# Unit OS3: Concurrency

## 3.2. Windows Trap Dispatching, Interrupts, Synchronization

# Copyright Notice

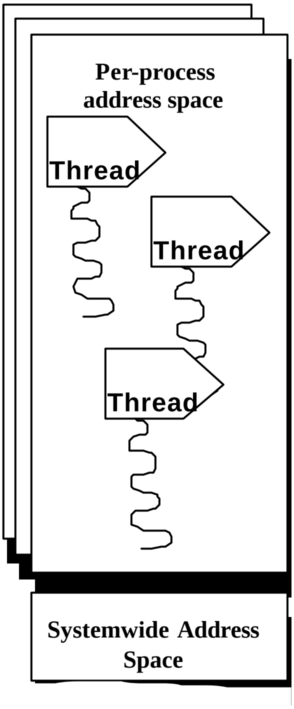
© 2000-2005 David A. Solomon and Mark Russinovich

- 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)

# Roadmap for Section 3.2.

- Trap and Interrupt dispatching
- IRQL levels & Interrupt Precedence
- Spinlocks and Kernel Synchronization
- Executive Synchronization

# Processes and Threads



- **What is a process?**
  - Represents an instance of a running program
    - you create a process to run a program
    - starting an application creates a process
  - Process defined by:
    - Address space
    - Resources (e.g. open handles)
    - Security profile (token)
- **What is a thread?**
  - An execution context within a process
  - Unit of scheduling (threads run, processes don't run)
  - All threads in a process share the same per-process address space
    - Services provided so that threads can synchronize access to shared resources (critical sections, mutexes, events, semaphores)
  - All threads in the system are scheduled as peers to all others, without regard to their "parent" process
- **System calls**
  - Primary argument to CreateProcess is image file name (or command line)
  - Primary argument to CreateThread is a function entry point address

# Kernel Mode Versus User Mode

- A processor state
  - Controls access to memory
  - Each memory page is tagged to show the required mode for reading and for writing
    - Protects the system from the users
    - Protects the user (process) from themselves
    - System is not protected from system
  - Code regions are tagged "no write in any mode"
  - Controls ability to execute privileged instructions
  - A Windows abstraction
    - Intel:  Ring 0, Ring 3
- Control flow (i.e.; a thread) ca change from user to kernel mode and back
  - Does not affect scheduling
  - Thread context includes info about execution mode (along with registers, etc)
- PerfMon counters:
  - "Privileged Time" and "User Time"
  - 4 levels of granularity: thread, process, processor, system

# Getting Into Kernel Mode

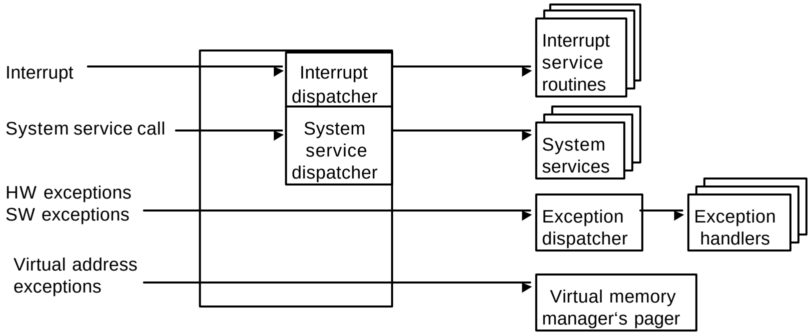
Code is run in kernel mode for one of three reasons:

1. Requests from user mode
   - Via the system service dispatch mechanism
   - Kernel-mode code runs in the context of the requesting thread
2. Interrupts from external devices
   - Windows interrupt dispatcher invokes the interrupt service routine
   - ISR runs in the context of the interrupted thread (so-called "arbitrary thread context")
   - ISR often requests the execution of a "DPC routine," which also runs in kernel mode
   - Time not charged to interrupted thread
3. Dedicated kernel-mode system threads
   - Some threads in the system stay in kernel mode at all times (mostly in the "System" process)
   - Scheduled, preempted, etc., like any other threads

# Trap dispatching

Trap: processor‘s mechanism to capture executing thread

- Switch from user to kernel mode
- Interrupts – asynchronous
- Exceptions - synchronous



Interrupt

System service call

HW exceptions
SW exceptions

Virtual address
exceptions

Interrupt dispatcher

System service dispatcher

Interrupt service routines

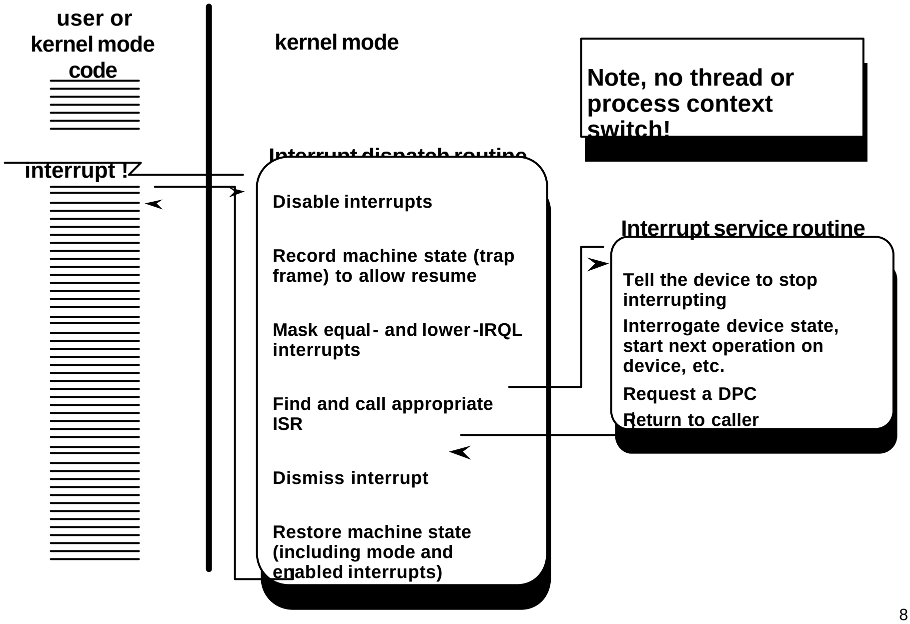
System services

Exception dispatcher

Exception handlers

Virtual memory manager‘s pager

# Interrupt Dispatching



user or
kernel mode
code

kernel mode

Note, no thread or process context switch!

interrupt !

Interrupt dispatch routine

Disable interrupts

Record machine state (trap frame) to allow resume

Mask equal- and lower-IRQL interrupts

Find and call appropriate ISR

Dismiss interrupt

Restore machine state (including mode and enabled interrupts)

Interrupt service routine

Tell the device to stop interrupting

Interrogate device state, start next operation on device, etc.

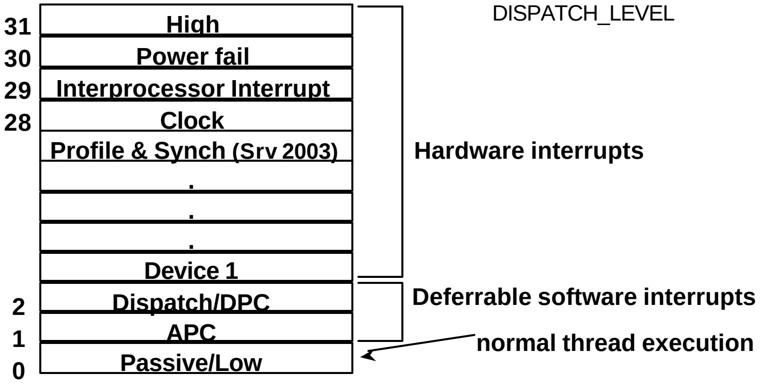
Request a DPC

Return to caller

# Interrupt Precedence via IRQLs (x86)

- **IRQL = Interrupt Request Level**
  - the “precedence” of the interrupt with respect to other interrupts
  - Different interrupt sources have different IRQLs
  - not the same as IRQ

- IRQL is also a state of the processor
- Servicing an interrupt raises processor IRQL to that interrupt’s IRQL
  - this masks subsequent interrupts at equal and lower IRQLs
- User mode is limited to IRQL 0
- No waits or page faults at IRQL >= DISPATCH_LEVEL



| | |
|---|---|
| **31** | **High** |
| **30** | **Power fail** |
| **29** | **Interprocessor Interrupt** |
| **28** | **Clock** |
| | **Profile & Synch (Srv 2003)** |
| | **.** |
| | **.** |
| | **.** |
| | **Device 1** |
| **2** | **Dispatch/DPC** |
| **1** | **APC** |
| **0** | **Passive/Low** |

**Hardware interrupts**

**Deferrable software interrupts**

**normal thread execution**

# Interrupt processing

- Interrupt dispatch table (IDT)
  - Links to interrupt service routines

- x86:
  - Interrupt controller interrupts processor (single line)
  - Processor queries for interrupt vector; uses vector as index to IDT

- After ISR execution, IRQL is lowered to initial level

# Interrupt object

- Allows device drivers to register ISRs for their devices
  - Contains dispatch code (initial handler)
  - Dispatch code calls ISR with interrupt object as parameter (HW cannot pass parameters to ISR)
- Connecting/disconnecting interrupt objects:
  - Dynamic association between ISR and IDT entry
  - Loadable device drivers (kernel modules)
  - Turn on/off ISR
- Interrupt objects can synchronize access to ISR data
  - Multiple instances of ISR may be active simultaneously (MP machine)
  - Multiple ISR may be connected with IRQL

# Predefined IRQLs

- **High**
  - used when halting the system (via *KeBugCheck()*)
- **Power fail**
  - originated in the NT design document, but has never been used
- **Inter-processor interrupt**
  - used to request action from other processor (dispatching a thread, updating a processors TLB, system shutdown, system crash)
- **Clock**
  - Used to update system's clock, allocation of CPU time to threads
- **Profile**
  - Used for kernel profiling (see Kernel profiler – Kernprof.exe, Res Kit)

# Predefined IRQLs (contd.)

- **Device**
  - Used to prioritize device interrupts
- **DPC/dispatch** and **APC**
  - Software interrupts that kernel and device drivers generate
- **Passive**
  - No interrupt level at all, normal thread execution

# 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 |

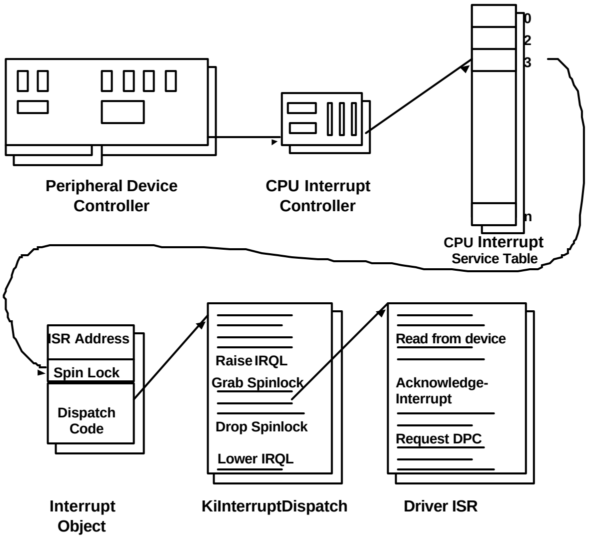
# Interrupt Prioritization & Delivery

- IRQLs are determined as follows:
  - x86 UP systems: IRQL = 27 - IRQ
  - x86 MP systems: bucketized (random)
  - x64 & IA64 systems: IRQL = IDT vector number / 16
- On MP systems, which processor is chosen to deliver an interrupt?
  - By default, any processor can receive an interrupt from any device
    - Can be configured with IntFilter utility in Resource Kit
  - On x86 and x64 systems, the IOAPIC (I/O advanced programmable interrupt controller) is programmed to interrupt the processor running at the lowest IRQL
  - On IA64 systems, the SAPIC (streamlined advanced programmable interrupt controller) is configured to interrupt one processor for each interrupt source
    - Processors are assigned round robin for each interrupt vector

# Software interrupts

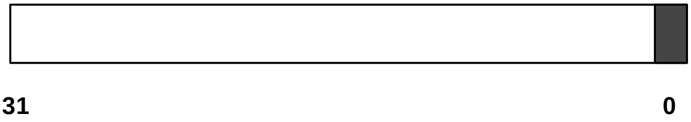
- Initiating thread dispatching
  - DPC allow for scheduling actions when kernel is deep within many layers of code
  - Delayed scheduling decision, one DPC queue per processor
- Handling timer expiration
- Asynchronous execution of a procedure in context of a particular thread
- Support for asynchronous I/O operations

# Flow of Interrupts



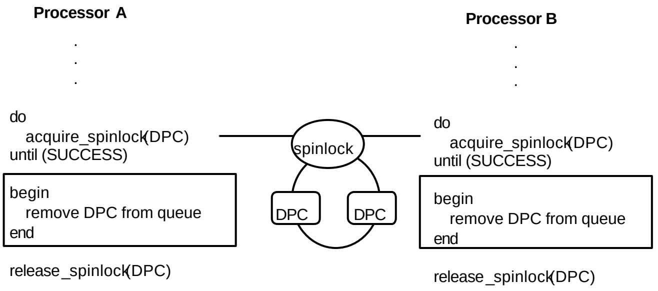
# Synchronization on SMP Systems

- Synchronization on MP systems use spinlocks to coordinate among the processors
- Spinlock acquisition and release routines implement a one-owner-at-a-time algorithm
  - A spinlock is either free, or is considered to be owned by a CPU
  - Analogous to using Windows API mutexes from user mode
- A spinlock is just a data cell in memory
  - Accessed with a test-and-modify operation that is atomic across all processors



  - KSPIN_LOCK is an opaque data type, typedef'd as a ULONG
  - To implement synchronization, a single bit is sufficient

# Kernel Synchronization



**Processor A**

```
.
.
.
do
   acquire_spinlock(DPC)
until (SUCCESS)
begin
   remove DPC from queue
end
release_spinlock(DPC)
```

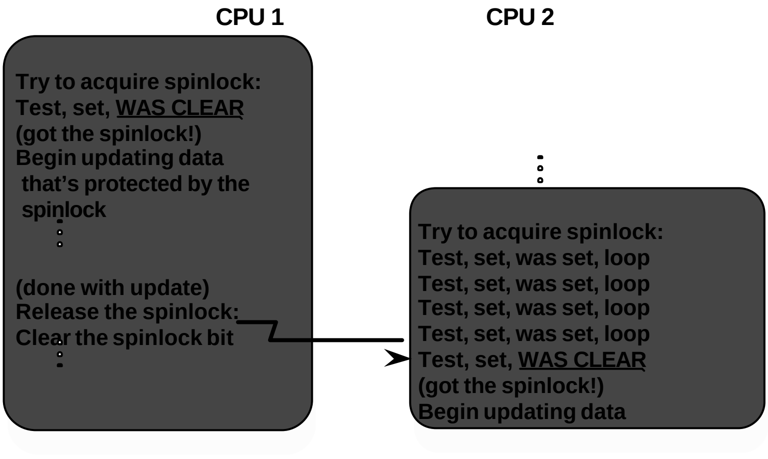
**Processor B**

```
.
.
.
do
   acquire_spinlock(DPC)
until (SUCCESS)
begin
   remove DPC from queue
end
release_spinlock(DPC)
```

▭ Critical section

A spinlock is a locking primitive associated with a global data structure, such as the DPC queue

# Spinlocks in Action



**CPU 1**

**Try to acquire spinlock:**
**Test, set, WAS CLEAR**
**(got the spinlock!)**
**Begin updating data that's protected by the spinlock**

**(done with update)**
**Release the spinlock:**
**Clear the spinlock bit**

**CPU 2**

**Try to acquire spinlock:**
**Test, set, was set, loop**
**Test, set, was set, loop**
**Test, set, was set, loop**
**Test, set, was set, loop**
**Test, set, WAS CLEAR**
**(got the spinlock!)**
**Begin updating data**

# 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

# SMP Scalability Improvements

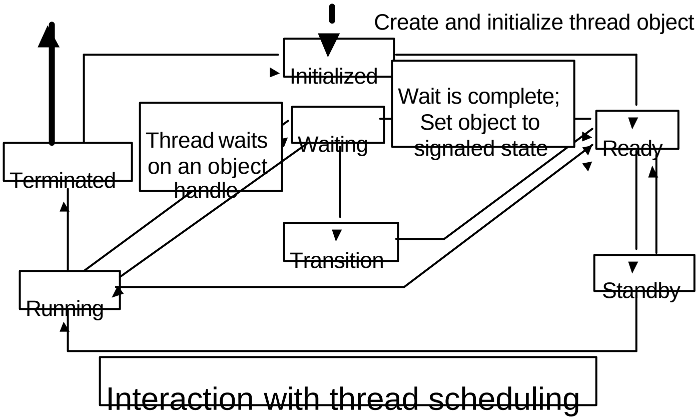
- Windows 2000: queued spinlocks
  - !qlocks in Kernel Debugger
- XP/2003:
  - Minimized lock contention for hot locks (PFN or Page Frame Database) lock
  - Some locks completely eliminated
    - Charging nonpaged/paged pool quotas, allocating and mapping system page table entries, charging commitment of pages, allocating/mapping physical memory through AWE functions
  - New, more efficient locking mechanism (pushlocks)
    - Doesn't use spinlocks when no contention
    - Used for object manager and address windowing extensions (AWE) related locks
- Server 2003:
  - More spinlocks eliminated (context swap, system space, commit)
  - Further reduction of use of spinlocks & length they are held
  - Scheduling database now per-CPU
    - Allows thread state transitions in parallel

# Waiting

- Flexible wait calls
  - Wait for one or multiple objects in one call
  - Wait for multiple can wait for "any" one or "all" at once
    - "All": all objects must be in the signalled state concurrently to resolve the wait
  - All wait calls include optional timeout argument
  - Waiting threads consume no CPU time
- Waitable objects include:
  - Events (may be auto-reset or manual reset; may be set or "pulsed")
  - Mutexes ("mutual exclusion", one-at-a-time)
  - Semaphores (n-at-a-time)
  - Timers
  - Processes and Threads (signalled upon exit or terminate)
  - Directories (change notification)
- No guaranteed ordering of wait resolution
  - If multiple threads are waiting for an object, and only one thread is released (e.g. it's a mutex or auto-reset event), which thread gets released is unpredictable
  - Typical order of wait resolution is FIFO; however APC delivery may change this order

# Executive Synchronization

- Waiting on Dispatcher Objects – outside the kernel

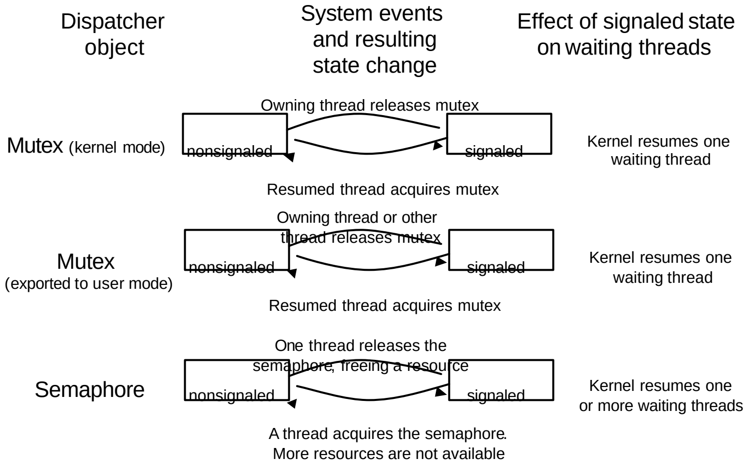

Create and initialize thread object
Initialized
Wait is complete; Set object to signaled state
Thread waits on an object handle
Waiting
Ready
Terminated
Transition
Standby
Running

Interaction with thread scheduling

# Interactions between Synchronization and Thread Dispatching

1. User mode thread waits on an event object‘s handle
2. Kernel changes thread‘s scheduling state from ready to waiting and adds thread to wait-list
3. Another thread sets the event
4. Kernel wakes up waiting threads; variable priority threads get priority boost
5. Dispatcher re-schedules new thread – it may preempt running thread it it has lower priority and issues software interrupt to initiate context switch
6. If no processor can be preempted, the dispatcher places the ready thread in the dispatcher ready queue to be scheduled later

# What signals an object?



Dispatcher object

System events and resulting state change

Effect of signaled state on waiting threads

Mutex (kernel mode)

Owning thread releases mutex

nonsignaled

signaled

Resumed thread acquires mutex

Kernel resumes one waiting thread

Mutex (exported to user mode)

Owning thread or other thread releases mutex

nonsignaled

signaled

Resumed thread acquires mutex

Kernel resumes one waiting thread

Semaphore

One thread releases the semaphore, freeing a resource

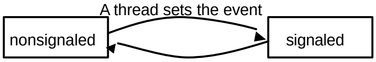
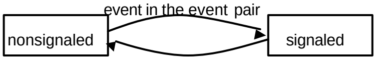
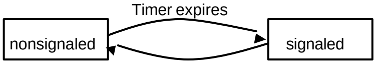
nonsignaled

signaled

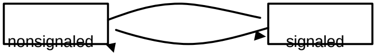
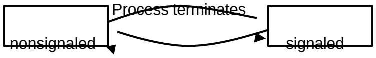
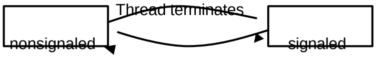
A thread acquires the semaphore. More resources are not available

Kernel resumes one or more waiting threads

# What signals an object? (contd.)

| Dispatcher object | System events and resulting state change | Effect of signaled state on waiting threads |
|---|---|---|
| Event | <br>A thread sets the event<br>nonsignaled<br>signaled<br>Kernel resumes one or more threads | Kernel resumes one or more waiting threads |
| Event pair | <br>Dedicated thread sets one event in the event pair<br>nonsignaled<br>signaled<br>Kernel resumes the other dedicated thread | Kernel resumes waiting dedicated thread |
| Timer | <br>Timer expires<br>nonsignaled<br>signaled<br>A thread (re) initializes the timer | Kernel resumes all waiting threads |

# What signals an object? (contd.)

| Dispatcher object | System events and resulting state change | Effect of signaled state on waiting threads |
|---|---|---|
| File | <br>IO operation completes<br>nonsignaled<br>signaled<br>Thread initiates wait on an IO port | Kernel resumes waiting dedicated thread |
| Process | <br>Process terminates<br>nonsignaled<br>signaled<br>A process reinitializes the process object | Kernel resumes all waiting threads |
| Thread | <br>Thread terminates<br>nonsignaled<br>signaled<br>A thread reinitializes the thread object | Kernel resumes all waiting threads |

# Further Reading

- Mark E. Russinovich and David A. Solomon, Microsoft Windows Internals, 4th Edition, Microsoft Press, 2004.
- Chapter 3 - System Mechanisms
  - Trap Dispatching (pp. 85 ff.)
  - Synchronization (pp. 149 ff.)
  - Kernel Event Tracing (pp. 175 ff.)