# Windows and Some Differences from Linux

Brian Railing
1st Year PhD Student

Previously of
Windows Server Performance Team

# Copyright Notice



- These materials are part of the *Windows Operating System Internals Curriculum Development Kit,* developed by David A. Solomon and Mark E. Russinovich with Andreas Polze
- Microsoft has licensed these materials from David Solomon Expert Seminars, Inc. for distribution to academic organizations solely for use in academic environments (and not for commercial use)

# Further Notices

- Original slide deck has modified to reflect updates to Windows since 2005
  - Mark Russinovich’s 2009 TechEd Talk WCL402: “Windows 7 and Windows Server 2008 R2 Kernel Changes”
- Further updates to reflect particular APIs of interest
- Being a performance engineer, I only know certain components in great detail
  - Storage / Networking Stack
  - Kernel Debugging / Performance Tools
  - “Fast” Synchronization

# Final Notices

- This slide deck is
  - To provide some specific details for developing applications and drivers with Windows
  - To see some of the implementation decisions and consider trade-offs
  - To show that Windows, Linux, etc make many similar design choices
- This slide deck is not
  - Meant to judge one between OSes

# Takeaways

- This slide deck is
  - Long
- What should you look for?
  - IRQLs
  - Schedulers are similar.  Priorities are different.
  - Paged vs NonPaged Memory
  - Wait for objects

# Outline

- **Overview of Windows**
- IO Processing
- Thread Scheduling
- Synchronization
- Memory
- Performance and Debugging
- Where to go from here

# A Rose by any other Name

- Most Operating System decisions are defined by fundamentals of computer science, performance considerations, etc
  - Virtual Memory Abstraction
  - Monolithic Kernels

- So many of the OS internals reflect two different implementations of similar approaches
  - read()  vs ReadFile()
  - DLLs vs SharedObjects

# Windows Architecture

- HAL (Hardware Abstraction Layer):
  - support for x86 (initial), MIPS (initial), Alpha AXP, PowerPC (NT 3.51), Itanium (Windows XP/2003)
  - Machine-specific functions located in HAL
  - Additional functionality found in pci.sys, acpi.sys, etc
- At present, two main architectures: x64 and IA64
  - Allows a degree of focus in implementation
  - But, cedes certain fields to other OSes

# Windows Kernel

- Windows is a monolithic but modular system
  - No protection among pieces of kernel code and drivers
- Support for Modularity is somewhat weak:
  - Windows Drivers allow for dynamic extension of kernel functionality
  - Windows XP Embedded has special tools / packaging rules that allow coarse-grained configuration of the OS
- Windows Drivers are dynamically loadable kernel modules
  - Significant amount of code run as drivers (including network stacks such as TCP/IP and many services)
  - Built independently from the kernel
  - Can be loaded on-demand
  - Dependencies among drivers can be specified

# Comparing Layering, APIs, Complexity

- Windows
  - Kernel exports about 250 system calls (accessed via ntdll.dll)
  - Layered Windows/POSIX subsystems
  - Rich Windows API (17 500 functions on top of native APIs)
- Linux
  - Kernel supports about 200 different system calls
  - Layered BSD, Unix Sys V, POSIX shared system libraries
  - Compact APIs (1742 functions in Single Unix Specification Version 3; not including X Window APIs)

# Outline

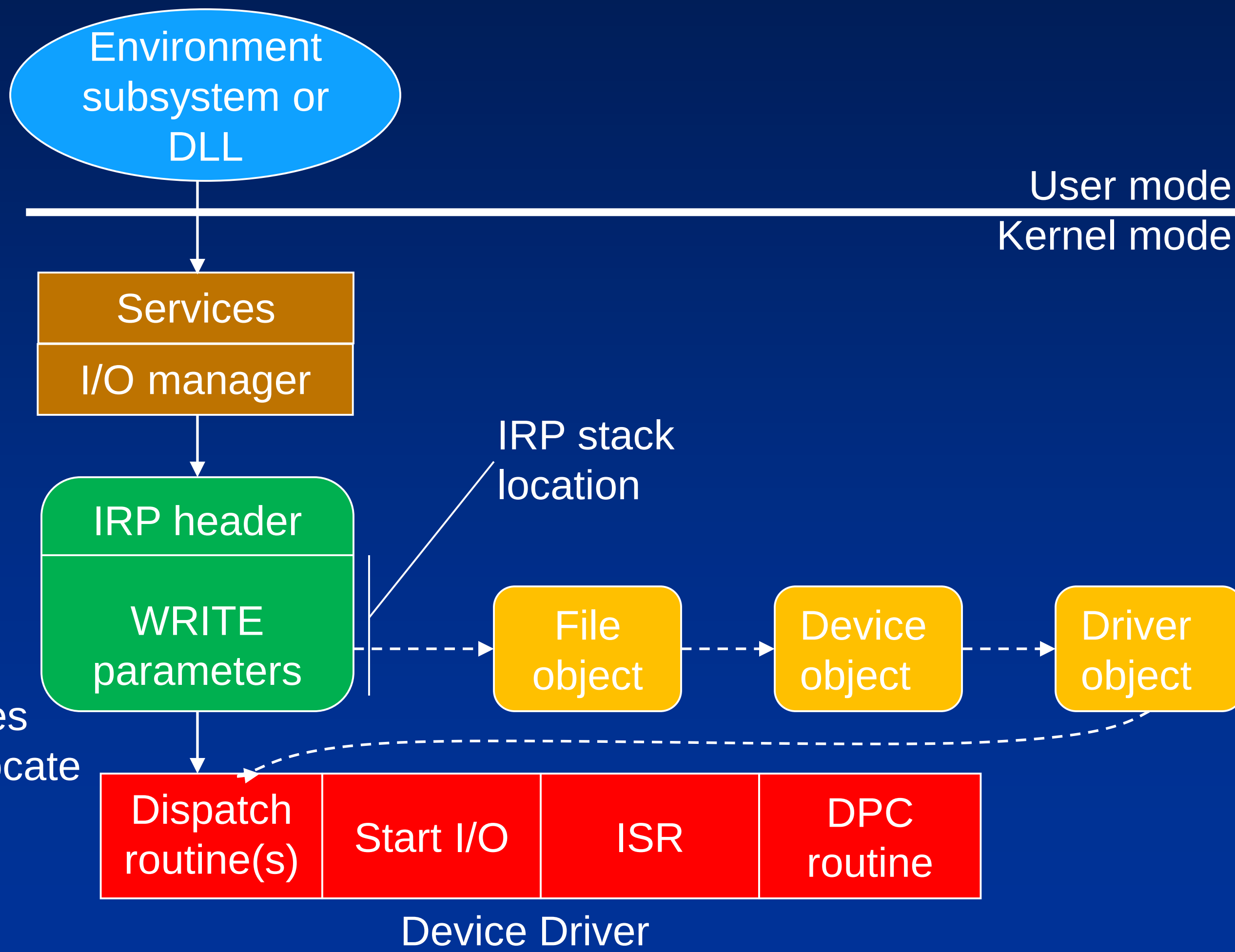
- Overview of Windows
- **IO Processing**
- Thread Scheduling
- Synchronization
- Memory
- Performance and Debugging
- Where to go from here

# I/O Processing

1) An application writes a file to the printer, passing a handle to the file object

2) The I/O manager creates an IRP and initializes first stack location

3) The I/O manager uses the driver object to locate the WRITE dispatch routine and calls it, passing the IRP



# IRP data

IRP consists of two parts:

- Fixed portion (header):
  - Type and size of the request
  - Whether request is synchronous or asynchronous
  - Pointer to buffer for buffered I/O
  - State information (changes with progress of the request)
- One or more stack locations:
  - Function code
  - Function-specific parameters
  - Pointer to caller‘s file object
- While active, IRPs are stored in a thread-specific queue
  - I/O system may free any outstanding IRPs if thread terminates

# Completing an I/O request

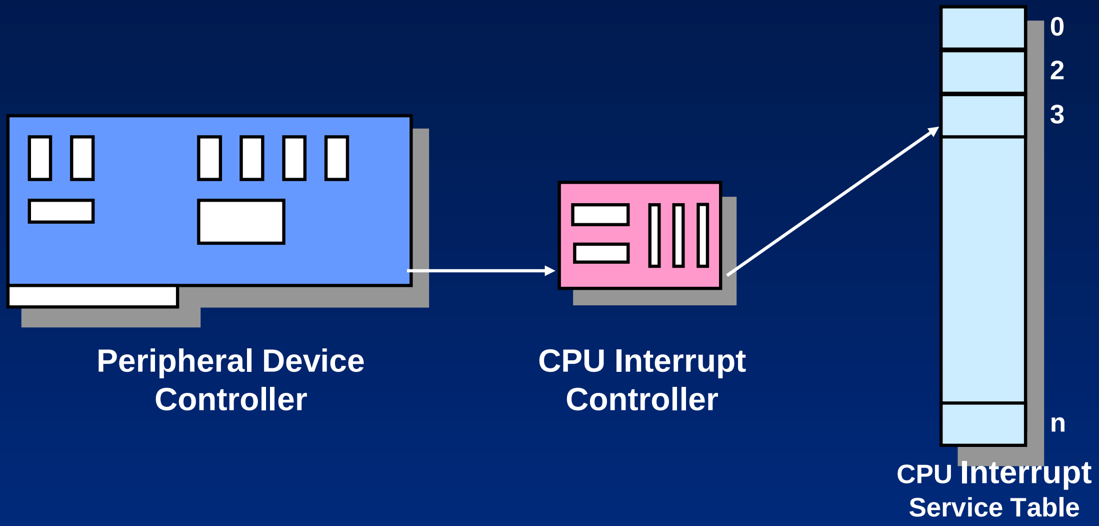
Servicing an interrupt:

- ISR schedules Deferred Procedure Call (**DPC**); dismisses int.
- **DPC** routine starts next I/O request and completes interrupt servicing
- May call completion routine of higher-level driver

I/O completion:

- Record the outcome of the operation in an I/O status block
- Return data to the calling thread – by queuing a kernel-mode Asynchronous Procedure Call (**APC**)
- **APC** executes in context of calling thread; copies data; frees IRP; sets calling thread to signaled state
- I/O is now considered complete; waiting threads are released

# Flow of Interrupts



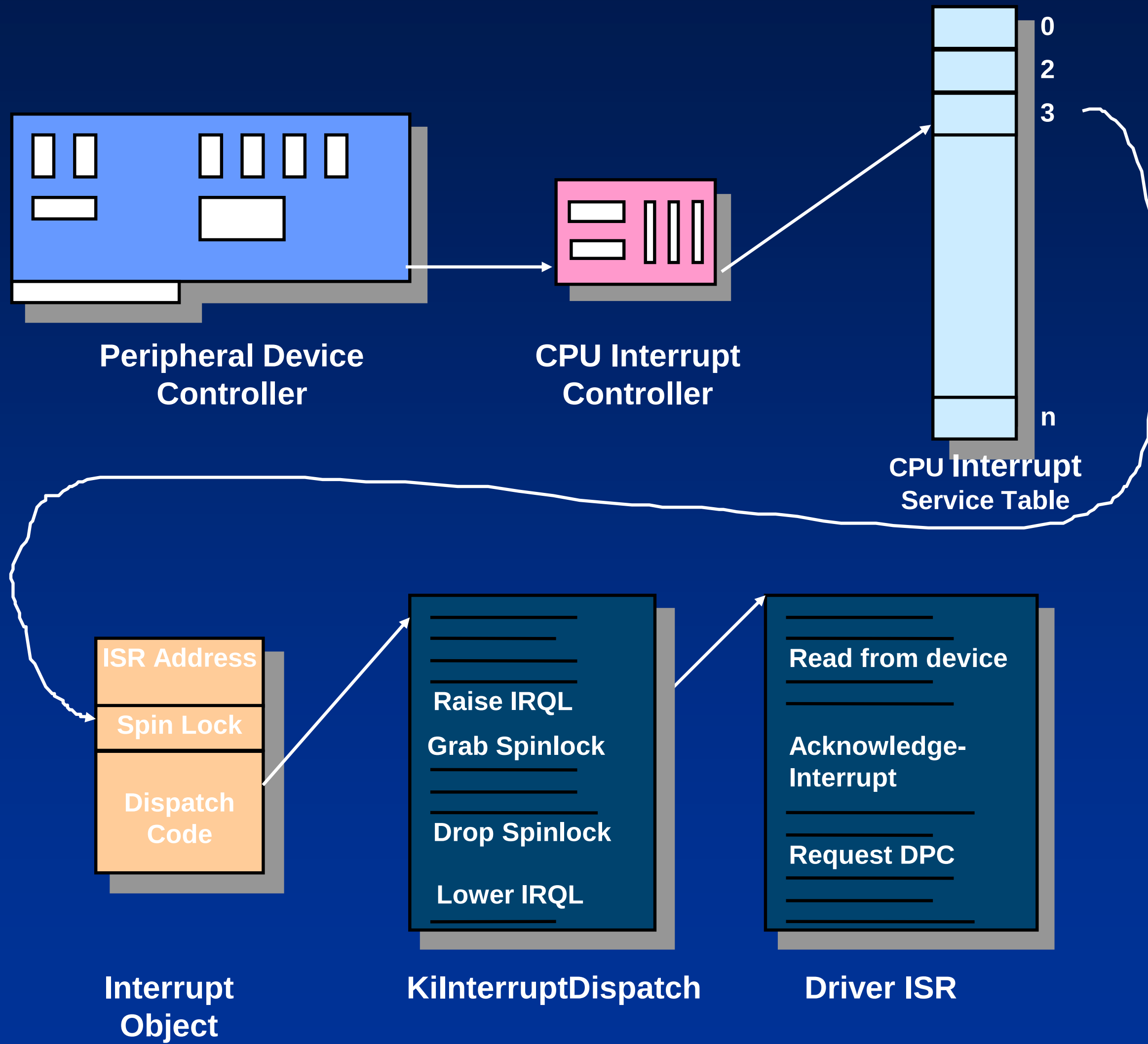
# IRQLs on 64-bit Systems

| | x64 | IA64 |
|---|---|---|
| 15 | High/Profile | High/Profile/Power |
| 14 | Interprocessor Interrupt/Power | Interprocessor Interrupt |
| 13 | Clock | Clock |
| 12 | Synch (Srv 2003) | Synch (MP only) |
| | Device n | Device n |
| | . | . |
| 4 | . | Device 1 |
| 3 | Device 1 | Correctable Machine Check |
| 2 | Dispatch/DPC | Dispatch/DPC & Synch (UP only) |
| 1 | APC | APC |
| 0 | Passive/Low | Passive/Low |

# IRQLs on 64-bit Systems

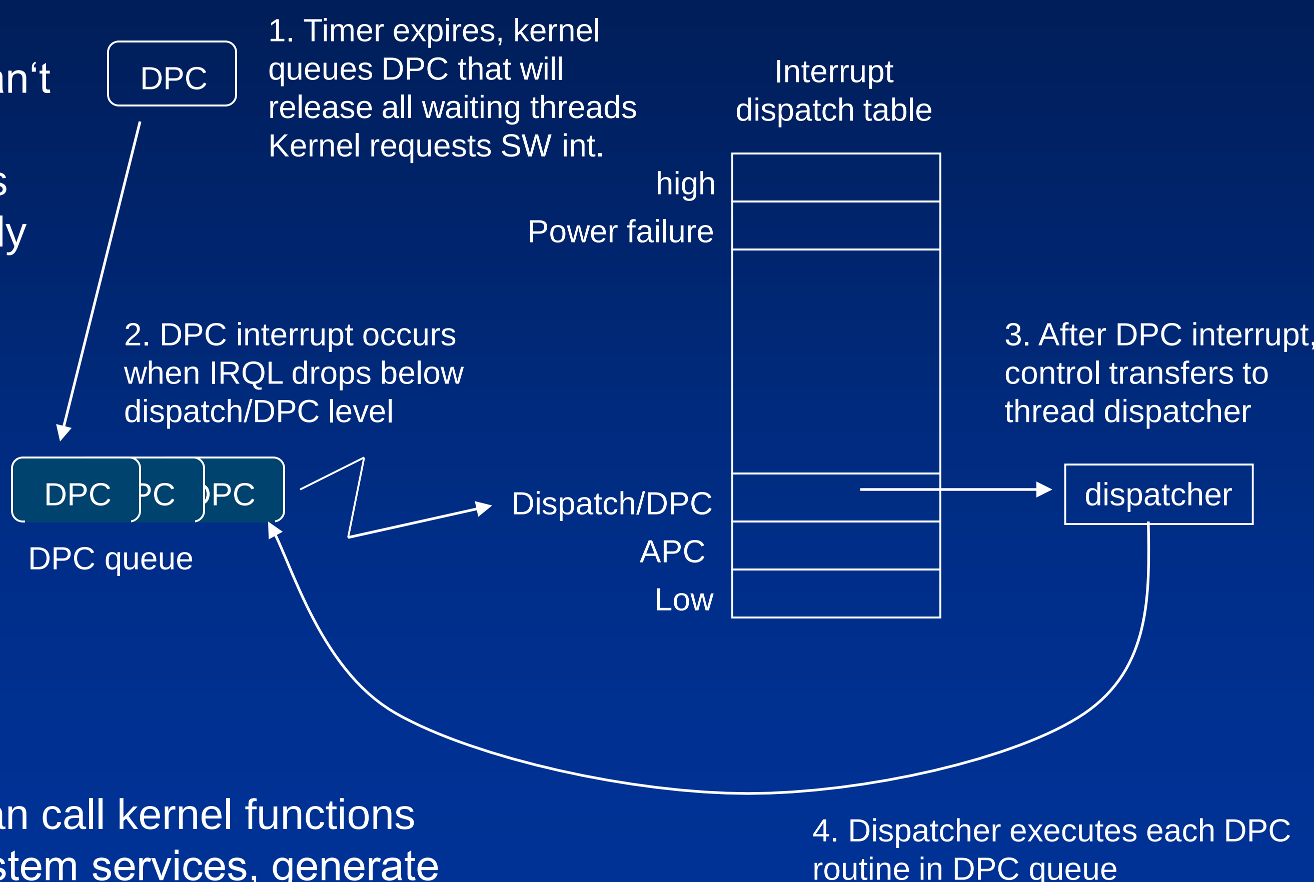
- When writing kernel code, five IRQLs matter:
  - HIGH – Mask all interrupts
  - DIRQL – IRQL for a particular device
  - DISPATCH / DPC –
    - No thread scheduling
    - No page faults
  - APC – Run code in a specific thread's context
  - PASSIVE – Default

# Flow of Interrupts



# Delivering a DPC

DPC routines can't assume what process address space is currently mapped




DPC routines can call kernel functions but can't call system services, generate page faults, or create or wait on objects

# I/O Processing

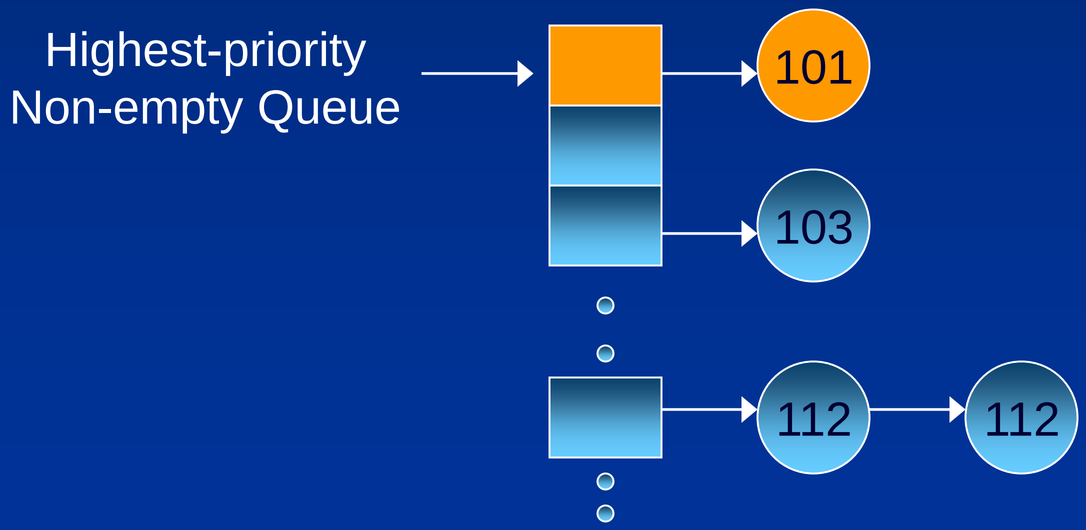
- Linux 2.2 had the notion of bottom halves (BH) for low-priority interrupt processing
  - Fixed number of BHs
  - Only one BH of a given type could be active on a SMP
- Linux 2.4 introduced *tasklets*, which are non-preemptible procedures called with interrupts enabled
- Tasklets are the equivalent of Windows Deferred Procedure Calls (DPCs)

# Outline

- Overview of Windows
- IO Processing
- **Thread Scheduling**
- Synchronization
- Memory
- Performance and Debugging
- Where to go from here

# Linux Scheduling

- Linux 2.6 has a revamped scheduler that's O(1) from Ingo Molnar that:
  - Calculates a task's priority at the time it makes scheduling decision
  - Has per-CPU ready queues where the tasks are pre-sorted by priority



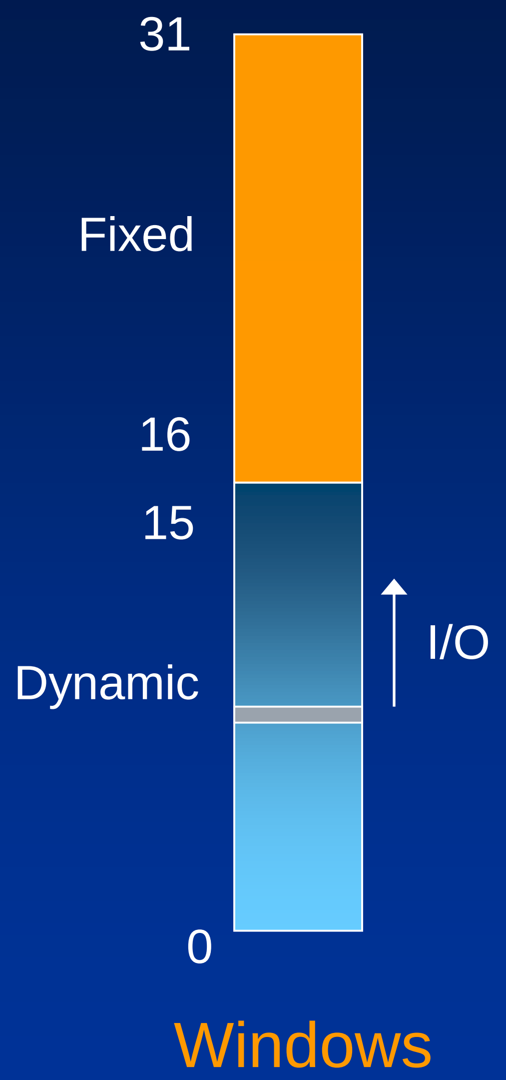
# Scheduling

- Windows NT has always had an O(1) scheduler based on pre-sorted thread priority queues
- Server 2003 introduced per-CPU ready queues
  - Linux load balances queues
  - Windows does not
    - Not seen as an issue in performance testing by Microsoft
    - Applications where it might be an issue are expected to use affinity

# Scheduling Priorities

## Windows

- Two scheduling classes
  - "Real time" (fixed) - priority 16-31
  - Dynamic - priority 1-15
- Higher priorities are favored
  - Priorities of dynamic threads get boosted on wakeups
  - Thread priorities are never lowered



# Windows Scheduling Details

- Most threads run in variable priority levels
  - Priorities 1-15;
  - A newly created thread starts with a base priority
  - Threads that complete I/O operations experience priority boosts (but never higher than 15)
  - A thread's priority will never be below base priority
- The Windows API function SetThreadPriority() sets the priority value for a specified thread
  - This value, together with the priority class of the thread's process, determines the thread's base priority level
  - Windows will dynamically adjust priorities for non-realtime threads

# Process Management

## Windows

- Process
  - Address space, handle table, statistics and at least one thread
  - No inherent parent/child relationship
- Threads
  - Basic scheduling unit
  - Fibers - cooperative user-mode threads
- Win7: User-Mode Scheduling(UMS)
  - User scheduled
  - Kernel supported

## Linux

- Process is called a Task
  - Basic Address space, handle table, statistics
  - Parent/child relationship
  - Basic scheduling unit
- Threads
  - No threads per-se
  - Tasks can act like Windows threads by sharing handle table, PID and address space
  - PThreads – cooperative user-mode threads

# Scheduling Timeslices

## Windows

- The thread timeslice (quantum) is 10ms-120ms
  - When quanta can vary, has one of 2 values
- Reentrant and preemptible

Fixed: 120ms

20ms Background

Foreground: 60ms

## Linux

- The thread quantum is 10ms-200ms
  - Default is 100ms
  - Varies across entire range based on priority, which is based on interactivity level
- Reentrant and preemptible

10ms ← → 200ms

100ms

# Outline

- Overview of Windows
- IO Processing
- Thread Scheduling
- **Synchronization**
- Memory
- Performance and Debugging
- Where to go from here

# Windows Synchronization

- Two types of Synchronization:
  - “Fast”
    - Protect small amounts of data
    - Busy waits
  - “Slow”
    - Producer / consumer, etc
    - Scheduler event
    - Notification of system events
      - E.G. wait for thread to exit

# Windows Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems.
- Uses *spinlocks* on multiprocessor systems.
- Provides *dispatcher objects* which may act as mutexes and semaphores.
- Dispatcher objects may also provide *events*. An event acts much like a condition variable.

# Queued Spinlocks

- **Problem**: Checking status of spinlock via test-and-set operation creates bus contention
- Queued spinlocks maintain queue of waiting processors
- First processor acquires lock; other processors wait on processor-local flag
  - Thus, busy-wait loop requires no access to the memory bus
- When releasing lock, the first processor's flag is modified
  - Exactly one processor is being signaled
  - Pre-determined wait order

# Other High-Perf Synchronization

- **Problem**: If the data under synchronization is small, is an interlocked operation sufficient or is a “lock” required
- **Semaphores:** The count can be the data
- **Test and Set, Swap, etc –** Exchange small sets of flags or other simple data
- **How about a linked list?**
- **Windows SLists or Interlocked Singly Linked Lists**
  - **Push Entry**
  - **Pop Entry**
  - **Flush List**

# Synchronizing Threads with Kernel Objects

```
DWORD WaitForSingleObject( HANDLE hObject, DWORD dwTimeout );

DWORD WaitForMultipleObjects( DWORD cObjects,
                              LPHANDLE lpHandles, BOOL bWaitAll,
                              DWORD dwTimeout );
```

The following kernel objects can be used to synchronize threads:

- Processes
- Threads
- Files
- Console input
- File change notifications
- Mutexes
- Events (auto-reset + manual-reset)
- Waitable timers

# Wait Functions - Details

- WaitForSingleObject():
  - hObject specifies kernel object
  - dwTimeout specifies wait time in msec
    - dwTimeout == 0 - no wait, check whether object is signaled
    - dwTimeout == INFINITE - wait forever
- WaitForMultipleObjects():
  - cObjects <= MAXIMUM_WAIT_OBJECTS (64)
  - lpHandles - pointer to array identifying these objects
  - bWaitAll - whether to wait for first signaled object or all objects
    - Function returns index of first signaled object
- Side effects:
  - Mutexes, auto-reset events and waitable timers will be reset to non-signaled state after completing wait functions

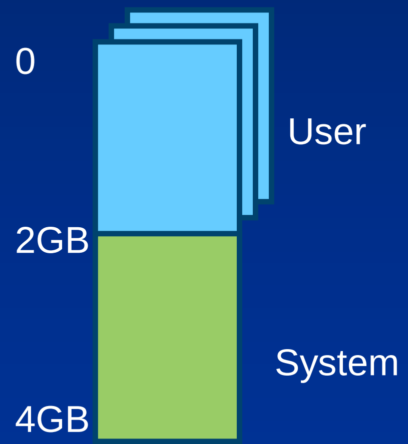
# Outline

- Overview of Windows
- IO Processing
- Thread Scheduling
- Synchronization
- **Memory**
- Performance and Debugging
- Where to go from here

# Virtual Memory Management

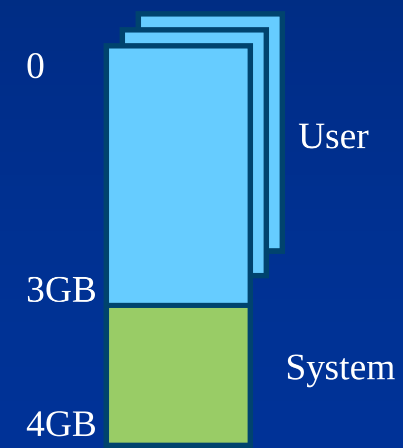
## Windows

- 32-bit versions split user-mode/kernel-mode from 2GB/2GB to 3GB/1GB
- Demand-paged virtual memory
  - 32 or 64-bits
  - Copy-on-write
  - Shared memory
  - Memory mapped files




## Linux

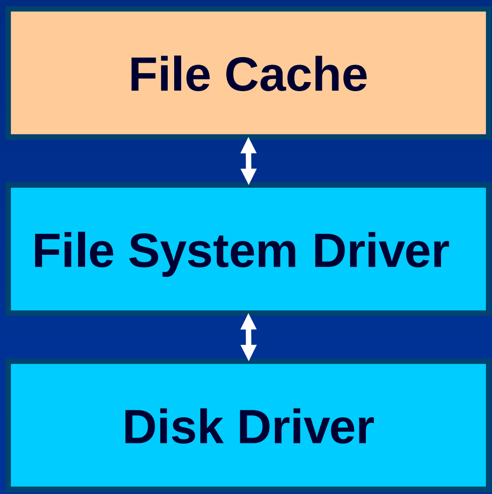
- Splits user-mode/kernel-mode from 1GB/3GB to 3GB/1GB
  - 2.6 has “4/4 split” option where kernel has its own address space
- Demand-paged virtual memory
  - 32-bits and/or 64-bits
  - Copy-on-write
  - Shared memory
  - Memory mapped files



# File Caching

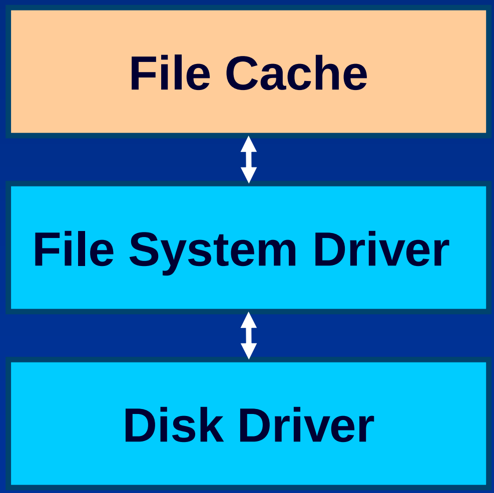
## Windows

- Single global common cache
- Virtual file cache
  - Caching is at file vs. disk block level
  - Files are memory mapped into kernel memory
- Cache allows for zero-copy file serving




## Linux

- Single global common cache
- Virtual file cache
  - Caching is at file vs. disk block level
  - Files are memory mapped into kernel memory
- Cache allows for zero-copy file serving



# Kernel Memory Allocation

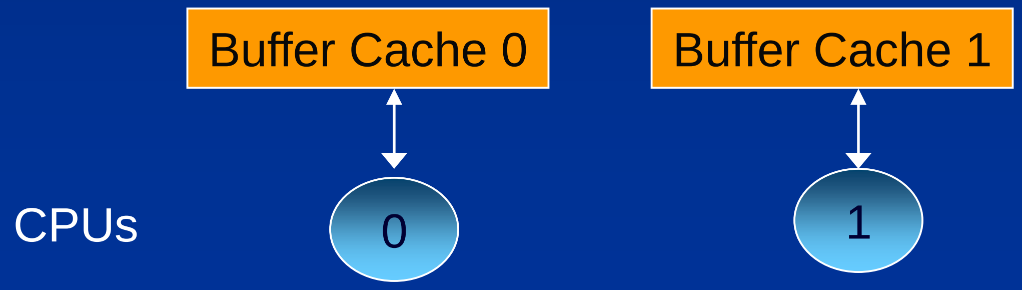
- Pool Allocations
  - ExAllocatePoolWithTag(type, size, tag)
    - Paged vs NonPaged
    - Size in bytes
    - Tag identifies allocations for debugging purposes
- Allocations for device operations
  - **MmAllocateNonCachedMemory**, **MmAllocateContiguousMemorySpecifyCache**, **AllocateCommonBuffer**
- http://msdn.microsoft.com/en-us/library/aa489507.aspx

# Kernel Memory Allocation cont.

- When drivers need to work with many kernel allocations
  - **Ex*Xxx*LookasideList**
    - Initialize / Allocate / Free
    - Paged vs NonPaged
  - System managed list of allocations of specified size
    - Heuristics for availability of system memory
    - Frequency of allocation
  - Faster allocation / free times
    - Can be per-processor / per-Node

# Per-CPU Memory Allocation

- Keeping accesses to memory localized to a CPU minimizes CPU cache thrashing
  - Hurts performance on enterprise SMP workloads
- Linux 2.4 introduced per-CPU kernel memory buffers
- Windows introduced per-CPU buffers in an NT 4 Service Pack in 1997



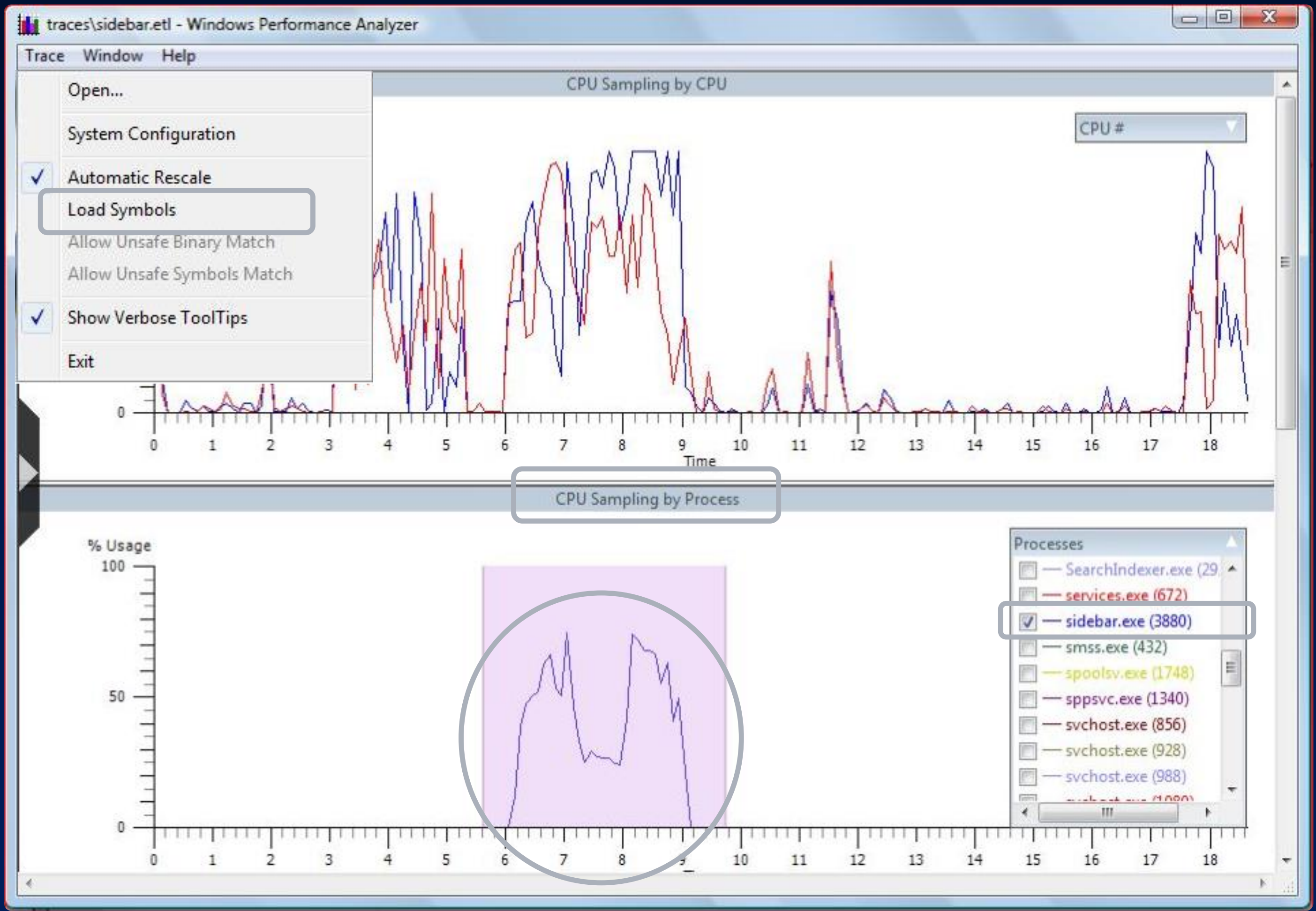
# Outline

- Overview of Windows
- IO Processing
- Thread Scheduling
- Synchronization
- Memory
- **Performance and Debugging**
- Where to go from here

# Performance Testing

- Windows Performance Instrumentation
  - Support for profiling
- XPerf – common interface for most instrumentation
  - Take CPU profiles
  - Collect instrumented events
  - Collect stacktraces for (almost) any profile source
- Let’s see an example
- http://msdn.microsoft.com/en-us/performance/default.aspx

# What is CPU Time Spent On?




xperf trace.etl

# CPU Summary Table

Grouping columns

Aggregated columns

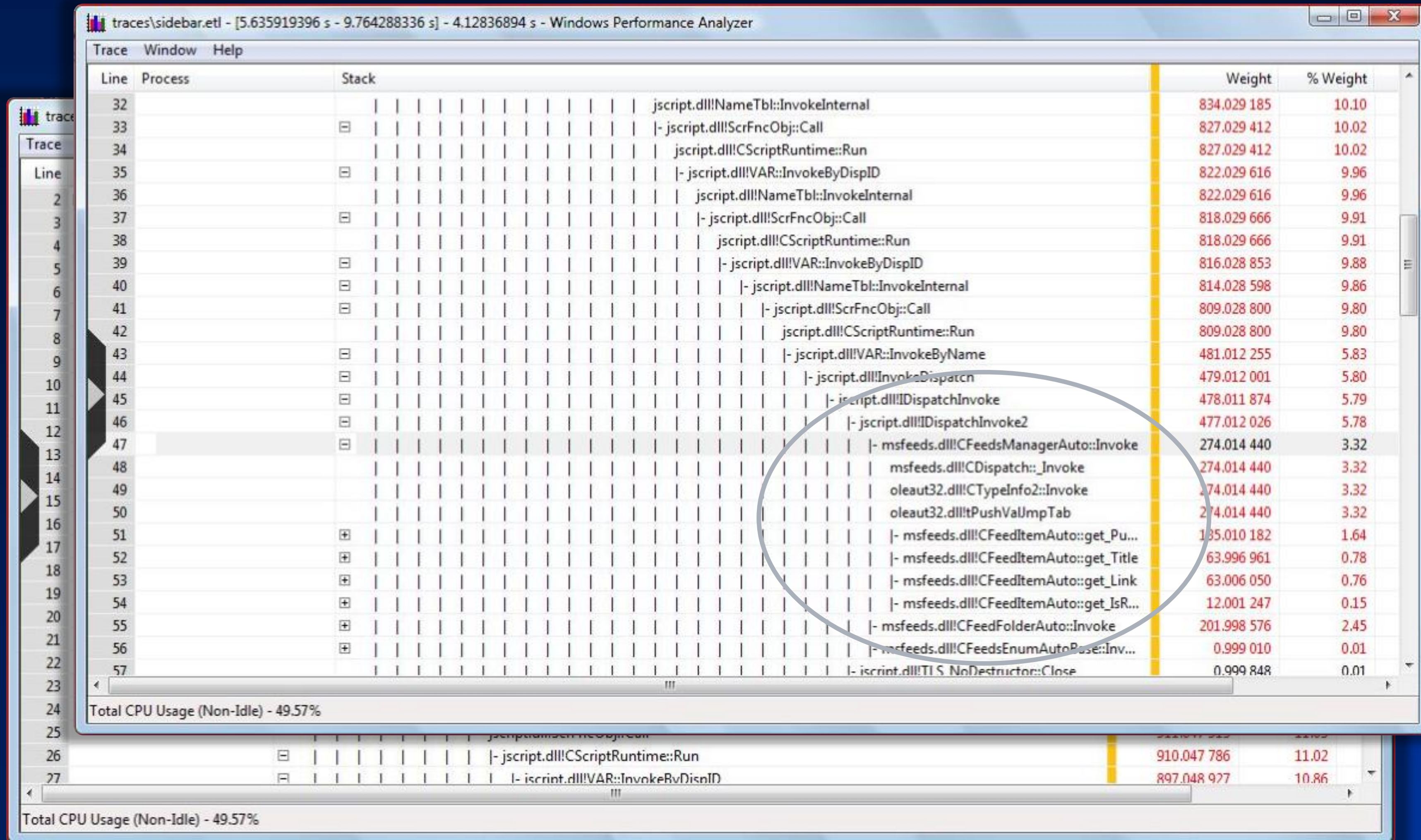
traces\sidebar.etl - [5.635919396 s - 9.764288336 s] - 4.12836894 s - Windows Performance Analyzer

Trace Window Help

| Line | Process | Stack | Weight | % Weight | Count | TimeStamp |
|---|---|---|---|---|---|---|
| 1 | ⊞ Idle (0) | | 4,163.665 136 | 50.43 | 2,287 | |
| 2 | ⊟ sidebar.exe (3880) | | 2,793.029 829 | 33.83 | 2,793 | |
| 3 | | ⊞ [Root] | 2,784.030 639 | 33.72 | 2,784 | |
| 4 | | ⊞ ? | 8.999 190 | 0.11 | 9 | |
| 5 | ⊞ svchost.exe (1116) | | 823.062 734 | 9.97 | 823 | |
| 6 | ⊞ dwm.exe (4828) | | 141.040 023 | 1.71 | 141 | |
| 7 | ⊞ explorer.exe (2760) | | 132.982 010 | 1.61 | 133 | |
| 8 | ⊞ System (4) | | 58.986 168 | 0.71 | 59 | |
| 9 | InoRT.exe (448) | ⊞ [Root] | 48.993 936 | 0.59 | 49 | |
| 10 | taskeng.exe (2012) | ⊞ [Root] | 27.001 351 | 0.33 | 27 | |
| 11 | svchost.exe (1164) | ⊞ [Root] | 18.994 313 | 0.23 | 19 | |
| 12 | svchost.exe (988) | ⊞ [Root] | 12.999 849 | 0.16 | 13 | |
| 13 | svchost.exe (928) | ⊞ [Root] | 6.994 746 | 0.08 | 7 | |
| 14 | csrss.exe (640) | ⊞ [Root] | 6.988 878 | 0.08 | 7 | |
| 15 | SearchIndexer.exe (2932) | ⊞ [Root] | 5.999 797 | 0.07 | 6 | |
| 16 | lsass.exe (684) | ⊞ [Root] | 5.999 085 | 0.07 | 6 | |
| 17 | svchost.exe (856) | ⊞ [Root] | 4.999 518 | 0.06 | 5 | |
| 18 | svchost.exe (1772) | ⊞ [Root] | 1.001 245 | 0.01 | 1 | |
| 19 | lsm.exe (692) | ⊞ [Root] | 1.001 244 | 0.01 | 1 | |
| 20 | svchost.exe (1388) | ⊞ [Root] | 1.000 000 | 0.01 | 1 | |
| 21 | spoolsv.exe (1748) | ⊞ [Root] | 0.999 568 | 0.01 | 1 | |
| 22 | svchost.exe (1520) | ⊞ [Root] | 0.998 450 | 0.01 | 1 | |

Total CPU Usage (Non-Idle) - 49.57%

# CPU Summary Table



# Kernel Debugging

- Useful for investigating internal system state not available from other tools
  - Requires 2 computers (host and target)
  - Target would be halted while host debugger in use
- XP & Server 2003 support live local kernel debugging
  - kd -kl
  - Technically requires system to be booted /DEBUG to work correctly
  - You can edit kernel memory on the live system (!)
  - But, not all commands work
- http://www.microsoft.com/whdc/devtools/debugging/default.mspx

# Outline

- Overview of Windows
- IO Processing
- Thread Scheduling
- Synchronization
- Memory
- Performance and Debugging
- **Where to go from here**

# Where to go from here

- Windows Driver Kit:
  - http://www.microsoft.com/whdc/devtools/WDK/default.mspx
- Msdn.microsoft.com
  - Every function is documented
  - Many have example code

# The Big Picture