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

4

EXPERIMENT: Viewing the PIC and APIC

You can view the configuration of the PIC on a uniprocessor and the APIC on a multiprocessor by using the !pic and !apic kernel debugger commands, respectively. (You can't use LiveKd for this experiment because LiveKd can't access hardware.) Here's the output of the !pic command on a uniprocessor. (Note that the !pic command doesn't work if your system is using an APIC HAL.)

```
lkd>!pic
 ---IRQ Number---- 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
Physically in service: . . . . . .
Physically masked:    . . . Y . . Y Y . . Y . . Y . .
Physically requested: . . . . . .
Level Triggered:      . . . . . Y . . . Y . Y . . . .
```

Here's the output of the !apic command on a system running with the MPS HAL. The "0:" prefix for the debugger prompt indicates that commands are running on processor 0, so this is the I/O APIC for processor 0:

```
lkd>!apic
Apic @ fffe0000 ID:0 (40010) LogDesc:01000000 DestFmt:ffffffff TPR 20
TimeCnt:0bebc200 clk SpurVec:3f FaultVec:e3 error:0
Ipi Cmd: 0004001f Vec:1F FixedDel Dest=Self edg high
Timer..:000300fd Vec:FD FixedDel Dest=Self edg high masked
Linti0.:0001003f Vec:3F FixedDel Dest=Self edg high masked
Linti1.:000184ff Vec:FF NMI Dest=Self lvl high masked
TMR: 61,82,91-92,B1
IRR:
ISR:
```

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.

5

EXPERIMENT: Viewing the IRQL

If you are running the kernel debugger on Windows Server 2003, you can view a processor's IRQL with the !irql debugger command:

kd> !irql Debuggersaved IRQL for processor 0x0--0 (LOW_LEVEL)

Note that there is a field called IRQL in a data structure called the processor control region (PCR) and its extension the processor control block (PRCB), which contain information about the state of each processor in the system, such as the current IRQL, a pointer to the hardware IDT, the currently running thread, and the next thread selected to run. The kernel and the HAL use this information to perform architecture-specific and machine-specific actions. Portions of the PCR and PRCB structures are defined publicly in the Windows Device Driver Kit (DDK) header file Ntddk.h, so examine that file if you want a complete definition of these structures. You can view the contents of the PCR with the kernel debugger by using the !pcr command:

kd> !pcr

PCR Processor 0@ffdff000
NtTib.ExceptionList: f8effc68
NtTib.StackBase: f8effdf0
NtTib.StackLimit: f8efd000
NtTib.SubSystemTib: 00000000
NtTib.Version: 00000000
….

Unfortunately, Windows does not maintain the Irql field on systems that do not use lazy IRQL, so on most systems the field will always be 0.

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)

6

EXPERIMENT: Viewing the IDT

You can view the contents of the IDT, including information on what trap handlers Windows has assigned to interrupts (including exceptions and IRQs), using the !idt kernel debugger command. The !idt command with no flags shows vectors that map to addresses in modules other than Ntoskrnl.exe. The following example shows what the output of the !idt command looks like:

```
kd> !idt
Dumping IDT:

30: 806b14c0hal!HalpClockInterrupt
31: 8a39dc3ci8042prt!I8042KeyboardInterruptService(KINTERRUPT 8a39dc00)
34: 8a436dd4serial!SerialCIsrSw (KINTERRUPT8a436d98)
35: 8a44ed74NDIS!ndisMIsr(KINTERRUPT 8a44ed38)
                portcls!CInterruptSync::Release+0x10 (KINTERRUPT899c44a0)
38: 806abe80hal!HalpProfileInterrupt
39: 8a4a8abcACPI!ACPIInterruptServiceRoutine (KINTERRUPT 8a4a8a80
3b: 8a48d8c4pcmcia!PcmciaInterrupt (KINTERRUPT8a48d888)
                ohci1394!OhciIsr(KINTERRUPT8a41da18)
                VIDEOPRT!pVideoPortInterrupt(KINTERRUPT 8a1bc2c0)
                USBPORT!USBPORT_InterruptService (KINTERRUPT 8a2302b8)
                USBPORT!USBPORT_InterruptService (KINTERRUPT 8a0b8008)
                USBPORT!USBPORT_InterruptService (KINTERRUPT 8a170008)
                USBPORT!USBPORT_InterruptService (KINTERRUPT 8a258380)
                NDIS!ndisMIsr(KINTERRUPT 8a0e0430)
3c: 8a39d3eci8042prt!I8042MouseInterruptService(KINTERRUPT 8a39d3b0)
3e: 8a47264catapi!IdePortInterrupt (KINTERRUPT8a472610)
3f: 8a489b3catapi!IdePortInterrupt (KINTERRUPT8a489b00)
```

On the system used to provide the output for this experiment, the keyboard device driver's (I8042prt.sys) keyboard ISR is at interrupt number 0x3C and several devices—including the video adapter, PCMCIA bus, USB and IEEE 1394 ports, and network adapter—share interrupt 0x3B.

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

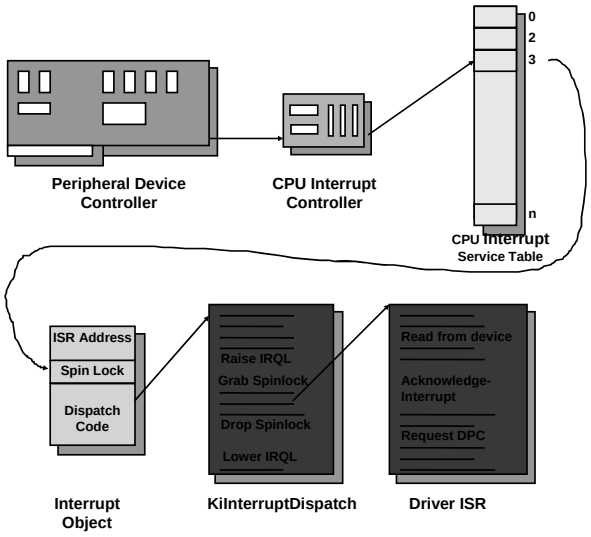
7

EXPERIMENT: Using Kernel Profiler to Profile Execution

You can use the Kernel Profiler tool to enable the system profiling timer, collect samples of the code that is executing when the timer fires, and display a summary showing the frequency distribution across image files and functions. It can be used to track CPU usage consumed by individual processes and/or time spent in kernel mode independent of processes (for example, interrupt service routines). Kernel profiling is useful when you want to obtain a breakdown of where the system is spending time. In its simplest form, Kernrate samples where time has been spent in each kernel module (for example, Ntoskrnl, drivers, and so on). For example, after installing the Krview package referred to previously, try performing the following steps:

1. Open a command prompt.
2. Type cd c:\program files\krview\kernrates.
3. Type dir. (You will see kernrate images for each platform.)
4. Run the image that matches your platform (with no arguments or switches). For example, Kernrate_i386_XP.exe is the image for Windows XP running on an x86 system.
5. While Kernrate is running, go perform some other activity on the system. For example, run Windows Media Player and play some music, run a graphics-intensive game, or perform network activity such as doing a directory of a remote network share.
6. Press Ctrl+C to stop Kernrate. This causes Kernrate to display the statistics from the sampling period.

Flow of Interrupts



8

EXPERIMENT: Examining Interrupt Internals

Using the kernel debugger, you can view details of an interrupt object, including its IRQL, ISR address, and custom interrupt dispatching code. First, execute the !idt command and locate the entry that includes a reference to I8042KeyboardInterruptService, the ISR routine for the PS2 keyboard device:

```
31: 8a39dc3ci8042prt!I8042KeyboardInterruptService(KINTERRUPT 8a39dc00)
```

To view the contents of the interrupt object associated with the interrupt, execute dt nt!_kinterrupt with the address following KINTERRUPT:

```
kd> dt nt!_kinterrupt 8a39dc00
nt!_KINTERRUPT
+0x000Type : 22
+0x002Size : 484
+0x004InterruptListEntry : _LIST_ENTRY [0x8a39dc04- 0x8a39dc04 ]
+0x00cServiceRoutine : 0xba7e74a2 i8042prt!I8042KeyboardInterruptService+0
+0x010ServiceContext : 0x8a067898
+0x014SpinLock : 0
+0x018TickCount : 0xffffffff
+0x01cActualLock : 0x8a067958 -> 0
+0x020DispatchAddress : 0x80531140 nt!KiInterruptDispatch+0
+0x024Vector : 0x31  +0x028Irql : 0x1a''
+0x029SynchronizeIrql : 0x1a''
+0x02aFloatingSave : 0''
…
```

In this example, the IRQL Windows assigned to the interrupt is 0x1a (which is 26 in decimal). Because this output is from a uniprocessor x86 system, we calculate that the IRQ is 1, because IRQLs on x86 uniprocessors are calculated by subtracting the IRQ from 27. We can verify this by opening the Device Manager, locating the PS/2 keyboard device, and viewing its resource assignments.

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)

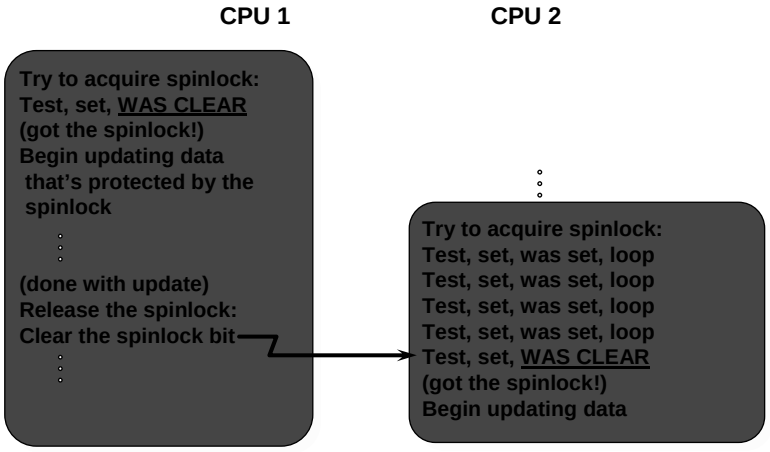
9

You can also trace the execution of specific interrupt service routines and deferred procedure calls with the built-in event tracing support in Windows XP Service Pack 2 and Windows Server 2003 Service Pack 1 and later.

1. Start capturing events by typing the following command: tracelog -start -fkernel.etl -b 64 -UsePerfCounter eflag 8 0x307 0x4084000000
2. Stop capturing events by typing: tracelog -stop tostop logging.
3. Generate reports for the event capture by typing: tracerpt kernel.etl -df -o -report This will generate two files: workload.txt and dumpfile.csv.
4. Open “workload.txt” and you will see summaries of the time spent in ISRs and DPCs by each driver type.
5. Open the file “dumpfile.csv” created in step 4; search for lines with “DPC” or “ISR” in the second value. For example, the following three lines from a dumpfile.csv generated using the above commands show a timer DPC, a DPC, and an ISR:
   PerfInfo, TimerDPC, 0xFFFFFFFF, 127383953645422825, 0, 0,127383953645421500, 0xFB03A385,0,0
   PerfInfo, DPC, 0xFFFFFFFF, 127383953645424040, 0, 0,127383953645421394, 0x804DC87D,0,0
   PerfInfo, ISR, 0xFFFFFFFF, 127383953645470903, 0, 0,127383953645468696, 0xFB48D5E0,0,0, 0

Doing an “ln” command in the kernel debugger on the start address in each event record (the eighth value on each line) shows the name of the function that executed the DPC or ISR:

Spinlocks in Action



10

EXPERIMENT: Viewing Global Queued Spinlocks

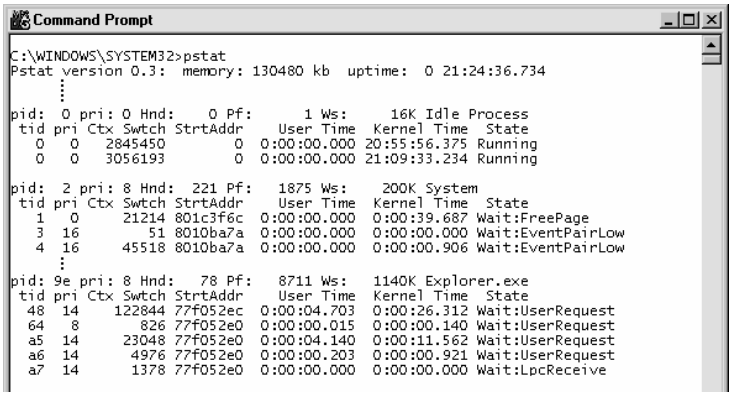
You can view the state of the global queued spinlocks (the ones pointed to by the queued spinlock array in each processor's PCR) by using the !qlock kernel debugger command. This command is meaningful only on a multiprocessor system because uniprocessor HALs don't implement spinlocks.

In the following example, taken from a Windows 2000 system, the dispatcher database queued spinlock is held by processor 1, and the other queued spinlocks are not acquired. (The dispatcher database is described in Book Chapter 6.)

```
kd> !qlocks  Key: O = Owner,1-n = Waitorder, blank = notowned/waiting, C = Corrupt
                  Processor Number
   LockName 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
KE-DispatcherO
KE-ContextSwap
MM-PFN
MM-SystemSpace
CC-Vacb
CC- Master
```

Looking at Waiting Threads

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



- To find out <u>what</u> a thread is waiting on, must use kernel debugger

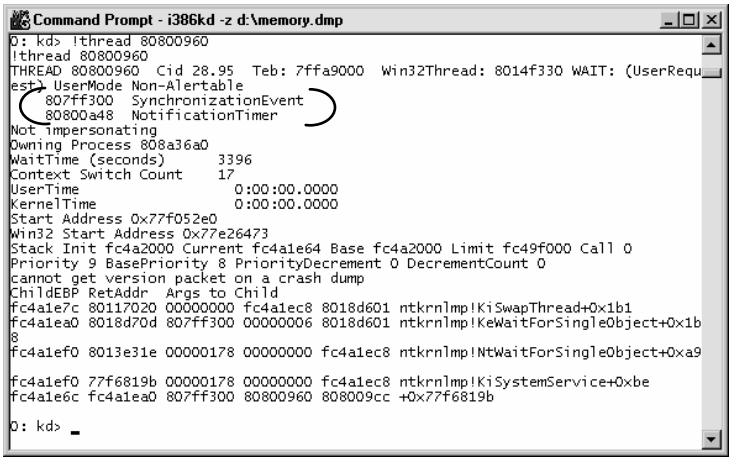
EXPERIMENT: Looking at Wait Queues

Although many process viewer utilities indicate whether a thread is in a wait state (and if so, they also indicate what kind of wait), you can see the list of objects a thread is waiting for only with the kernel debugger !thread command. For example, the following excerpt from the output of a !process command shows that the thread is waiting for an event object:

```
kd> !process
THREAD 8a12a328 Cid 0bb8.0d50 Teb: 7ffdd000 Win32 Thread: e7c9aeb0 WAIT:
(WrUserRequest) UserMode Non-Alertable 8a21bf58
                SynchronizationEvent
```

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



You can use the dt command to interpret the dispatcher header of the object like this:

```
kd>dtnt!_dispatcher_header8a21bf58
nt!_DISPATCHER_HEADER +0x000
Type :0x1 '' +0x001
Absolute :0'' +0x002
Size :0x4 '' +0x003
Inserted :0'' +0x004
SignalState :0 +0x008
WaitListHead :_LIST_ENTRY [0x8a12a398-0x8a12a398]
```

From this, we can ascertain that no other threads are waiting for this event object because the wait list head forward and backward pointers point to the same location (a single wait block). Dumping the wait block (at address 0x8a12a398) yields the following:

```
kd>dtnt!_kwait_block 0x8a12a398
nt!_KWAIT_BLOCK +0x000
WaitListEntry :_LIST_ENTRY [0x8a21bf60-0x8a21bf60] +0x008
Thread :0x8a12a328 +0x00c
Object :0x8a21bf58 +0x010
NextWaitBlock :0x8a12a398 +0x014
WaitKey :0 +0x016
WaitType :1
```

If the wait list had more than one entry, you could execute the same command on the second pointer value in the WaitListEntry field of each wait block (by executing !thread on the thread pointer in the wait block) to traverse the list and see what other threads are waiting for the object.