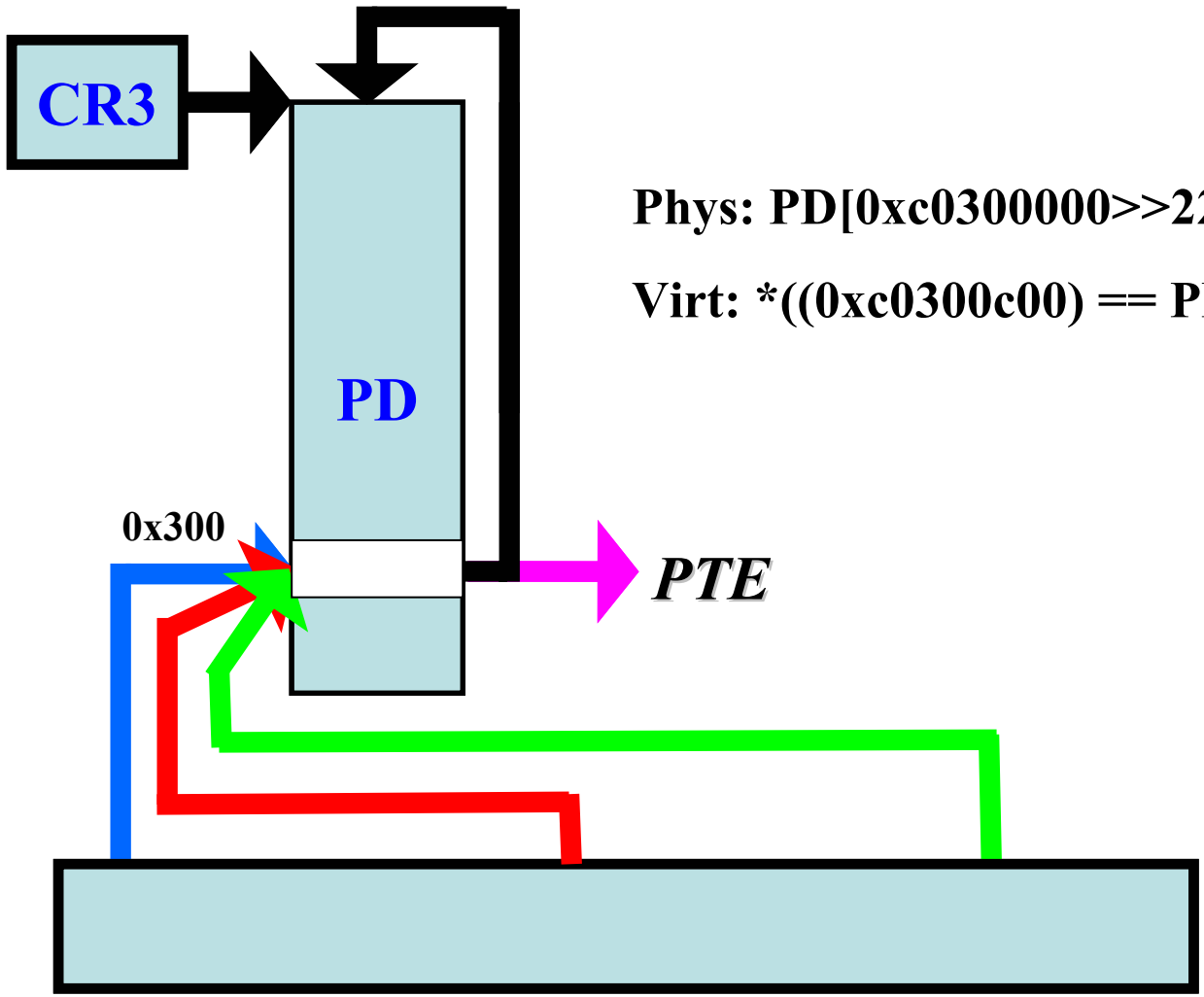
Self-mapping page tables

Normal Virtual Address Translation



Self-mapping page tables

Virtual Access to PageDirectory[0x300]

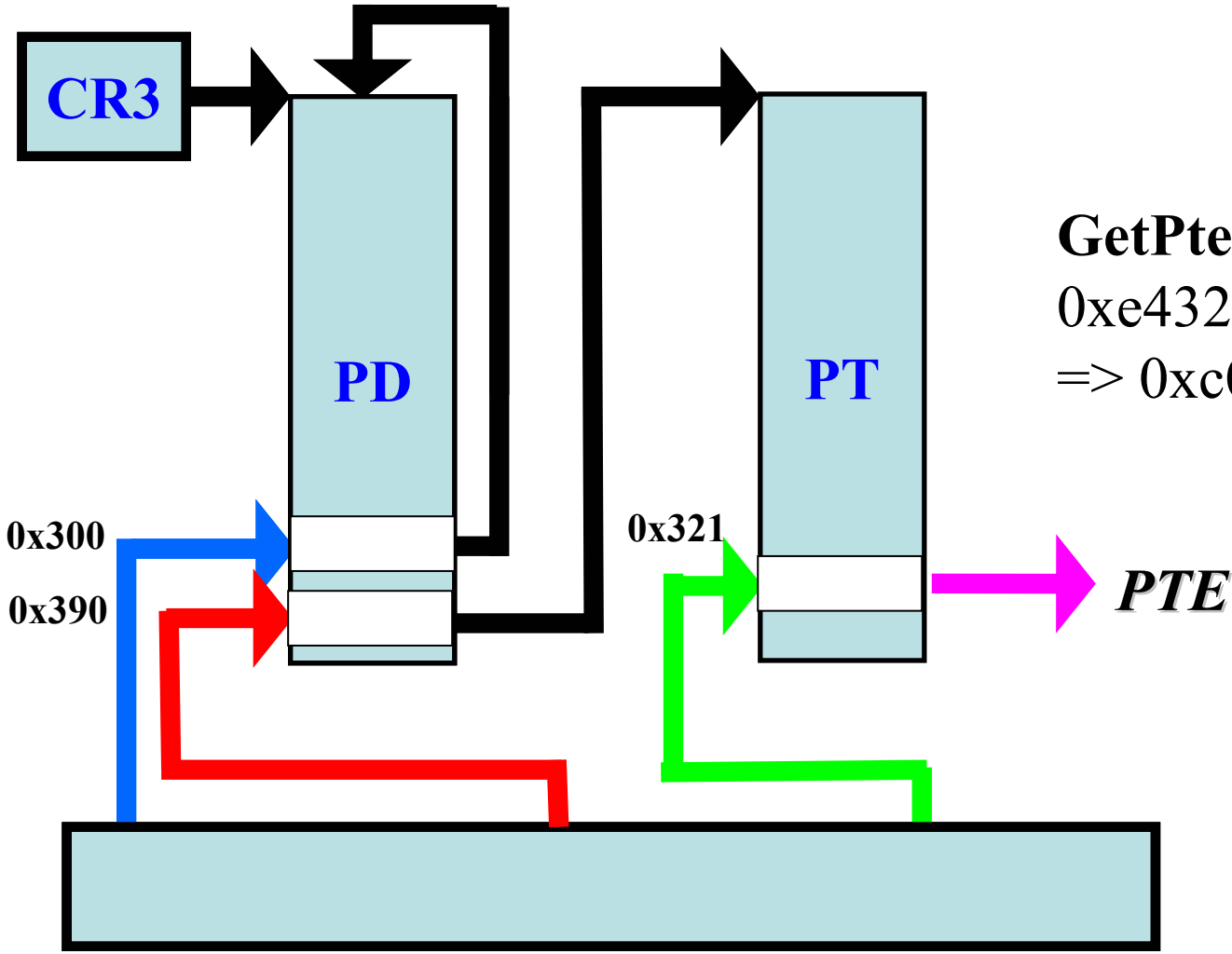


Phys: PD[0xc0300000>>22] = PD

Virt: *((0xc0300c00) == PD

Self-mapping page tables

Virtual Access to PTE for va 0xe4321000



GetPteAddress:
0xe4321000
=> 0xc0390c84

Physical Memory

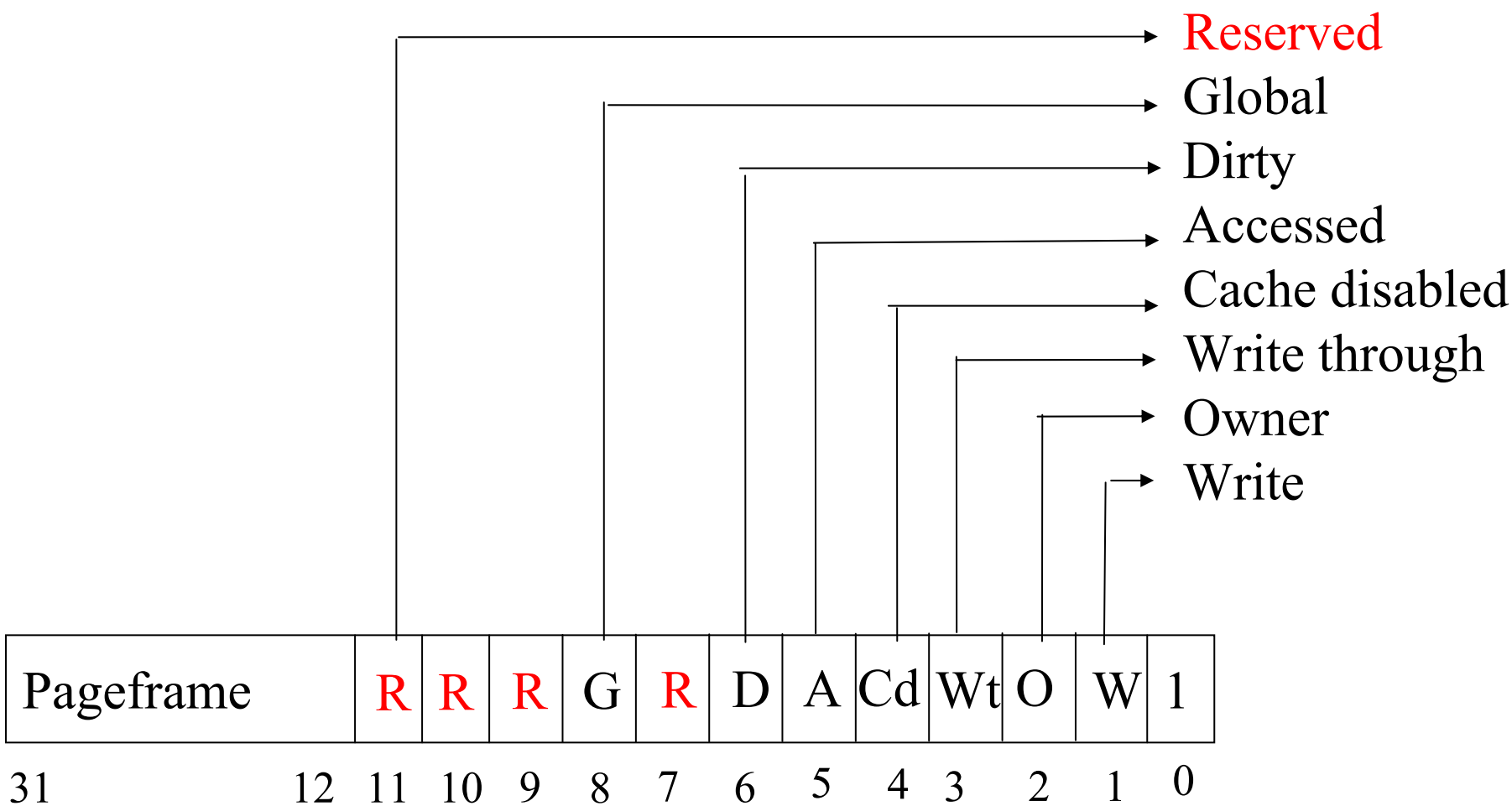
- Maximum is 64GB
  - Intel server processors support up to 64 GB physical memory through PAE mode (cr4)
    - four more bits of physical address in PTEs
  - Requires different kernel
    - Standard kernels use 32bit PTE format addressing 4Gb physical
    - PAE kernels use 36bit PTE format addressing 64Gb physical

Overcoming Physical Memory Limitations

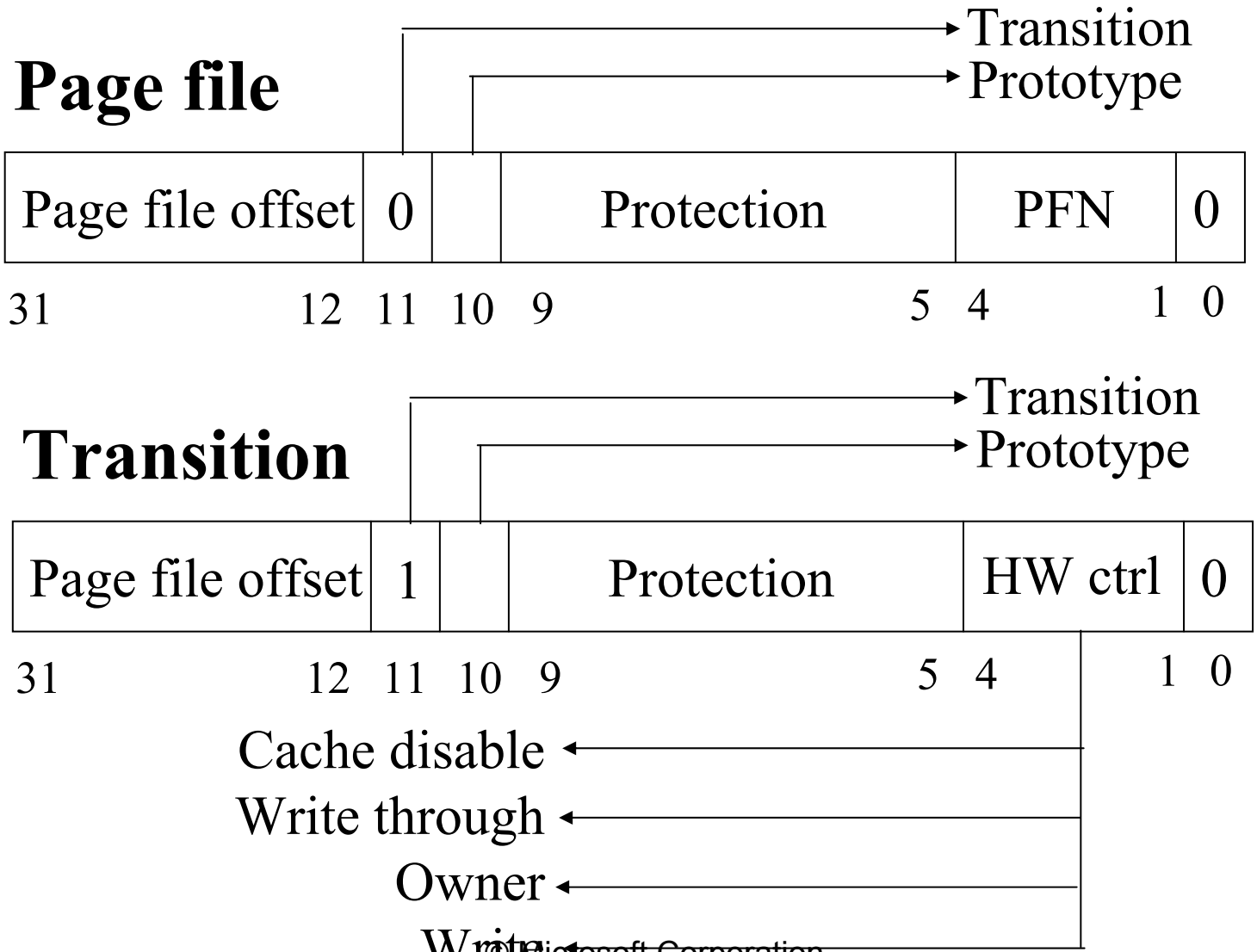
Virtual address space is 4 GB, so why physical memory > 4 GB?

1. Multiple processes.
2. Mapped (previously cached) files remain in physical memory (on standby list)
3. Extended addressing services:
   - Allow Win32 processes to allocate up to 64 GB RAM & map “views” into 2 GB process virtual address space (can do I/Os to it)
   - See Win32 functions AllocateUserPhysicalPages, MapUserPhysicalPages (very fast)
   - Used heavily by large databases

Valid x86 Hardware PTEs



x86 Invalid PTEs

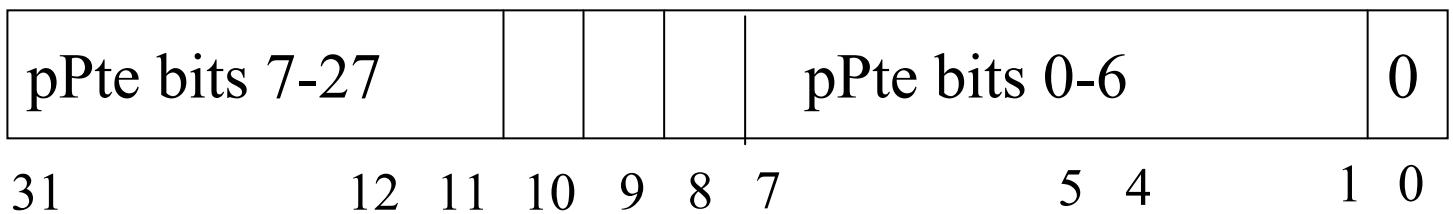


x86 Invalid PTEs

Demand zero: Page file PTE with zero offset and PFN

Unknown: PTE is completely zero or Page Table doesn't exist yet. Examine VADs.

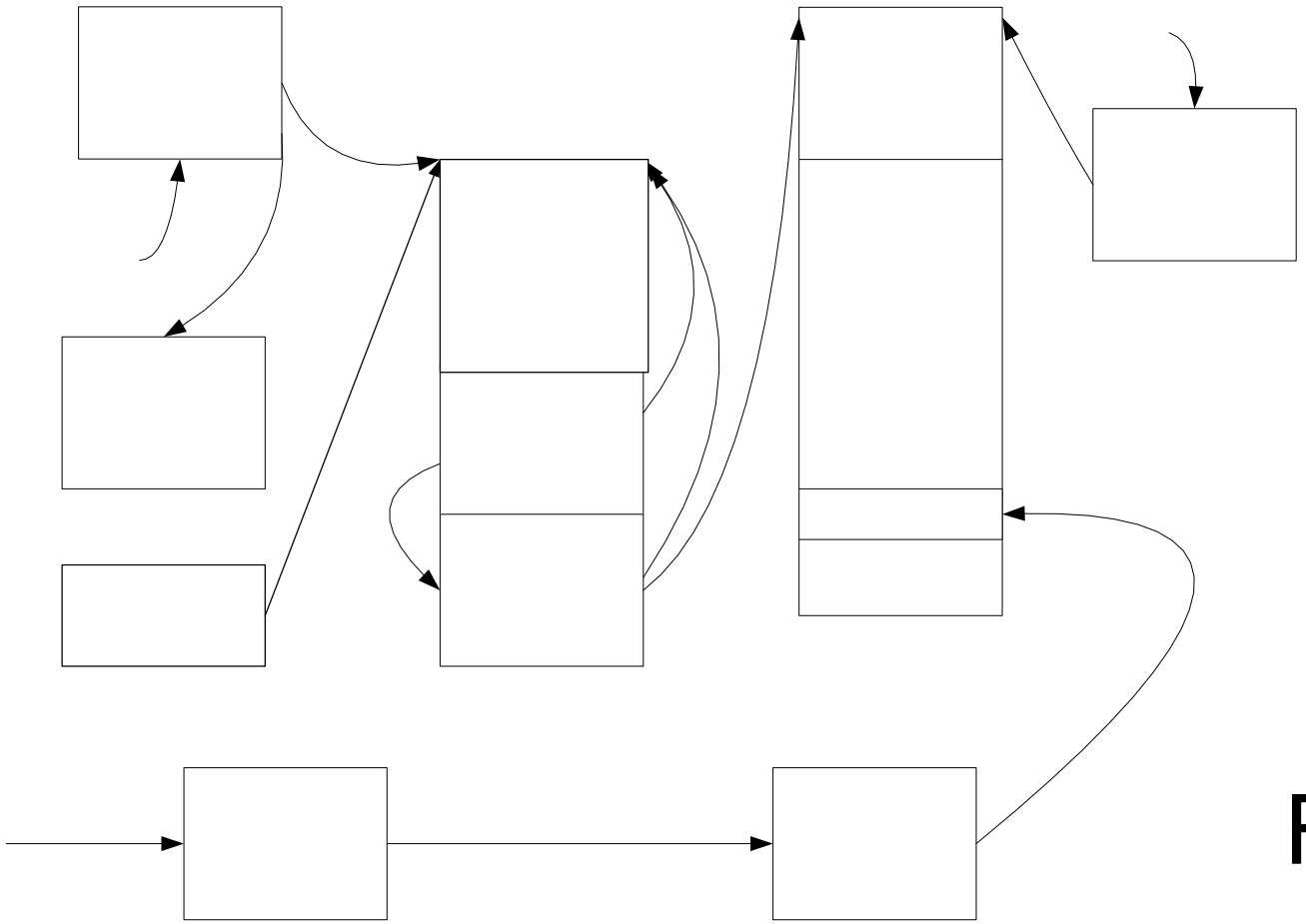
Pointer to Prototype PTE



Prototype PTEs

- Kept in array in the *segment* structure associated with section objects
- Six PTE states:
  - Active/valid
  - Transition
  - Modified-no-write
  - Demand zero
  - Page file
  - Mapped file

Shared Memory Data Structures

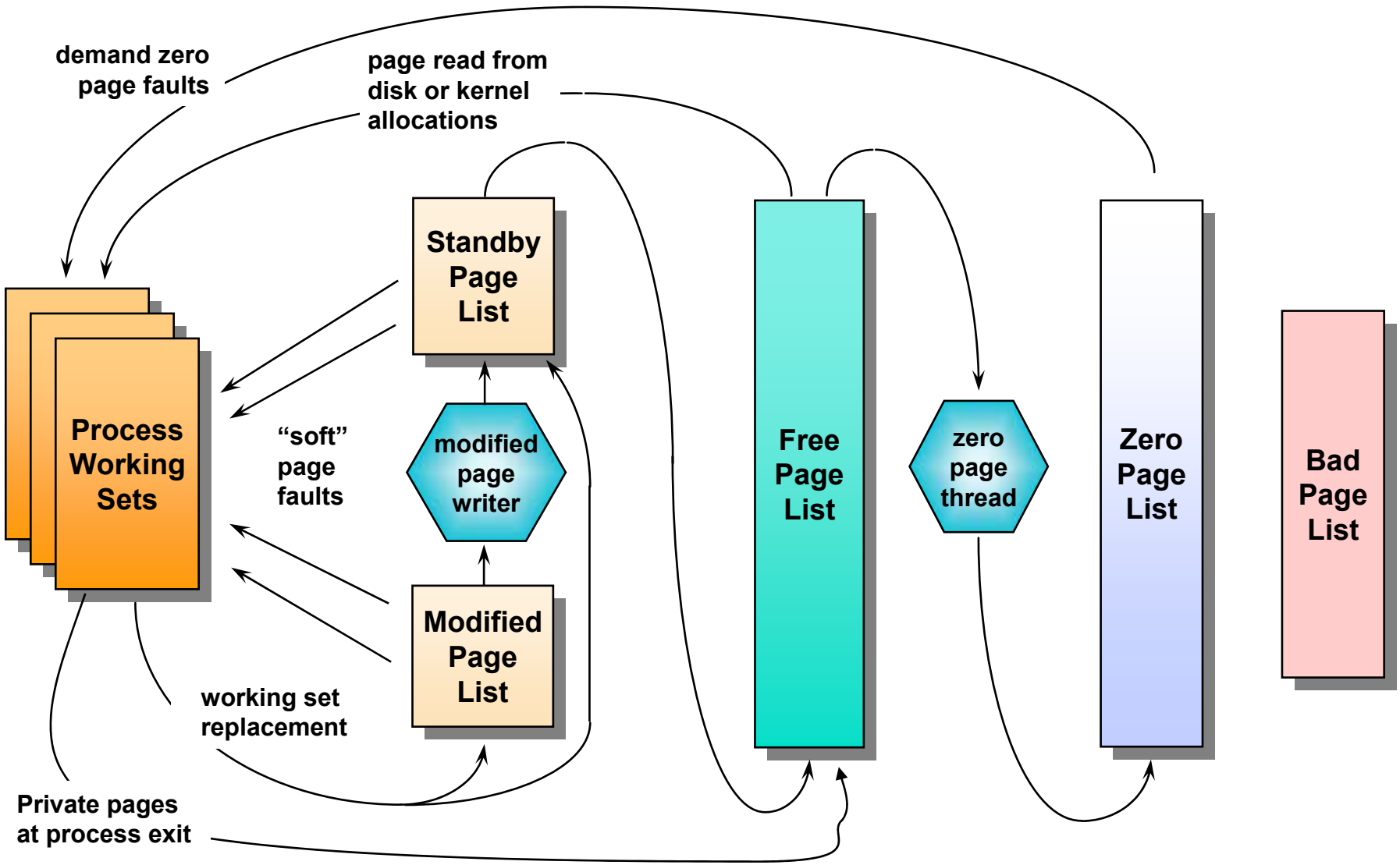


File Ob

Soft vs Hard Page Faults

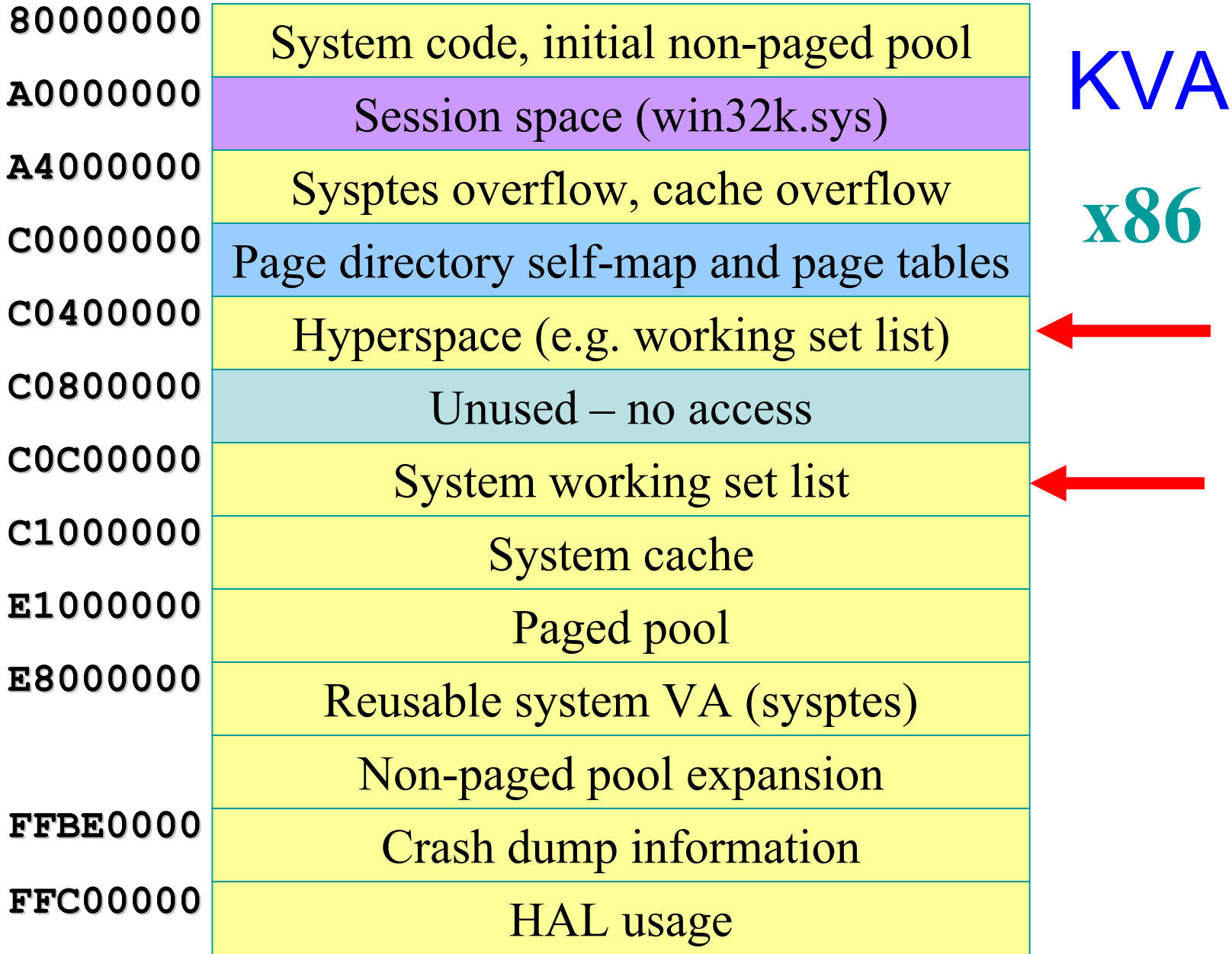
- Hard page faults involve a disk read
  - Some hard page faults are unavoidable
    - Code is brought into physical memory (from .EXEs and .DLLs) via page faults
    - The file system cache reads data from cached files in response to page faults
- Soft page faults are satisfied in memory
  - A shared page that's valid for one process can be faulted into other processes
  - Pages can be faulted back into a process from the standby and modified page list

Paging Dynamics



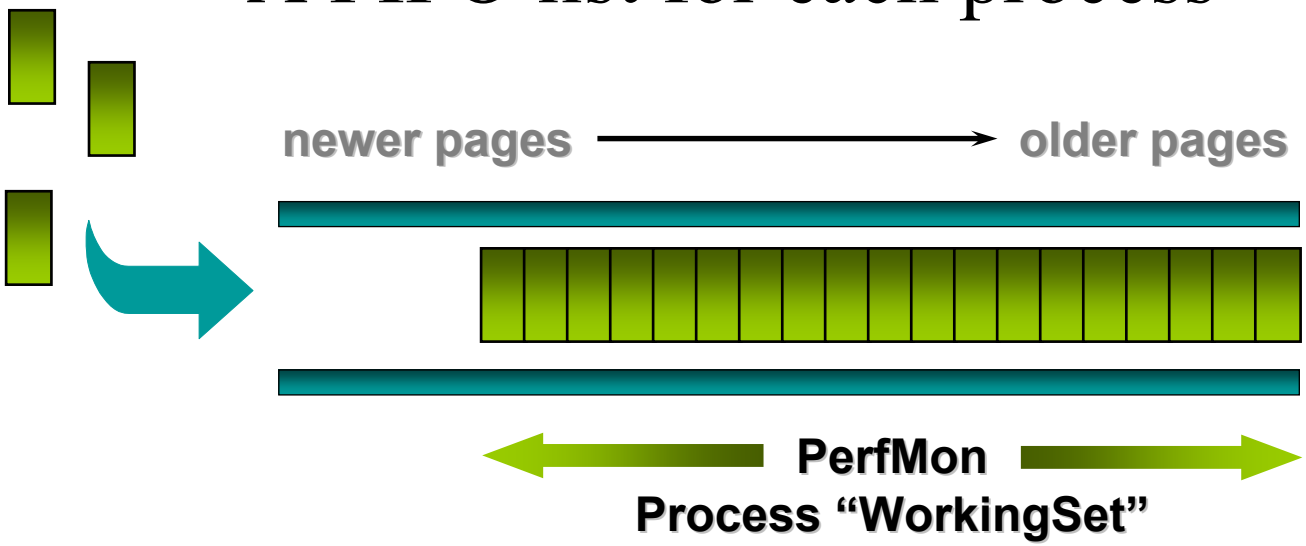
Standby and Modified Page Lists

- Used to:
  - Avoid writing pages back to disk too soon
  - Avoid releasing pages to the free list too soon
- The system can replenish the free page list by taking pages from the top of the standby page list
  - This breaks the association between the process and the physical page
  - i.e. the system no longer knows if the page still contains the process's info
- Pages move from the modified list to the standby list
  - Modified pages' contents are copied to the pages' backing stores (usually the paging file) by the modified page writer
  - The pages are then placed at the bottom of the standby page list
- Pages can be faulted back into a process from the standby and modified page list
  - The SPL and MPL form a system-wide cache of "pages likely to be needed again"



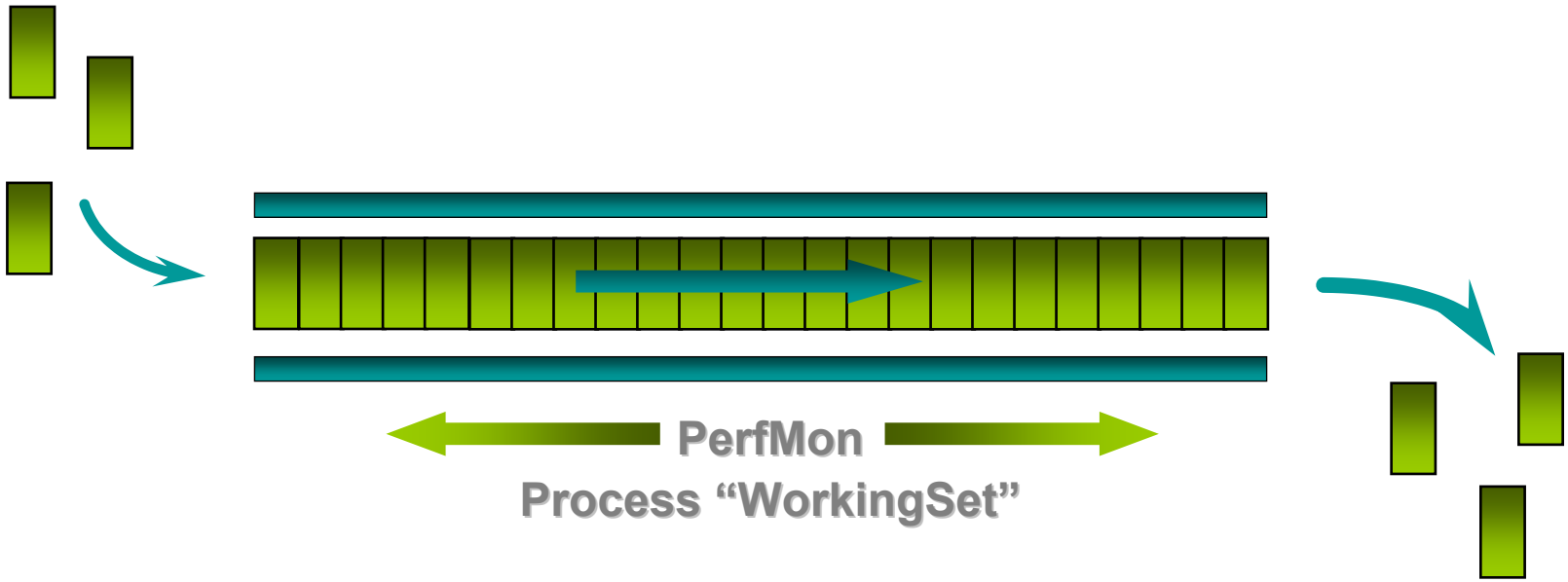
Working Set List

A FIFO list for each process



- **Pages in the working set are accessible without incurring a fault**
- **A process always starts with an empty working set**
  - **Pages itself into existence**
  - **Many page faults may be resolved from memory (to be described later)**

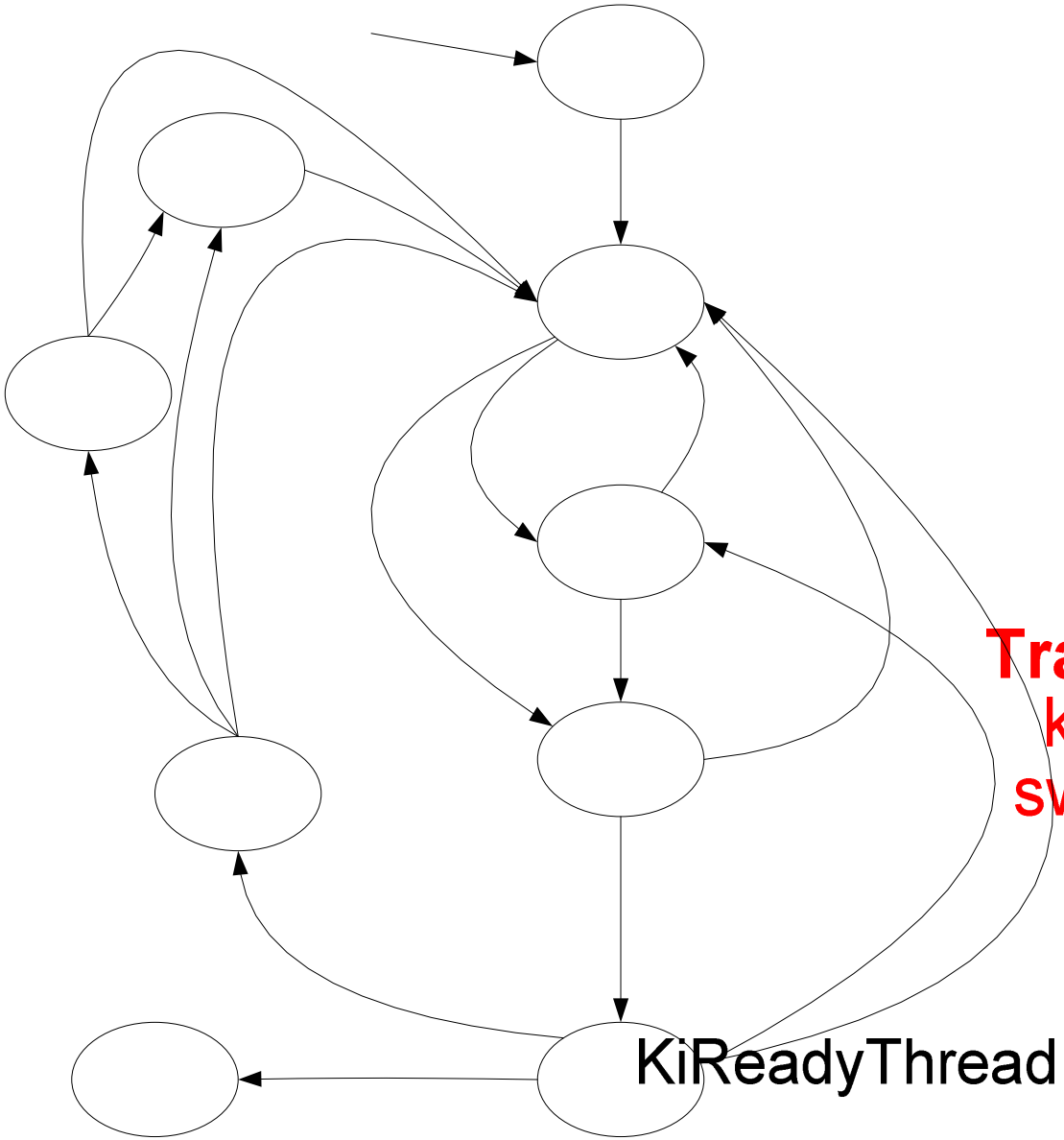
Working Set Replacement



- When working set "count" = working set size, must give up pages to make room for new pages
- Working set size is settable through SetProcessWorkingSetSize()
- Pages may be locked in a working set using VirtualLock
- Page replacement is least recently accessed

Balance Set Manager

- NT's swapper
  - Balance set = sum of all inswapped working sets
- Balance Set Manager is a system thread
  - Wakes up every second. If paging activity high or memory needed:
    - trims working sets of processes
    - if thread in a long user-mode wait, marks kernel stack pages as pageable
    - if process has no nonpageable kernel stacks, "outswaps" process
    - triggers a separate thread to do the "outswap" by gradually reducing target process's working set limit to zero



Thread scheduling states

KeInitThr

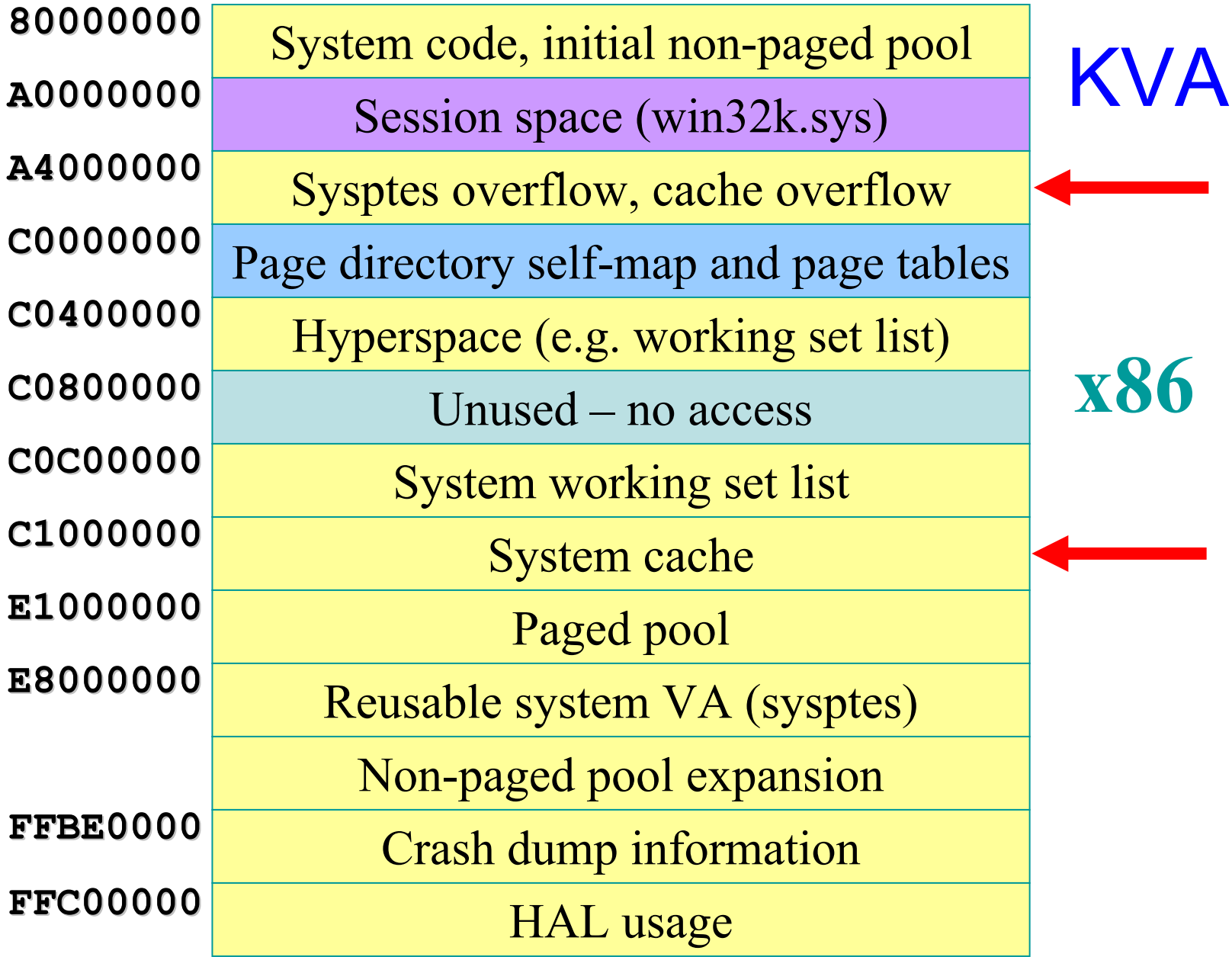
Transition
k stack
swapped

KiInsertDeferred

Ready

Thread scheduling states

- Main quasi-states:
  - Ready – able to run
  - Running – current thread on a processor
  - Waiting – waiting an event
- For scalability Ready is three real states:
  - DeferredReady – queued on any processor
  - Standby – will be imminently start Running
  - Ready – queue on target processor by priority
- Goal is granular locking of thread priority queues
- Red states related to swapped stacks and processes



File System Virtual Block Cache

- Shared by all file systems (local or remote)
- Caches all files
  - Including file system metadata files
- Virtual block cache (not logical block)
  - Managed in terms of blocks within files, not blocks within partition
  - Uses standard Windows NT virtual memory mechanisms
  - Coherency maintained between mapped files and read/write access

Cache Size

- Virtual size: 64-960mb
  - In system virtual address space, so visible to all
  - Divided into 256kb “views”
- Physical size: depends on available memory
  - Competes for physical memory with processes, paged pool, pageable system code
  - Part of system working set
    - Automatically expanded / shrunk by system
    - Normal working set adjustment mechanisms

Cache Functions And Control

- Automatic asynchronous readahead
  - Done by separate “Readahead” system thread
  - 64kb readaheads by default
  - Predicts next read location based on history of last 3 reads
  - Readahead hints can be provided to CreateFile:
    - FILE_FLAG_SEQUENTIAL does 192kb read ahead
    - FILE_FLAG_RANDOM_ACCESS disables read ahead
- Write-back, not write-through
  - Can override via CreateFile with FILE_FLAG_WRITE_THROUGH
    - Or explicitly call FlushFileBuffers when you care (does flush mapped files)
- Can disable cache completely on a per-file basis
  - CreateFile with FILE_FLAG_NO_BUFFERING

Summary

- Manages physical memory and pagefiles
- Manages user/kernel virtual space
- Working-set based management
- Provides shared-memory
- Supports physical I/O
- Address Windowing Extensions for large memory
- Provides session-memory for Win32k GUI processes
- File cache based on shared sections
- Single implementation spans multiple architectures

Discussion