Windows Kernel Internals Overview

9 October 2006
Singapore

Dave Probert, Ph.D.
Architect, Windows Kernel Group
Windows Core Operating Systems Division
Microsoft Corporation

History of NT/OS2

- 1988: Bill Gates recruits VMS Architect Dave Cutler
- Business goals:
- an advanced commercial OS for desktops/servers
- compatible with OS/2
- Technical goals:
- scalable on symmetric multiprocessors
- secure, reliable, performant
- portable

NT Timeline first 17 years

2/1989	Coding Begins
7/1993	NT 3.1
9/1994	NT 3.5
5/1995	NT 3.51
7/1996	NT 4.0
12/1999	NT 5.0 Windows 2000
8/2001	*NT 5.1 Windows XP*
3/2003	NT 5.2 Server 2003
8/2004	NT 5.2 Windows XP SP2
4/2005	NT 5.2 Windows XP 64 Bit Edition (& WS03SP1)
2006	NT 6.0 Windows Vista (client)

Important NT kernel features

- Highly multi-threaded
- Completely asynchronous I/O model
- Thread-based scheduling
- Object-manager provides unified management of
    - kernel data structures
    - kernel references
    - user references (handles)
    - namespace
    - synchronization objects
    - resource charging
    - cross-process sharing
- Centralized ACL-based security reference monitor
- Configuration store decoupled from file system

Important NT kernel features (cont)

- Extensible filter-based I/O model with driver layering, standard device models, notifications, tracing, journaling, namespace, services/subsystems
- Virtual address space managed separately from memory objects
- Advanced VM features for databases (app management of virtual addresses, physical memory, I/O, dirty bits, and large pages)
- Plug-and-play, power-management
- System library mapped in every process provides trusted entrypoints

Major Kernel Functions

• Manage naming & security	➢ **OB, SE**
• Manage address spaces	➢ **PS, MM**
• Manage physical memory	➢ **MM, CACHE**
• Manage CPU	➢ **KE**
• Provide I/O & net abstractions	➢ **IO, drivers**
• Implement cross-domain calls	➢ **LPC**
• Abstract low-level hardware	➢ **HAL**
• Internal support functions	➢ **EX, RTL**
• Internal configuration mgmt	➢ **CONFIG**

Major NT Kernel Components

- **OB – Object Manager**
- **SE – Security Reference Monitor**
- **PS – Process/Thread management**
- **MM – Memory Manager**
- **CACHE – Cache Manager**
- **KE – Scheduler**
- **IO – I/O manager, PnP, device power mgmt, GUI**
- **Drivers – devices, file systems, volumes, network**
- **LPC – Local Procedure Calls**
- **HAL – Hardware Abstraction Layer**
- **EX – Executive functions**
- **RTL – Run-Time Library**
- **CONFIG – Persistent configuration state (registry)**

Major Kernel Services

Object Manager

- Naming, referencing, synchronizing

Process management

- Process/thread creation

Security reference monitor

- Access checks, token management

Memory manager

- Virtual address mgmt, physical memory mgmt, paging, Services for sharing, copy-on-write, mapped files, GC support, large apps

Lightweight Procedure Call (LPC)

- Native transport for RPC and user-mode system services.

I/O manager (& plug-and-play & power)

- Maps user requests into IRP requests, configures/manages I/O devices, implements services for drivers

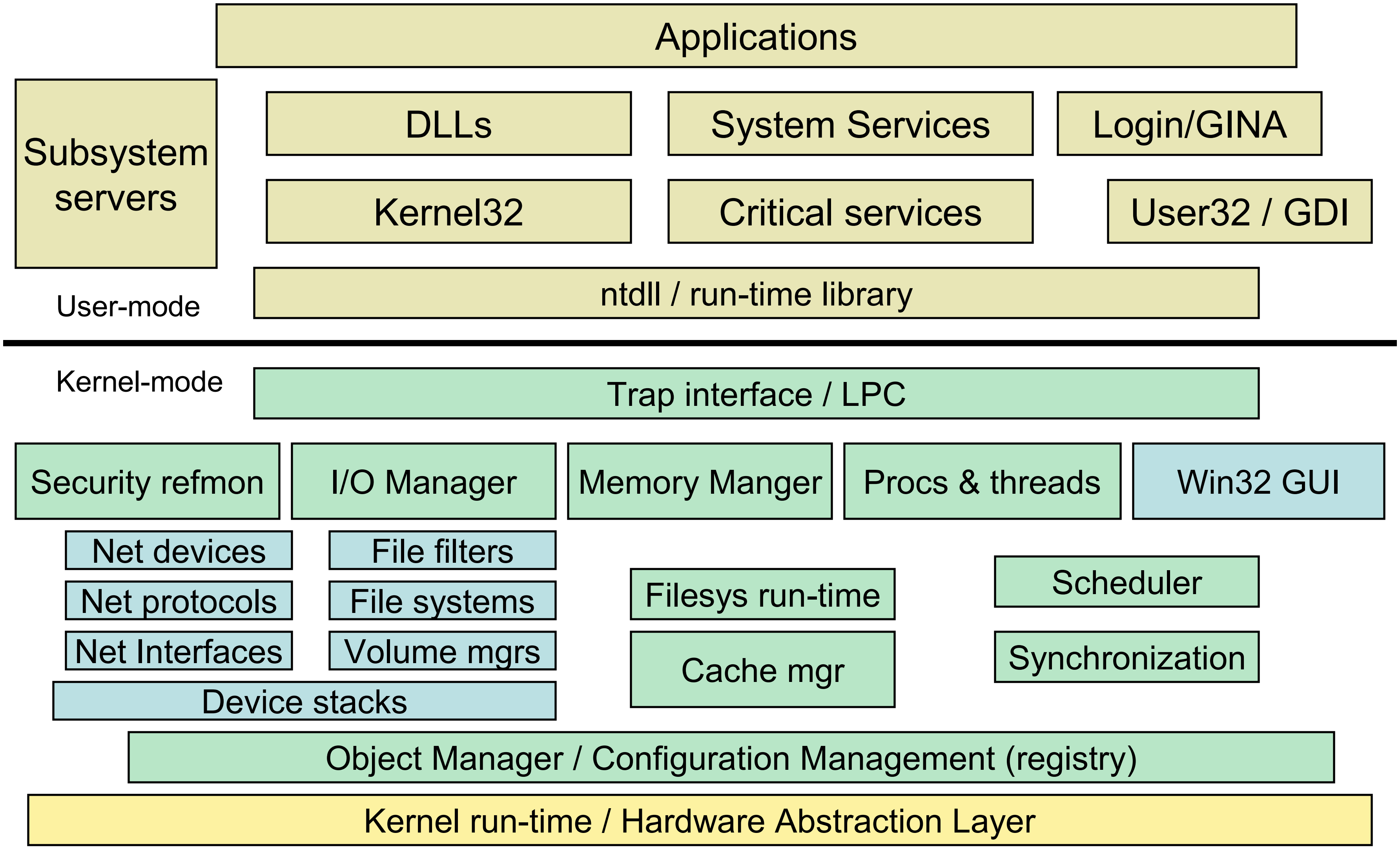
Cache manager

- Provides file-based caching to buffer file system I/O

Scheduler (aka 'kernel')

- Schedules thread execution on each processor

Windows Architecture



Windows Kernel Organization

Kernel-mode organized into

NTOS (kernel-mode services)

- Run-time Library, Scheduling, Executive services, object manager, services for I/O, memory, processes, …

HAL (hardware-adaptation layer)

- Insulates NTOS & drivers from hardware details
- Providers facilities, such as device access, timers, interrupt servicing, clocks, spinlocks

Drivers

- Kernel extensions (devices, file systems, network)

Namespace Components

Manage naming and security

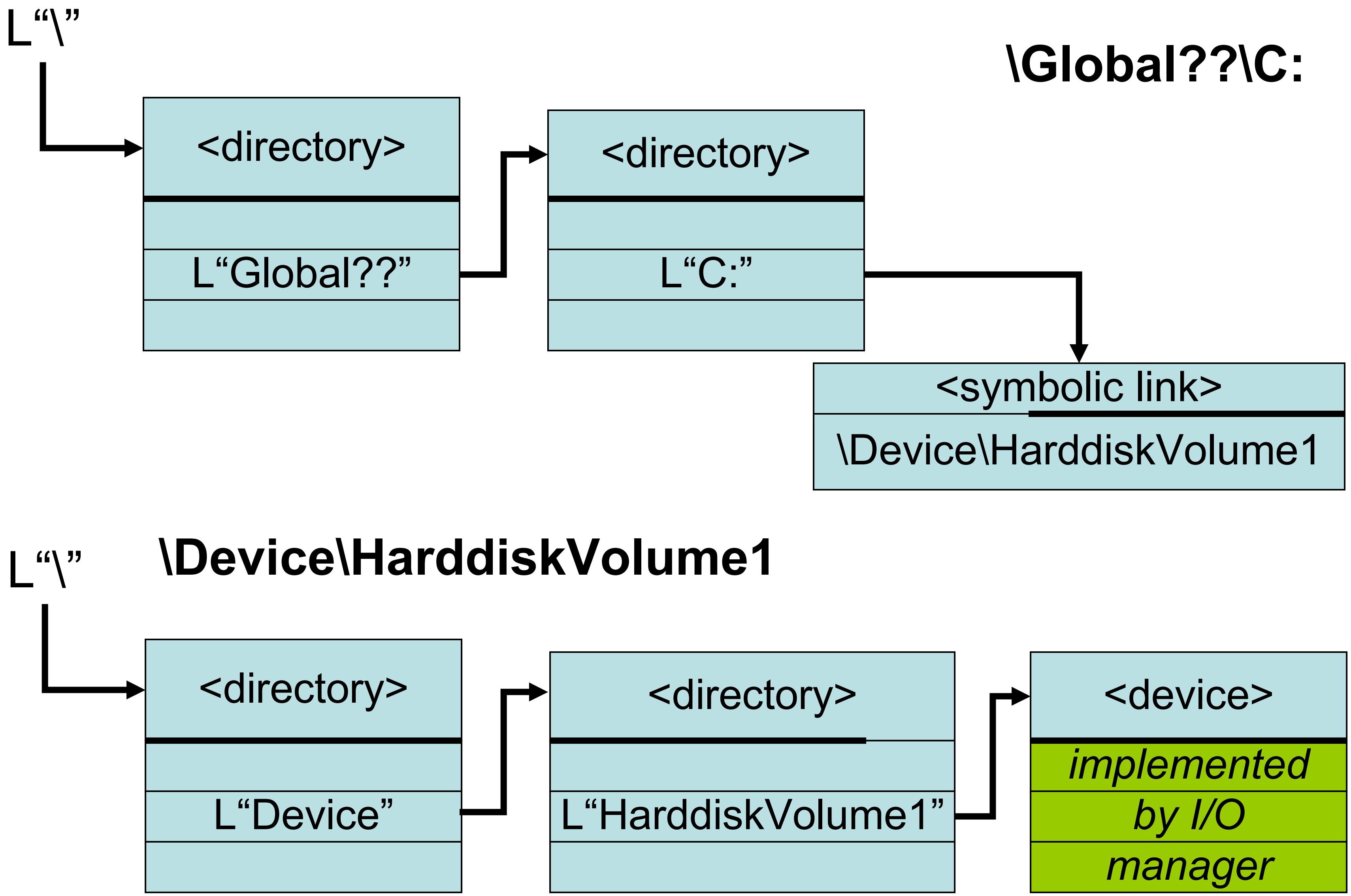
Manage references to kernel data structures

Extensible mechanisms, scalable

Provides general synchronization

NT Object Manager

- Provides underlying NT namespace
- Unifies kernel data structure referencing
- Unifies user-mode referencing via handles
- Simplifies resource charging
- Central facility for security protection
- Other namespaces ‘mount’ on OB nodes
- Provides device & I/O support

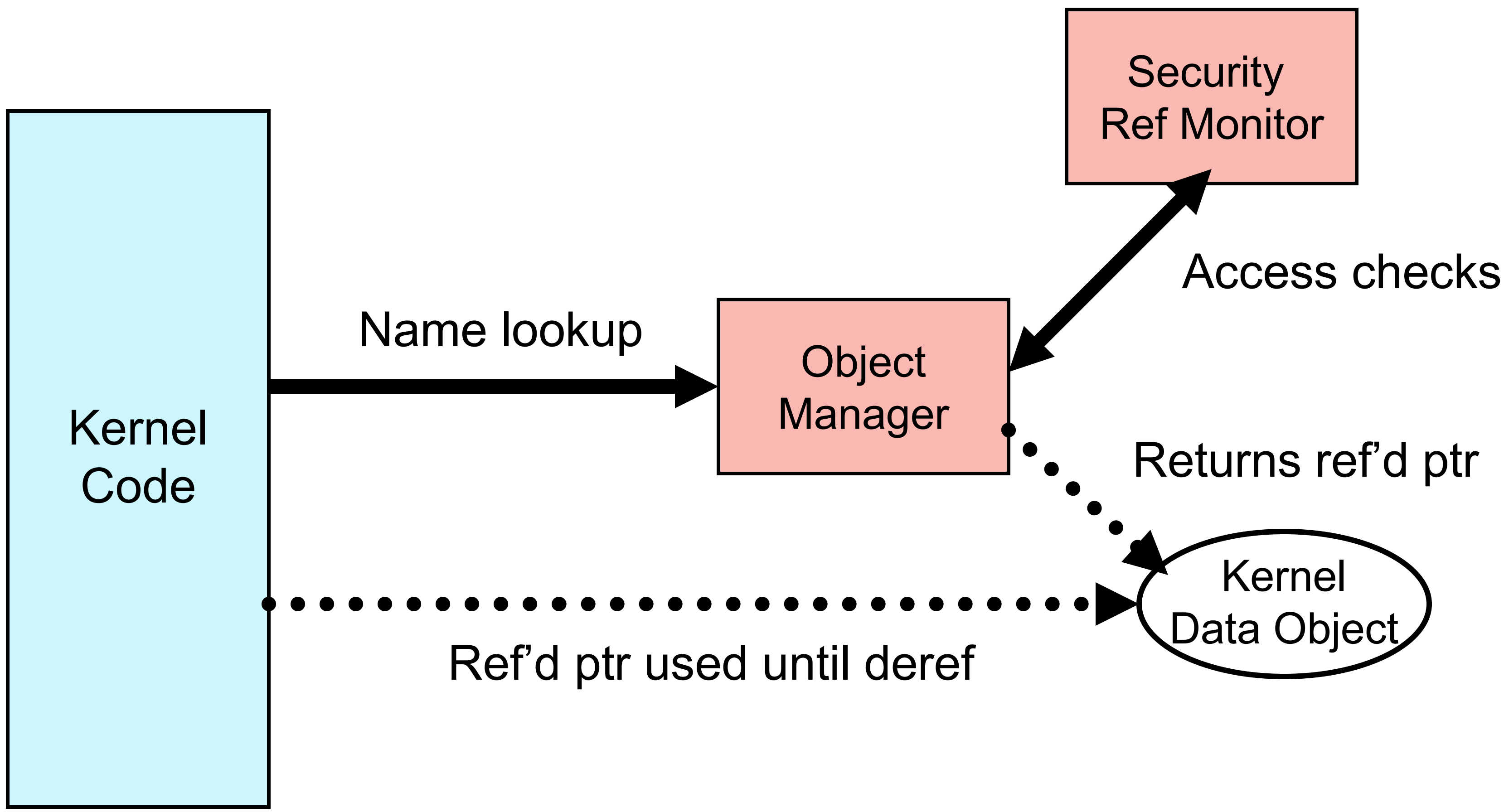


L“\”
\Global??\C:
<directory>
L“Global??”
<directory>
L“C:”
<symbolic link>
\Device\HarddiskVolume1
L“\”
\Device\HarddiskVolume1
<directory>
L“Device”
<directory>
L“HarddiskVolume1”
<device>
implemented
by I/O
manager

Security Reference Monitor

- Based on discretionary access controls
  - Single module for access checks
  - Implements Security Descriptors, System and Discretionary ACLs, Privileges, and Tokens
  - Collaborates with Local Security Authority Service to obtain authenticated credentials
  - Provides auditing and fulfills other Common Criteria requirements

Object Mgr and Sec Monitor



OB Namespace: objdir \

ArcName	Directory	NLS	Directory
BaseNamedObjects	Directory	Ntfs	Device
Callback	Directory	ObjectTypes	Directory
Cdfs	Device	REGISTRY	Key
Device	Directory	RPC Control	Directory
Dfs	Device	SAM_SERVICE_STARTED	Event
DosDevices	SymbolicLink - \??	Security	Directory
Driver	Directory	SeLsaCommandPort	Port
ErrorLogPort	Port	SeLsaInitEvent	Event
FileSystem	Directory	SeRmCommandPort	Port
GLOBAL??	Directory	Sessions	Directory
i8042PortAccessMutex	Event	SmApiPort	Port
KernelObjects	Directory	SmSsWinStationApiPort	Port
KnownDlls	Directory	SystemRoot	SymbolicLink - \Device\Harddisk0\Partition1\WINDOWS
LanmanServerAnnounceEvent	Event	ThemeApiPort	Port
LsaAuthenticationPort	Port	UniqueSessionIdEvent	Event
NETLOGON_SERVICE_STARTED	Event	Windows	Directory
NLAPrivatePort	WaitablePort	XactSrvLpcPort	Port
NLAPublicPort	WaitablePort		

OB Extensibility: Object Methods

Note that the methods are unrelated to actual operations on the underlying objects:

OPEN:	Create/Open/Dup/Inherit handle
CLOSE:	Called when each handle closed
DELETE:	Called on last dereference
PARSE:	Called looking up objects by name
SECURITY:	Usually *SeDefaultObjectMethod*
QUERYNAME:	Return object-specific name

OB Extensibility: \ObjectTypes

Adapter	File	Semaphore
Callback	IoCompletion	SymbolicLink
Controller	Job	Thread
DebugObject	Key	Timer
Desktop	KeyedEvent	Token
Device	Mutant	Type
Directory	Port	WaitablePort
Driver	Process	WindowsStation
Event	Profile	WMIGuid
EventPair	Section	

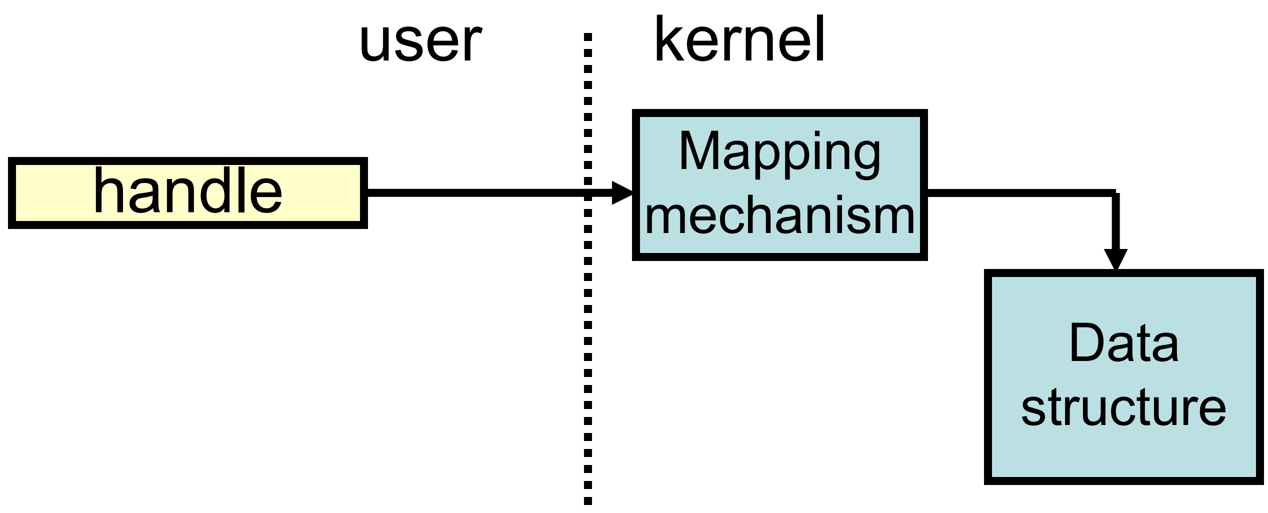
OB Extensibility: \ObjectTypes

Adapter	**File**	Semaphore
Callback	IoCompletion	**SymbolicLink**
Controller	Job	Thread
DebugObject	**Key**	Timer
Desktop	KeyedEvent	Token
Device	Mutant	Type
Directory	Port	WaitablePort
Driver	Process	WindowsStation
Event	Profile	WMIGuid
EventPair	Section	

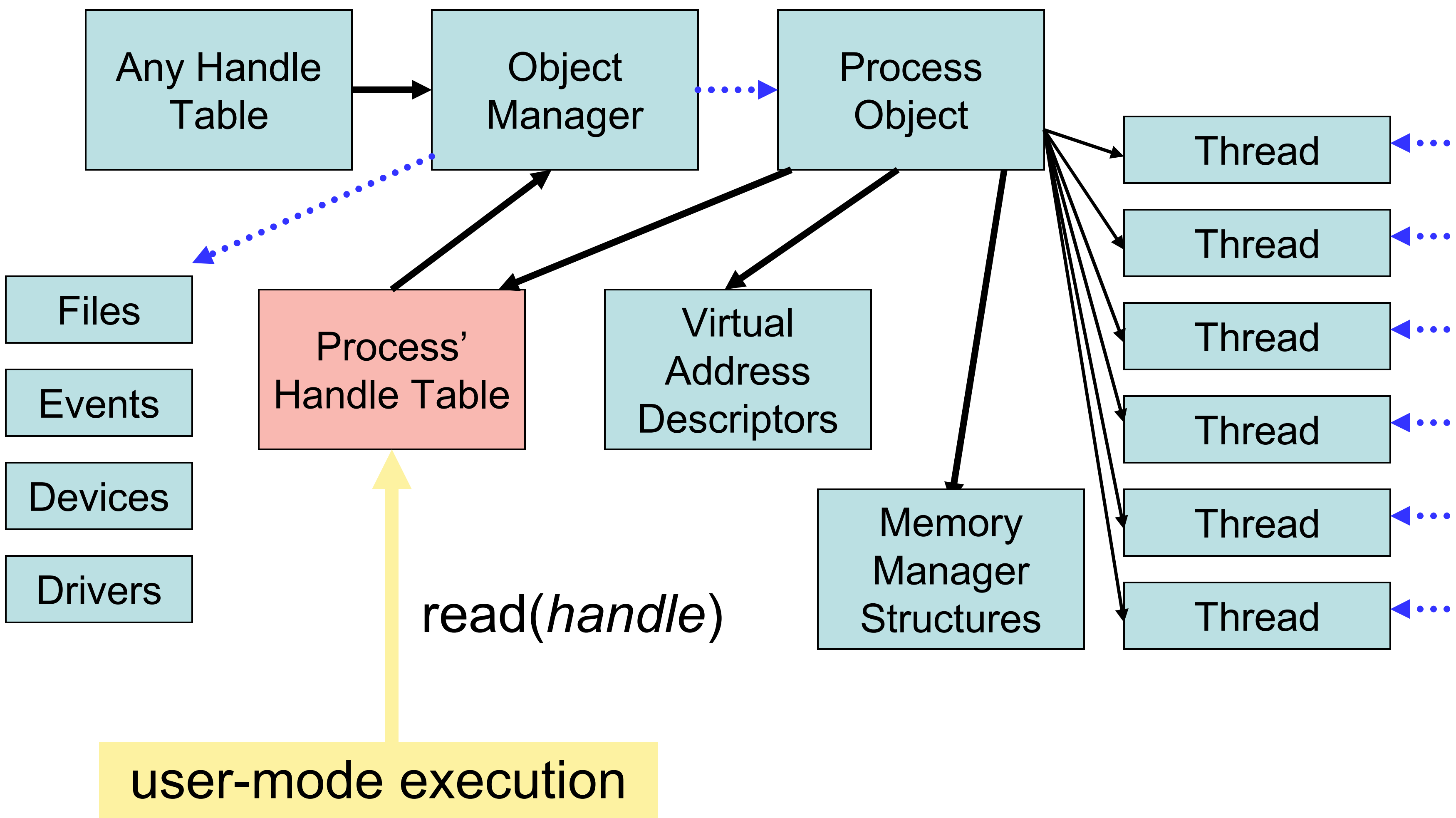
Object referencing: Handles

General mechanism: shorthand for referencing an opaque data structure

e.g. a kernel structure (file, process, …)



Process/Thread structure



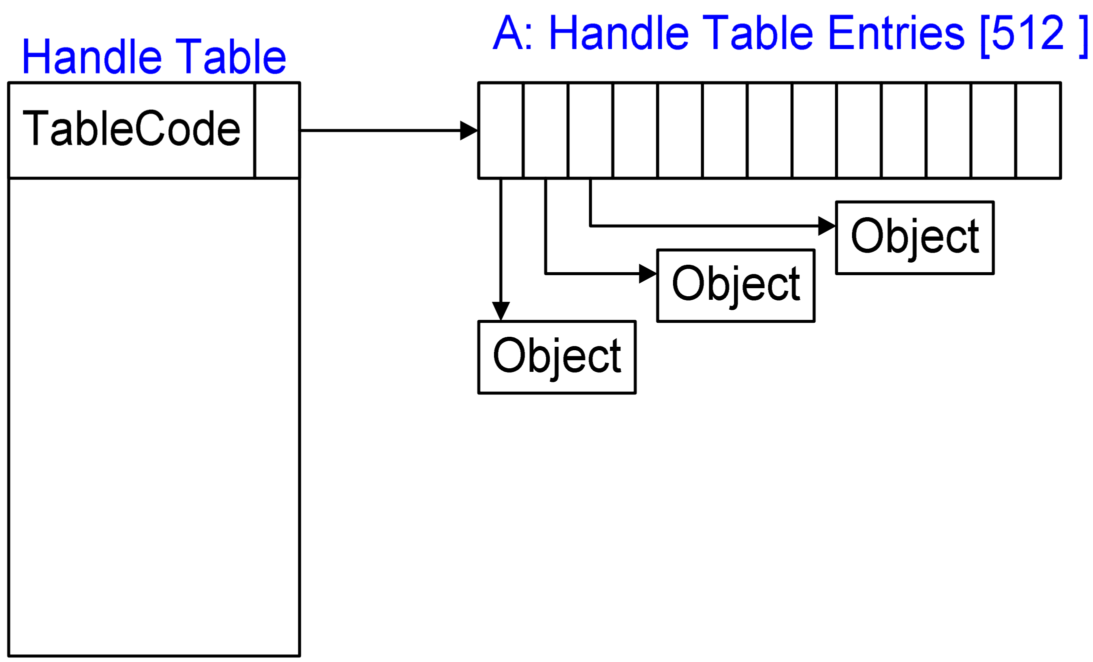
Handle Table

- NT handles allow user code to reference kernel data structures (similar, but more general than UNIX file descriptors)
- NT APIs use explicit handles to refer to objects (simplifying cross-process operations)
- Handles can be used for synchronization, including WaitMultiple
- Implementation is highly scalable

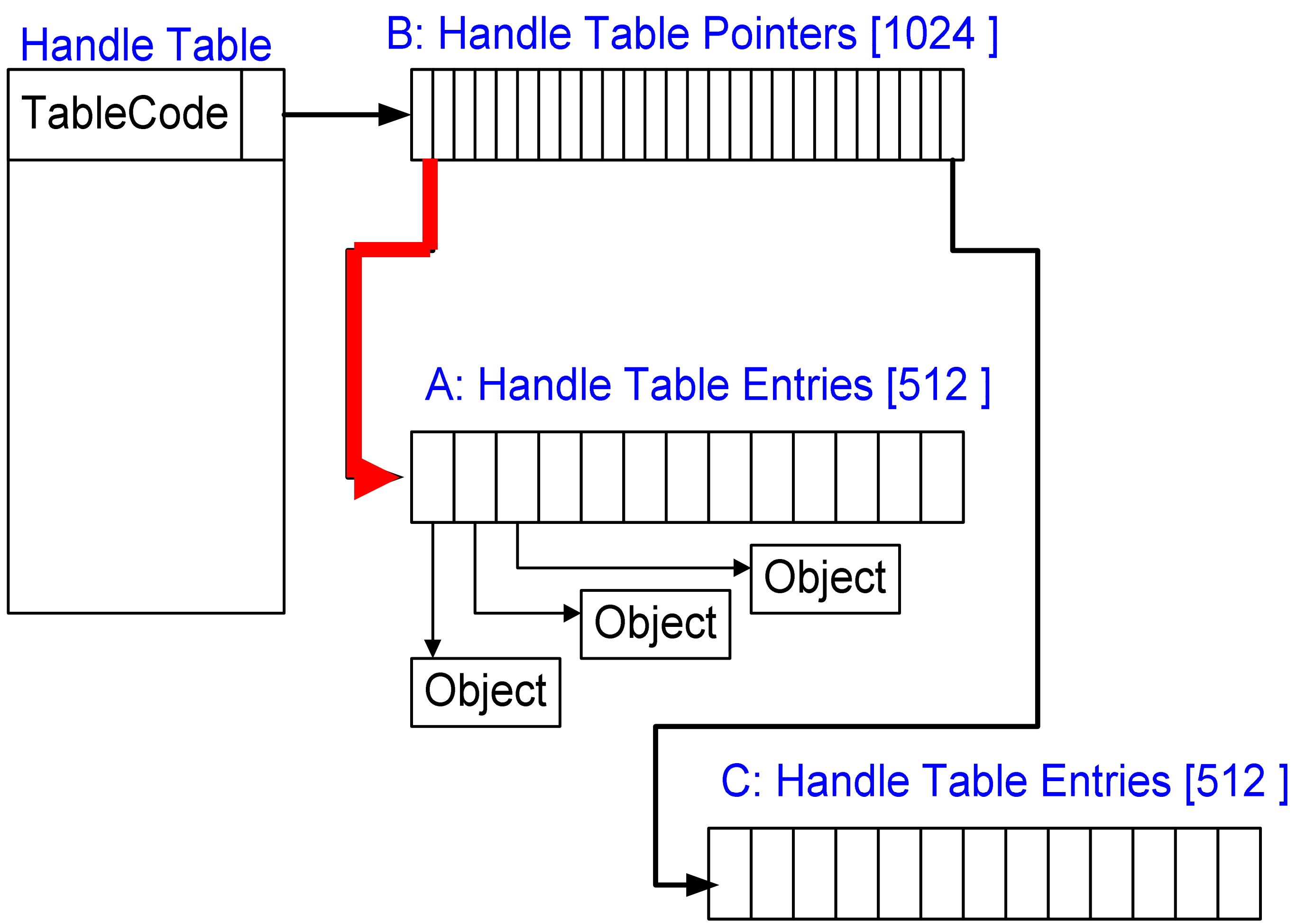
Handle Table Requirements

- Perform well (time & memory) across a broad range of handle table sizes
- Handles can’t change as table expands
- Efficient *allocate*, *duplicate*, *free* operations
- Scalable performance on high-MP systems

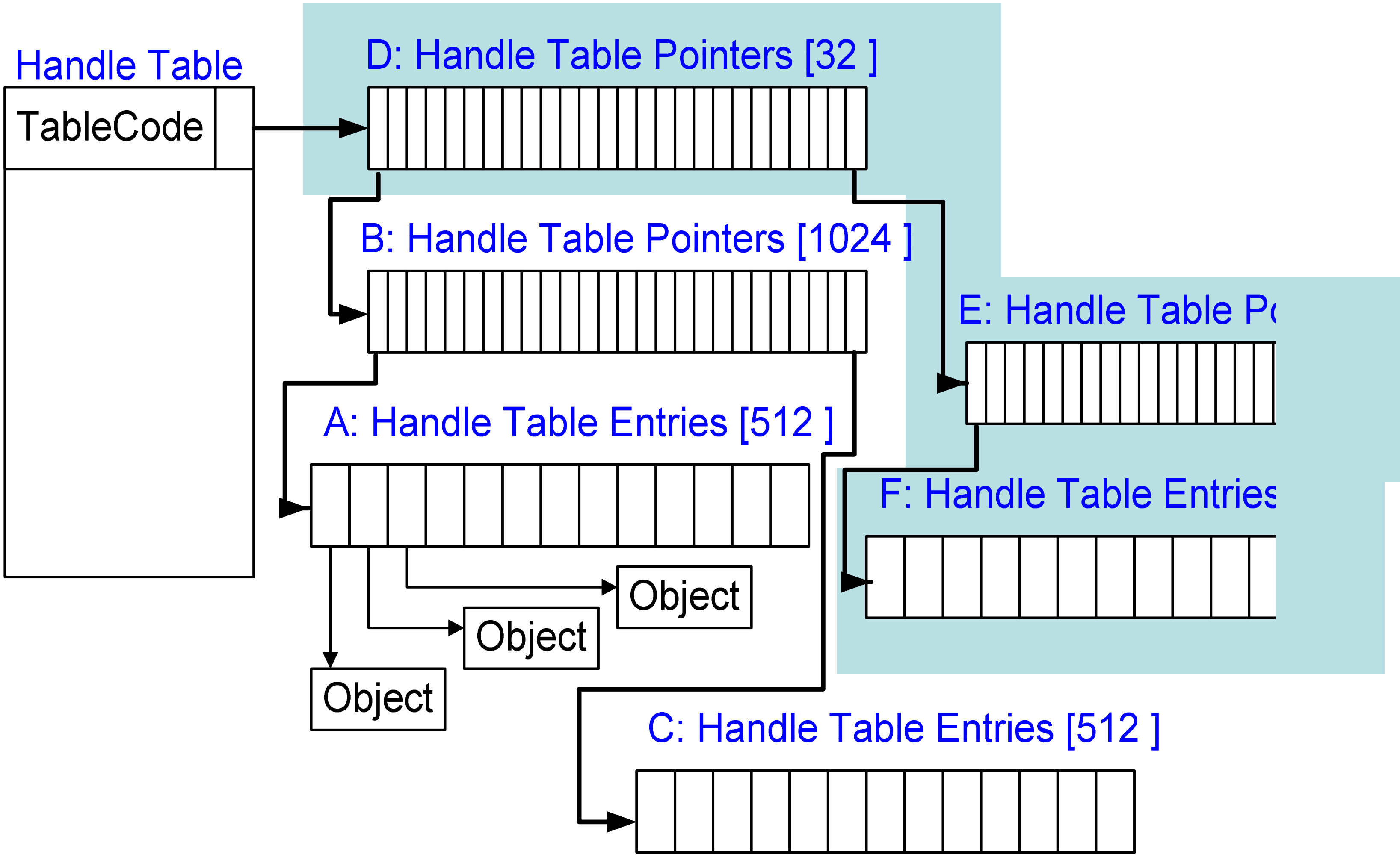
One level: (to 512 handles)



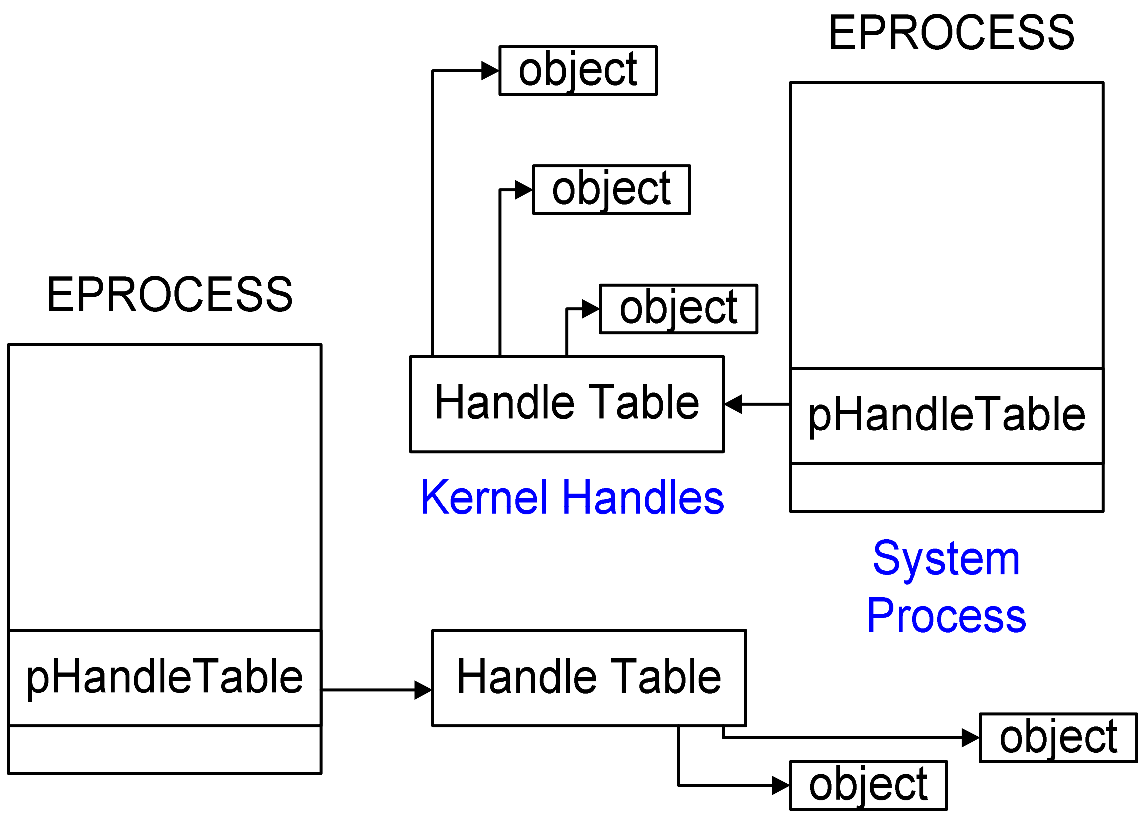
Two levels: (to 512K handles)



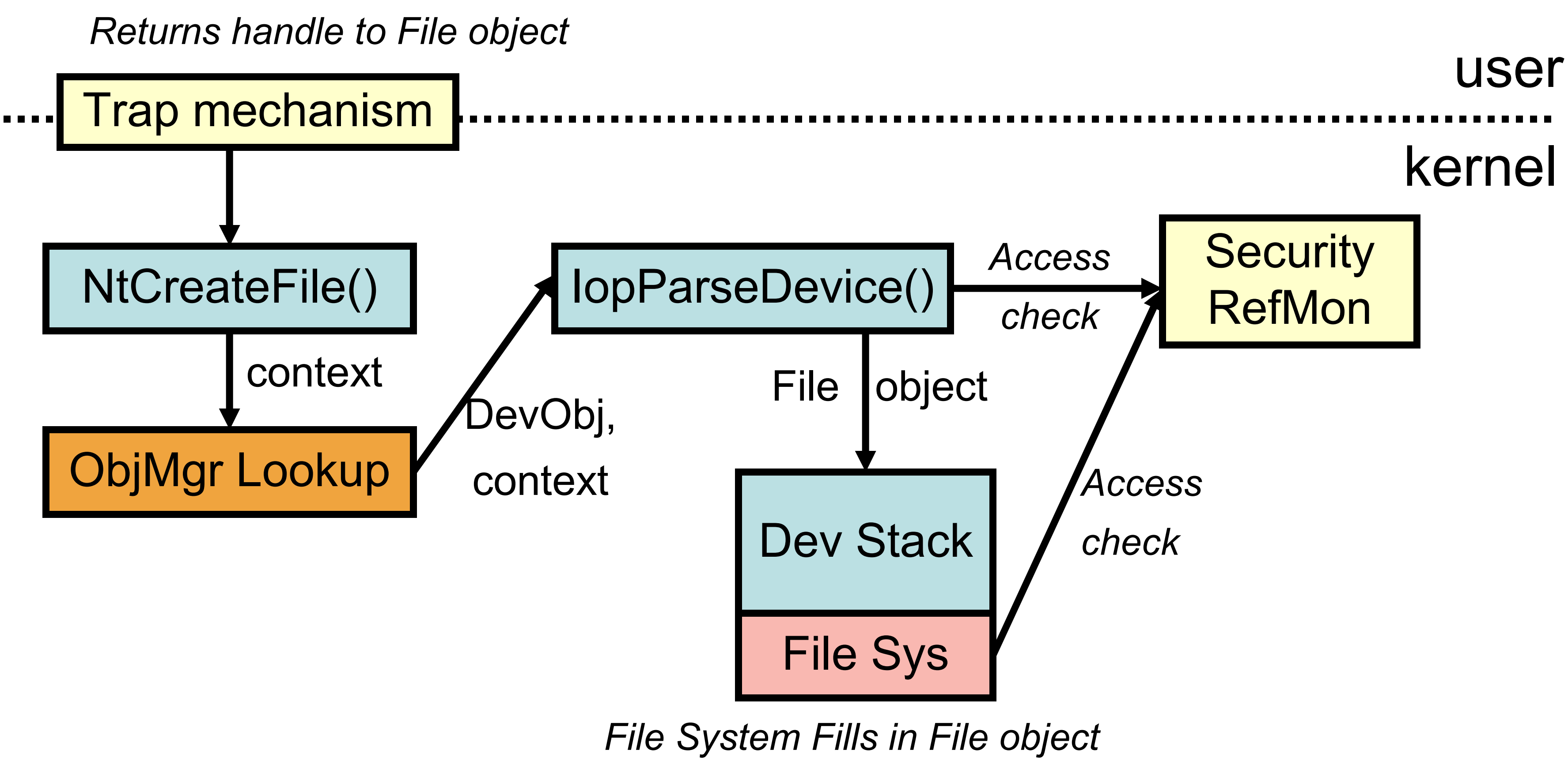
Three levels: (to 16M handles)



Kernel Handles



IO Support: IopParseDevice



Object Manager Implementation

- Implements standard operations
  - Open, close, delete, parse, security, query
- Dynamic definition of OB types, including callbacks for standard ops and allocation
- Implements a unified API
  - OpenByName, reference, dereference
  - Namespace and synchronization functions
- Relies on Security Reference Monitor
- Every object has standard OBJECT_HEADER

OBJECT_HEADER

<table>
<tr><td colspan="4">PointerCount</td></tr>
<tr><td colspan="4">HandleCount</td></tr>
<tr><td colspan="4">pObjectType</td></tr>
<tr><td>oNameInfo</td><td>oHandleInfo</td><td>oQuotaInfo</td><td>Flags</td></tr>
<tr><td colspan="4">pQuotaBlockCharged</td></tr>
<tr><td colspan="4">pSecurityDescriptor</td></tr>
<tr><td colspan="4">CreateInfo + NameInfo + HandleInfo + QuotaInfo</td></tr>
<tr><td colspan="4">OBJECT BODY
[with optional DISPATCHER_HEADER]</td></tr>
</table>

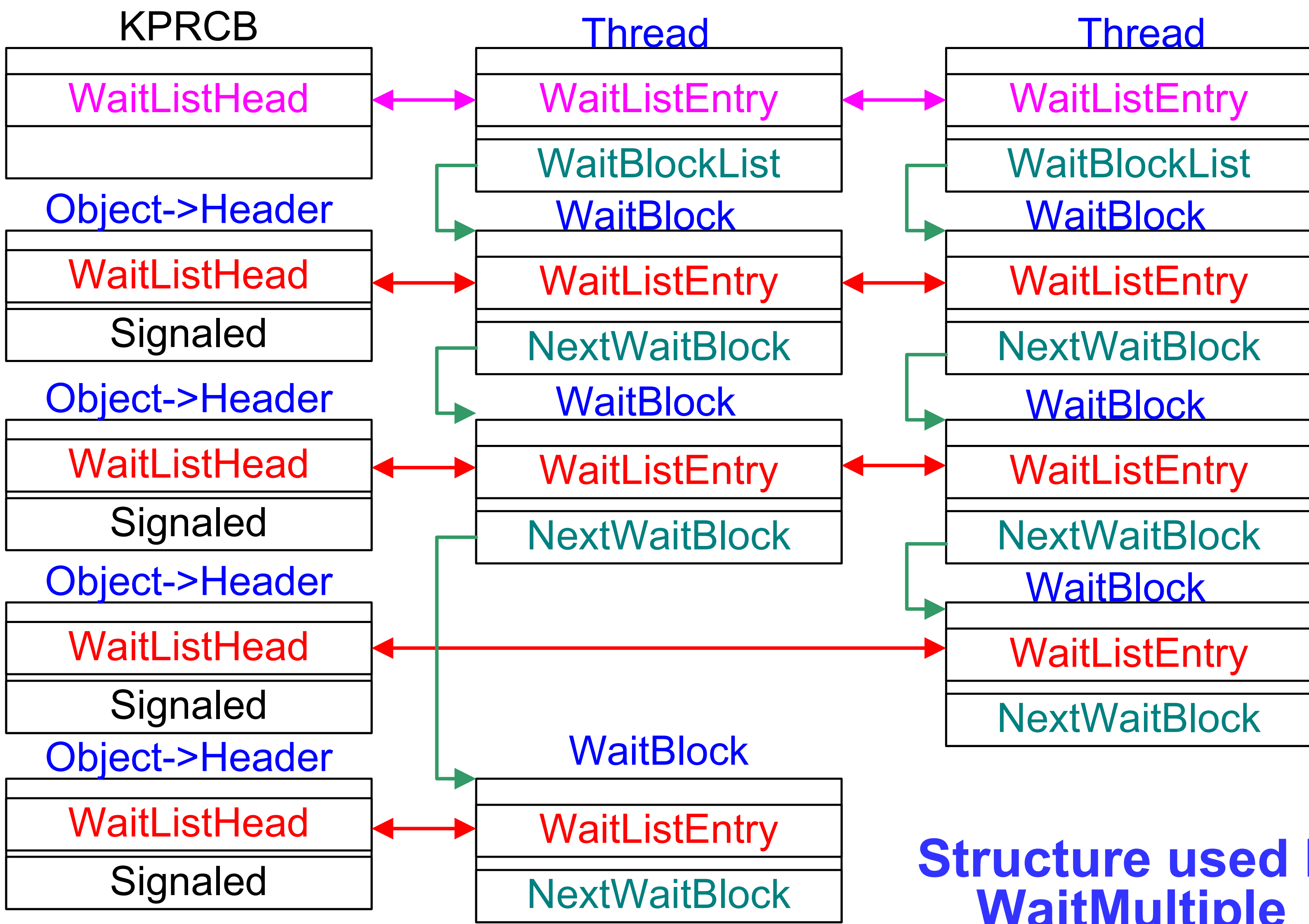
Uniform Synchronization: **DISPATCHER_HEADER**

Fundamental kernel synchronization mechanism

Equivalent to a KEVENT at front of *dispatcher objects*

Object Body →

<table>
<tr><td>Inserted</td><td>Size</td><td>Absolute</td><td>Type</td></tr>
<tr><td colspan="4">SignalState</td></tr>
<tr><td colspan="4">WaitListHead.flink</td></tr>
<tr><td colspan="4">WaitListHead.blink</td></tr>
</table>

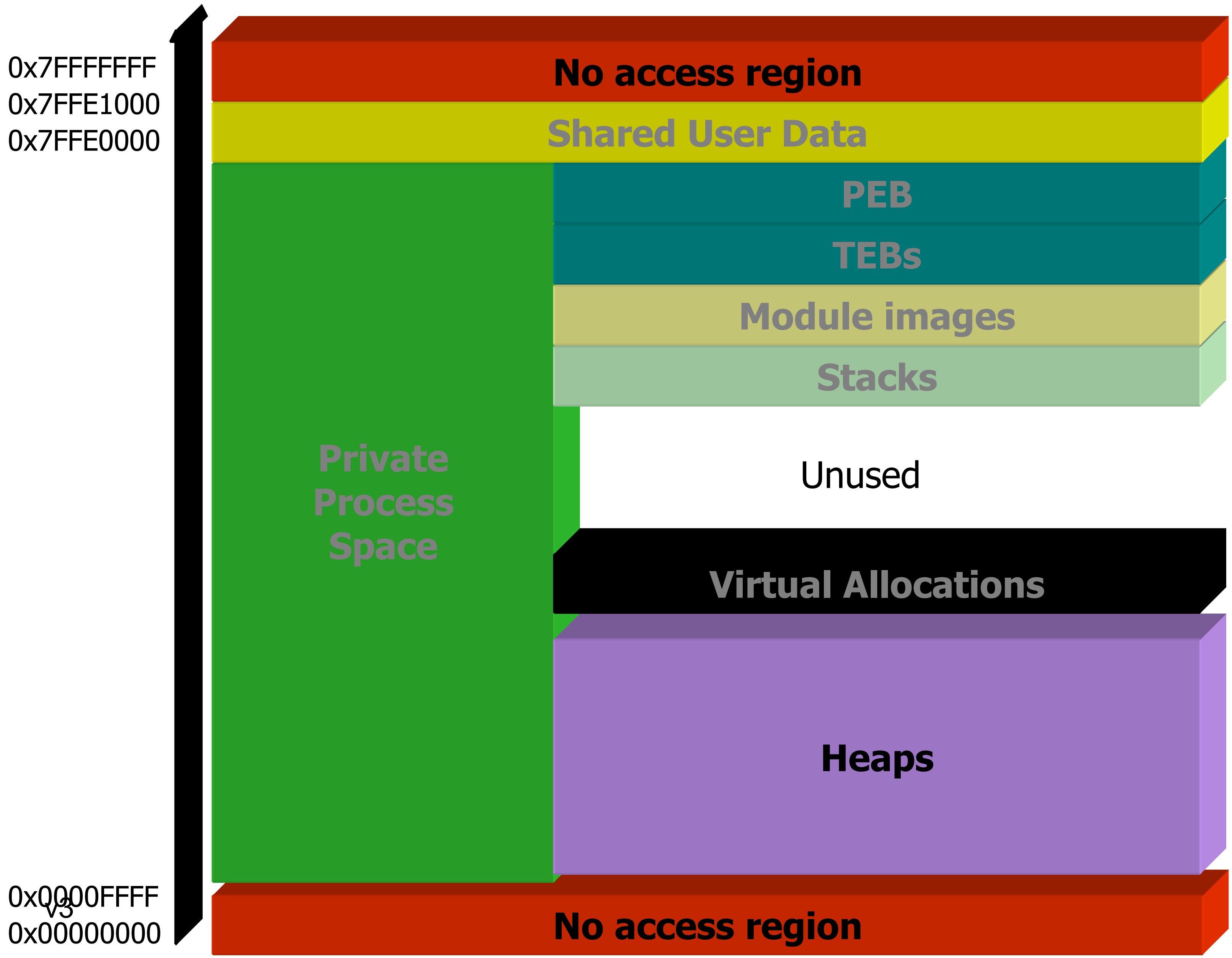


Structure used by WaitMultiple

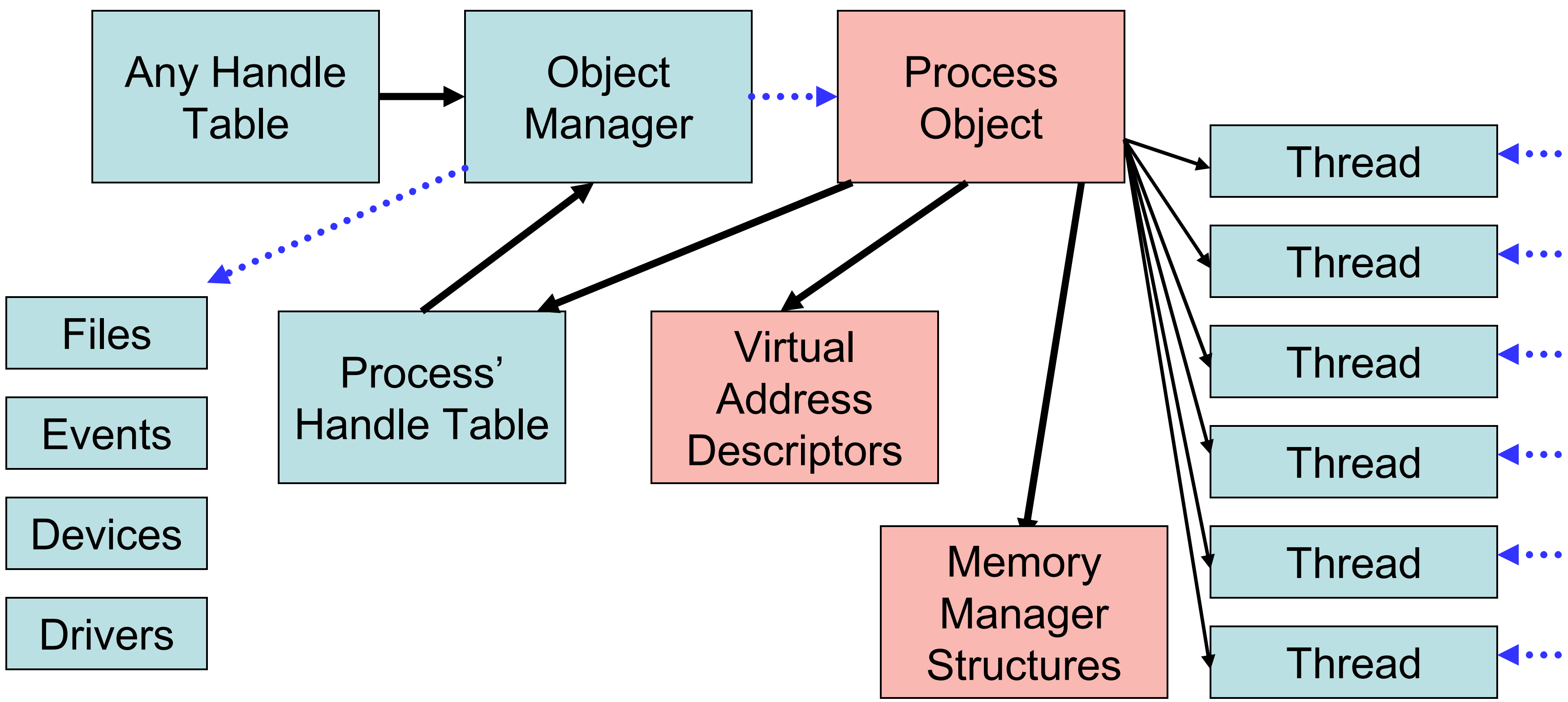
Address Spaces Memory Mgmt

- Virtual Address management, processes
- Shared memory, cache management
- Virtual Address Translation, page tables
- Physical pageframe (& pagefile) management
- Large app support

Address Space Layout (2GB mode)



Process/Thread structure



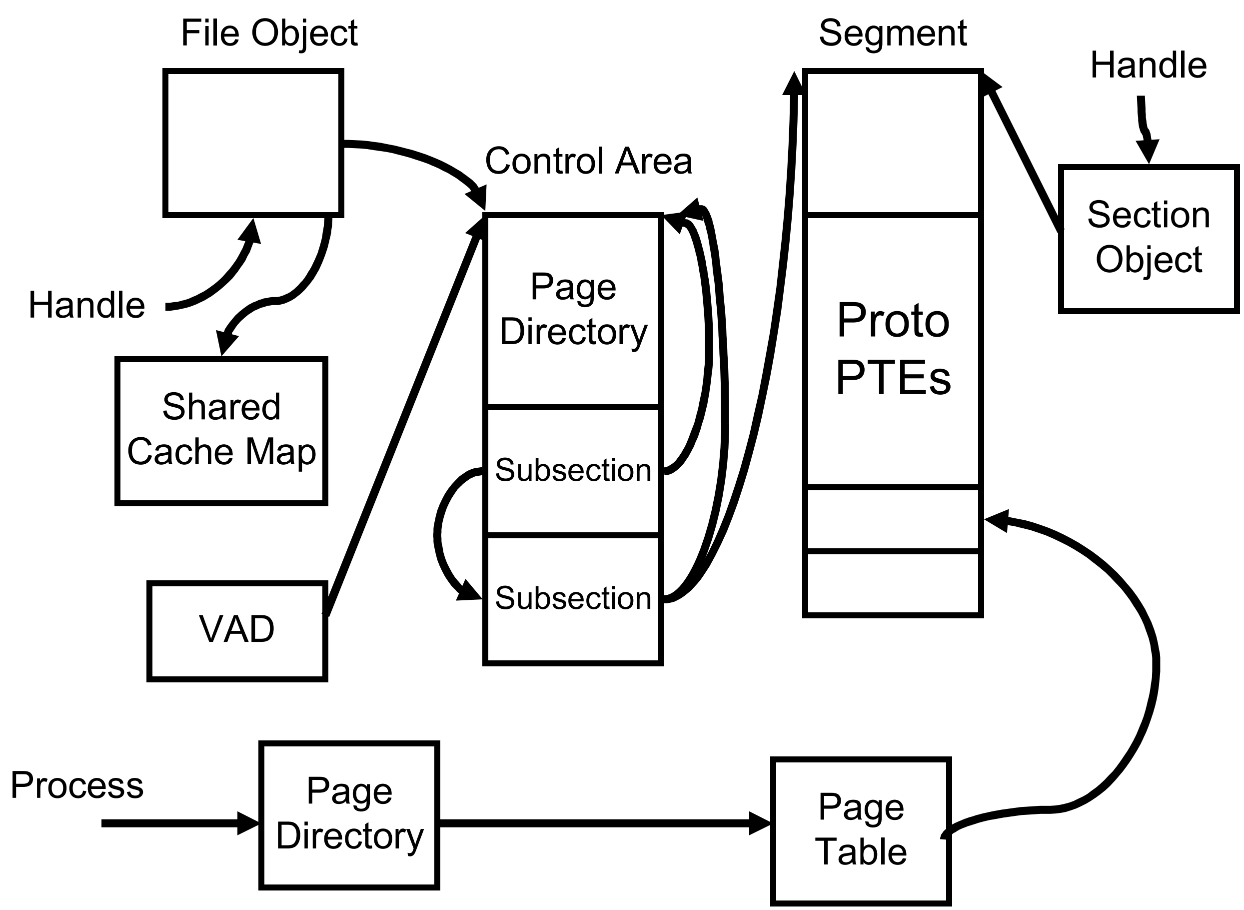
Processes

- An environment for program execution (conceptually)
- Binds
  - namespaces
  - virtual address mappings
  - ports (debug, exceptions)
  - threads
- *Not* a virtualization of a processor

Virtual Address Descriptors

- Tree representation of an address space
- Types of VAD nodes
  - invalid
  - reserved
  - committed
  - committed to backing store
  - app-managed (large pages, AWE, physical)
- Backing store represented by section objects

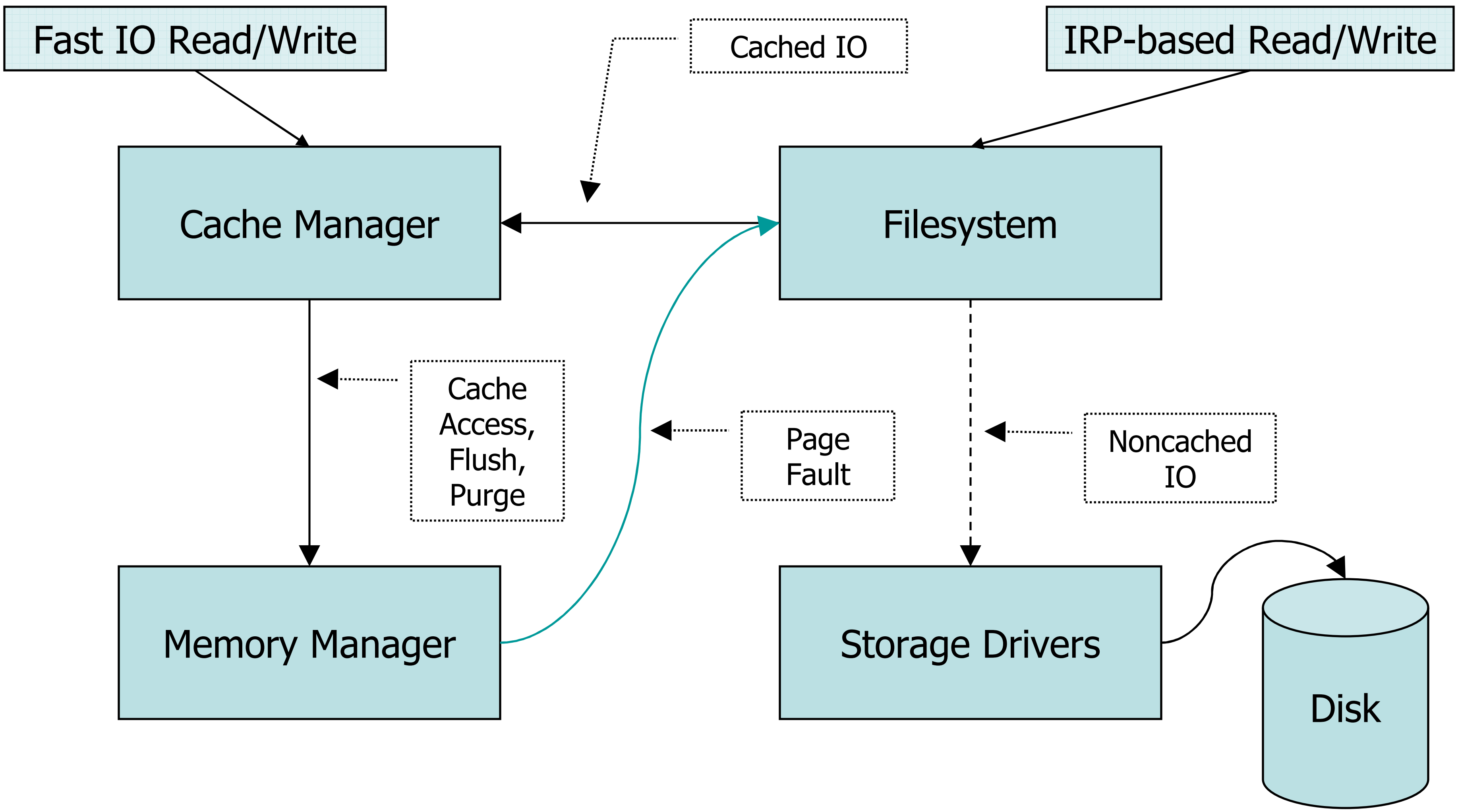
Shared Memory Data Structures



Cache Manager Summary

- Virtual block cache for files not logical block cache for disks
- Memory manager is the ACTUAL cache manager
- Cache Manager context integrated into FileObjects
- Cache Manager manages views on files in kernel virtual address space
- I/O has special fast path for cached accesses
- The Lazy Writer periodically flushes dirty data to disk
- Filesystems need two interfaces to CC: map and pin

The Big Block Diagram



Filesystem & Cache Manager

- 3 basic types of I/O: *cached, noncached and “paging”*
- Paging I/O is I/O generated by Mm – *flushing or faulting*
  - the data section implies the file is big enough
  - can never extend a file
- A filesystem will recurse on the same callstack as Mm dispatches cache pagefaults
  - This makes things exciting! (ERESOURCEs)

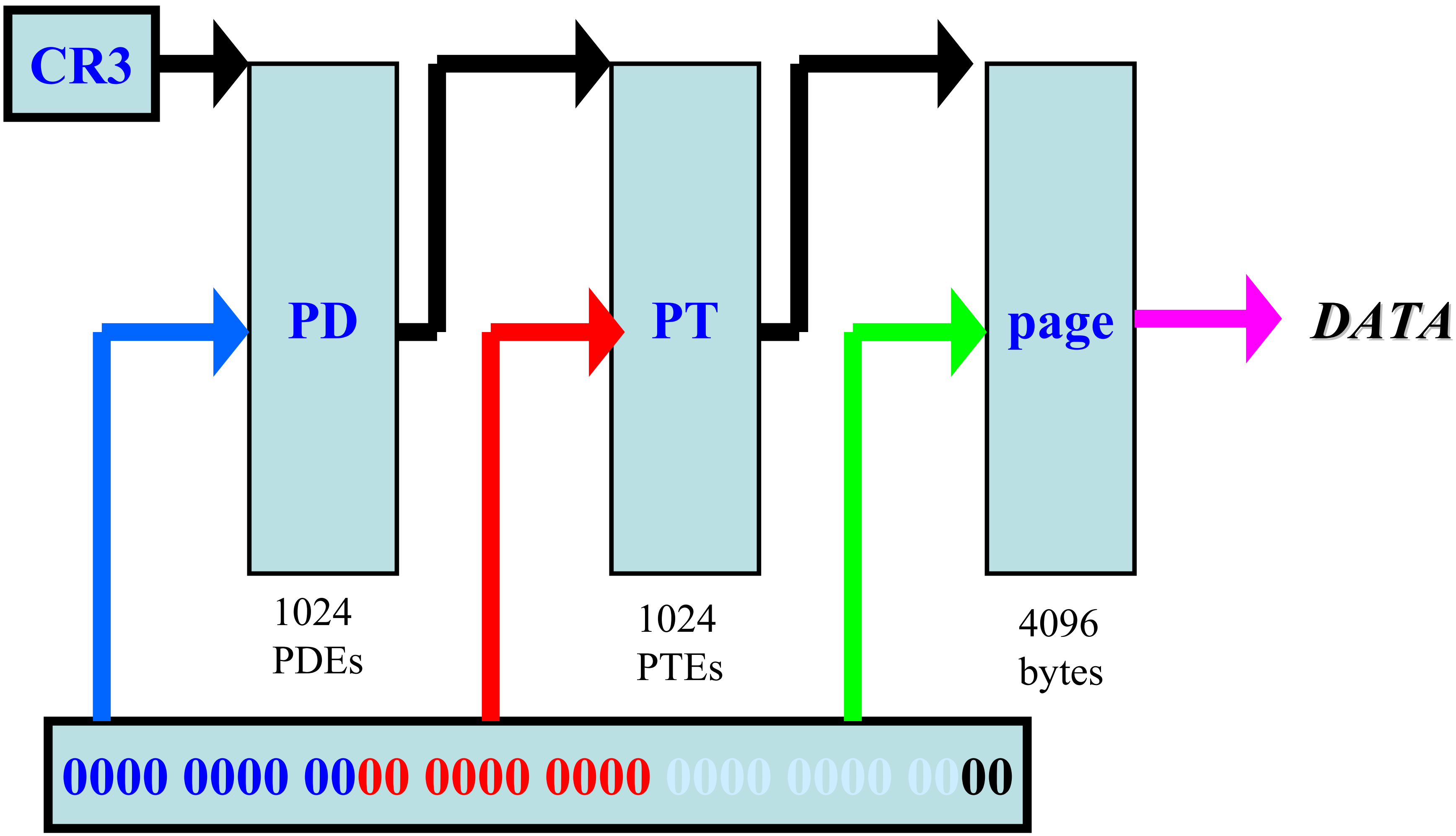
Three File Sizes

- FileSize – normal length expected by the user
- AllocationSize – backing store allocated on the volume
  - multiple of cluster size, which is 2^n * sector size
- ValidDataLength – size written so far
  - ValidDataLength <= FileSize <= AllocationSize

Letting the Filesystem Into The Cache

- 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

Virtual Address Translation



Self-mapping page tables

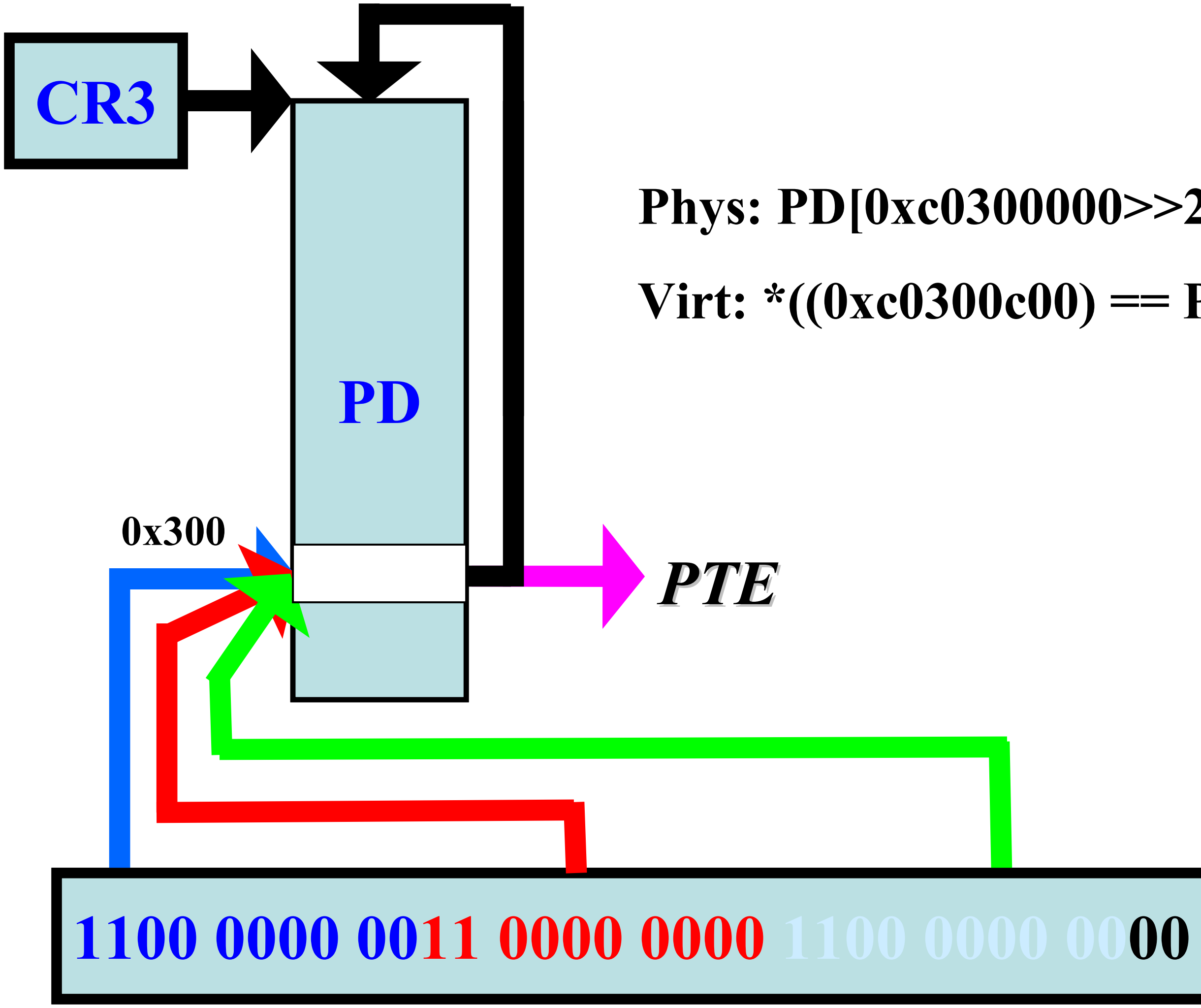
- Page Table Entries (PTEs) and Page Directory Entries (PDEs) contain **Physical Frame Numbers (PFNs)**
  - But Kernel runs with **Virtual Addresses**
- To access PDE/PTE from kernel use the self-map for the current process:

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

Virtual Access to PageDirectory[0x300]

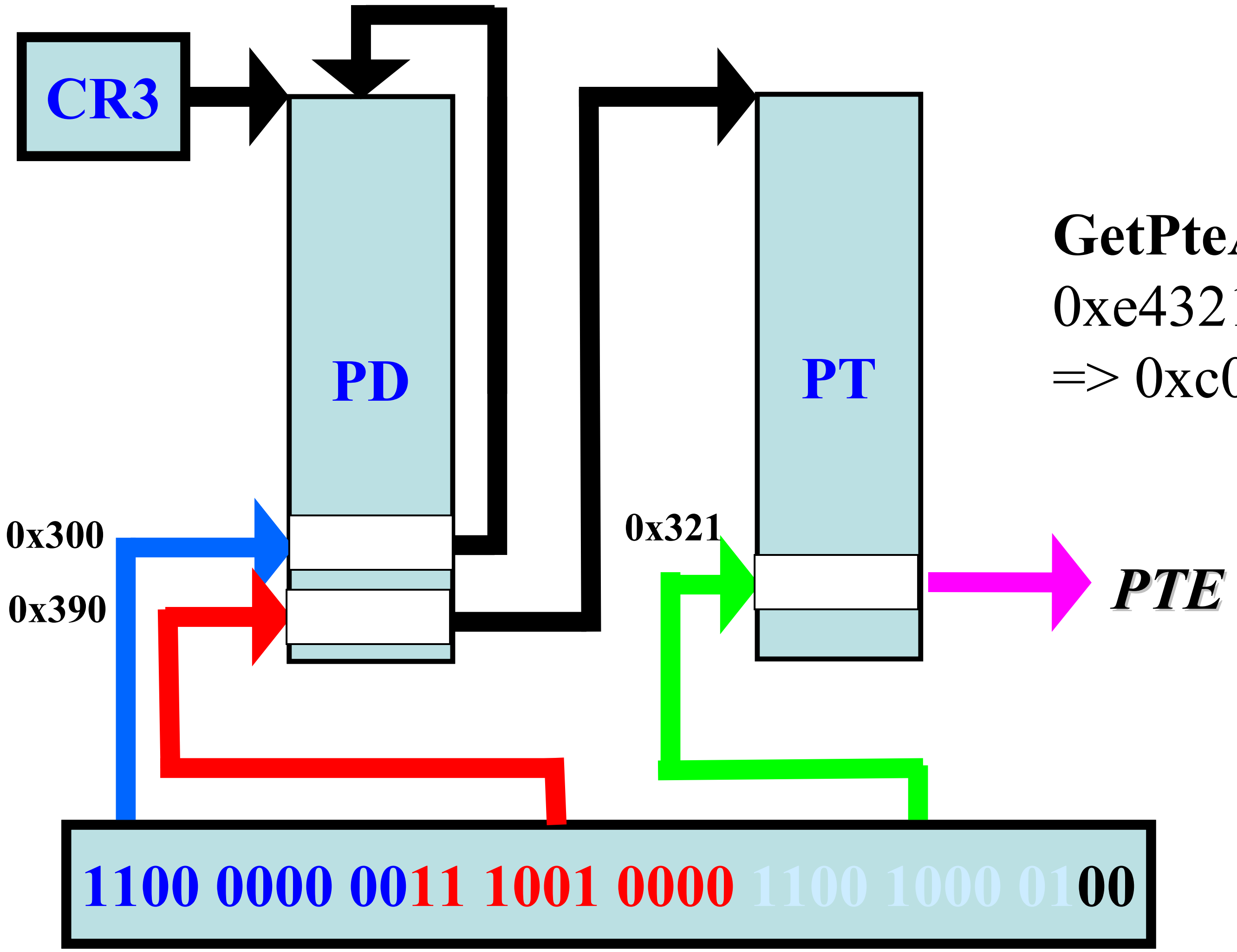


Phys: PD[0xc0300000>>22] = PD

Virt: *((0xc0300c00) == PD

Self-mapping page tables

Virtual Access to PTE for va 0xe4321000



GetPteAddress:
0xe4321000
=> 0xc0390c84

Writing Cached Data

- There are three basic sets of threads involved, only one of which is Cc’s
  - Mm’s modified page writer (paging file)
  - Mm’s mapped page writer (mapped file)
  - Cc’s lazy writer pool (cleans data in cache)

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

Physical Frame Management

- Table of PFN data structures
  - represent all pageable pages
  - synchronize page-ins
  - linked to management lists
- Page Tables
  - hierarchical index of page directories and tables
  - leaf-node is *page table entry* (PTE)
  - PTE states:
    - Active/valid
    - Transition
    - Modified-no-write
    - Demand zero
    - Page file
    - Mapped file

Paging Overview

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

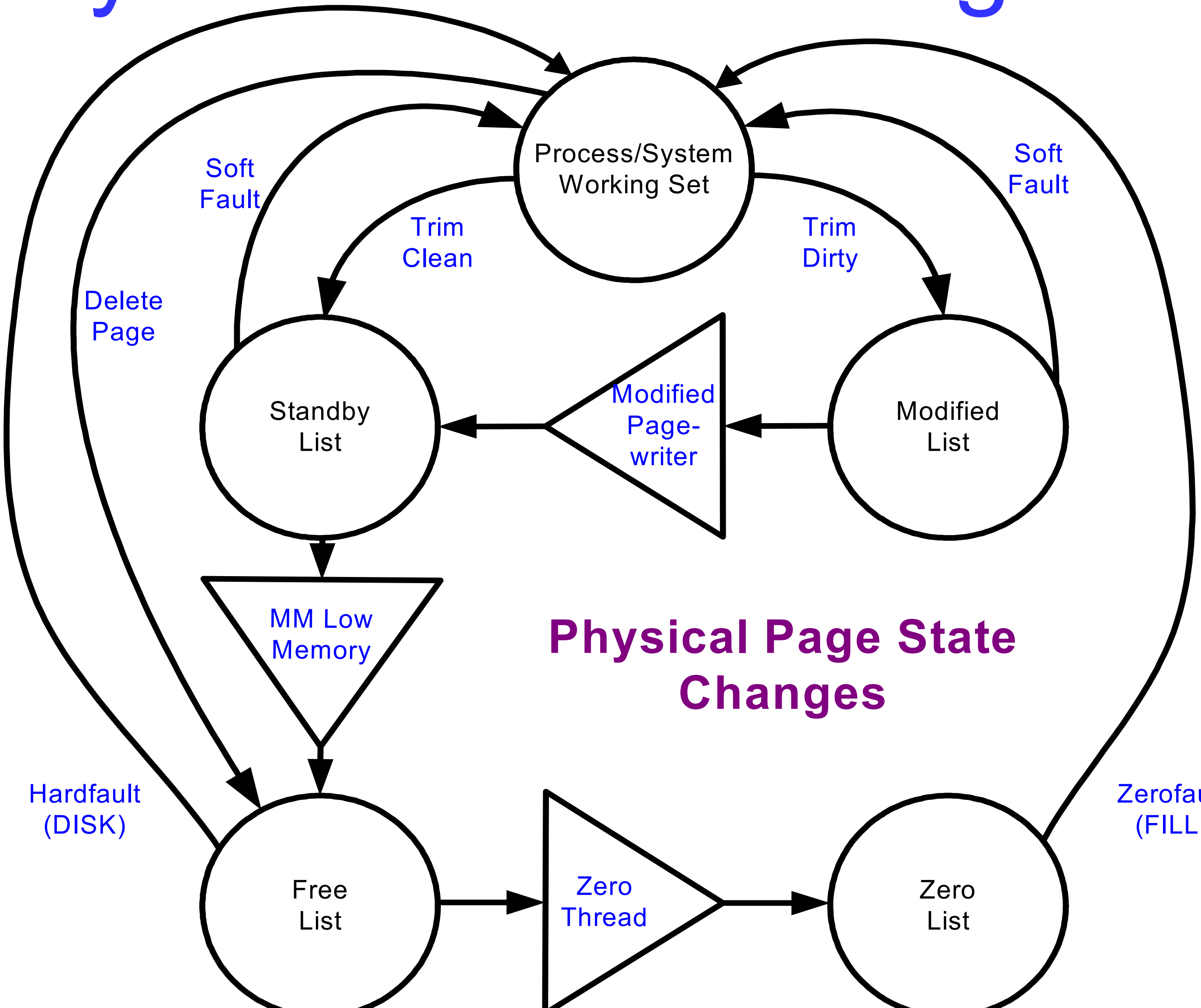
Pages 'trimmed' from working set on lists

- **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

Physical Frame Management



Managing Working Sets

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

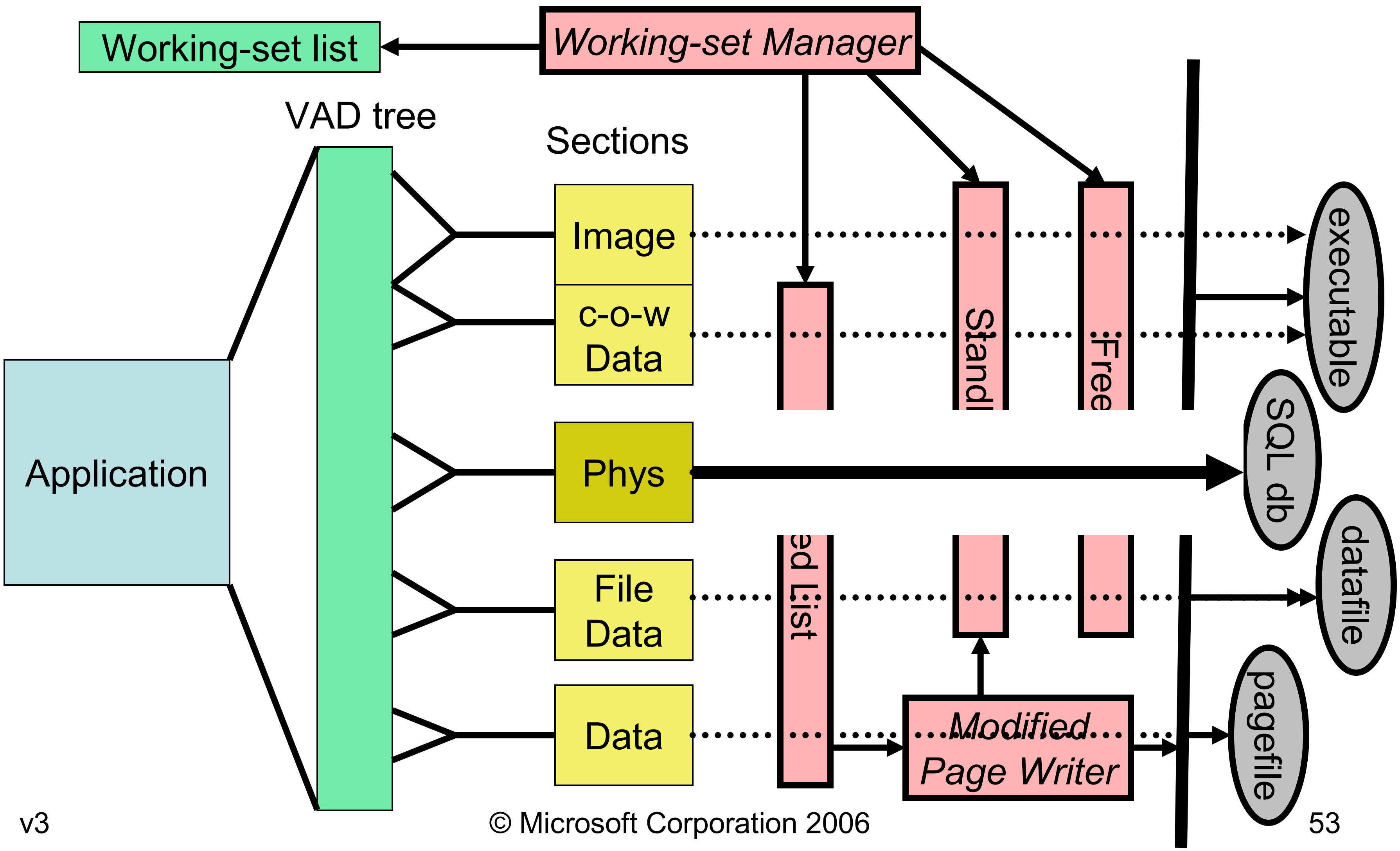
Estimate unused pages: count in working set and keep a global count of estimate

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

Bypassing Memory Management



CPU

Processes versus Threads

Lighterweight multi-threading

CPU scheduling

CPU mechanisms:

- APCs, ISRs/DPCs, system worker threads

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

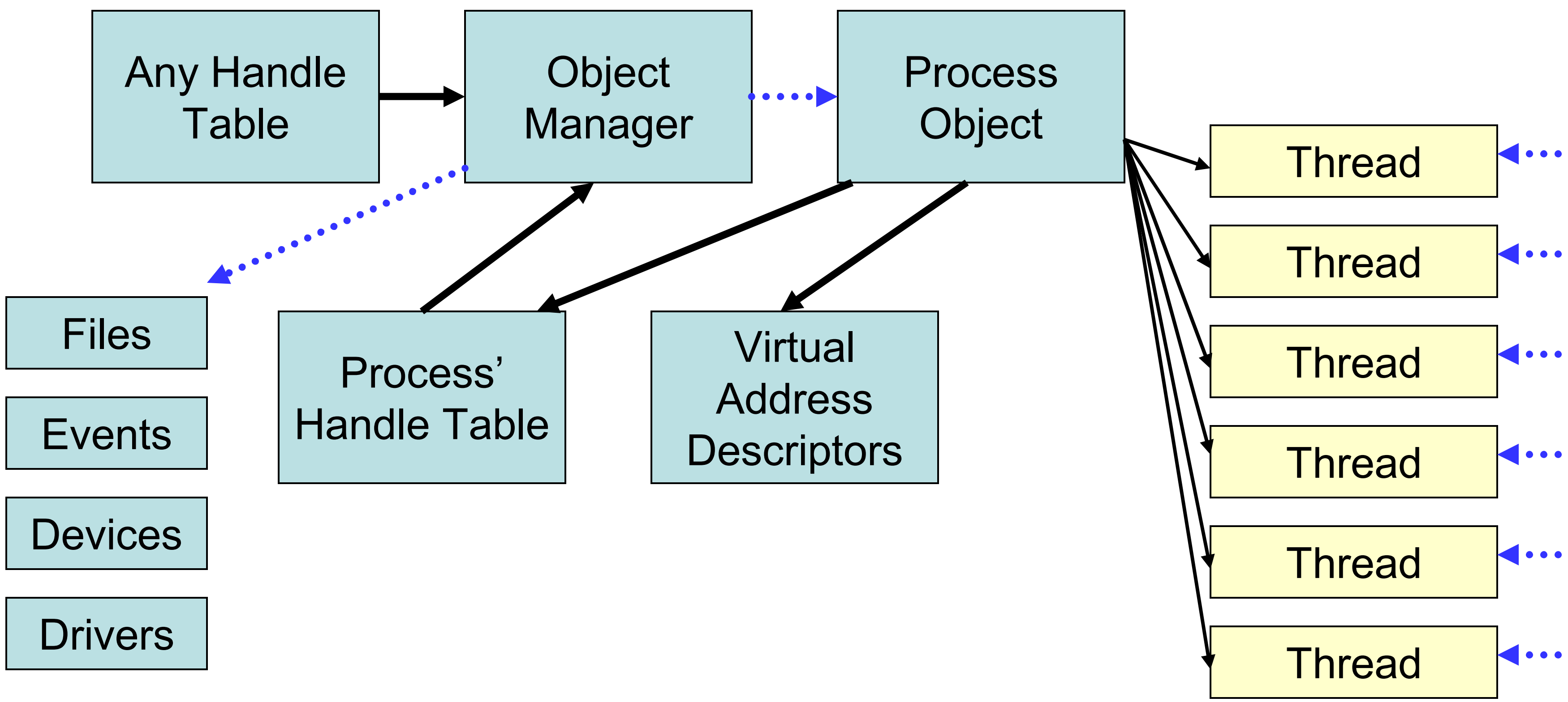
Queued to the Job, global process list and Session list

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

Process/Thread structure



Mitigating thread costs

Thread pools

- Driven by work items
- User-mode thread pool
- Kernel-mode worker threads

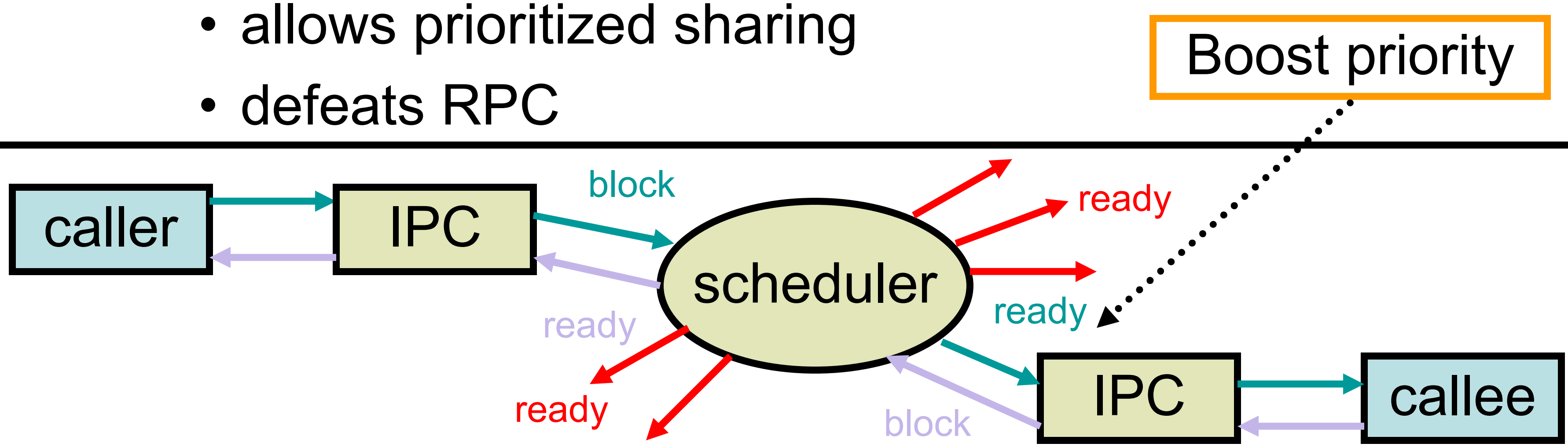
Fibers

- user-mode threads
- allows user-mode control of scheduling
- better performance for certain apps, but generally discouraged
- has most of the usual user vs. kernel thread issues

Thread latencies

Scheduling introduces bad latencies

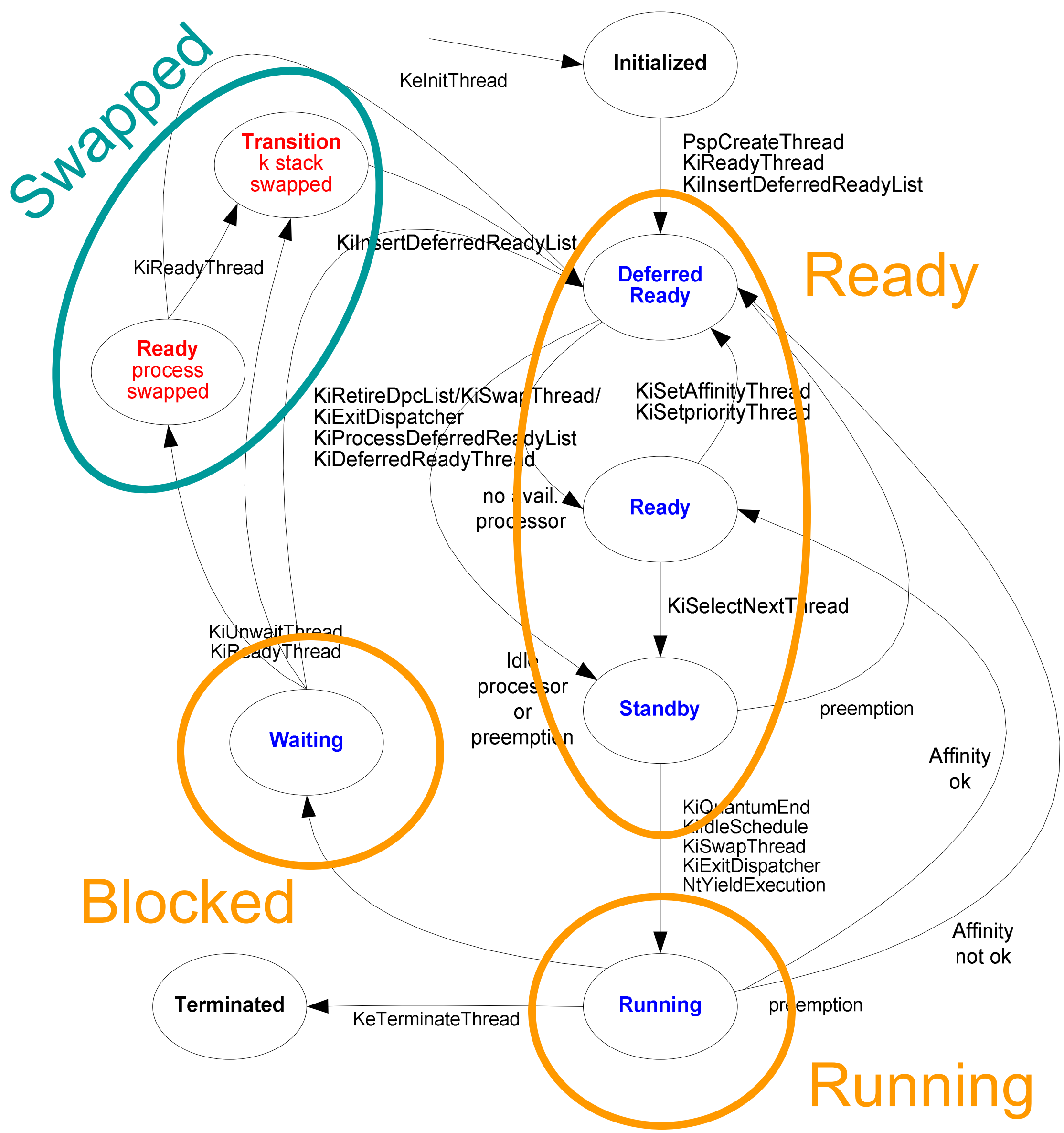
- Preemption
  - introduces fairness and responsiveness
  - creates priority inversion if holding locks/resources
- Scheduling
  - allows prioritized sharing
  - defeats RPC



Scheduling

- Windows schedules threads, not processes
  - Scheduling is preemptive, priority-based, and round-robin at the highest-priority
  - 16 real-time priorities above 16 normal priorities
  - Scheduler tries to keep a thread on its ideal processor/node to avoid perf degradation of cache/NUMA-memory
  - Threads can specify affinity mask to run only on certain processors
- Each thread has a current & base priority
  - Base priority initialized from process
  - Non-realtime threads have priority boost/decay from base
  - Boosts for GUI foreground, waking for event
  - Priority decays, particularly if thread is CPU bound (running at quantum end)
- Scheduler is state-driven by timer, setting thread priority, thread block/exit, etc
- Priority inversions can lead to starvation
  - balance manager periodically boosts non-running runnable threads

Scheduler

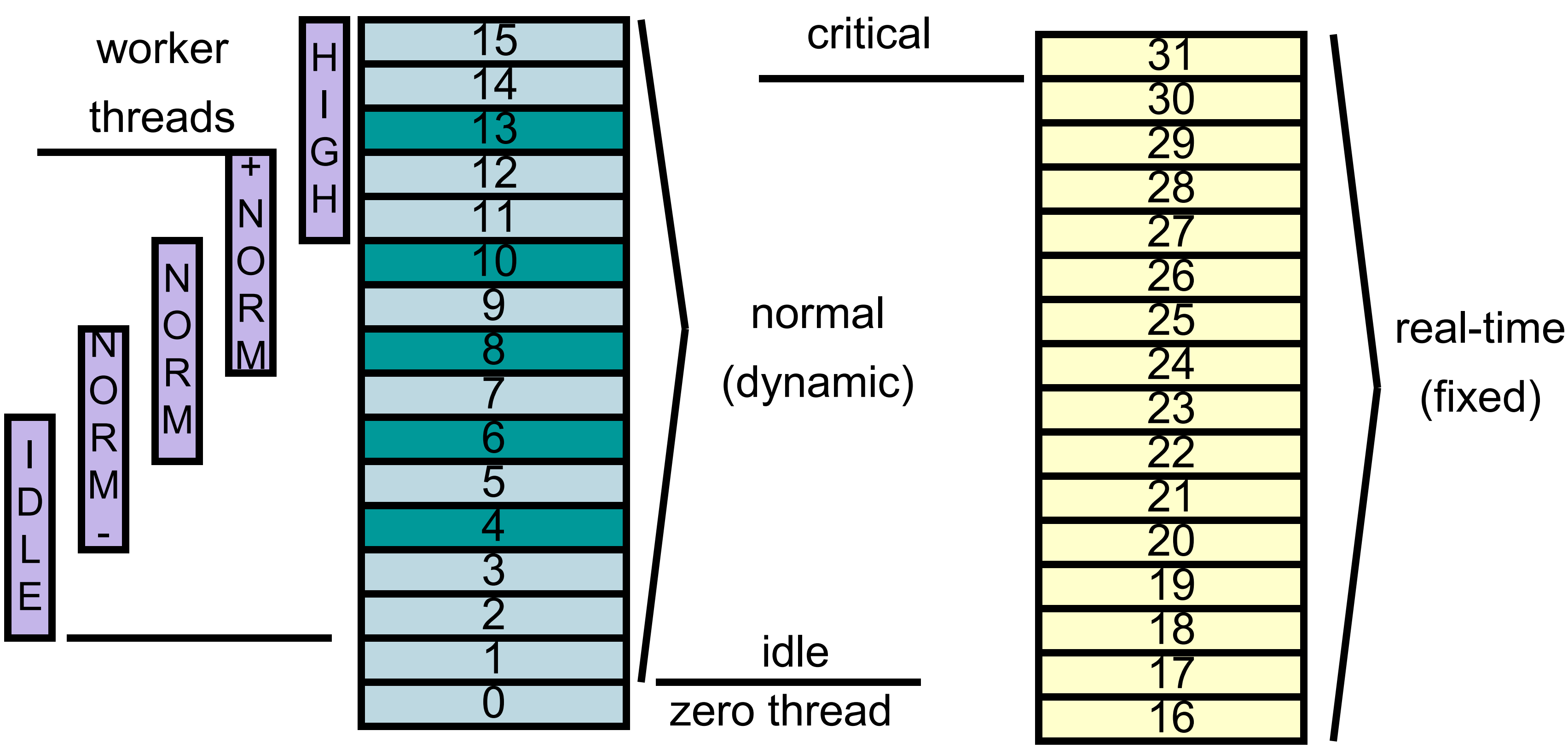


Kernel Thread Transition Diagram
DavePr@Microsoft.com
2003/04/06 v0.4b

Thread scheduling states

- Main quasi-states:
  - Ready – able to run (queued on Prcb ReadyList)
  - Running – current thread (Prcb CurrentThread)
  - 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

NT thread priorities



CPU Control-flow

Thread scheduling occurs at PASSIVE or APC level
 (IRQL < 2)

APCs (Asynchronous Procedure Calls) deliver I/O completions, thread/process termination, etc (IRQL == 1)

- Not a general mechanism like unix signals (user-mode code must explicitly block pending APC delivery)

Interrupt Service Routines run at IRL > 2

ISRs defer most processing to run at IRQL==2 (DISPATCH level) by queuing a DPC to their current processor

A pool of *worker threads* available for kernel components to run in a normal thread context when user-mode thread is unavailable or inappropriate

Normal thread scheduling is round-robin among priority levels, with priority adjustments (except for fixed priority real-time threads)

Asynchronous Procedure Calls

- APCs execute routine in thread context
  - not as general as UNIX signals
  - user-mode APCs run when blocked & alertable
  - kernel-mode APCs used extensively: timers, notifications, swapping stacks, debugging, set thread ctx, I/O completion, error reporting, creating & destroying processes & threads, …
- APCs generally blocked in critical sections
  - e.g. don’t want thread to exit holding resources

Deferred Procedure Calls

- DPCs run a routine on a particular processor
  - DPCs are higher priority than threads
  - common usage is deferred interrupt processing
  - ISR queues DPC to do bulk of work
    - *long DPCs harm perf, by blocking threads*
    - *Drivers must be careful to flush DPCs before unloading*
  - also used by scheduler & timers (e.g. at quantum end)
  - kernel-mode APCs used extensively: timers, notifications, swapping stacks, debugging, set thread ctx, I/O completion, error reporting, creating & destroying processes & threads, …
- High-priority routines use IPI (inter-processor intr)
  - used by MM to flush TLB in other processors

System Threads

- System threads have no user-mode context
  - Run in 'system' context, use system handle table
- System thread examples
  - Dedicated threads
    - Lazy writer, modified page writer, balance set manager, mapped pager writer, other housekeeping functions
  - General worker threads
    - Used to move work out of context of user thread
    - Must be freed before drivers unload
    - Sometimes used to avoid kernel stack overflows
  - Driver worker threads
    - Extends pool of worker threads for heavy hitters, like file server

Synchronization

Multiple tailored mechanisms for synchronization and resource sharing

Examples:

- PushLocks
- Fast Referencing

Kernel synchronization mechanisms

- Pushlocks
- Fastref
- Rundown protection
- Spinlocks
- Queued spinlocks
- IPI
- SLISTs
- DISPATCHER_HEADER
- KQUEUEs
- KEVENTs
- Guarded mutexes
- Mutants
- Semaphores
- EventPairs
- ERESOURCEs
- Critical Sections

Push Locks

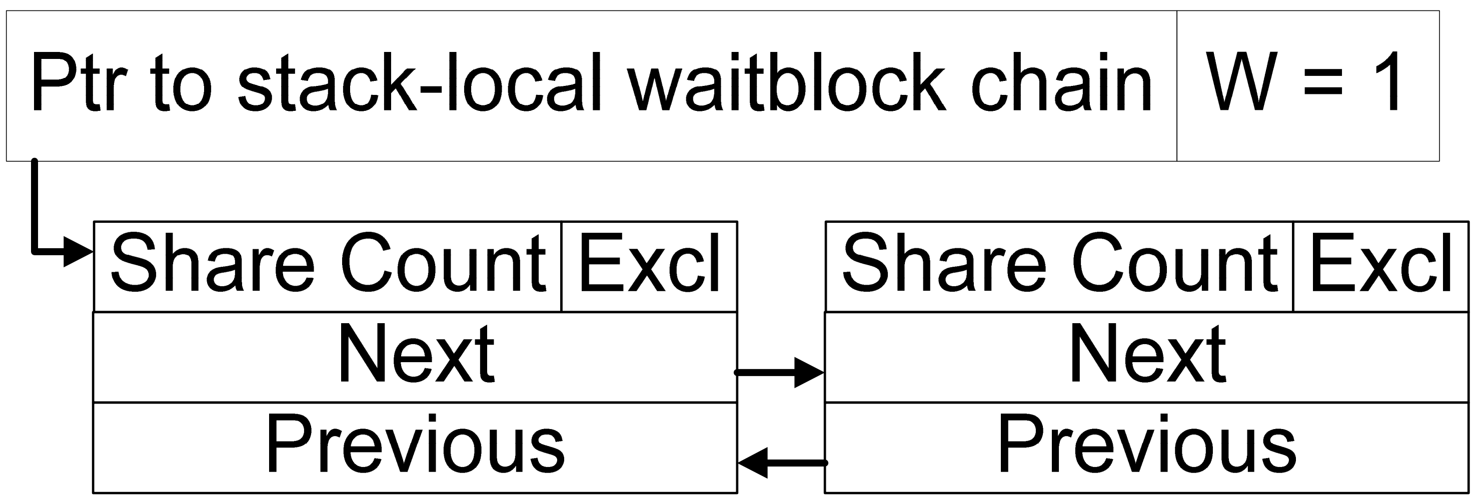
- Acquired shared or exclusive
- NOT recursive
- Locks granted in order of arrival
- Fast non-contended / Slow contended
- Sizeof(pushlock) == Sizeof(void*)
- Pageable
- Acquire/release are lock-free
- Contended case blocks using local stack

Pushlock format

Normal case

Share Count	Excl	W = 0

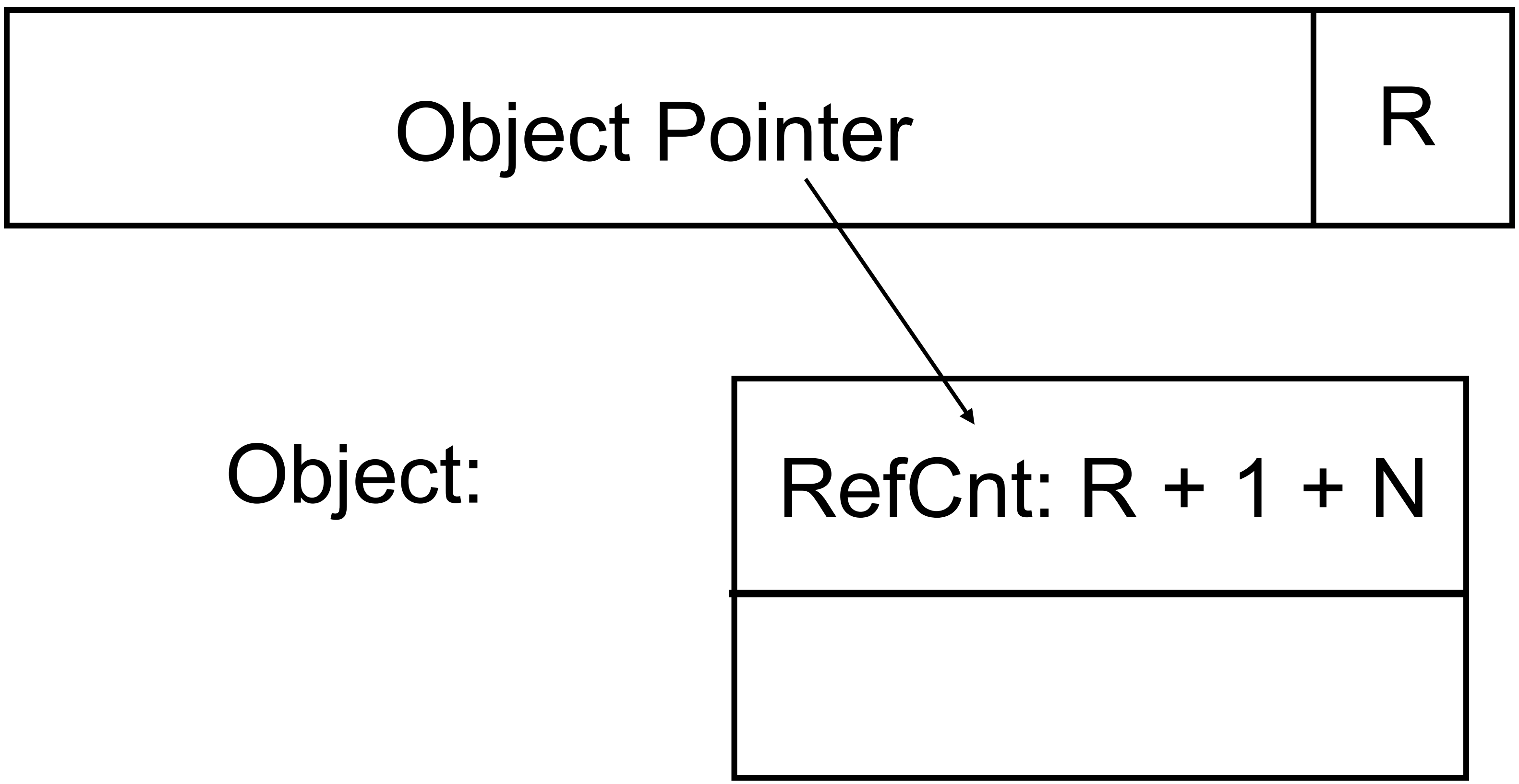
Contended case



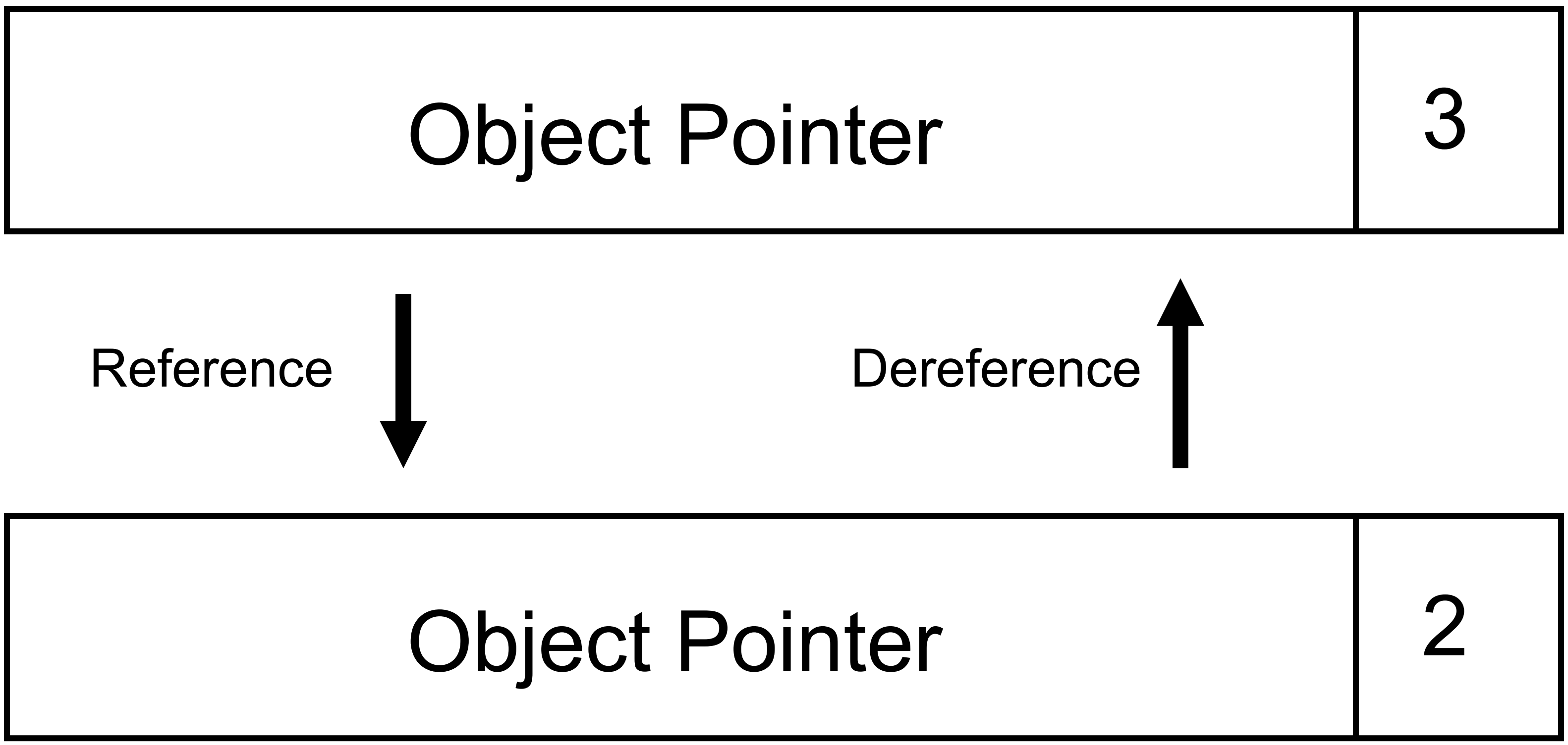
Fast Referencing

- Used to protect rarely changing reference counted data
- Small pageable structure that's the size of a pointer
- Scalable since it requires no lock acquires in over 99% of calls

Fast Referencing Internals

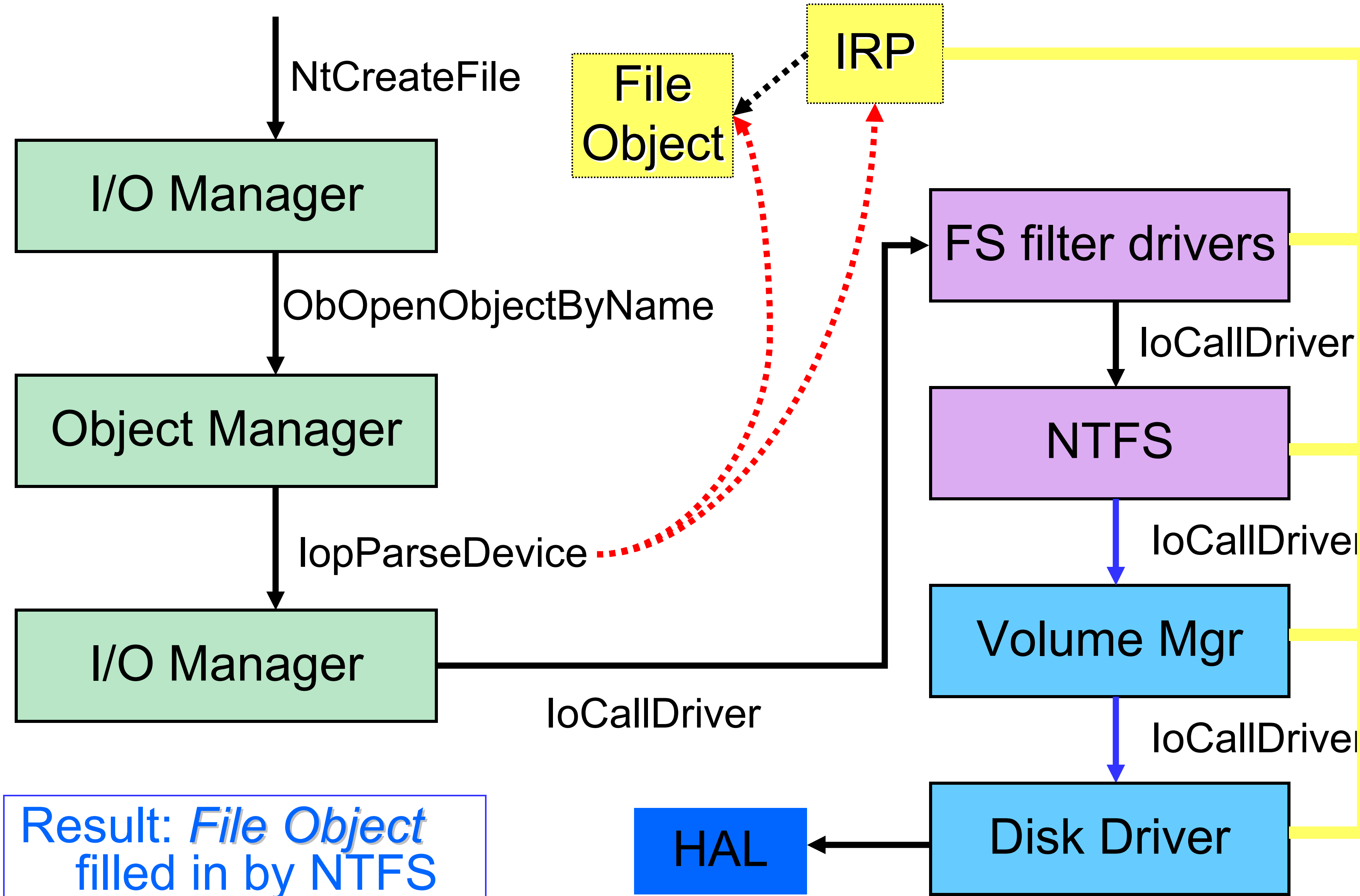


Obtaining a Fast Reference



I/O

- Driver stacks
- I/O Request Packets
- Synchronous vs Asynchronous I/O
- I/O completion ports
- File Systems



Result: *File Object*
filled in by NTFS

Layering Drivers

Device objects attach one on top of another using IoAttachDevice* APIs creating device stacks

– IO manager sends IRP to top of the stack

– drivers store next lower device object in their private data structure

– stack tear down done using IoDetachDevice and IoDeleteDevice

Device objects point to driver objects

– driver represent driver state, including dispatch table

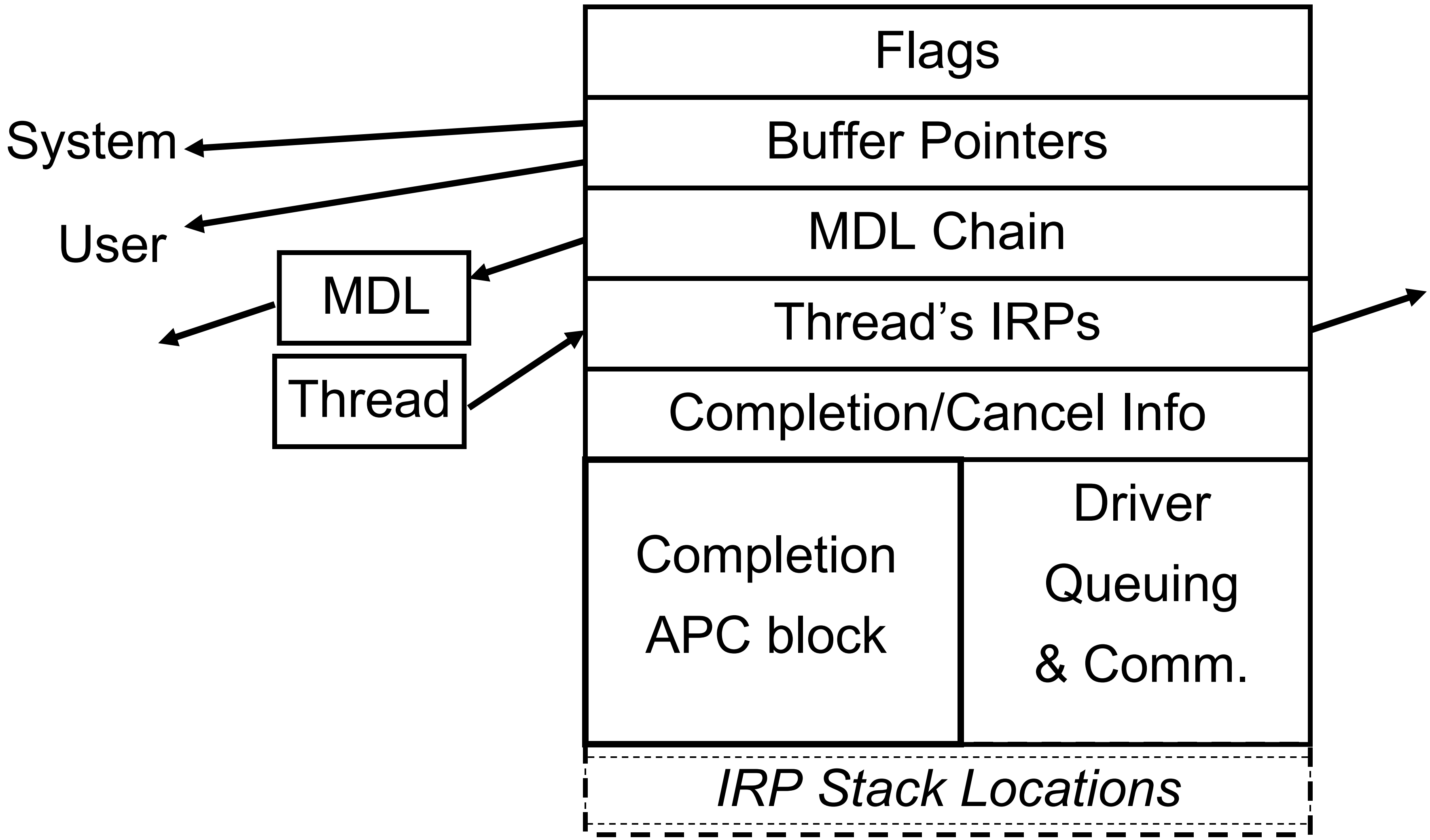
File objects point to open files

File systems are drivers which manage file objects for volumes (described by VolumeParameterBlocks)

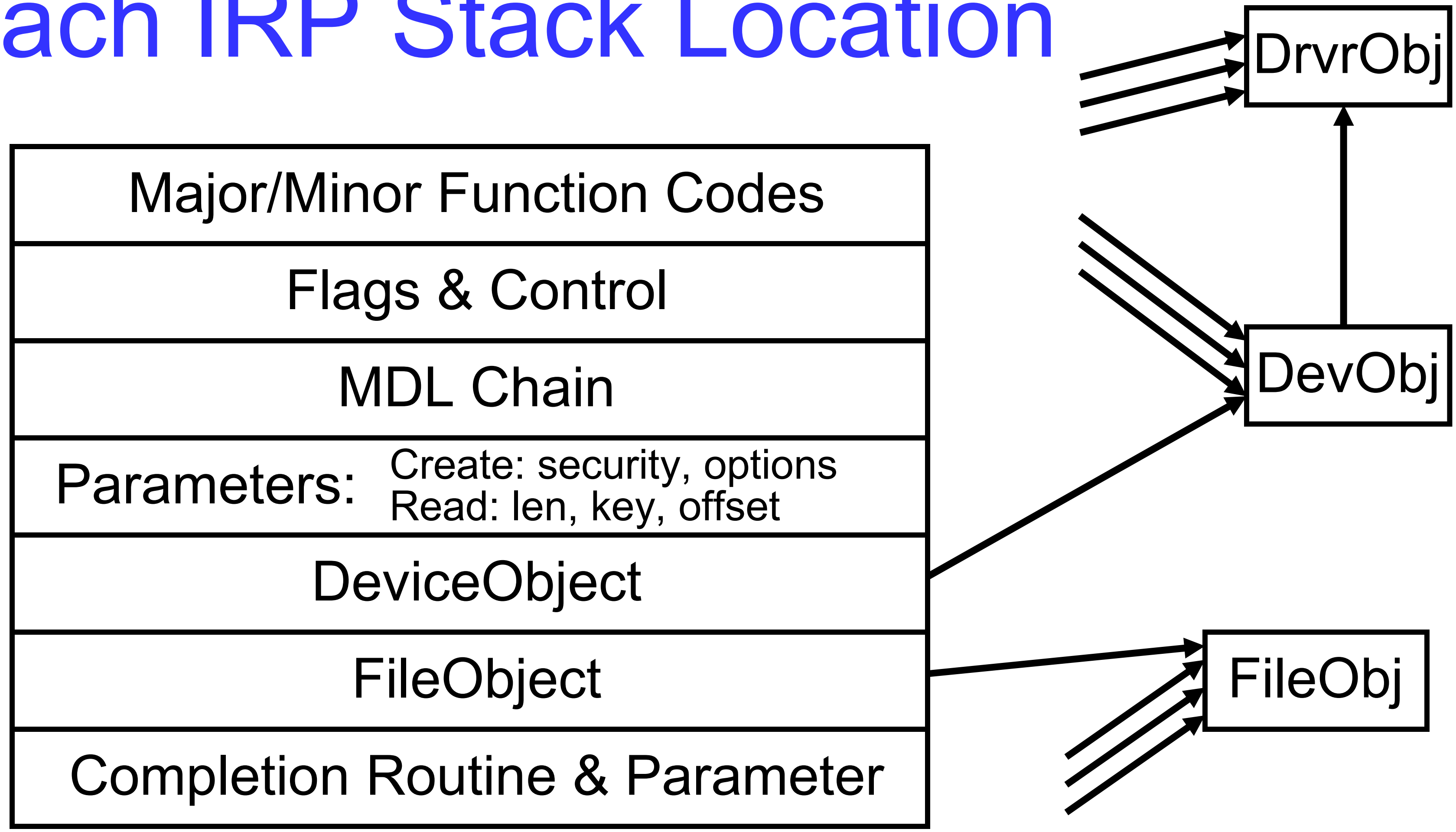
IO Request Packet (IRP)

- IO operations encapsulated in IRPs.
- IO requests travel down a driver stack in an IRP.
- Each driver gets a stack location which contains parameters for that IO request.
- IRP has major and minor codes to describe IO operations.
- Major codes include create, read, write, PNP, devioctl, cleanup and close.
- Irps are associated with a thread that made the IO request.

IRP Fields


Flags
Buffer Pointers
MDL Chain
Thread's IRPs
Completion/Cancel Info
Completion
APC block
Driver
Queuing
& Comm.
IRP Stack Locations
System
User
MDL
Thread

Each IRP Stack Location



IRP flow of control (synchronous)

- **IOMgr (e.g. IopParseDevice) creates IRP, fills in top stack location, calls IoCallDriver to pass to stack**
  - driver determined by top device object on device stack
  - driver passed the device object and IRP
- **IoCallDriver**
  - copies stack location for next driver
  - driver routine determined by major function in drvobj
- **Each driver in turn**
  - does work on IRP, if desired
  - keeps track in the device object of the next stack device
- **Calls IoCallDriver on next device**
- **Eventually bottom driver completes IO and returns on callstack**

IRP flow of control (asynch)

Eventually a driver decides to be asynchronous

- driver queues IRP for further processing
- driver returns STATUS_PENDING up call stack
- higher drivers may return all the way to user, or may wait for IO to complete (synchronizing the stack)

Eventually a driver decides IO is complete

- usually due to an interrupt/DPC completing IO
- each completion routine in device stack is called, possibly at DPC or in arbitrary thread context
- IRP turned into APC request delivered to original thread
- APC runs final completion, accessing process memory

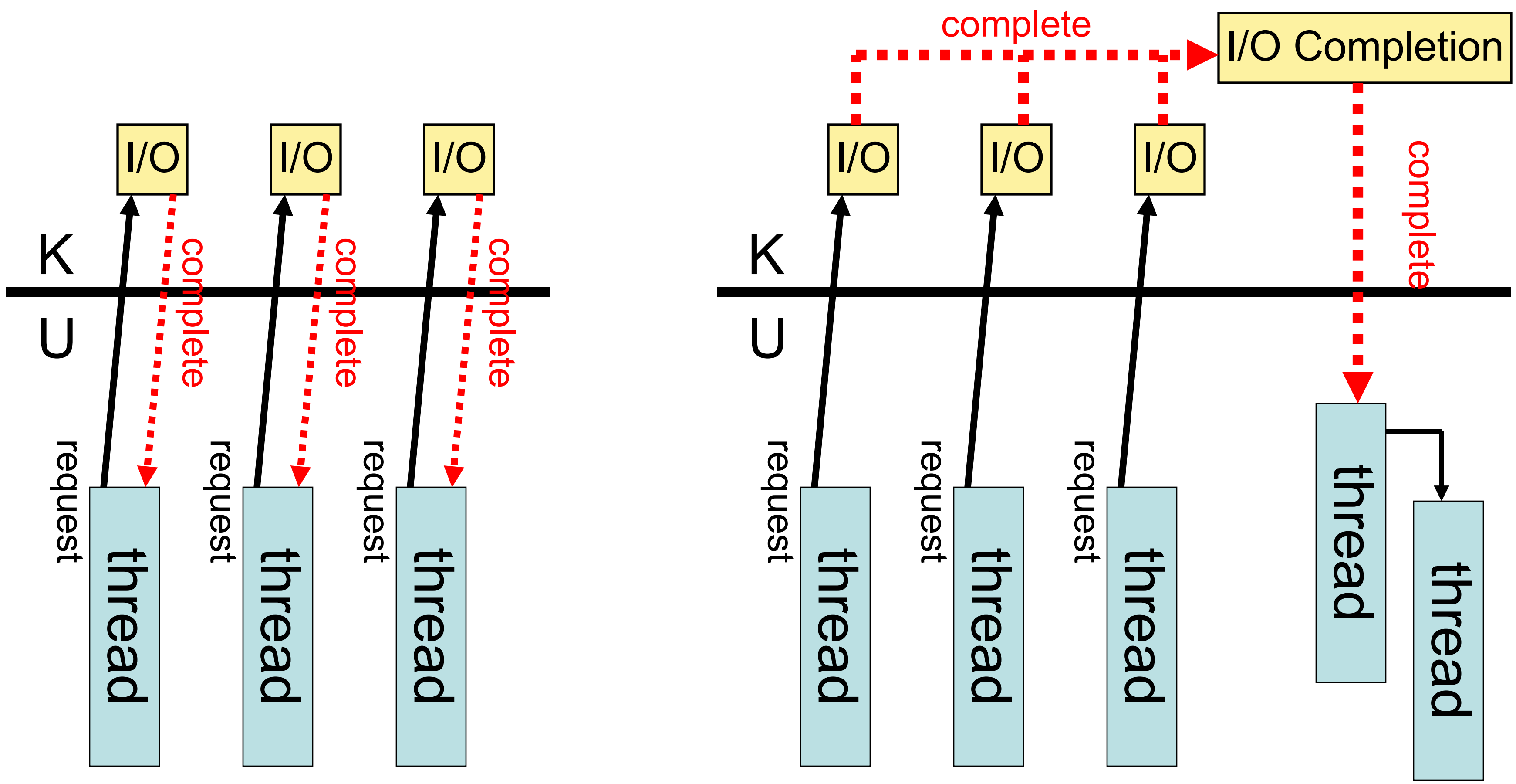
Asychronous I/O

- Applications can issue asynchronous IO requests to files opened with FILE_FLAG_OVERLAPPED and passing an LPOVERLAPPED parameter to the IO API (e.g., ReadFile(…))
- Five methods available to wait for IO completion,
  - Wait on the file handle
  - Wait on an event handle passed in the overlapped structure (e.g., GetOverlappedResult(…))
  - Specify a routine to be called on IO completion
  - Use completion ports
  - Poll status variable

I/O Completion Ports

- Five methods to receive notification of completion for asynchronous I/O:
  - poll status variable
  - wait for the file handle to be signalled
  - wait for an explicitly passed event to be signalled
  - specify a routine to be called on the originating ports
  - use and I/O completion port

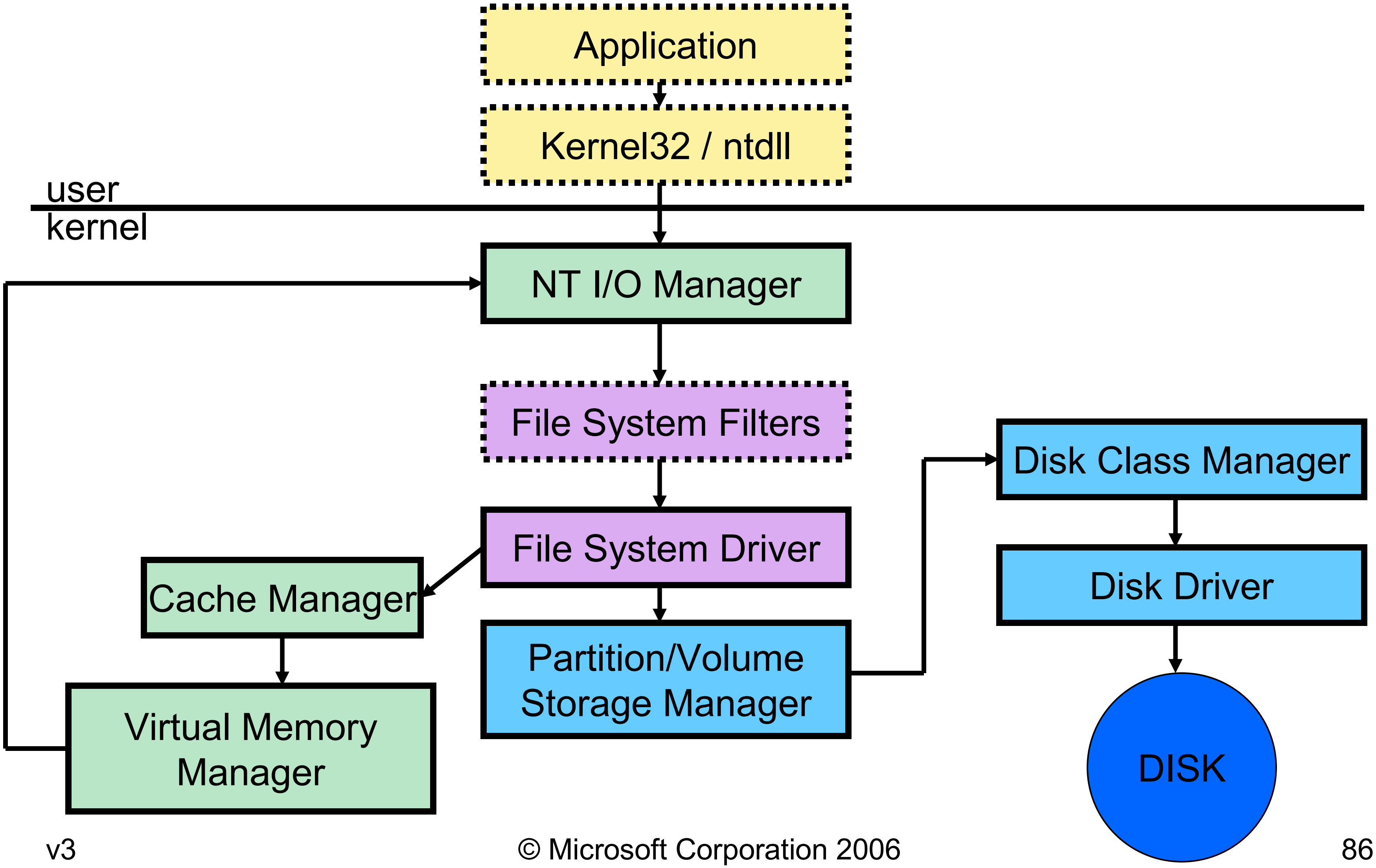
Completing Asynchronous I/O



normal completion

I/O completion ports

File System Device Stack



Discussion