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
- Supports shared memory and copy-on-write
- 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

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

80000000	System code, initial non-paged pool
A0000000	Session space (win32k.sys)
A4000000	Sysptes overflow, cache overflow
C0000000	Page directory self-map and page tables
C0400000	Hyperspace (e.g. working set list)
C0800000	Unused – no access
C0C00000	System working set list
C1000000	System cache
E1000000	Paged pool
E8000000	Reusable system VA (sysptes)
	Non-paged pool expansion
FFBE0000	Crash dump information
FFC00000	HAL usage

x86

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

```
 31               23          16 15               7             0
 +------------------------------------------------------------+
 | Current Size |  PoolType+1 |  Pool Index  |Previous Size |
 +------------------------------------------------------------+
 |   ProcessCharged  (NULL if not allocated with quota)       |
 +------------------------------------------------------------+
 | Zero or more longwords of pad such that the pool header    |
 | is on a cache line boundary and the pool body is also      |
 | on a cache line boundary.                                  |
 +------------------------------------------------------------+
PoolBody:
 +------------------------------------------------------------+
 |  Used by allocator, or when free FLINK into sized list     |
 +------------------------------------------------------------+
 |  Used by allocator, or when free BLINK into sized list     |
 +------------------------------------------------------------+
 ... rest of pool block...
```

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

KVA

x86

80000000	System code, initial non-paged pool
A0000000	Session space (win32k.sys)
A4000000	Sysptes overflow, cache overflow
C0000000	Page directory self-map and page tables
C0400000	Hyperspace (e.g. working set list)
C0800000	Unused – no access
C0C00000	System working set list
C1000000	System cache
E1000000	Paged pool
E8000000	Reusable system VA (sysptes)
	Non-paged pool expansion
FFBE0000	Crash dump information
FFC00000	HAL usage

Sysptes

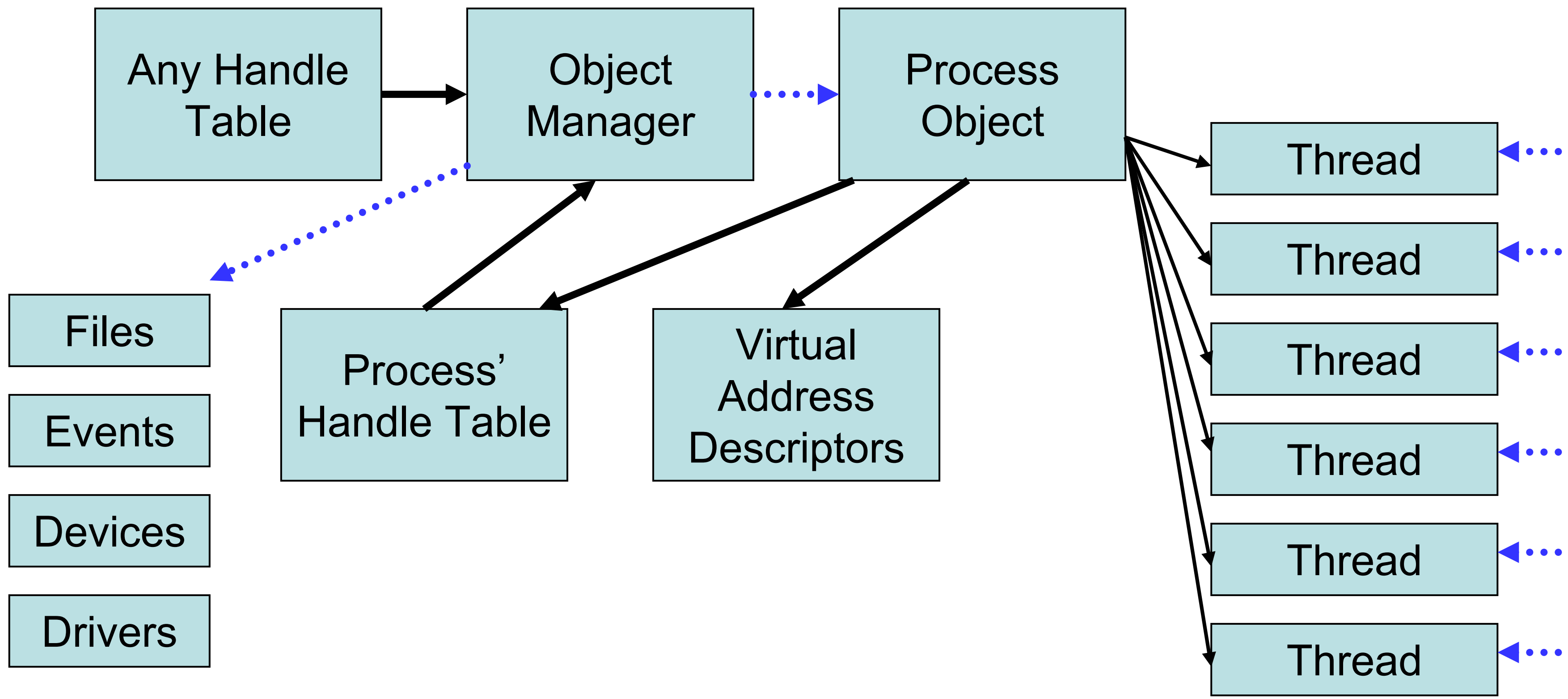
Used to manage random use of kernel virtual memory, e.g. by device drivers.

Kernel implements functions like:

- MiReserveSystemPtes (n, type)
- MiMapLockedPagesInUserSpace
  (mdl, virtaddr, cachetype,basevirtaddr)

Often a critical resource!

Process/Thread structure



Process

Container for an address space and threads

Associated User-mode Process Environment Block (PEB)

Primary Access Token

Quota, Debug port, Handle Table etc

Unique process ID

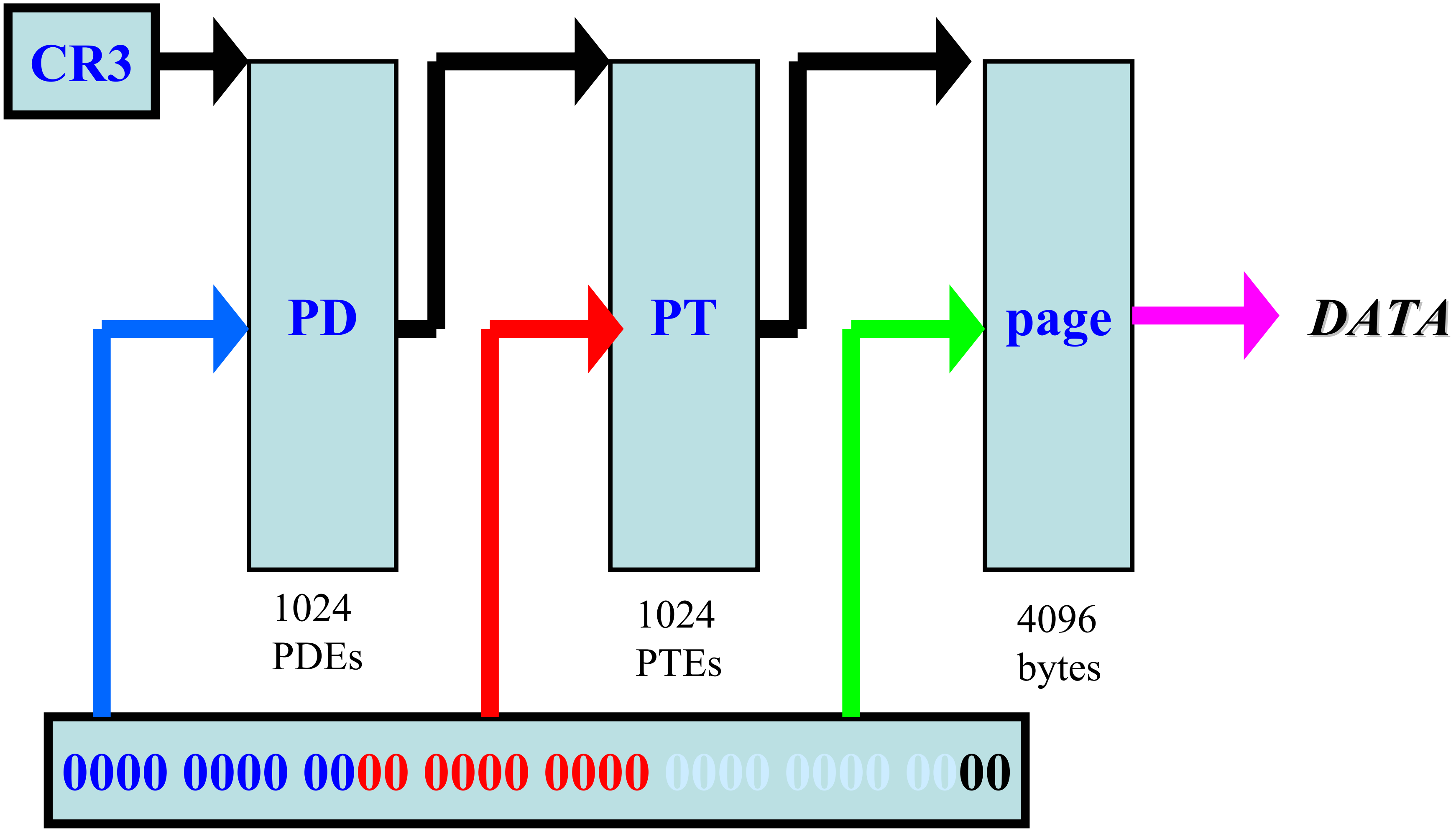
Queued to the Job, global process list and Session list

MM structures like the WorkingSet, VAD tree, AWE etc

Thread

- Fundamental schedulable entity in the system
- Represented by ETHREAD that includes a KTHREAD
- Queued to the process (both E and K thread)
- IRP list
- Impersonation Access Token
- Unique thread ID
- Associated User-mode Thread Environment Block (TEB)
- User-mode stack
- Kernel-mode stack
- Processor Control Block (in KTHREAD) for cpu state when not running

Virtual Address Translation



Windows Virtual Memory Model

User-mode (2GB or 3GB with boot option)

- Code/data from executables (e.g. .exe, .dll)
- Thread/process environment blocks (TEB/PEB)
- User-mode stacks, heaps

Kernel-mode (2GB or 1GB)

- Kernel, hal, drivers
- Kernel-mode stacks, heap (i.e. pool), pagetables
- File-cache (cache and pool small if 1GB)
- Terminal-server session space (for Win32k)
- Kernel data structures and objects

Physical Memory Model (IA32)

Limit is 4GB (or 64GB w/ PAE support)

- PAE support requires 64-bit PTE entries
  - Separate kernel needed as all MM data recompiled
- Thread/process environment blocks (TEB/PEB)
- User-mode stacks, heaps

Large server applications can use AWE

- Address Window Extension
- Processes allocate contiguous physical memory
- Memory Manager allows map/unmap via AWE APIs

Large (4MB) pages supported for TLB efficiency

64b Windows makes virtual/physical limits moot

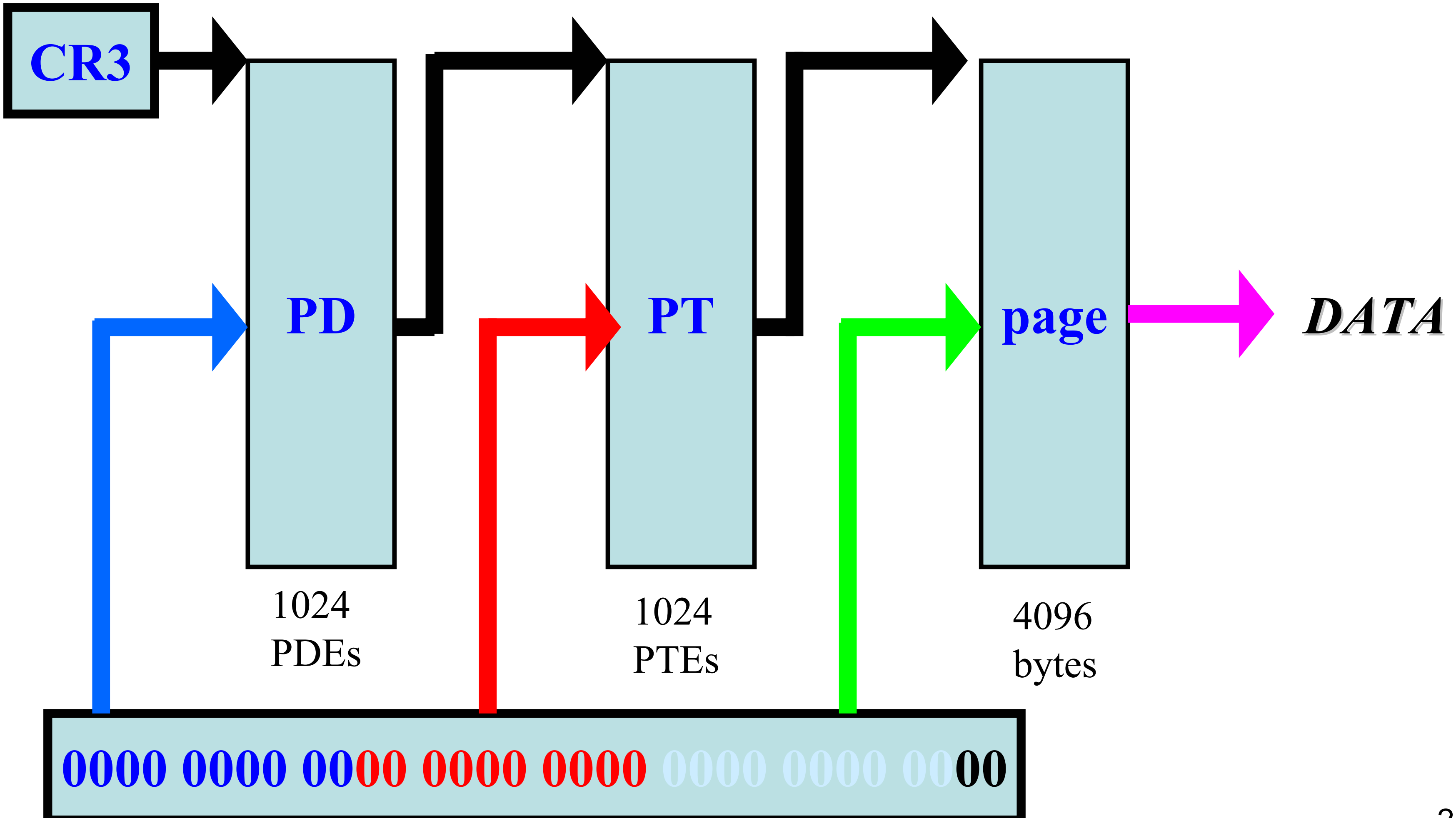
KVA

x86

80000000	System code, initial non-paged pool
A0000000	Session space (win32k.sys)
A4000000	Sysptes overflow, cache overflow
C0000000	Page directory self-map and page tables
C0400000	Hyperspace (e.g. working set list)
C0800000	Unused – no access
C0C00000	System working set list
C1000000	System cache
E1000000	Paged pool
E8000000	Reusable system VA (sysptes)
	Non-paged pool expansion
FFBE0000	Crash dump information
FFC00000	HAL usage

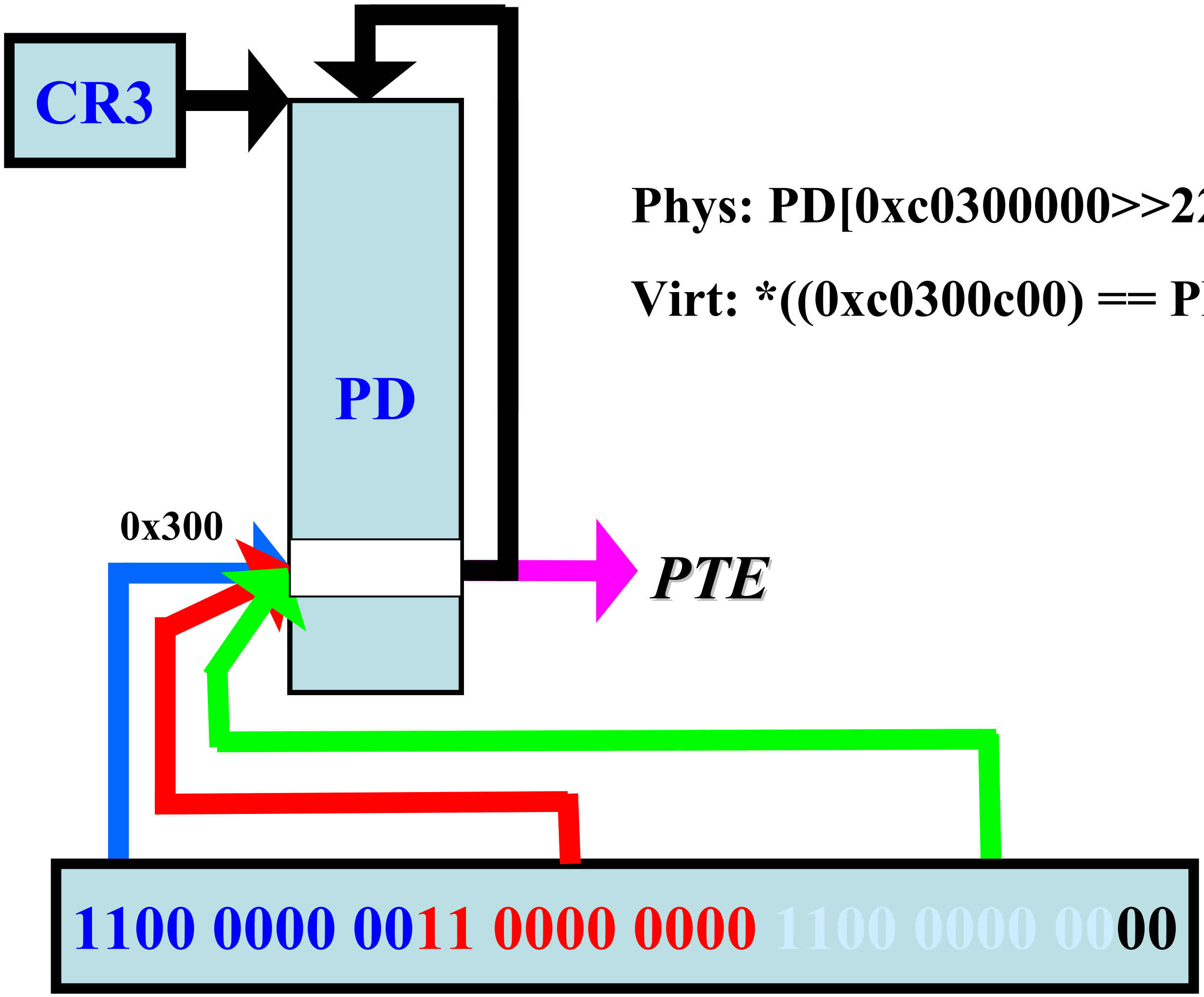
Self-mapping page tables

Normal Virtual Address Translation



Self-mapping page tables

Virtual Access to PageDirectory[0x300]

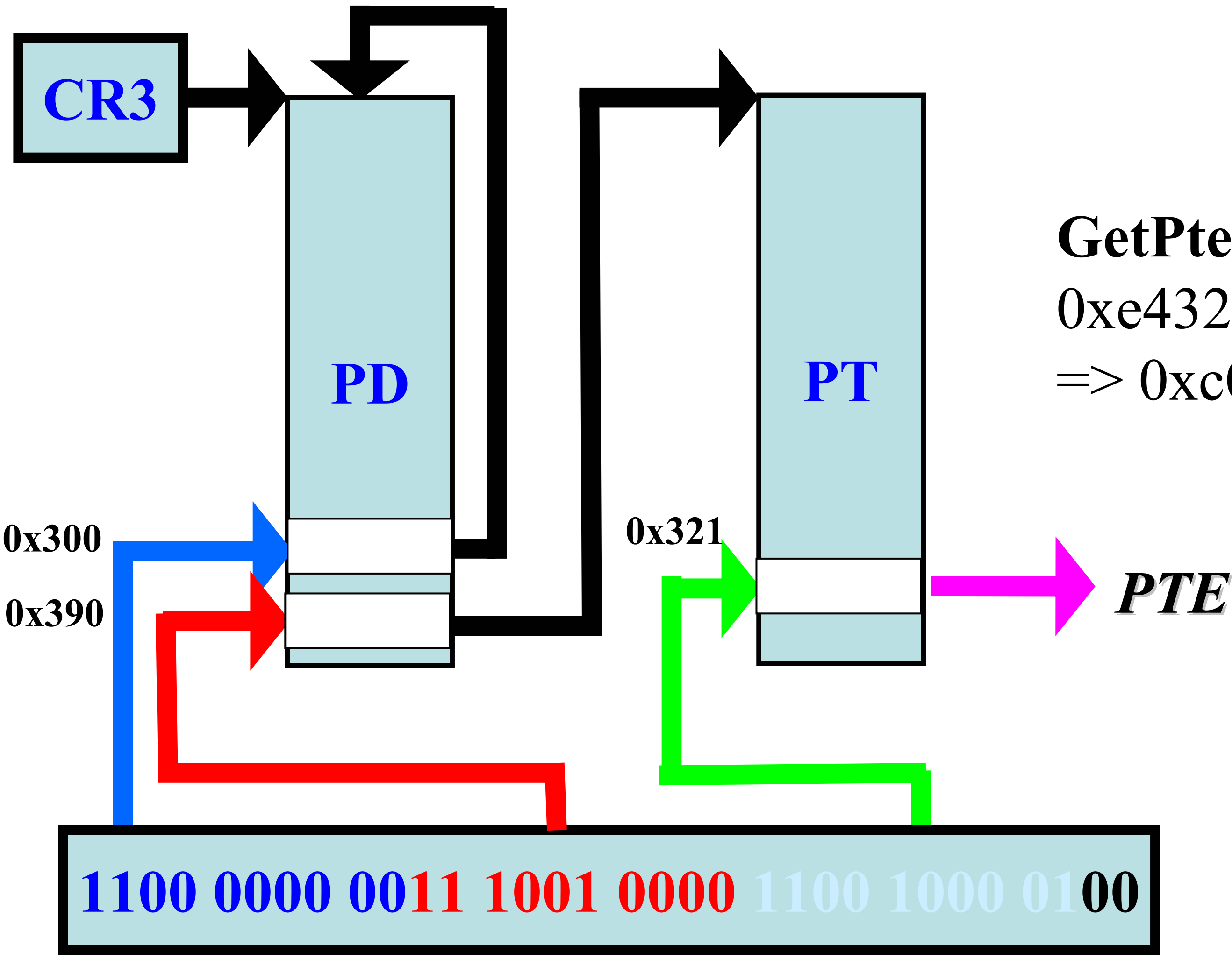


Phys: PD[0xc0300000>>22] = PD

Virt: *((0xc0300c00) == PD

Self-mapping page tables

Virtual Access to PTE for va 0xe4321000



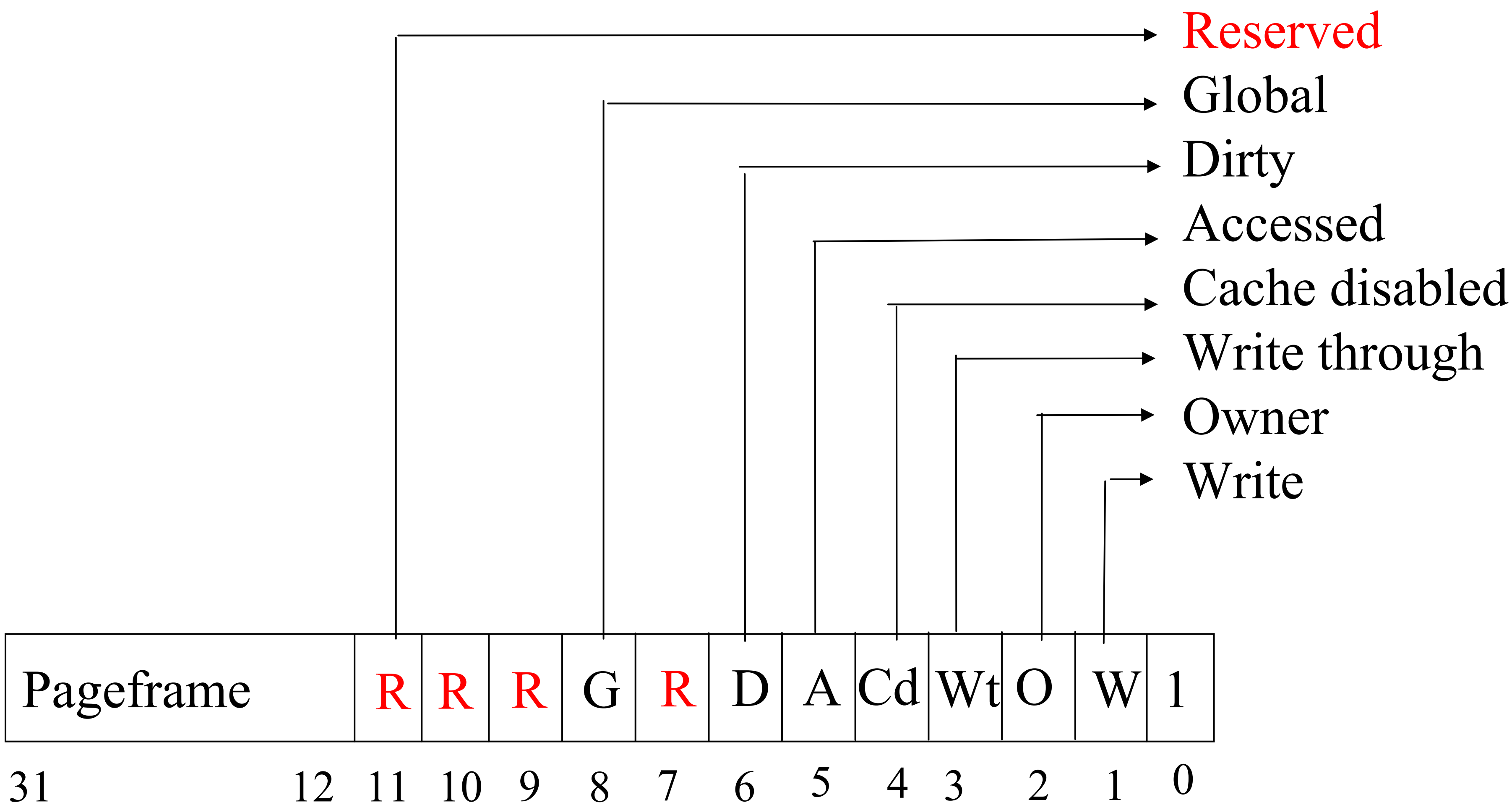
GetPteAddress:
0xe4321000
=> 0xc0390c84

Self-mapping page tables

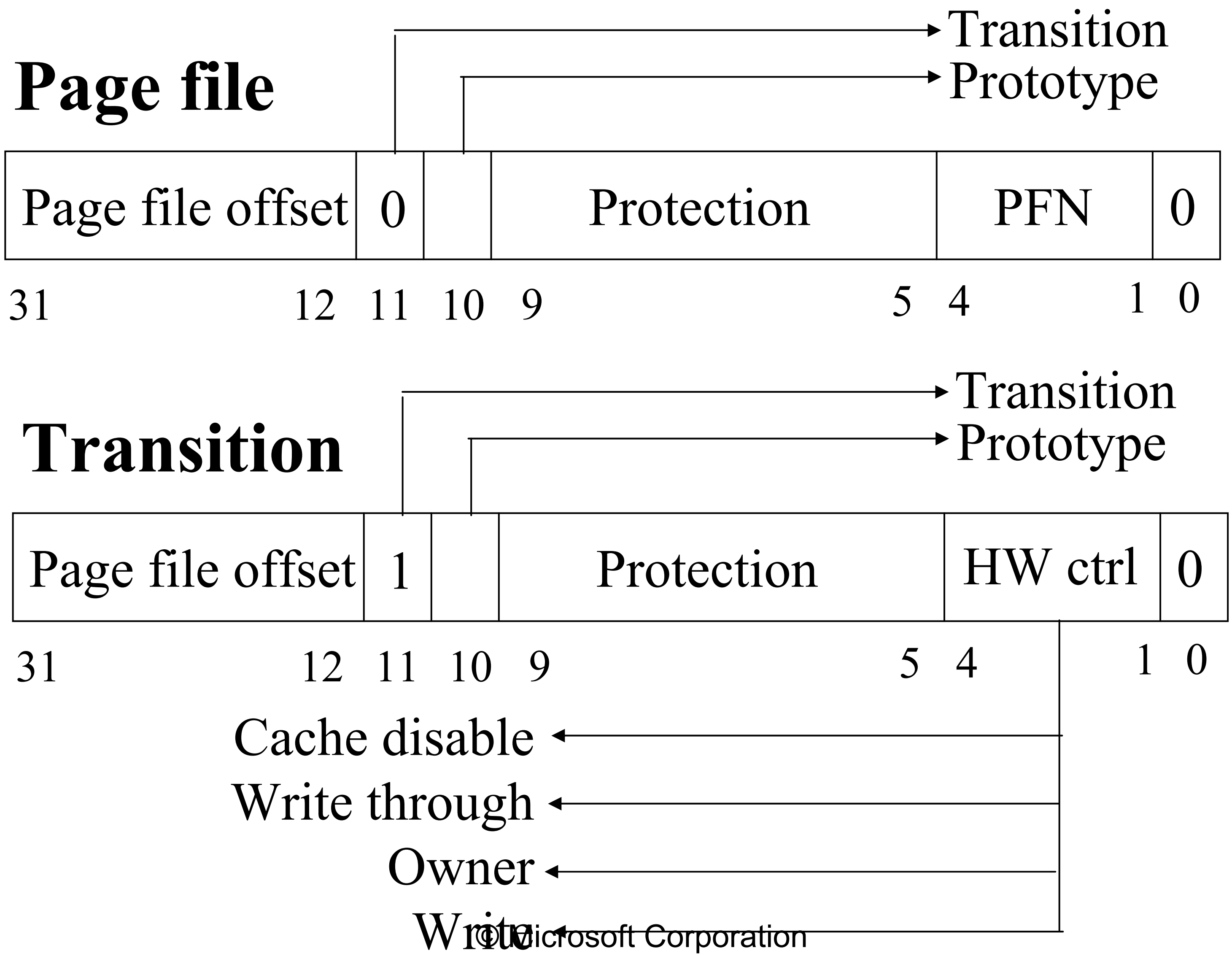
- Page Table Entries (PTEs) and Page Directory Entries (PDEs) contain **Physical Frame Numbers (PFNs)**
  - But Kernel runs with **Virtual Addresses**
- To access PDE/PTE from kernel use the self-map for the current process:

  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’

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

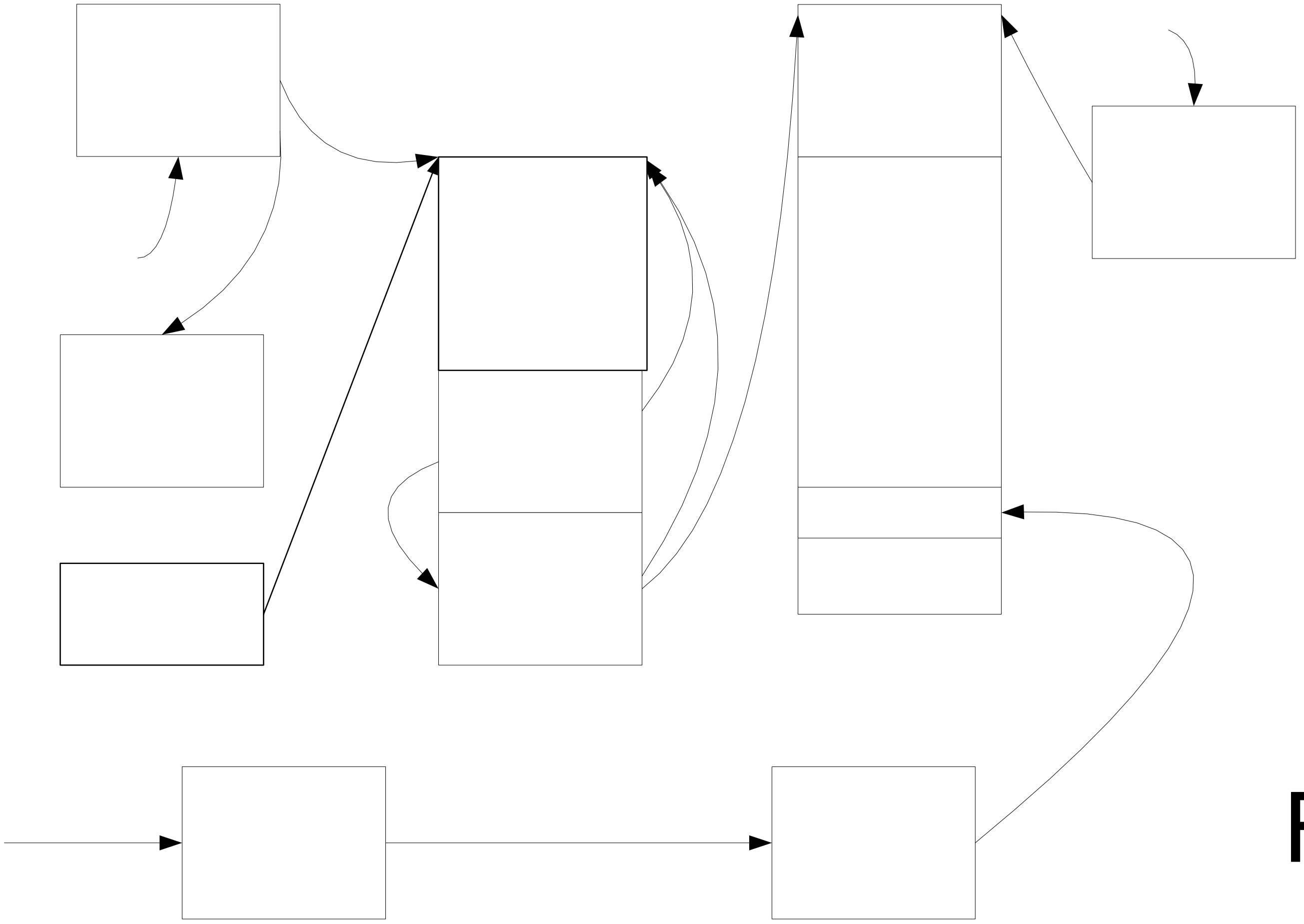
pPte bits 7-27				pPte bits 0-6	0

31 12 11 10 9 8 7 5 4 1 0

Prototype PTEs

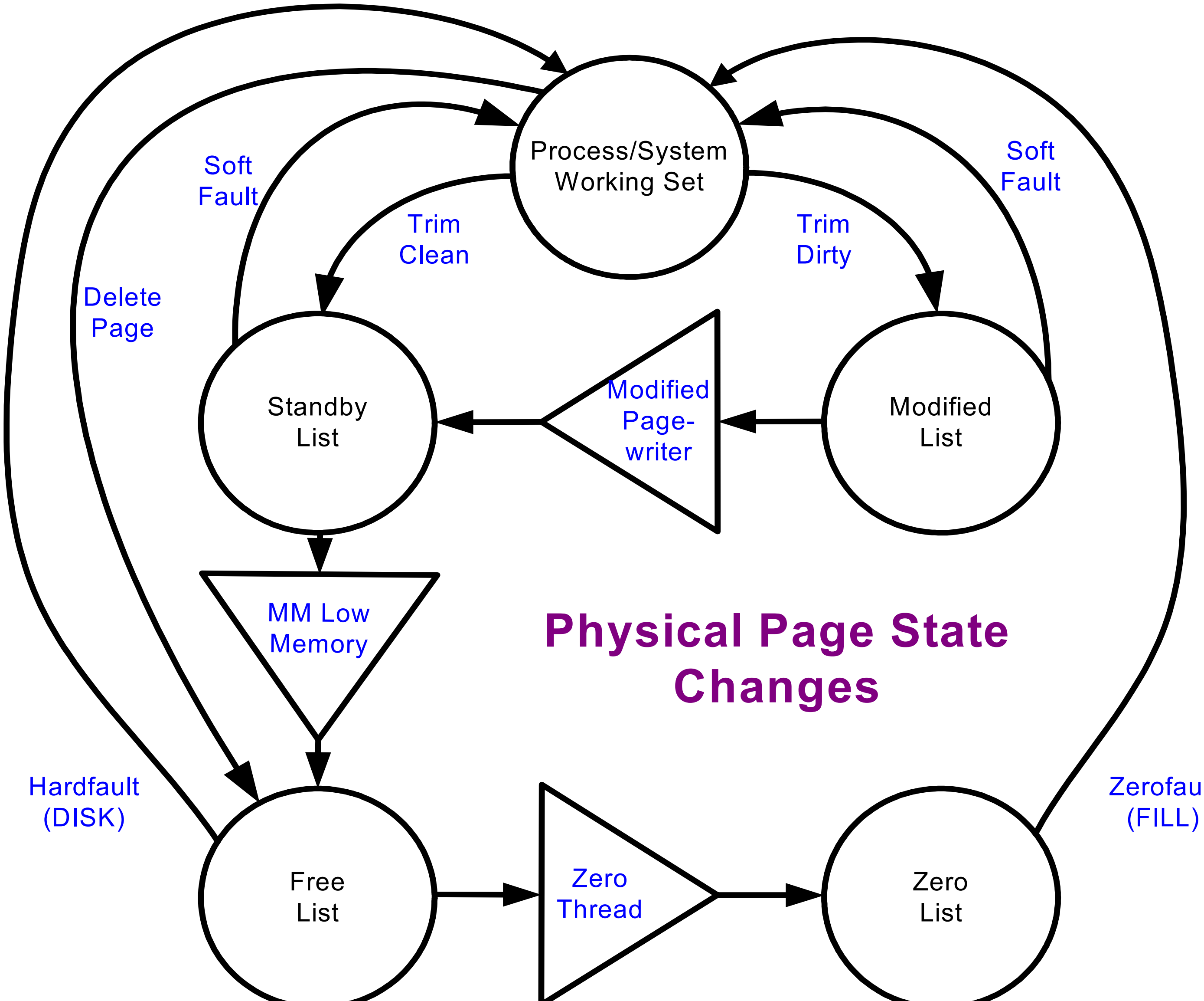
- Kept in array in the *segment* structure associated with section objects
- Six PTE states:
  - Active/valid
  - Transition
  - Modified-no-write
  - Demand zero
  - Page file
  - Mapped file

Shared Memory Data Structures



File Ob

Physical Memory Management



Paging Overview

Working Sets: list of valid pages for each process (and the kernel)

Pages ‘trimmed’ from working set on lists

- **Standby list**: pages backed by disk
- **Modified list**: dirty pages to push to disk
- **Free list**: pages not associated with disk
- **Zero list**: supply of demand-zero pages

Modify/standby pages can be faulted back into a working set w/o disk activity (soft fault)

Background system threads trim working sets, write modified pages and produce zero pages based on memory state and config parameters

KVA x86

80000000	System code, initial non-paged pool
A0000000	Session space (win32k.sys)
A4000000	Sysptes overflow, cache overflow
C0000000	Page directory self-map and page tables
C0400000	Hyperspace (e.g. working set list)
C0800000	Unused – no access
C0C00000	System working set list
C1000000	System cache
E1000000	Paged pool
E8000000	Reusable system VA (sysptes)
	Non-paged pool expansion
FFBE0000	Crash dump information
FFC00000	HAL usage

Managing Working Sets

Aging pages: Increment age counts for pages which haven't been accessed

Estimate unused pages: count in working set and keep a global count of estimate

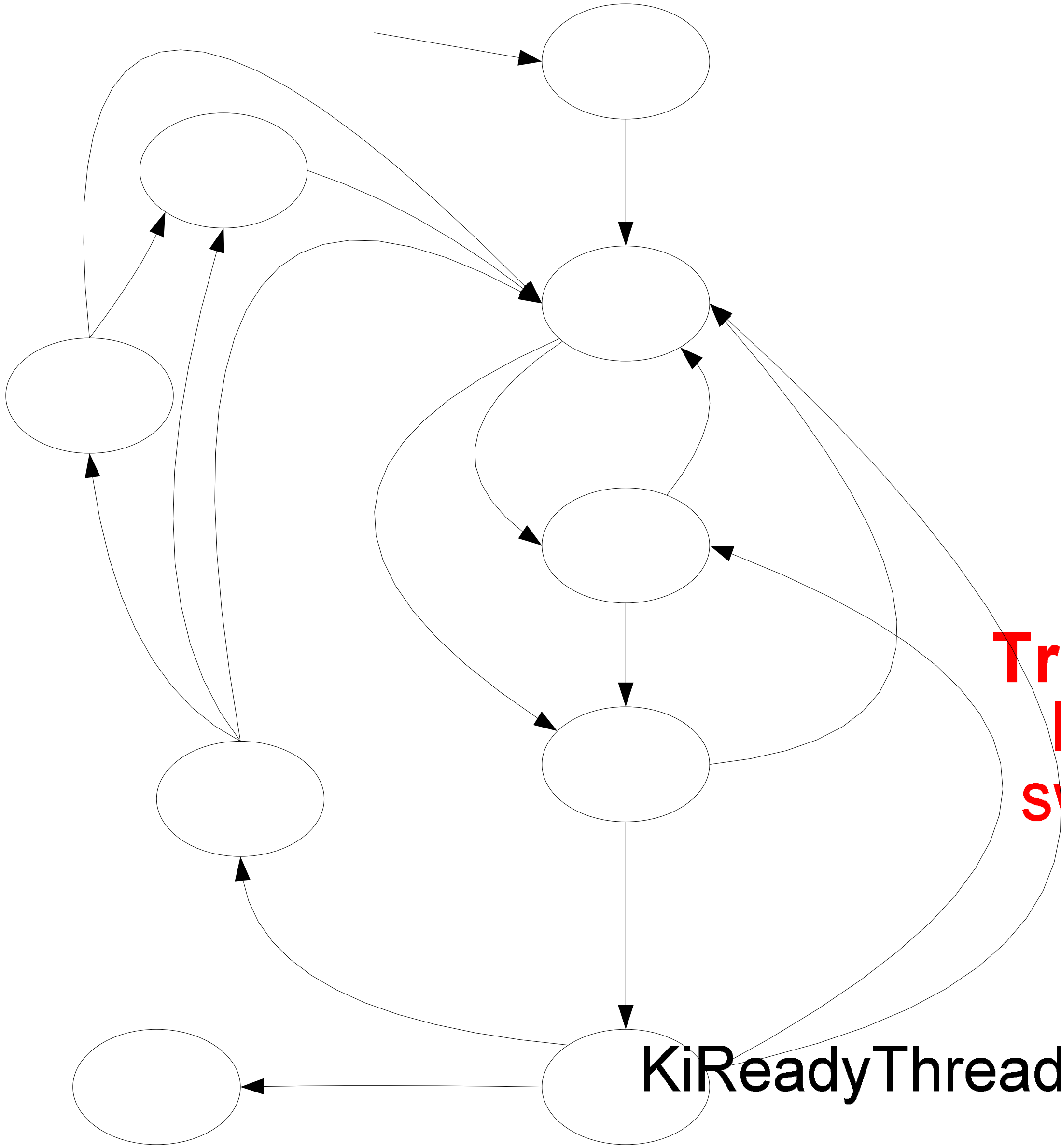
When *getting* tight on memory: replace rather than add pages when a fault occurs in a working set with significant unused pages

When memory *is* tight: reduce (trim) working sets which are above their maximum

Balance Set Manager: periodically runs Working Set Trimmer, also swaps out kernel stacks of long-waiting threads

Thread scheduling states

KeInitThr



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

KVA

x86

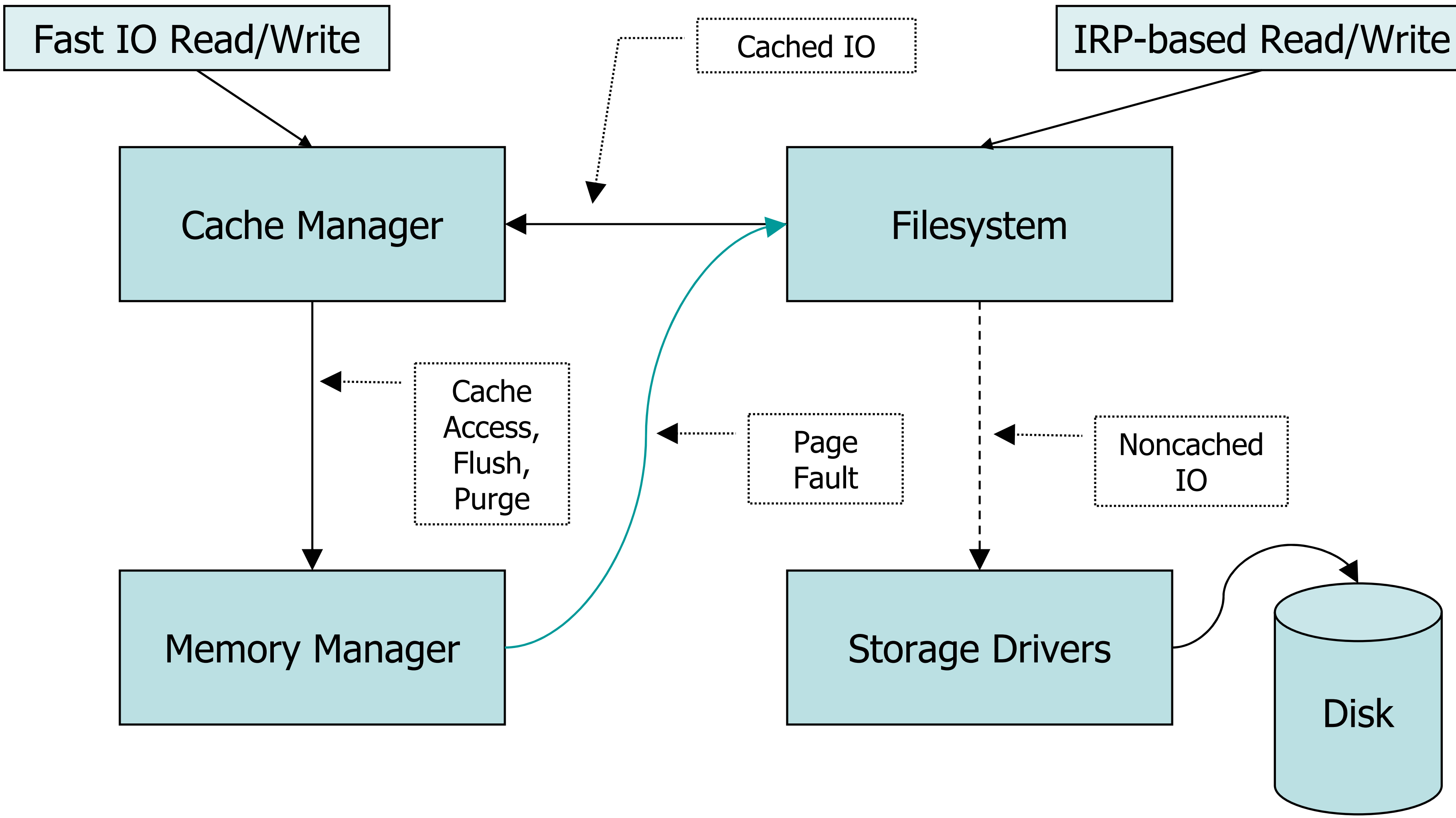
80000000	System code, initial non-paged pool
A0000000	Session space (win32k.sys)
A4000000	Sysptes overflow, cache overflow
C0000000	Page directory self-map and page tables
C0400000	Hyperspace (e.g. working set list)
C0800000	Unused – no access
C0C00000	System working set list
C1000000	System cache
E1000000	Paged pool
E8000000	Reusable system VA (sysptes)
	Non-paged pool expansion
FFBE0000	Crash dump information
FFC00000	HAL usage

File Cache Manager

Kernel APIs & worker threads that interface the file systems to memory manager

- File-based, not block-based
- Access methods for pages of opened files
- Automatic asynch read ahead
- Automatic asynch write behind (lazy write)
- Supports “Fast I/O” – IRP bypass
- Works with file system metadata (pseudo files)

Cache Manager Block Diagram



Cache Manager and MM

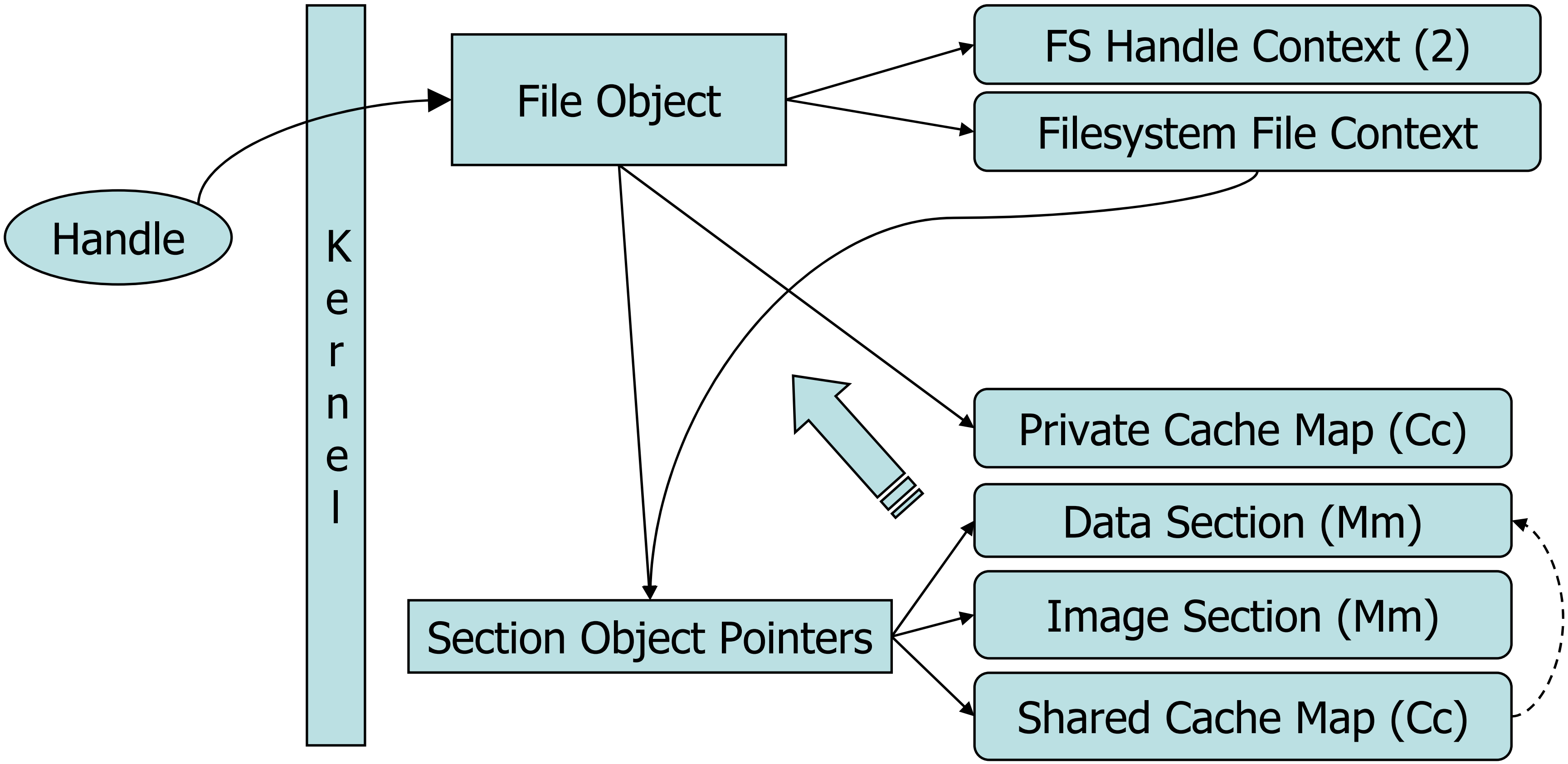
The Cache Manager sits between the file systems and the memory manager

- Mapped stream model integrated with memory management
- Cached streams are mapped with fixed-size views (256KB)
- Pages are faulted into memory via MM
- Pages may be modified in memory and written back
- MM manages global memory policy

Pagefault Cluster Hints

- Taking a pagefault can result in Mm opportunistically bringing surrounding pages in (up 7/15 depending)
- Since Cc takes pagefaults on streams, but knows a lot about which pages are useful, Mm provides a hinting mechanism in the TLS
  - MmSetPageFaultReadAhead()
- Not exposed to usermode …

Cache Manager Data Structures

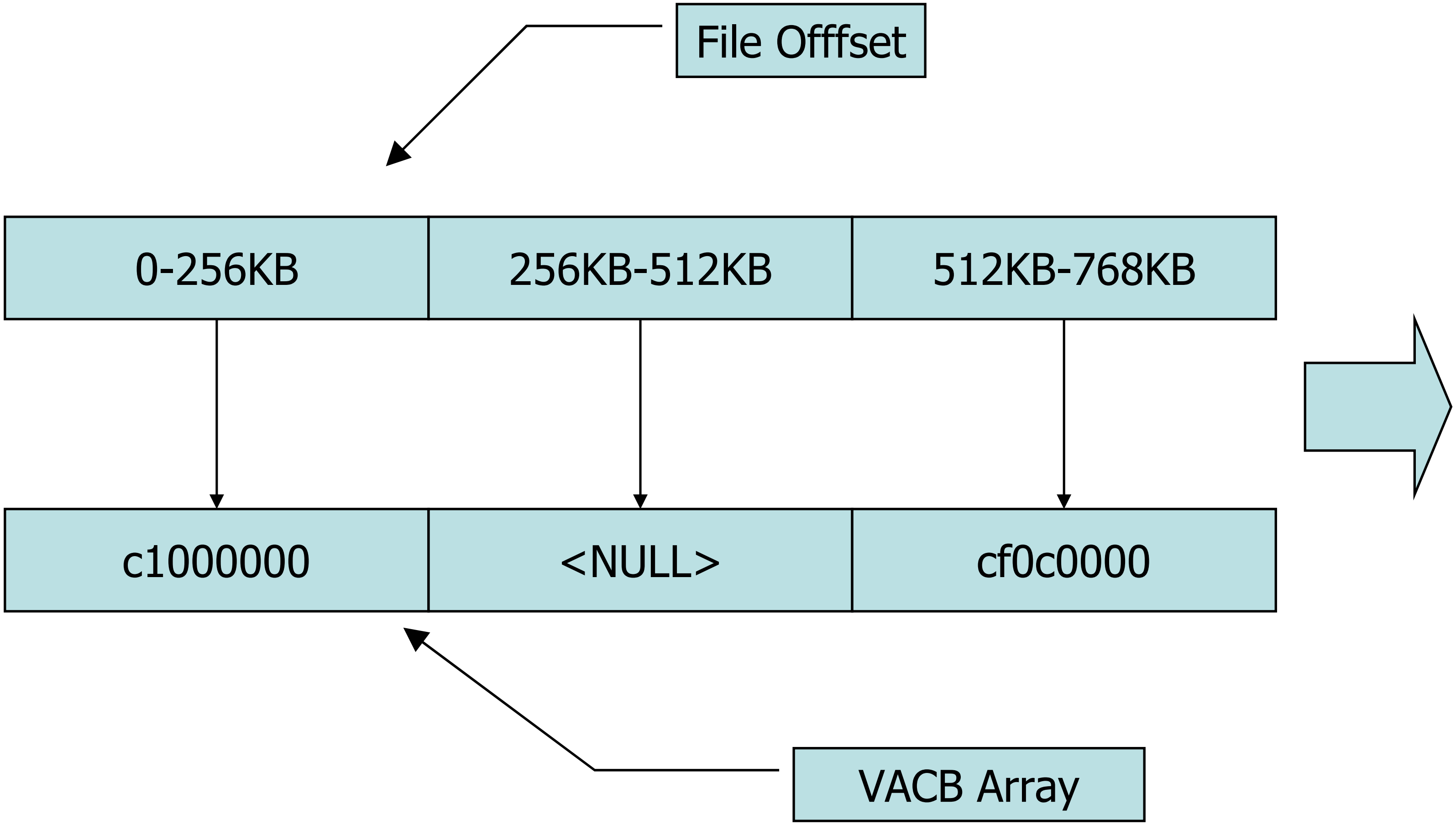


- File Object == Handle (U or K), *not one per file*
- Section Object Pointers and FS File Context are the same for all file objects for the same stream

Cache View Management

- A Shared Cache Map has an array of View Access Control Block (VACB) pointers which record the base cache address of each view
  - promoted to a sparse form for files > 32MB
- Access interfaces map File+FileOffset to a cache address
- Taking a view miss results in a new mapping, possibly unmapping an unreferenced view in another file (views are recycled LRU)
- Since a view is fixed size, mapping across a view is impossible – Cc returns one address
- Fixed size means no fragmentation …

Cache View Mapping



Cache Manager Readahead

- CcScheduleReadAhead detects patterns on a handle and schedules readahead into the next suspected ranges
  - Regular motion, backwards and forwards, with gaps
  - Private Cache Map contains the per-handle info
  - Called by CcCopyRead and CcMdlRead
- Readahead granularity (64KB) controls the scheduling trigger points and length
  - Small IOs – don’t want readahead every 4KB
  - Large IOs – ya get what ya need (up to 8MB, thanks to Jim Gray)
- CcPerformReadAhead maps and touch-faults pages in a Cc worker thread, will use the new Mm prefetch APIs in a future release

Cache Manager Unmap Behind

- Views are managed on demand (by misses)
- On view miss, Cc will unmap two views behind the current (missed) view before mapping
- Unmapped valid pages go to the standby list in LRU order and can be soft-faulted
- Unmap behind logic is default due to large file read/write operations causing huge swings in working set.
- Mm’s working set trim falls down at the speed a disk can produce pages, Cc must help.

Cache Hints

- Cache hints affect both read ahead and unmap behind
- Two flags specifiable at Win32 CreateFile()
  - FILE_FLAG_SEQUENTIAL_SCAN
    - doubles readahead unit on handle, unmaps to the *front* of the standby list (MRU order) if all handles are SEQUENTIAL
  - FILE_FLAG_RANDOM_ACCESS
    - turns off readahead on handle, turns off unmap behind logic if any handle is RANDOM
- Unfortunately, there is no way to split the effect

Cache Write Throttling

- Avoids out of memory problems by delaying writes to the cache
  - Filling memory faster than writeback speed is not useful, we may as well run into it sooner
- Throttle limit is twofold
  - CcDirtyPageThreshold – dynamic, but ~1500 on all current machines (small, but see above)
  - MmAvailablePages & pagefile page backlog
- CcCanIWrite sees if write is ok, optionally blocking, also serving as the restart test
- CcDeferWrite sets up for callback when write should be allowed (async case)
- *!defwrites debugger extension triages and shows the state of the throttle*

Writing Cached Data

- There are three basic sets of threads involved, only one of which is Cc’s
  - Mm’s modified page writer
    - the paging file
  - Mm’s mapped page writer
    - almost anything else
  - Cc’s lazy writer pool
    - executing in the kernel critical work queue
    - writes data produced through Cc interfaces

The Lazy Writer

- Name is misleading, its really *delayed*
- All files with dirty data have been queued onto CcDirtySharedCacheMapList
- Work queueing – CcLazyWriteScan()
  - Once per second, queues work to arrive at writing 1/8th of dirty data given current dirty and production rates
  - Fairness considerations are interesting
- CcLazyWriterCursor rotated around the list, pointing at the next file to operate on (fairness)
  - 16th pass rule for user and metadata streams
- Work issuing – CcWriteBehind()
  - Uses a special mode of CcFlushCache() which flushes front to back (HotSpots – fairness again)

Valid Data Length (VDL) Calls

- Cache Manager knows highest offset successfully written to disk – via the lazy writer
- File system is informed by special FileEndOfFileInformation call after each write which extends/maintains VDL
- FS which persist VDL to disk (NTFS) push that down here
- FS use it as a hint to update directory entries (recall Fast IO extension, one among several)
- CcFlushCache() flushing front to back is important so we move VDL on disk as soon as possible.

Filesystem Cache Interfaces

- Two distinct access interfaces
  - Map – given File+FileOffset, return a cache address
  - Pin – same, but acquires synchronization – this is a range lock on the stream
    - Lazy writer acquires synchronization, allowing it to serialize metadata production with metadata writing
- Pinning also allows setting of a log sequence number (LSN) on the update, for transactional FS
  - FS receives an LSN callback from the lazy writer prior to range flush

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