Dave Probert, Ph.D. - Windows Kernel Architect
Core Operating Systems Division – Microsoft

EVOLUTION OF THE WINDOWS KERNEL ARCHITECTURE

SWITZERLAND – JULY 2009

UNIX vs NT Design Environments

Environment which influenced fundamental design decisions	
UNIX [1969]	**Windows (NT)** [1989]
16-bit program address space	32-bit program address space
Kbytes of physical memory	Mbytes of physical memory
Swapping system with memory mapping	Virtual memory
Kbytes of disk, fixed disks	Mbytes of disk, removable disks
Uniprocessor	Multiprocessor (4-way)
State-machine based I/O devices	Micro-controller based I/O devices
Standalone interactive systems	Client/Server distributed computing
Small number of friendly users	Large, diverse user populations

Effect on OS Design

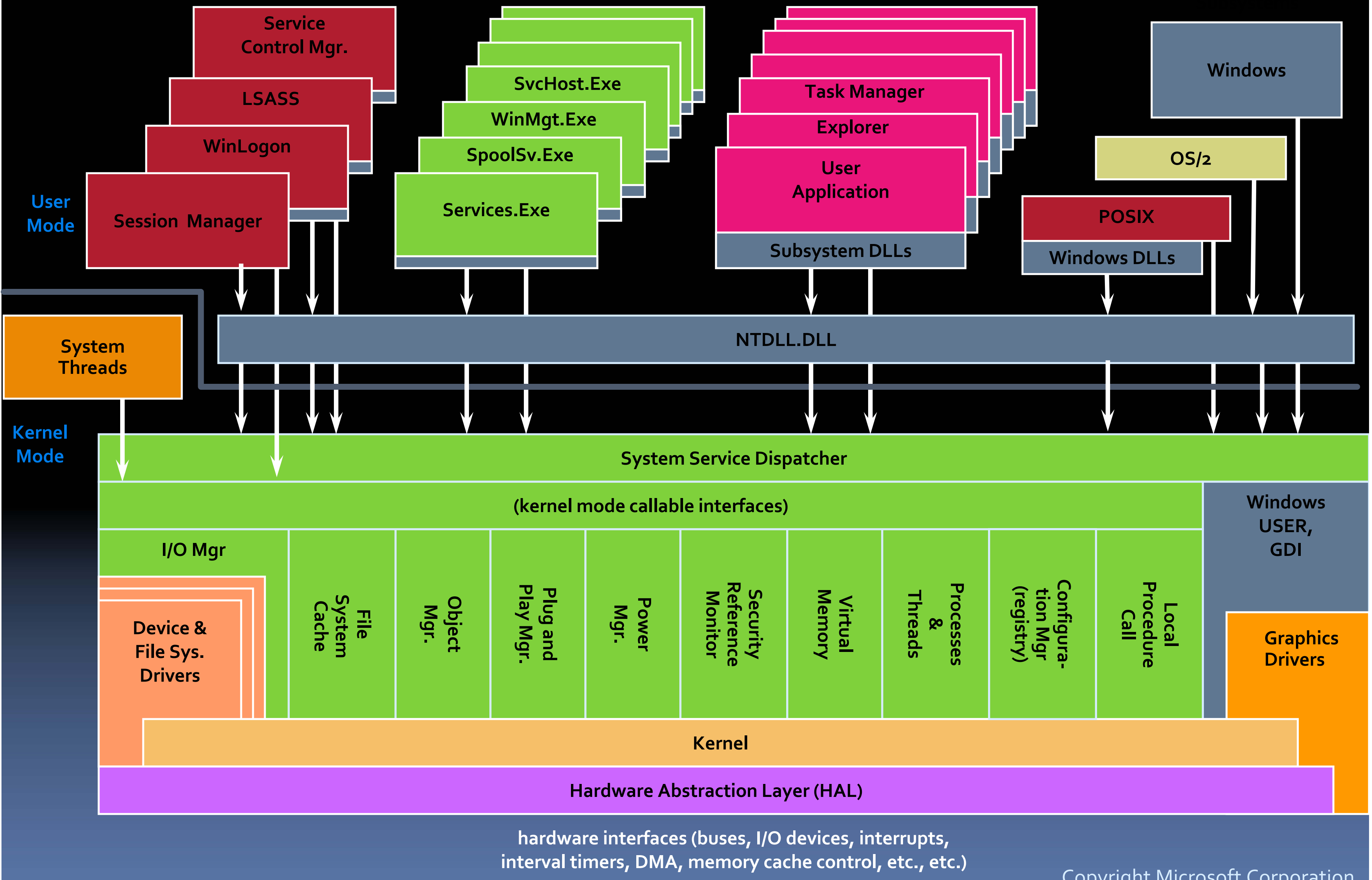
<table>
<tr><th colspan="3">NT vs UNIX</th></tr>
<tr><td colspan="3">Although both Windows and Linux have adapted to changes in the environment, the original design environments (i.e. in 1989 and 1969) heavily influenced the design choices:</td></tr>
<tr><td>Unit of concurrency:</td><td>Threads vs processes</td><td>Addr space, uniproc</td></tr>
<tr><td>Process creation:</td><td>CreateProcess() vs fork()</td><td>Addr space, swapping</td></tr>
<tr><td>I/O:</td><td>Async vs sync</td><td>Swapping, I/O devices</td></tr>
<tr><td>Namespace root:</td><td>Virtual vs Filesystem</td><td>Removable storage</td></tr>
<tr><td>Security:</td><td>ACLs vs uid/gid</td><td>User populations</td></tr>
</table>

Today's Environment [2009]

64-bit addresses
GBytes of physical memory
TBytes of rotational disk
New Storage hierarchies (SSDs)
Hypervisors, virtual processors
Multi-core/Many-core
Heterogeneous CPU architectures, Fixed function hardware
High-speed internet/intranet, Web Services
Media-rich applications
Single user, but vulnerable to hackers worldwide

Convergence: Smartphone / Netbook / Laptop / Desktop / TV / Web / Cloud

Windows Architecture



Service Control Mgr.

LSASS

WinLogon

Session Manager

SvcHost.Exe

WinMgt.Exe

SpoolSv.Exe

Services.Exe

Task Manager

Explorer

User Application

Subsystem DLLs

Windows

OS/2

POSIX

Windows DLLs

User Mode

System Threads

NTDLL.DLL

Kernel Mode

System Service Dispatcher

(kernel mode callable interfaces)

Windows USER, GDI

I/O Mgr

Device & File Sys. Drivers

File System Cache

Object Mgr.

Plug and Play Mgr.

Power Mgr.

Security Reference Monitor

Virtual Memory

Processes & Threads

Configura-tion Mgr (registry)

Local Procedure Call

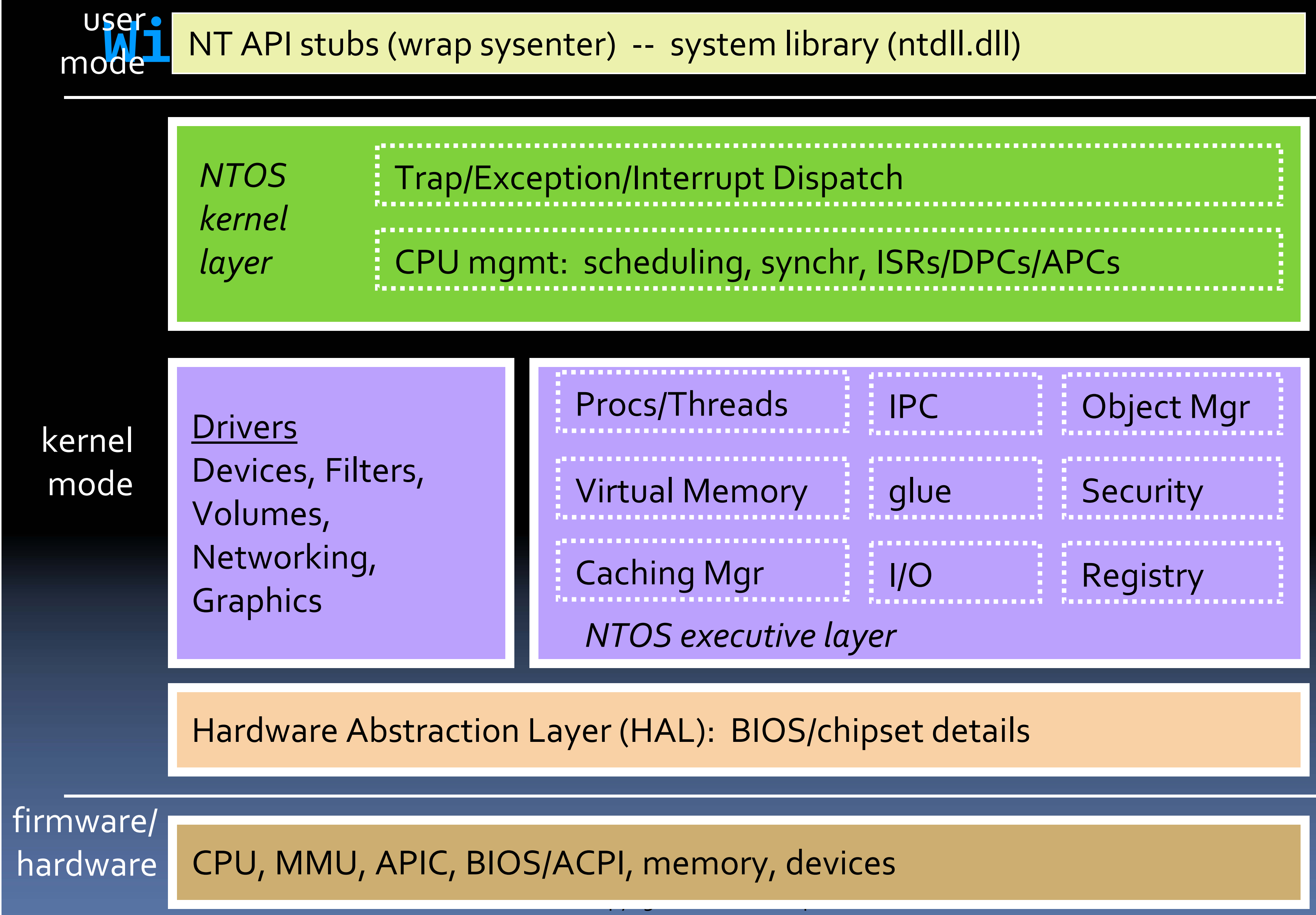
Graphics Drivers

Kernel

Hardware Abstraction Layer (HAL)

hardware interfaces (buses, I/O devices, interrupts,
interval timers, DMA, memory cache control, etc., etc.)

Kernel-mode Architecture of Wi



user mode

NT API stubs (wrap sysenter) -- system library (ntdll.dll)

kernel mode

NTOS kernel layer

Trap/Exception/Interrupt Dispatch

CPU mgmt: scheduling, synchr, ISRs/DPCs/APCs

Drivers

Devices, Filters, Volumes, Networking, Graphics

Procs/Threads

IPC

Object Mgr

Virtual Memory

glue

Security

Caching Mgr

I/O

Registry

NTOS executive layer

Hardware Abstraction Layer (HAL): BIOS/chipset details

firmware/ hardware

CPU, MMU, APIC, BIOS/ACPI, memory, devices

Kernel/Executive layers

- Kernel layer – ntos/ke – ~ 5% of NTOS source)
  - Abstracts the CPU
    - Threads, Asynchronous Procedure Calls (APCs)
    - Interrupt Service Routines (ISRs)
    - Deferred Procedure Calls (DPCs – aka Software Interrupts)
  - Providers low-level synchronization
- Executive layer
  - OS Services running in a multithreaded environment
  - Full virtual memory, heap, handles
  - Extensions to NTOS: drivers, file systems, network, …

NT (Native) API examples

NtCreateProcess (&ProcHandle, Access, SectionHandle, DebugPort, ExceptionPort, …)

NtCreateThread (&ThreadHandle, ProcHandle, Access, ThreadContext, bCreateSuspended, …)

NtAllocateVirtualMemory (ProcHandle, Addr, Size, Type, Protection, …)

NtMapViewOfSection (SectionHandle, ProcHandle, Addr, Size, Protection, …)

NtReadVirtualMemory (ProcHandle, Addr, Size, …)

NtDuplicateObject (srcProcHandle, srcObjHandle, dstProcHandle, dstHandle, Access, Attributes, Options)

Windows Vista Kernel Changes

- Kernel changes mostly minor improvements
  - Algorithms, scalability, code maintainability
  - CPU timing: Uses Time Stamp Counter (TSC)
    - Interrupts not charged to threads
    - Timing and quanta are more accurate
  - Communication
    - ALPC: Advanced Lightweight Procedure Calls
    - Kernel-mode RPC
    - New TCP/IP stack (integrated IPv4 and IPv6)
  - I/O
    - Remove a context switch from I/O Completion Ports
    - I/O cancellation improvements
  - Memory management
    - Address space randomization (DLLs, stacks)
    - Kernel address space dynamically configured
  - Security: BitLocker, DRM, UAC, Integrity Levels

Windows 7 Kernel Changes

- Miscellaneous kernel changes
  - MinWin
    - Change how Windows is built
    - Lots of DLL refactoring
    - API Sets (virtual DLLs)
  - Working-set management
    - Runaway processes quickly start reusing own pages
    - Break up kernel working-set into multiple working-sets
      - System cache, paged pool, pageable system code
  - Security
    - Better UAC, new account types, less BitLocker blockers
  - Energy efficiency
    - Trigger-started background services
    - Core Parking
    - Timer-coalescing, tick skipping
- Major scalability improvements for large server apps
  - Broke apart last two major kernel locks, >64p
- Kernel support for ConcRT
  - User-Mode Scheduling (UMS)

Core Parking Operation

Core 0

Core 1

Socket 0

Core 0

Core 1

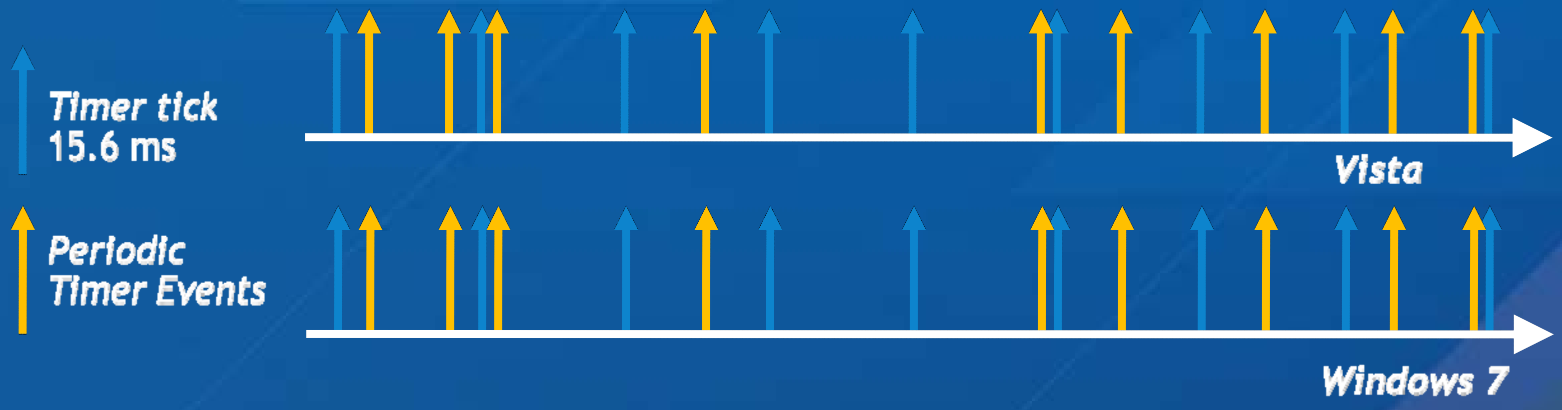
Socket 1

● Workload

MarkRuss

Timer Coalescing

- Staying idle requires minimizing timer interrupts
- Before, periodic timers had independent cycles even when period was the same
- New timer APIs permit timer coalescing
  - Application or driver specifies tolerable delay
- Timer system shifts timer firing to align periods on a coalescing interval:
  - 50ms, 100ms, 250ms, 1s



Broke apart the Dispatcher Lock

- Scheduler Dispatcher lock hottest on server workloads
  - Lock protects all thread state changes (wait, unwait)
  - Very *hot* lock at >64x
- Dispatcher lock broken up in Windows 7 / Server 2008 R2
  - Each object protected by its own lock
  - Many operations are lock-free

Removed PFN Lock

- Windows tracks the state of pages in physical memory
  - In use: in working sets:
  - Not assigned: on paging lists: freemodified, standby, …
- Before, all page state changes protected by global PFN (Physical Frame Number) lock
- As of Windows 7 the PFN lock is gone
  - Pages are now locked individually
  - Improves scalability for large memory applications

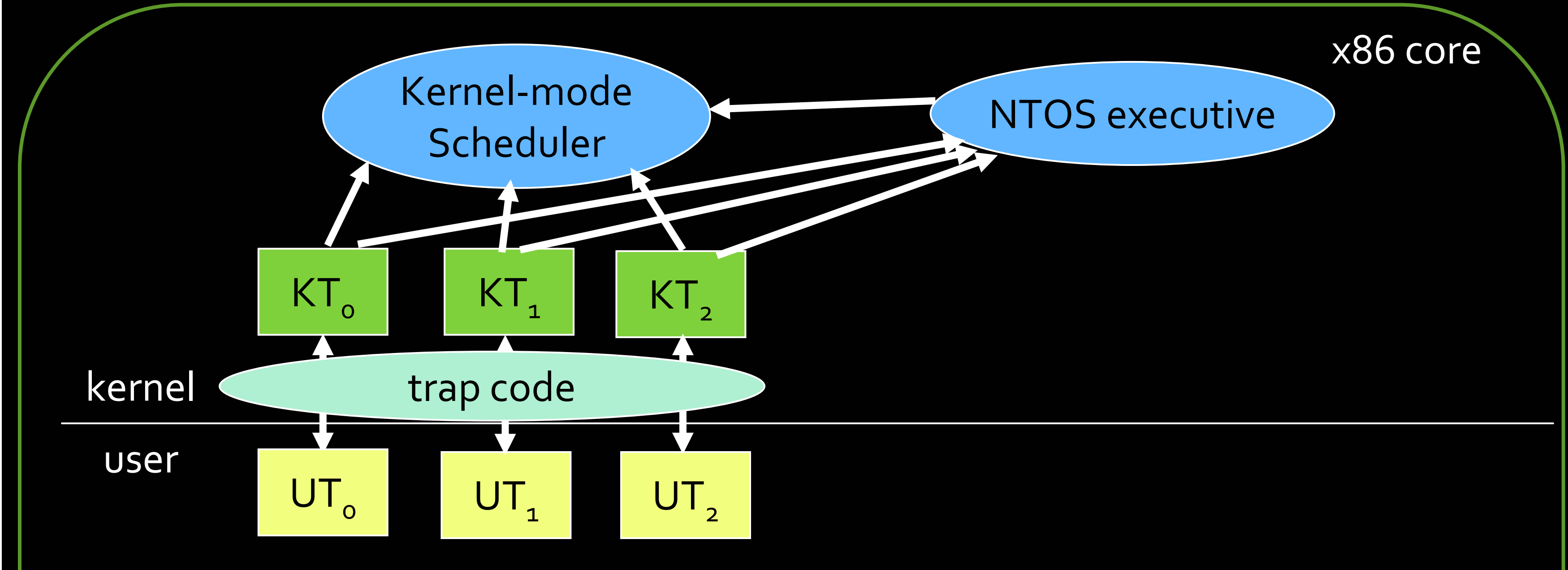
User Mode Scheduling (UMS)

- Improve support for efficient cooperative multithreaded scheduling of small tasks (over-decomposition)
  - ⇒ Want to schedule tasks in user-mode
  - ⇒ Use NT threads to simulate CPUs, multiplex tasks onto these threads
- When a task calls into the kernel and blocks, the CPU may get scheduled to a different app
  - ⇒ If a single NT thread per CPU, when it blocks it blocks.
  - ⇒ Could have extra threads, but then kernel and user-mode are competing to schedule the CPU
- Tasks run arbitrary Win32 code (but only x64/IA64)
  - ⇒ Assumes running on an NT thread (TEB, kernel thread)
- Used by ConcRT (Visual Studio 2010’s Concurrency Run-Time)

Windows 7 User-Mode Scheduling

- UMS breaks NT thread into two parts:
    - UT: user-mode portion (TEB, ustack, registers)
    - KT: kernel-mode portion (ETHREAD, kstack, registers)
- Three key properties:
    - User-mode scheduler switches UTs w/o ring crossing
    - KT switch is lazy: at kernel entry (e.g. syscall, pagefault)
    - CPU returned to user-mode scheduler when KT blocks
- KT “returns” to user-mode by queuing completion
    - User-mode scheduler schedules corresponding UT
    - (similar to scheduler activations, etc)

Normal NT Threading

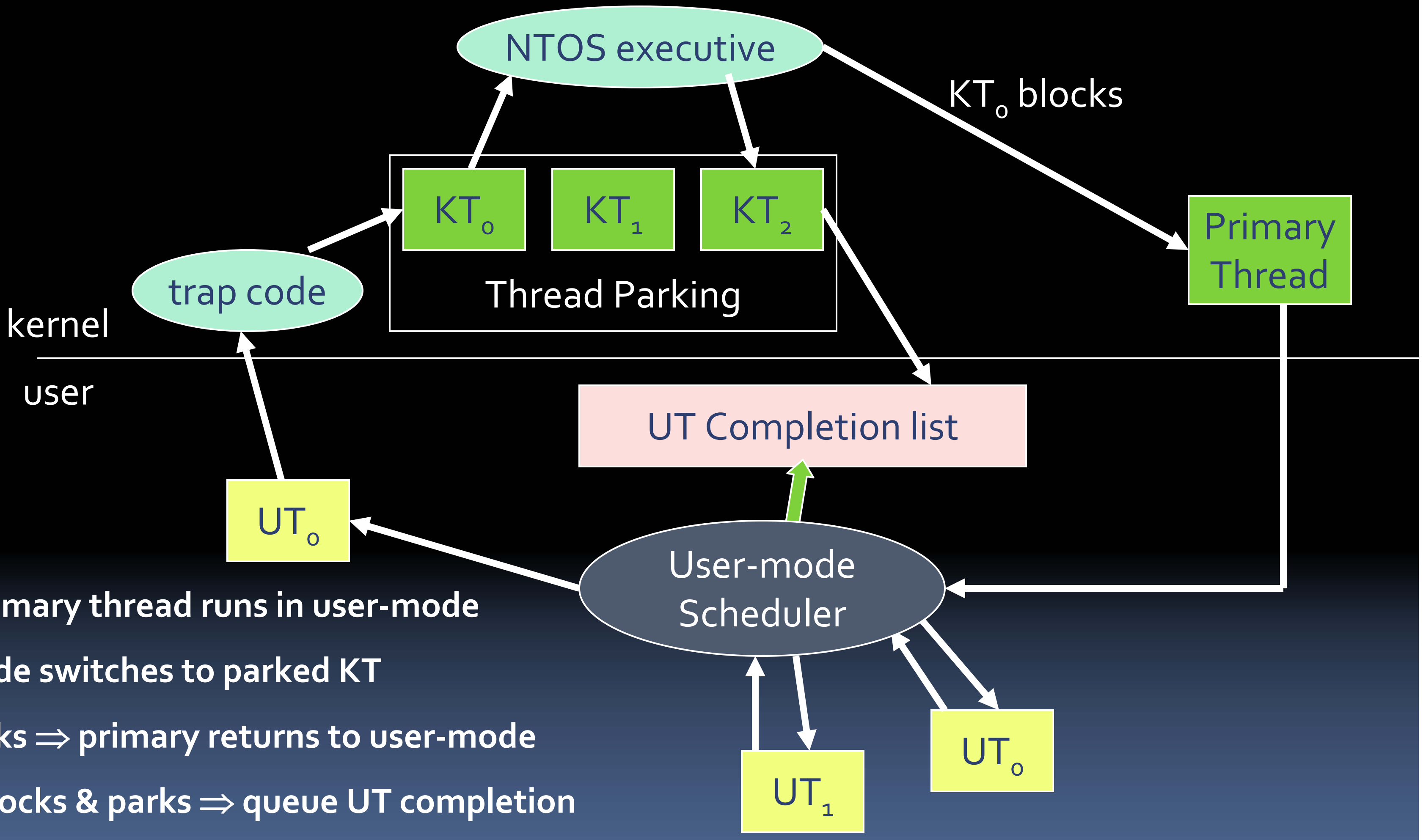


NT Thread is Kernel Thread (KT) and User Thread (UT)

UT/KT form a single logical thread representing NT thread in user or kernel

- **KT:** ETHREAD, KSTACK, link to EPROCESS
- **UT:** TEB, USTACK

User-Mode Scheduling (UMS)



- Only primary thread runs in user-mode
- Trap code switches to parked KT
- KT blocks ⇒ primary returns to user-mode
- KT unblocks & parks ⇒ queue UT completion

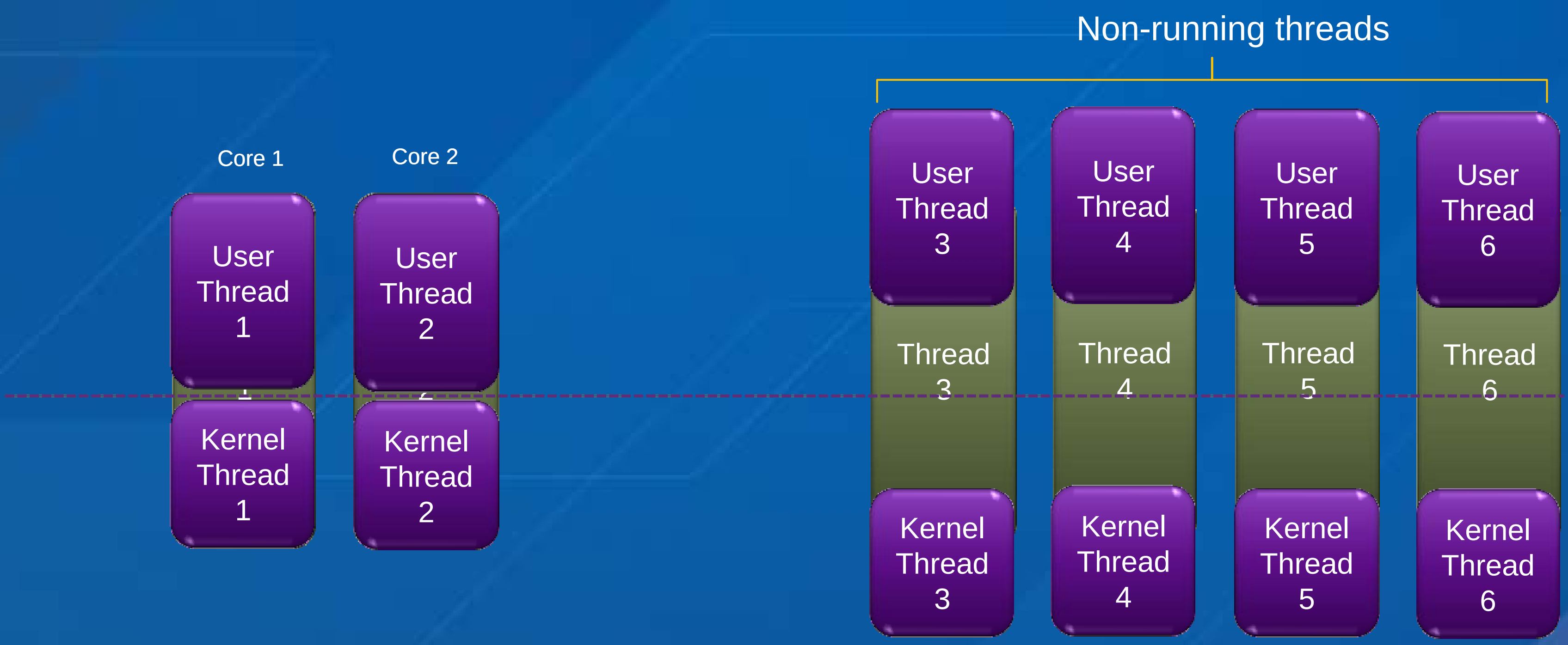
UMS

- Based on NT threads
  - ⇒ Each NT thread has user & kernel parts (UT & KT)
  - ⇒ When a thread becomes UMS, KT never returns to UT
    - ⇒ *(Well, sort of)*
  - ⇒ Instead, the *primary* thread calls the USched
- USched
  - ⇒ Switches between UTs, all in user-mode
  - ⇒ When a UT enters kernel and blocks, the primary thread will hand CPU back to the USched declaring UT blocked
  - ⇒ When UT unblocks, kernel queues notification
  - ⇒ USched consumes notifications, marks UT runnable
- Primary Thread
  - ⇒ Self-identified by entering kernel with wrong TEB
  - ⇒ So UTs can migrate between threads
  - ⇒ Affinities of primaries and KTs are orthogonal issues

UMS Thread Roles

- **Primary threads:** represent CPUs, normal app threads enter the USched world and become primaries, primaries also can be created by UScheds to allow parallel execution
  - Primaries represent concurrent execution
- **UMS threads (UT/KTs):** allow blocking in the kernel without losing the CPU
  - UMS thread represent concurrent blocking in kernel

Thread Scheduling vs UMS



Non-running threads

Core 1

Core 2

User Thread 1

Kernel Thread 1

User Thread 2

Kernel Thread 2

User Thread 3

Thread 3

Kernel Thread 3

User Thread 4

Thread 4

Kernel Thread 4

User Thread 5

Thread 5

Kernel Thread 5

User Thread 6

Thread 6

Kernel Thread 6

Cooperative Thread Scheduling

MarkRuss

Win32 compat considerations

Why not Win32 fibers?

- TEB issues
  - ⇒ Contains TLS and Win32-specific fields (incl LastError)
  - ⇒ Fibers run on multiple threads, so TEB state doesn't track
- Kernel thread issues
  - ⇒ Visibility to TEB
  - ⇒ I/O is queued to thread
  - ⇒ Mutexes record thread owner
  - ⇒ Impersonation
  - ⇒ Cross-thread operations expect to find threads and IDs
  - ⇒ Win32 code has thread and affinity awareness

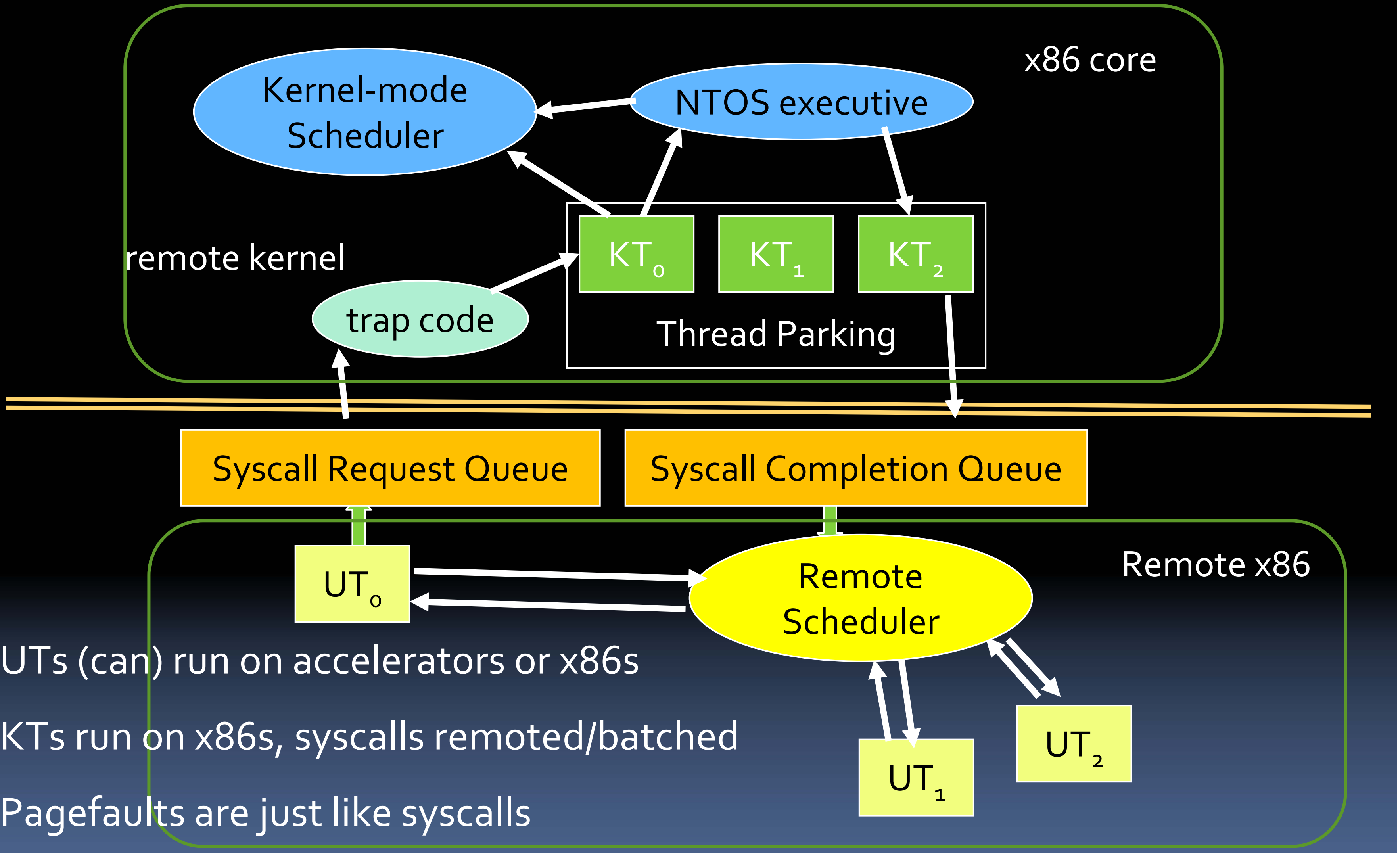
APCs

- Types of APCs
  - ⇒ User and Kernel (both Normal kernel and Special kernel)
  - ⇒ User APCs only delivered when waiting in the kernel
- Only a few APCs are important w.r.t. user-mode
  - ⇒ Suspend/resume thread
  - ⇒ Get/set thread context
  - ⇒ Thread termination
- Need to force the UT into the kernel
  - ⇒ *UMSContextLock* – coordinates between USched/kernel
  - ⇒ Tells USched not to schedule UT
  - ⇒ Tells kernel which primary to force into kernel
- Thread termination
  - ⇒ Queues notification of termination to completion list

Debug Registers

- User-mode scheduling cannot set debug registers
  - ⇒ store debug registers in user-mode context
  - ⇒ when switching to a UT with debug registers, the USched must enter the kernel to do the actual switch

Futures: Master/Slave UMS?



- UTs (can) run on accelerators or x86s
- KTs run on x86s, syscalls remoted/batched
- Pagefaults are just like syscalls
- Accelerator never "loses the CPU" (implicit primary)

Operating Systems Futures

- Many-core challenge
  - New driving force in software innovation:
    Amdahl's Law overtakes Moore's Law as high-order bit
  - Heterogeneous cores?
- OS Scalability
  - Loosely –coupled OS: mem + cpu + services?
  - Energy efficiency
- Hypervisor/Kernel/Runtime relationships
  - Move kernel scheduling (cpu/memory) into run-times?
  - Move kernel resource management into Hypervisor?
- Shrink-wrap and Freeze-dry applications?

Questions and open discussion