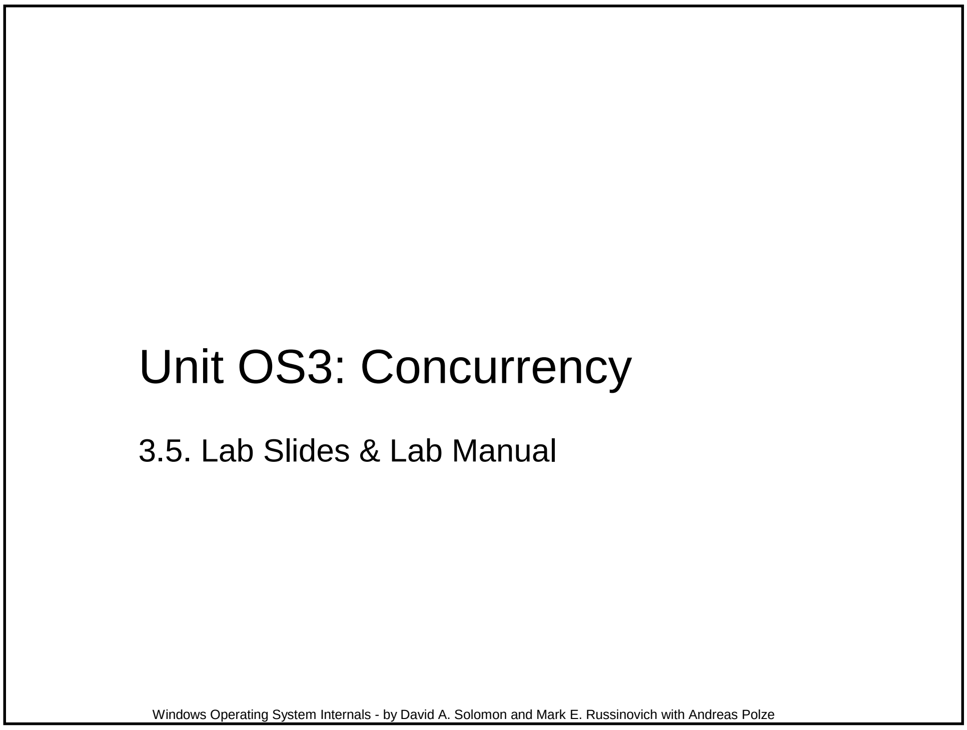
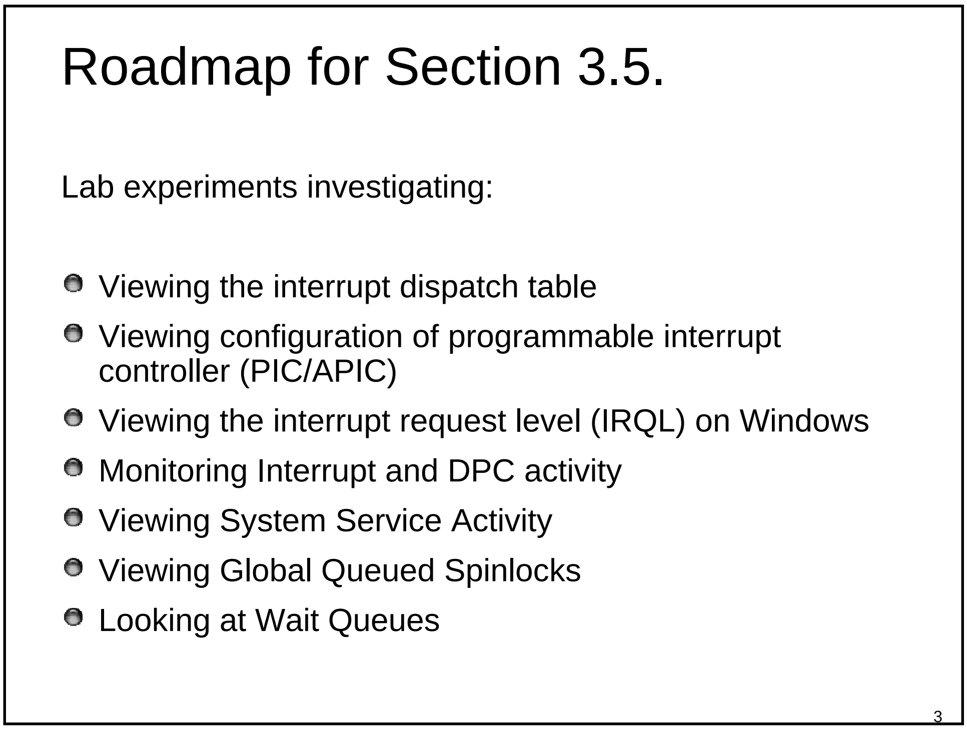


Unit OS3: Concurrency

3.5. Lab Slides & Lab Manual

Windows Operating System Internals - by David A. Solomon and Mark E. Russinovich with Andreas Polze



Roadmap for Section 3.5.

Lab experiments investigating:

- Viewing the interrupt dispatch table
- Viewing configuration of programmable interrupt controller (PIC/APIC)
- Viewing the interrupt request level (IRQL) on Windows
- Monitoring Interrupt and DPC activity
- Viewing System Service Activity
- Viewing Global Queued Spinlocks
- Looking at Wait Queues

x86 Interrupt Controllers -

Hardware Interrupt Processing

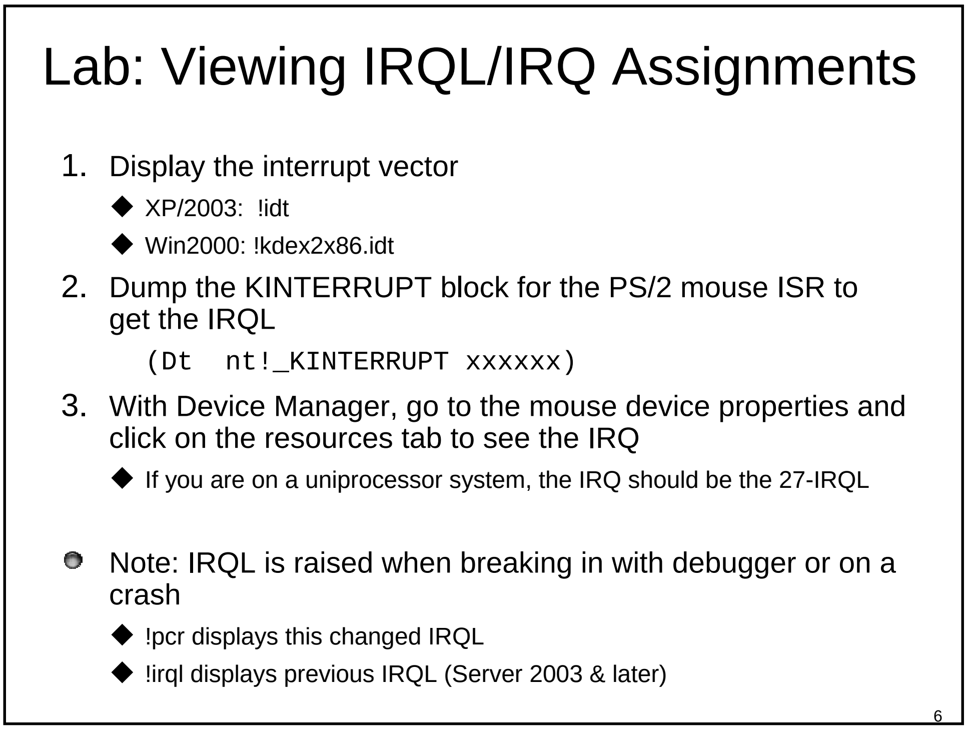
- Most x86 systems rely on
  - i8259A Programmable Interrupt Controller (PIC) or
  - a variant of the i82489 Advanced Programmable Interrupt Controller (APIC) - most new computers
- PICs work only with uniprocessor systems
  - APICs work with multiprocessor systems
- Lab: Observe PIC / APIC configuration
  - Use **!pic** and **!apic** kernel debugger commands

Viewing the IRQL on Windows

- On Windows Server 2003, kernel debugger displays IRQL:
  - !irql debugger command:

```
kd> !irql
Debugger saved IRQL for processor 0x0 -- 0 (LOW_LEVEL)
```

- Processor control region (PCR) and processor control block (PRCB) store:
  - current IRQL,
  - pointer to the hardware IDT,
  - currently running thread,
  - next thread selected to run.



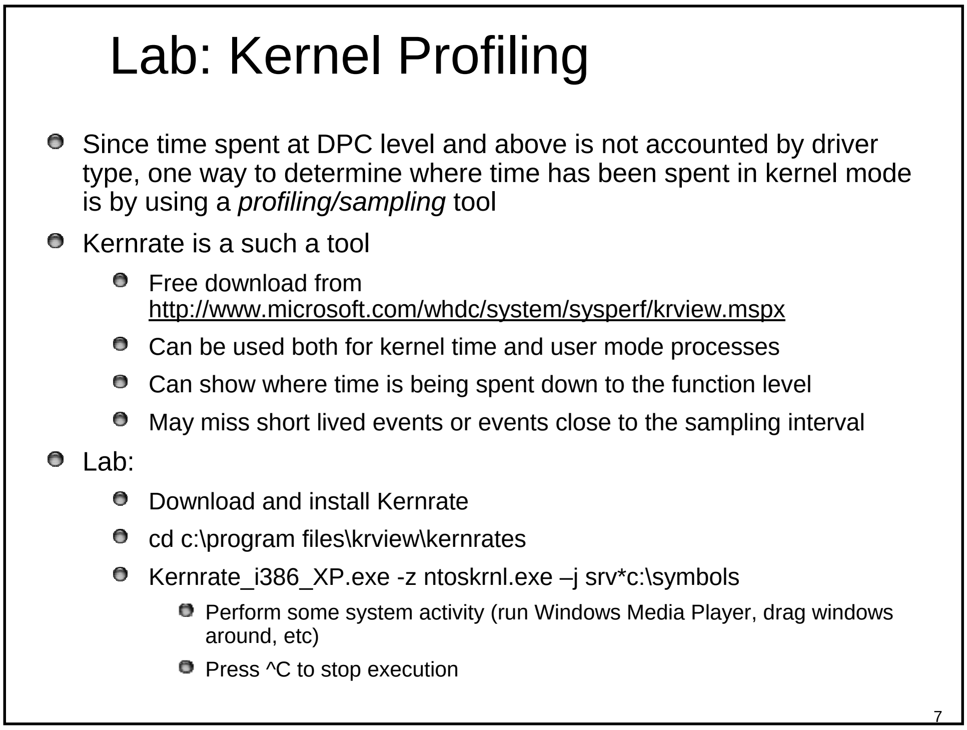
Lab: Viewing IRQL/IRQ Assignments

1. Display the interrupt vector
   - ◆ XP/2003: !idt
   - ◆ Win2000: !kdex2x86.idt
2. Dump the KINTERRUPT block for the PS/2 mouse ISR to get the IRQL

```
(Dt  nt!_KINTERRUPT xxxxxx)
```

3. With Device Manager, go to the mouse device properties and click on the resources tab to see the IRQ
   - ◆ If you are on a uniprocessor system, the IRQ should be the 27-IRQL

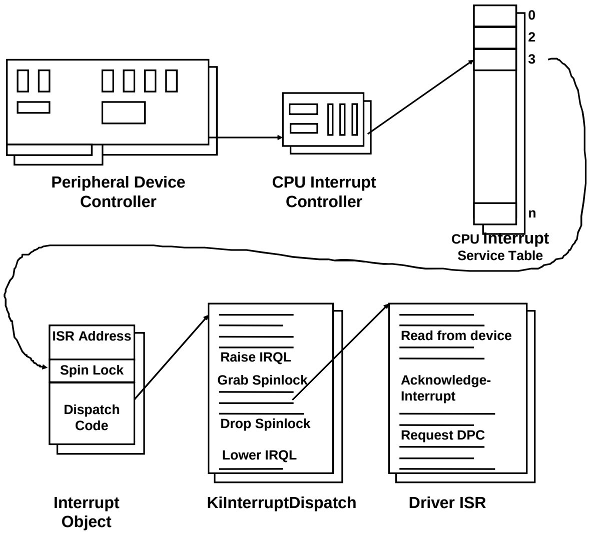
- Note: IRQL is raised when breaking in with debugger or on a crash
  - ◆ !pcr displays this changed IRQL
  - ◆ !irql displays previous IRQL (Server 2003 & later)



Lab: Kernel Profiling

- Since time spent at DPC level and above is not accounted by driver type, one way to determine where time has been spent in kernel mode is by using a *profiling/sampling* tool
- Kernrate is a such a tool
  - Free download from http://www.microsoft.com/whdc/system/sysperf/krview.mspx
  - Can be used both for kernel time and user mode processes
  - Can show where time is being spent down to the function level
  - May miss short lived events or events close to the sampling interval
- Lab:
  - Download and install Kernrate
  - cd c:\program files\krview\kernrates
  - Kernrate_i386_XP.exe -z ntoskrnl.exe –j srv*c:\symbols
    - Perform some system activity (run Windows Media Player, drag windows around, etc)
    - Press ^C to stop execution

Flow of Interrupts



Lab: ISR/DPC Tracing

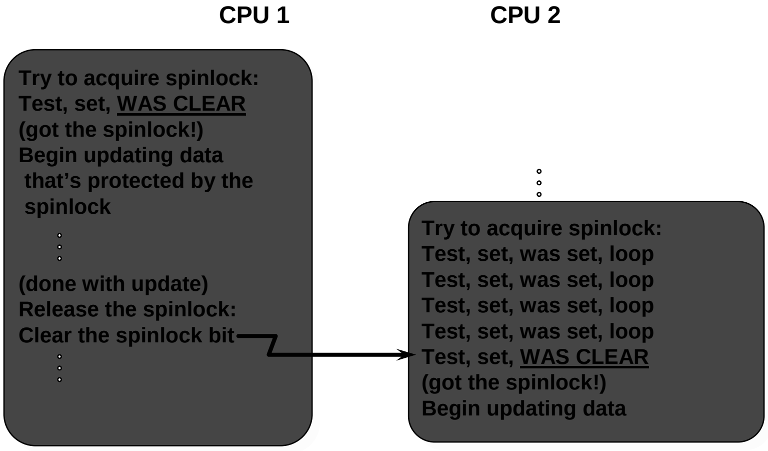
- XP SP2 and Server 2003 SP1 and later support tracing ISRs and DPCs

1. Start capturing events (tracelog.exe is in Support Tools):
   tracelog -start -f kernel.etl -b 64 -UsePerfCounter -eflag 8 0x307 0x4084 0 0 0 0 0 0
2. Stop capturing events:

   tracelog -stop
3. Generate reports (tracerpt.exe is part of Windows):

   tracerpt kernel.etl -df –report -o
4. Review workload.txt to determine where ISR/DPC time spent
5. Open "dumpfile.csv" & search for lines with "DPC" or "ISR" in the second value. In kernel debugger, do an “ln” on 8th argument (start address)

Spinlocks in Action



CPU 1

CPU 2

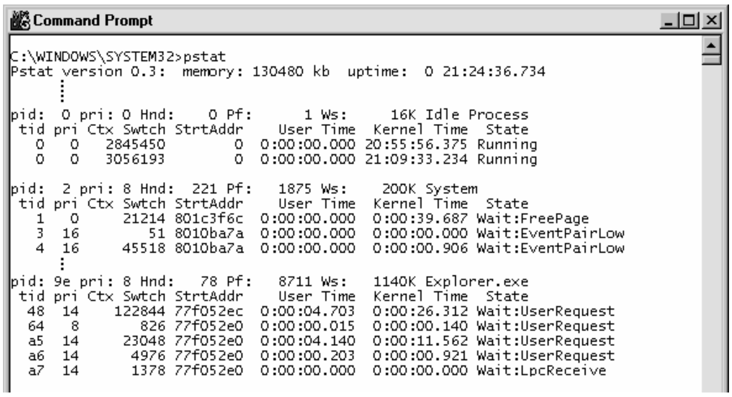
Try to acquire spinlock:
Test, set, WAS CLEAR
(got the spinlock!)
Begin updating data that's protected by the spinlock

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

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

Looking at Waiting Threads

- For waiting threads, user-mode utilities only display the wait reason
- Example: pstat



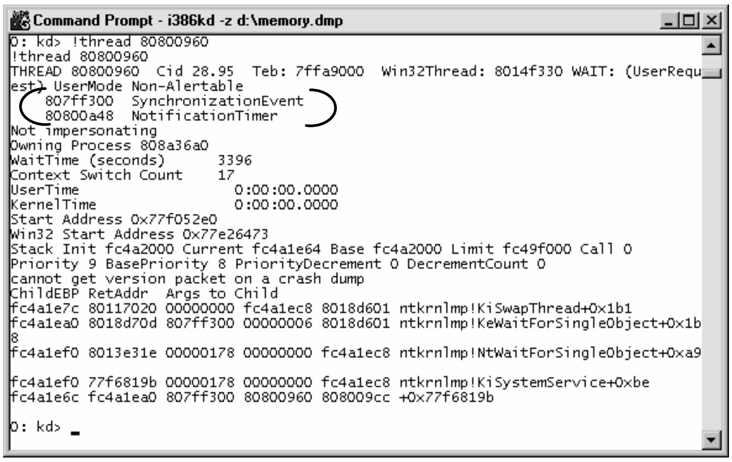
```
C:\WINDOWS\SYSTEM32>pstat
Pstat version 0.3:  memory: 130480 kb  uptime:  0 21:24:36.734
      ⋮
pid:  0 pri: 0 Hnd:    0 Pf:      1 Ws:     16K Idle Process
 tid pri Ctx Swtch StrtAddr    User Time  Kernel Time  State
   0   0   2845450        0  0:00:00.000 20:55:56.375 Running
   0   0   3056193        0  0:00:00.000 21:09:33.234 Running

pid:  2 pri: 8 Hnd:  221 Pf:   1875 Ws:    200K System
 tid pri Ctx Swtch StrtAddr    User Time  Kernel Time  State
   1   0     21214 801c3f6c  0:00:00.000  0:00:39.687 Wait:FreePage
   3  16        51 8010ba7a  0:00:00.000  0:00:00.000 Wait:EventPairLow
   4  16     45518 8010ba7a  0:00:00.000  0:00:00.906 Wait:EventPairLow
      ⋮
pid: 9e pri: 8 Hnd:   78 Pf:   8711 Ws:   1140K Explorer.exe
 tid pri Ctx Swtch StrtAddr    User Time  Kernel Time  State
  48  14    122844 77f052ec  0:00:04.703  0:00:26.312 Wait:UserRequest
  64   8       826 77f052e0  0:00:00.015  0:00:00.140 Wait:UserRequest
  a5  14     23048 77f052e0  0:00:04.140  0:00:11.562 Wait:UserRequest
  a6  14      4976 77f052e0  0:00:00.203  0:00:00.921 Wait:UserRequest
  a7  14      1378 77f052e0  0:00:00.000  0:00:00.000 Wait:LpcReceive
```

- To find out what a thread is waiting on, must use kernel debugger

Looking at Wait Queues

- !thread command to kernel debugger
  - Lists addresses of objects being waited on (if a mutex, shows owner)
  - !irpfind can search IRPs for an event object address



```
0: kd> !thread 80800960
!thread 80800960
THREAD 80800960  Cid 28.95  Teb: 7ffa9000  Win32Thread: 8014f330 WAIT: (UserRequ
est) UserMode Non-Alertable
    807ff300  SynchronizationEvent
    80800a48  NotificationTimer
Not impersonating
Owning Process 808a36a0
WaitTime (seconds)      3396
Context Switch Count    17
UserTime                  0:00:00.0000
KernelTime                0:00:00.0000
Start Address 0x77f052e0
Win32 Start Address 0x77e26473
Stack Init fc4a2000 Current fc4a1e64 Base fc4a2000 Limit fc49f000 Call 0
Priority 9 BasePriority 8 PriorityDecrement 0 DecrementCount 0
cannot get version packet on a crash dump
ChildEBP RetAddr  Args to Child
fc4a1e7c 80117020 00000000 fc4a1ec8 8018d601 ntkrnlmp!KiSwapThread+0x1b1
fc4a1ea0 8018d70d 807ff300 00000006 8018d601 ntkrnlmp!KeWaitForSingleObject+0x1b
8
fc4a1ef0 8013e31e 00000178 00000000 fc4a1ec8 ntkrnlmp!NtWaitForSingleObject+0xa9

fc4a1ef0 77f6819b 00000178 00000000 fc4a1ec8 ntkrnlmp!KiSystemService+0xbe
fc4a1e6c fc4a1ea0 807ff300 80800960 808009cc +0x77f6819b

0: kd> _
```