Unit OS6: Device Management

6.4. Lab Manual

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

Roadmap for Section 6.4.

Lab experiments investigating:

- Viewing Security Processes
- Looking at the SAM
- Viewing Access Tokens
- Looking at Security Identifiers (SIDs)
- Viewing a Security Descriptor structure
- Investigating ordering of Access Control Entries (ACEs)
- Investigating Privileges

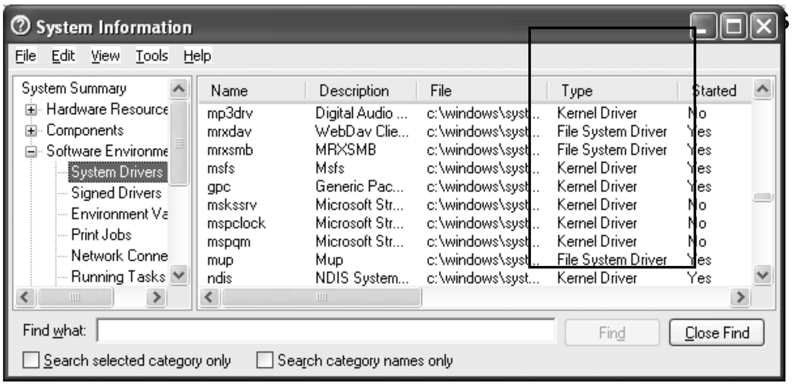
3

This Lab Manual includes experiments investigating the the I/O system mechanisms and concepts implemented inside the Windows operating system. Students are expected to carry out Labs in addition to studying the learning materials in Unit OS6.

A thorough understanding of the concepts presented in Unit OS6: Device Management is a prerequisite for these Labs.

Lab: Viewing the Installed Driver List

- View the list of System Drivers in the Software Environment section of the Windows Information utility (Msinfo32.exe)



4

Lab objective: Viewing the Loaded Driver List

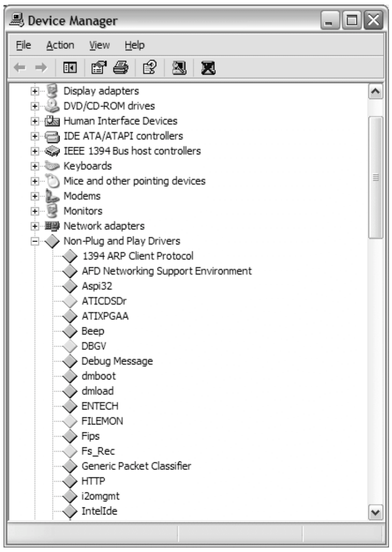
You can see a list of registered drivers on a Windows 2000 system by going to the Drivers section of the Computer Management Microsoft Management Console (MMC) snapin or by right-clicking the My Computer icon on the desktop and selecting Manage from the context menu. (The Computer Management snap-in is in the Programs/ Administrative Tools folder of the Start menu.) Navigate to the Drivers section within Computer Management by expanding System Tools, System Information, Software Environment and selecting Drivers.

In Windows XP and Windows Server 2003, you can obtain the identical information as reported by the Windows 2000 Computer Management MMC snap-in by executing the Msinfo32.exe utility from the Run dialog box of the Start menu. Select the System Drivers entry under Software Environment to see the list of drivers configured on the system. Those that are loaded have the text “Yes” in the Started column.

You can also view the list of loaded kernel-mode drivers with Process Explorer from www.sysinternals.com. Run Process Explorer, select the System process, and select DLLs from the Lower Pane menu entry in the View menu. Process Explorer lists the loaded drivers, their names, version information including company and description, and load address (assuming you have configured Process Explorer to display the corresponding columns).

Lab: Viewing Installed Drivers

- Open a command prompt and type “set devmgr_show_nonpresent _devices=1”
- Then enter “devmgmt.msc”
- Select “show hidden devices” in the view menu



5

Lab Objective: Viewing Installed Drivers

This lab presents the installed drivers from the viewpoint of the Plug and Play database. The first set of devices are plug and play devices. The non-plug and play devices are listed afterwards. Setting the environment variable devmgr_show_nonpresent_devices to 1 causes all devices that have ever been installed on the system to be shown (vs just devices that are currently present).

Lab: Viewing Loaded Drivers

- List the loaded drivers with Drivers.exe from the Resource Kit
- List the loaded drivers “lm kv” in the kernel debugger

6

Lab objective: Viewing the Loaded Driver List

If you’re looking at a crash dump (or live system) with the kernel debugger, you can get a similar display with the kernel debugger lm kv command:

```
kd>lmkv
start        end          module name
804d4000     806aa280     nt (pdbsymbols) c:\Symbols\ntoskrnl.pdb\
FB1EDACE71FB4812A5D5132819D72E523\ntoskrnl.pdb
  Loaded symbol image file: ntoskrnl.exe
  Image path: ntoskrnl.exe
  Timestamp: Thu Apr 24 10:57:43 2003 (3EA80977) Checksum: 001E311B
  ImageSize: 001D6280
  File version: 5.1.2600.1151
  Product version: 5.1.2600.1151
  File flags: 0(Mask 3F)
  File OS: 40004 NT Windows
  File type: 1.0 App
  File date: 00000000.00000000
  Translations: 0409.04b0
  CompanyName: MicrosoftCorporation
  ProductName: Microsoft" Windows«Operating System
  InternalName: ntoskrnl.exe
  OriginalFilename: ntoskrnl.exe
  ProductVersion: 5.1.2600.1151
  FileVersion: 5.1.2600.1151 (xpsp2.030422-1633)
  FileDescription: NTKernel& System
  LegalCopyright: Microsoft Corporation. All rights reserved.
806ab000806bde80 hal (deferred)
  Imagepath:halacpi.dll
  Timestamp: Thu Aug29 03:05:022002 (3D6DD5AE) Checksum: 000203BD
  ImageSize :00012E80
  Translations: 0000.04b0 0000.04e00409.04b00409.04e0
a8b8e000a8bb4e80 kmixer (deferred)
  Imagepath:\SystemRoot\system32\drivers\kmixer.sys
  Timestamp: Thu Aug29 03:32:282002 (3D6DDC1C) Checksum: 00032574
  ImageSize :00026E80
  Translations: 0000.04b0 0000.04e00409.04b00409.04e0
```

Lab: Driver Verifier

- Enable verification for all drivers with all options
- Reboot
- Does the system still boot?
  - If not, use Last Known Good to reboot
- After 7 minutes low resource simulation will begin
- Reboot again and within 7 minutes turn off verification and reboot again!

7

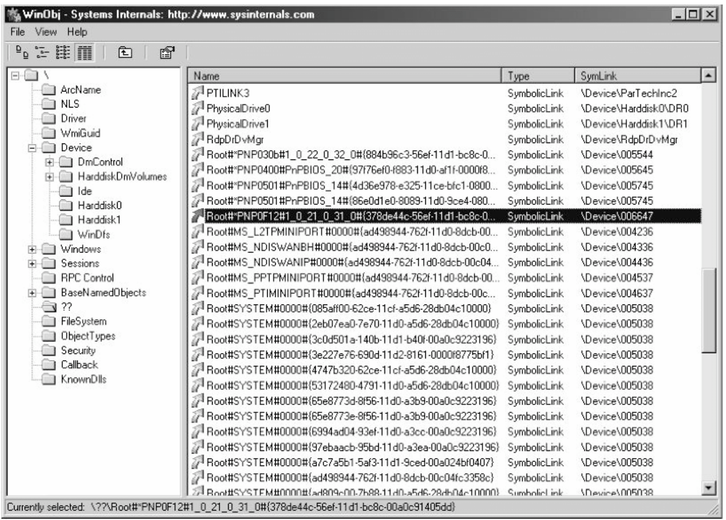
Lab objective: Driver Verifier

The Driver Verifier includes several options that check the correctness of I/O-related operations.

- I/O Verification When this option is selected, the I/O manager allocates IRPs for verified drivers from a special pool and their usage is tracked. In addition, the Verifier crashes the system when an IRP is completed that contains an invalid status and when an invalid device object is passed to the I/O manager. (In Windows 2000, this is called I/O Verification Level 1).
- I/O Verification Level 2 This option exists only in Windows 2000 and results in more rigorous testing of IRP completion operations and stack usage.
- Enhanced I/O Verification This option was introduced in Windows XP, and it monitors all IRPs to ensure that drivers mark them correctly when completing them asynchronously, that they manage device stack locations correctly, and that they delete device objects only once. In addition, the Verifier randomly stresses drivers by sending them fake power management and WMI IRPs, changing the order that devices are enumerated, and adjusting the status of PnP and power IRPs when they complete to test for drivers that return incorrect status from their dispatch routines.
- DMA Checking DMA – Direct Memory Access This is a hardware-supported mechanism that allows devices to transfer data to or from physical memory without involving the CPU. The I/O manager provides a number of functions that drivers use to schedule and control DMA operations, and this option enables checks for correct use of the functions and for the buffers that the I/O manager supplies for DMA operations.
- Disk Integrity Verification When you enable this option, which is available only in Windows Server 2003, the Verifier monitors disk read and write operations and checksums the associated data. When disk reads complete, it checks to see whether it has a previously stored checksum and crashes the system if the new and old checksum don’t match, because that would indicate corruption of the disk at the hardware level.
- SCSI Verification This was introduced in Windows XP and is not visible in the Driver Verifier option dialog box. However, it is enabled when you select a SCSI miniport driver for verification and enable at least one of the other options.

Lab: Viewing \Device Directory

- Use Winobj to view driver objects in the \Device directory



8

Lab objective: Looking at the \Device Directory

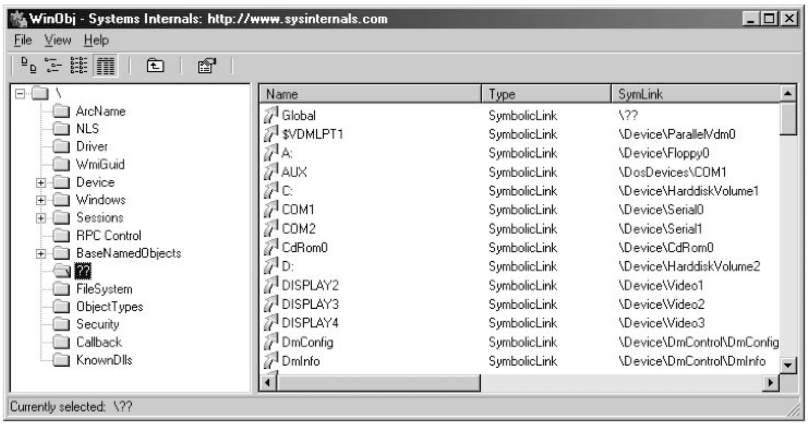
You can use the Winobj tool from www.sysinternals.com or the !object kernel debugger command to view the device names under \Device in the object manager namespace. The following screen shot shows an I/O manager–assigned symbolic link that points to a device object in \Device with an autogenerated name.

When you run the !object kernel debugger command and specify the \Device directory, you should see output similar to the following:

```
kd> !object \device

Object: e100c4a0 Type:(8a4f3178)Directory
          ObjectHeader: e100c488
          HandleCount:0 PointerCount:301
          Directory Object: e10011e8 Name:Device
          65535symbolic links snapped through this directory

          Hash      Address   Type      Name
          -----     ------    ----      ----
          00        8a437398  Device    KsecDD
                    8a4a56f0  Device    Ndis
                    8a0ed5c0  Device    ProcExp
                    8a1ddb40  Device    Beep
                    8a336d38  Device    0000008e
                    8a4ed730  Device    00000032
                    8a4ee4f0  Device    00000025
                    8a4b5030  Device    00000019
```

Lab: Device Name Mappings

- Use Winobj to view symbolic links that define the Windows device namespace



9

Lab objective: Viewing Windows Device Name to Windows Device Name Mappings

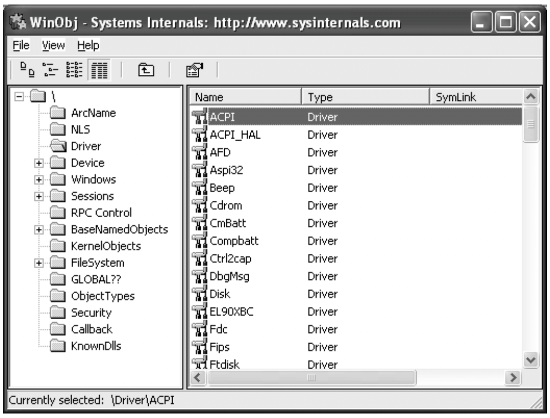
You can examine the symbolic links that define the Windows device namespace with the Winobj utility from www.sysinternals.com. Run Winobj, and click on the \?? Directory on Windows 2000 or \Global?? on Windows XP or Windows Server 2003.

Notice the symbolic links on the right. Try double-clicking on the device C:

C: is a symbolic link to the internal device named \Device\HarddiskVolume1, or the first volume on the first hard drive in the system. The COM1 entry in Winobj is a symbolic link to \Device\Serial0, and so forth. Try creating your own links with the subst command at a command prompt.

Lab: Viewing Defined Driver Objects

- Use Winobj to view driver objects in the \Drivers and \FileSystem directories
  - Drivers in the FileSystem directory are those that were marked as file system drivers in their Registry key's Type value



10

Lab objective: Displaying Driver and Device Objects

You can display driver and device objects with the kernel debugger !drvobj and !devobj commands, respectively. In the following example, the driver object for the keyboard class driver is examined, and its lone device object viewed:

```
kd> !drvobj kbdclass
Driver object (81869cb0) is for:
\Driver\Kbdclass
Driver ExtensionList:(id, addr)

Device Object list: 81869310
kd> !devobj 81869310
Device object (81869310) is for:
KeyboardClass0 \Driver\Kbdclass DriverObject 81869cb0
Current Irp a57a0e90 RefCount 0 Type 0000000b Flags 00002044
DevExt 818693c8 DevObjExt 818694b8
ExtensionFlags (0000000000)  AttachedDevice (Upper) 818691e0 \Driver\Ctrl2cap
AttachedTo (Lower) 81869500 \Driver\i8042prt
Device queue is busy -- Queueempty.
```

Notice that the !devobj command also shows you the addresses and names of any device objects that the object you're viewing is layered over (the AttachedTo line) as well as the device objects layered on top of the object specified (the AttachedDevice line)

Lab: Viewing the TCP/IP Driver Object and its Device Objects

- In the kernel debugger type “!drvobj tcpip 7”
  - Note the DriverEntry function, which the I/O Manager calls to start the driver
  - Note the I/O command dispatch function table
- Find the device objects for TCP, UDP and IP
  - Type “!devobj <address>” with the address of each of the listed device objects
- Find the TCPIP driver object in Winobj
- Find the TCP device object in Winobj

11

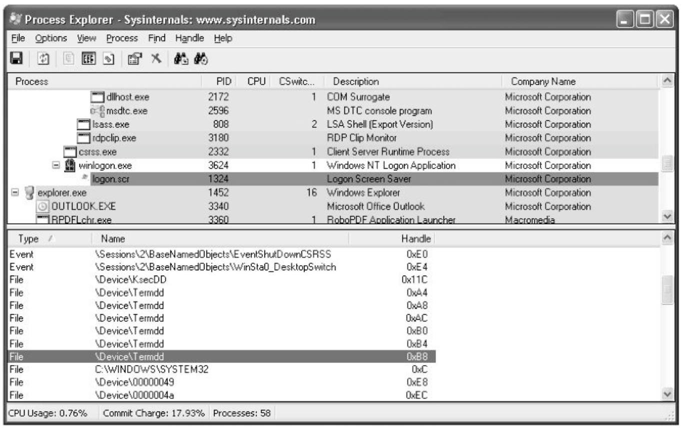
Lab objective: Looking at TCP/IP’s Device Objects

Using the kernel debugger to look at a live system, you can examine TCP/IP’s device objects. After performing the !drvobj command to see the addresses of each of the driver’s device objects, execute !devobj to view the name and other details about the device object.

```
lkd>.reload tcpip.sys

lkd>!drvobj tcpip 7

Driver object (8a01ada0) is for:
  \Driver\Tcpip
  Driver ExtensionList:(id, addr)

Device Object list:
8a0dbc88 8a0dc958 8a0dcd80 8a0eff18  8a0f32a0

lkd>!devobj 8a0dbc88
Device object (8a0dbc88) is for:
  RawIp \Driver\Tcpip DriverObject 8a01ada0
Current Irp 00000000 RefCount 3 Type 00000012 Flags 00000050
Dacl e100d19c DevExt 00000000 DevObjExt 8a0dbd40
ExtensionFlags (0000000000)
Device queue is not busy.

lkd>!devobj 8a0dc958
Device object (8a0dc958) is for:
  Udp \Driver\Tcpip DriverObject 8a01ada0
Current Irp00000000 RefCount 41 Type 00000012 Flags 00000050
Dacl e100d19c DevExt 00000000 DevObjExt 8a0dca10
ExtensionFlags (0000000000)
Device queue is not busy.

(..output shortened due to limited space..)
```

Lab: Viewing Device Handles

- Any process that has an open handle to a device will have a corresponding file object in its handle table
- Can be display with Process Explorer



12

Lab objective: Viewing Device Handles

Any process that has an open handle to a device will have a file object in its handle table corresponding to the open instance. You can view these handles with Process Explorer from www.sysinternals.com by selecting a process, checking Show Lower Pane in the View menu and Handles in the Lower Pane View submenu of the View menu. Sort by the Type column and scroll to where you see the handles that represent file objects, which are labeled as “File”.

In this example the Csrss process has handles open to file objects that represent open instances of devices with autogenerated names as well as ones that belong to the Terminal Server Driver. You can look at the specific file object in the kernel debugger by first identifying the address of the object.

The following command reports information on the highlighted handle (handle value 0xB8) in the preceding screen shot, which is in the Csrss.exe process that has a process ID of 2332 (0x91c): 0:

```
kd> !handle b8 f91c
processor number 0
Searching for Process with Cid==91c
PROCESS 86a6c020 SessionId: 0 Cid: 091c Peb: 7ffde000 ParentCid:028c
    DirBase: 1158a000 ObjectTable: e1b5d080 HandleCount: 643.
    Image: csrss.exe
New version of handle table at e2b44000 with 643 Entries in use
00B8: Object: 866ae9e8 GrantedAccess: 0012019f
Object: 866ae9e8 Type:(86fe8ad0) File
  ObjectHeader: 866ae9d0
    HandleCount:1 PointerCount:3
```

Because the object is a file object, you can get information about it with the !fileobj command: `0:kd>!fileobj 866ae9e8`

Lab: Looking at a file object

- Open the handle view in Process Explorer and look at handles of type “file”
  - Identify ones that represent real devices
- Type “dt _FILE_OBJECT” in the kernel debugger
- You can look at an actual file object with !fileobj

13

Lab objective: Viewing the File Object Data Structure

You can view the contents of the kernel-mode file object data structure with the kernel debugger’s dt command:

```
kd> dt nt!_file_object
nt!_FILE_OBJECT
            +0x000 Type : Int2B
            +0x002 Size : Int2B
            +0x004 DeviceObject : Ptr32_DEVICE_OBJECT
            +0x008 Vpb : Ptr32_VPB
            +0x00c FsContext : Ptr32Void
            +0x010 FsContext2 : Ptr32Void
            +0x014 SectionObjectPointer: Ptr32_SECTION_OBJECT_POINTERS
            +0x018 PrivateCacheMap : Ptr32Void
            +0x01c FinalStatus : Int4B
            +0x020 RelatedFileObject:Ptr32 _FILE_OBJECT
            +0x024 LockOperation : UChar
            +0x025 DeletePending : UChar
            +0x026 ReadAccess : UChar
            +0x027 WriteAccess : UChar
            +0x028 DeleteAccess : UChar
            +0x029 SharedRead : UChar
            +0x02a SharedWrite : UChar
            +0x02b SharedDelete : UChar
            +0x02c Flags : Uint4B
            +0x030 FileName : _UNICODE_STRING
            +0x038 CurrentByteOffset:_LARGE_INTEGER
            +0x040 Waiters : Uint4B
            +0x044 Busy : Uint4B
            +0x048 LastLock : Ptr32Void
            +0x04c Lock : _KEVENT
            +0x05c Event : _KEVENT
            +0x06c CompletionContext:Ptr32 _IO_COMPLETION_CONTEXT
```

Lab: Looking at Driver's Dispatch Routines

- Most drivers specify dispatch routines to handle only a subset of possible major function codes
  - create (open), read, write, device I/O control, power, Plug and Play, System (for WMI commands), and close
  - File system drivers are an example of a driver type that often fills in most or all of its dispatch entry points with functions
  - The I/O manager sets any dispatch entry points that a driver doesn't fill to point to its own IopInvalidDeviceRequest

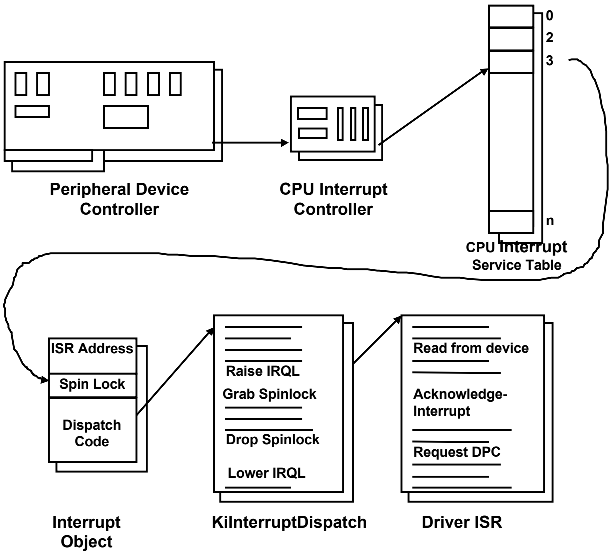
14

Lab objective: Looking at Driver Dispatch Routines

You can obtain a listing of the functions a driver has defined for its dispatch routines by entering a 7 after the driver object's name (or address) in the !drvobj kernel debugger command. The following output shows that drivers support 28 IRP types.

```
kd> !drvobj kbdclass 7
Driver object (8a238900) is for:
  \Driver\Kbdclass  Driver ExtensionList:(id, addr)
Device Object list:
  8a189030 8a2501f8

DriverEntry: f7822d22kbdclass!DriverEntry
DriverStartIo: 00000000
DriverUnload:  00000000
Dispatchroutines:
[00]IRP_MJ_CREATE                       f781fd3b kbdclass!KeyboardClassCreate
[01]IRP_MJ_CREATE_NAMED_PIPE            804eef8e nt!IopInvalidDeviceRequest
[02]IRP_MJ_CLOSE                        f781ff4c kbdclass!KeyboardClassClose
[03]IRP_MJ_READ                         f7820ba5 kbdclass!KeyboardClassRead
[04]IRP_MJ_WRITE                        804eef8e nt!IopInvalidDeviceRequest
[05]IRP_MJ_QUERY_INFORMATION            804eef8e nt!IopInvalidDeviceRequest
[06]IRP_MJ_SET_INFORMATION              804eef8e nt!IopInvalidDeviceRequest
[07]IRP_MJ_QUERY_EA                     804eef8e nt!IopInvalidDeviceRequest
[08]IRP_MJ_SET_EA                       804eef8e nt!IopInvalidDeviceRequest
[09]IRP_MJ_FLUSH_BUFFERS                f781fcbe kbdclass!KeyboardClassFlush
[0a]IRP_MJ_QUERY_VOLUME_INFORMATION     804eef8e nt!IopInvalidDeviceRequest
[0b] IRP_MJ_SET_VOLUME_INFORMATION      804eef8e nt!IopInvalidDeviceRequest
[0c] IRP_MJ_DIRECTORY_CONTROL           804eef8e nt!IopInvalidDeviceRequest
[0d] IRP_MJ_FILE_SYSTEM_CONTROL         804eef8e nt!IopInvalidDeviceRequest
[0e] IRP_MJ_DEVICE_CONTROL              f7821829 kbdclass!KeyboardClassDevice Control
[0f] IRP_MJ_INTERNAL_DEVICE_CONTROL     f7821200 kbdclass!KeyboardClassPass Through
[10] IRP_MJ_SHUTDOWN                    804eef8e nt!IopInvalidDeviceRequest
[11] IRP_MJ_LOCK_CONTROL                804eef8e nt!IopInvalidDeviceRequest
[12] IRP_MJ_CLEANUP                     f781fc84 kbdclass!KeyboardClassCleanup
[13] IRP_MJ_CREATE_MAILSLOT             804eef8e nt!IopInvalidDeviceRequest
[14] IRP_MJ_QUERY_SECURITY              804eef8e nt!IopInvalidDeviceRequest
[15] IRP_MJ_SET_SECURITY                804eef8e nt!IopInvalidDeviceRequest
```

Lab: Examine Interrupt Intervals



15

Lab objective: 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: Find an IRP

- Type “!irpfind” in the kernel debugger
- Locate an IRP aimed at the TCP/IP driver
- Type “!irp <address>” on the IRP
  - Look at the command type the active stack location (the one with the “>” symbol)
  - Correlate that against the TCP/IP driver’s dispatch table: “!drvobj \driver\tcpip 7”
  - Type “!devobj <address>” to view the device object
  - Type “!fileobj <address>” to view the file object

```
>[  c, 2]   1  1 86fb2488 861a4a40 00000000-00000000    pending
               \Driver\Tcpip
```

16

Lab objective: Examining IRPs

In this experiment, you’ll find an uncompleted IRP on the system, and you’ll determine the IRP type, the device at which it’s directed, the driver that manages the device, the thread that issued the IRP, and what process the thread belongs to. At any point in time, there are at least a few uncompleted IRPs on a system. This is because there are many devices to which applications can issue IRPs that a driver will only complete when a particular event occurs, such as data becoming available. One example is a blocking read from a network endpoint. You can see the outstanding IRPs on a system with the !irpfind kernel debugger command:

```
kd>!irpfind  unable to get large pool allocationtable - either wrong symbols
or pool tagging  is disabled

Searching NonPaged pool (82502000 :8a502000) for Tag: Irp?
  Irp     [Thread]  irpStack: (Mj,Mn) DevObj [Driver]
89695868 [00000000] Irp is complete (CurrentLocation4 >StackCount3)
0x43776f56
89712008 [8a29d7c0] irpStack: (e,9) 8a19e208 [\Driver\AFD]
89716008 [8a29d7c0] irpStack: (e,9) 8a19e208 [\Driver\AFD]
  ... 89cb3928 [8a3acbc0] irpStack: (3, 0) 8a09a030 [ \Driver\Kbdclass]
89cb3c88 [89cb1da8]irpStack: (c,2) 8a436020 [\FileSystem\Ntfs]
89cb4640 [8a165498]irpStack: (e,9) 8a19e208 [\Driver\AFD]
```

The highlighted entry in the output describes an IRP that is directed at the Kbdclass driver, so it is likely the IRP that was issued by the Windows subsystem raw input thread that reads keyboard input. Next step is examining the IRP with the !irp command:

```
kd>!irp 8a1716f0
```

Lab: Find an IRP

- Look at the issuing thread and process:

```
Irp is active with 3 stacks 1 is current Mdl = 809d45c8
       Associated Irp = 80988e68 Thread 80987da0: Irp stack trace.
```

- Open Process Explorer and go to the threads tab of the owning process
  - Look at the stack of the thread to determine what its purpose is

17

Lab objective: Looking at a Thread's Outstanding IRPs

When you use the !thread command, it prints any IRPs associated with the thread. Run the kernel debugger with live debugging, and locate the Service Control Manager process (Services.exe) in the output generated by the !process command:

```
lkd> !process 0 0
  **** NT ACTIVE PROCESS DUMP****
...
PROCESS 8a238da8 SessionId:0 Cid: 02a8 Peb:7ffdf000 ParentCid:027c
  DirBase:14fac000 ObjectTable:e1c3e008 HandleCount: 365.
  Image:SERVICES.EXE
...
```

Then dump the threads for the process by executing the !process command on the process object. You should see many threads, with most of them having IRPs reported in the IRP List area of the threads:

```
kd>!process 8a238da8
PROCESS 8a238da8 SessionId:0 Cid: 02a8 Peb:7ffdf000 ParentCid:027c
  DirBase:14fac000 ObjectTable:e1c3e008 HandleCount: 365.
  Image:SERVICES.EXE
  VadRoot 8a1be328 Vads 88 Clone 0 Private 346. Modified 37. Locked 0.
  DeviceMape10087c0
…
THREAD 8a124870 Cid 02a8.0338 Teb:7ffd8000 Win32Thread:00000000 WAIT:
(WrQueue) UserModeNon-Alertable
             8a2dc620 Unknown
             8a124960 NotificationTimer
                     IRP List:  8a2c2c00: (0006,0094) Flags:00000900 Mdl: 00000000
             8a20f770: (0006,0094) Flags:00000900 Mdl:00000000
             8a437780: (0006,0094)Flags:00000900 Mdl:00000000
```

```
Choose an IRP, and examine it with the !irp command:
lkd>!irp 8a2c2c00
Irp is active with 1stacks1is current(= 0x8a2c2c70)
No Mdl Thread 8a124870: Irpstack trace.
cmd          flg   cl     Device       File         Completion-Context
 >[ 3, 0]     0     1      8a0e5680     8a26e4b8     00000000-00000000 pending
\Driver\Npfs  Args: 00000400 00000000 00000000 00000000
```

Lab: Looking at a Device Stack

- Use the !devstack command to look at a driver stack

```
0: kd> !devstack keyboardclass0
  !DevObj   !DrvObj            !DevExt   ObjectName
  86e40530  \Driver\Ctrl2cap   86e405e8
> 86e42160  \Driver\Kbdclass   86e42218  KeyboardClass0
  86e3f020  \Driver\i8042prt   86e3f0d8
  86fc9650  \Driver\ACPI       86fccea0  0000006b
!DevNode 86fc85e8 :
  DeviceInst is "ACPI\PNP0303\4&11876118&0"
  ServiceName is "i8042prt"
```

18

Lab objective: Viewing a Device Stack

The kernel debugger command !devstack shows you the device stack of layered device objects associated with a specified device object. This example shows the device stack associated with a device object, \device\keyboardclass0, which is owned by the keyboard class driver:

```
lkd> !devstack keyboardclass0
  !DevObj          !DrvObj              !DevExt    ObjectName
   8a266d28        \Driver\Ctrl2cap     8a266de0
>  8a09a030        \Driver\Kbdclass     8a09a0e8   KeyboardClass0
   8a2672b0        \Driver\nmfilter     8a267368   0000008c
   8a09ba78        \Driver\i8042prt     8a09bb30
   8a4adce0        \Driver\ACPI         a4ab9c8    0000006b
!DevNode 8a4acee8:
  DeviceInstis "ACPI\PNP0303\4&61f3b4b&0"
  ServiceNameis "i8042prt"
```

The output highlights the entry associated with KeyboardClass0 with the “>“ prefix. The entries above that line are drivers layered above the keyboard class driver, and those below are layered beneath it. In general, IRPs flow from the top of the stack to the bottom.

Lab: See the volsnap.sys driver

- Using Winobj see what device corresponds to \Global??\C:
- In the kernel debugger look at that device object e.g. "!devstack \device\harddiskvolume1"
  - Note the volsnap.sys device object attached above the volume device

19

Lab objective: Viewing Windows Device Name to Windows Device Name Mappings

You can examine the symbolic links that define the Windows device namespace with the Winobj utility from www.sysinternals.com. Run Winobj, and click on the \?? Directory on Windows 2000 or \Global?? on Windows XP or Windows Server 2003.

Notice the symbolic links on the right. Try double-clicking on the device C:

C: is a symbolic link to the internal device named \Device\HarddiskVolume1, or the first volume on the first hard drive in the system. The COM1 entry in Winobj is a symbolic link to \Device\Serial0, and so forth. Try creating your own links with the subst command at a command prompt.

Lab: Viewing the Device Tree

- Use View->Devices by Connection in the Hardware Manager to see a system's device tree
- In the kernel debugger use "!devnode 0 7" to see the internal representation of the device tree

20

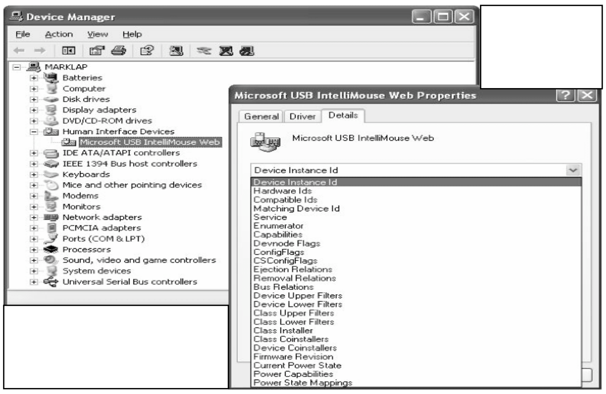
Lab objective: Dumping the Device Tree

A more detailed way to view the device tree than using Device Manager is to use the !devnode kernel debugger command. Specifying 0 1 as command options dumps the internal device tree devnode structures, indenting entries to show the hierarchy:

```
lkd>!devnode 01
Dumping IopRootDeviceNode (= 0x8a4b7ee8)
DevNode 0x8a4b7ee8 for PDO0x8a4b7020
  InstancePath is "HTREE\ROOT\0"
  State =DeviceNodeStarted(0x308)
  Previous State= DeviceNodeEnumerateCompletion(0x30d)
  DevNode0x8a4b7a50 for PDO 0x8a4b7b98
    InstancePathis "Root\ACPI_HAL\0000"
    State=DeviceNodeStarted(0x308)
    PreviousState =DeviceNodeEnumerateCompletion (0x30d)
    DevNode0x8a4af448 for PDO 0x8a4eb2c8
      InstancePath is "ACPI_HAL\PNP0C08\0"
      ServiceName is "ACPI"
      State= DeviceNodeStarted (0x308)
      Previous State=DeviceNodeEnumerateCompletion(0x30d)
      DevNode 0x8a4af198 for PDO 0x8a4b1350
        InstancePathis "ACPI\GenuineIntel_-_x86_Family_6_Model_9\_0"
        ServiceNameis "gv3"
        State =DeviceNodeStarted(0x308)
        PreviousState= DeviceNodeEnumerateCompletion(0x30d)
      DevNode 0x8a4e8008 for PDO 0x8a4a8950
        InstancePathis "ACPI\ThermalZone\THM_"
        State =DeviceNodeStarted(0x308)
        PreviousState= DeviceNodeEnumerateCompletion(0x30d)
      DevNode 0x8a4e82b8 for PDO 0x8a4eb640
        InstancePathis "ACPI\ACPI0003\2&daba3ff&0"
        ServiceNameis "CmBatt"
```

Lab: Viewing Devnode Information

- Windows XP and Server 2003 Device Manager can display details tab
  - Shows devnode’s device instance ID, hardware ID, service names, filters, and power capabilities
  - Run: `set devmgr_show_details=1`
    `devmgmt.msc`



21

Lab objective: Viewing Detailed Devnode Information in Device Manager

By default, the Device Manager applet that you can access from the Hardware tab of the System control panel application doesn’t show detailed information about a device node. However, in Windows XP and Windows Server 2003 you can enable a tab called Details by creating and setting the devmgr_show_details environment variable to a value of 1. The tab allows you to view an assortment of fields including the devnode’s device instance ID, hardware ID, service name, filters, and power capabilities.

The simplest way to launch the Device Manager with the Details tab is to open a command prompt and execute the following:

C:\>set devmgr_show_details=1
C:\>devmgmt.msc

The screen shot shows the selection combo box of the Details tab expanded to reveal the types of information you can access.

Lab: View the system power policy

- Use !popolicy to see the active power policy

```
lkd> !popolicy
SYSTEM_POWER_POLICY (R.1) @ 0x80544020
  PowerButton:         None  Flags: 00000003   Event: 00000010   Query UI
  SleepButton:        Sleep  Flags: 00000003   Event: 00000000   Query UI
  LidClose:           Sleep  Flags: 00000001   Event: 00000000   Query
  Idle:               Sleep  Flags: 00000001   Event: 00000000   Query
  OverThrottled:      Sleep  Flags: c0000004   Event: 00000000   Override NoWakes Critical
  IdleTimeout:            0  IdleSensitivity:        50%
  MinSleep:              S1  MaxSleep:               S1
  LidOpenWake:           S0  FastSleep:              S1
  WinLogonFlags:          1  S4Timeout:               0
  VideoTimeout:        1200  VideoDim:               56
  SpinTimeout:            0  OptForPower:             0
  FanTolerance:         100% ForcedThrottle:        100%
  MinThrottle:           20% DyanmicThrottle:      None (0)
```

22

Lab objective: Viewing the System Power Capabilities and Policy

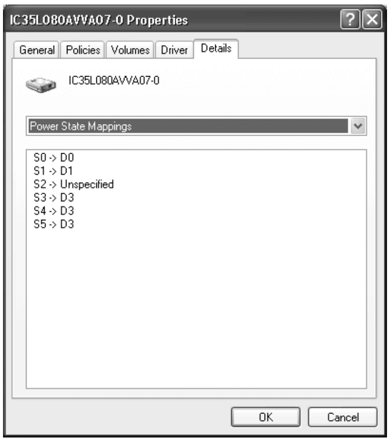
You can view a computer’s system power capabilities by using the !pocaps kernel debugger command. Here’s the output of the command when run on an ACPI-compliant laptop running Windows Professional:

```
kd>!pocaps
PopCapabilities @0x8046adc0
  MiscSupportedFeatures: PwrButtonSlpButton Lid S1 S3 S4S5
                  HiberFileFullWake
  Processor Features: ThermalThrottle (MinThrottle =03,Scale =08)
  DiskFeatures:     SpinDown
  BatteryFeatures:  BatteriesPresent
    Battery 0- Capacity: 00000000 Granularity:00000000
    Battery 1- Capacity: 00000000 Granularity:00000000
    Battery 2- Capacity: 00000000 Granularity:00000000
  WakeCaps
    AcOnLineWake: Sx
    Soft LidWake: Sx
    RTC Wake: S3
    Min Device Wake: Sx
    Default Wake: Sx
```

The Misc Supported Features line reports that, in addition to S0 (fully on), the system supports system power states S1, S3, S4, and S5 (it doesn’t implement S2) and has a valid hibernation file to which it can save system memory when it hibernates (state S4).

Lab: Looking at a Device's Power Mapping

- Open a command prompt and type "set devmgr_show_details=1"
- Then enter "devmgmt.msc"
- Go to the "Details" page on a device's properties page and look at "Power State Mapping"



23

Lab objective: Viewing a Driver's Power Mappings

In Windows XP and Windows Server 2003, you can see a driver's system power state to driver power state mappings with Device Manager. Open the Properties dialog box for a device, and choose the Power State Mappings entry in the drop-down list of the Details tab to see the mappings.

In Windows XP and Windows Server 2003 you can enable a tab called Details by creating and setting the devmgr_show_details environment variable to a value of 1. The tab allows you to view an assortment of fields including the devnode's device instance ID, hardware ID, service name, filters, and power capabilities.

The mappings for a disk driver show that besides fully on (D0) and fully off (D3), it supports an intermediate state, D1, for S1. This likely represents the disk spin-down power state.

Lab: Using Filemon to Trace File I/O

1. Run Filemon
2. Set filter to only include Notepad.exe
3. Run Notepad
4. Type some text
5. Save file as “test.txt”
6. Go back to Filemon
7. Stop logging
8. Set highlight to “test.txt”
9. Find line representing creation of new file
   - Hint: look for create operation

24

Lab objective: Examine File I/O with Filemon

The purpose of this lab is to examine the low level I/O activity involved in creating a file with Notepad. Filemon can be useful to check the efficiency of application file I/O.

For example, tracing the file I/O for creating a file with Notepad reveals that it first attempts to open the name as a folder, then as a file, to ensure there is no conflict. It then creates the file, then deletes the file, then checks if the file is there (twice), then re-creates the file and writes the data.

Lab: Seeing An Error's Root Cause with Filemon

- Many applications don't report access denied errors well

1. In Explorer, create a folder c:\noaccess
2. Remove all rights to the folder
3. Run Notepad & type some text
4. Run Filemon – set filter to Notepad.exe
5. In Notepad, File->Save As to c:\noaccess\test.txt
6. Look at Filemon trace and find Access Denied

25

Lab objective: Seeing an Error's Root Cause with Filemon

Applications sometimes present error messages in response to an error condition that do not reveal the root cause of the error. These error messages can be frustrating because they might lead you to spend time diagnosing or resolving problems that do not exist. If the error message is related to a file system issue, Filemon will show you what underlying errors might have occurred prior to the appearance of an error message.

In this experiment, you'll set permission on a directory and then perform a file save operation in Notepad that results in a misleading error message. Filemon's trace shows the actual error and the source of the message displayed in Notepad's error dialog box.

1.Run Filemon, and set the include filter to "notepad.exe".

2.Open Explorer, and create a directory named c:\noaccess on an NTFS volume.

3.Edit the security permissions on the directory to remove all access. This might require you to open the Advanced Security Settings dialog box and use the settings on the Permissions tab to remove inherited security.

When you apply the modified security, Explorer should warn you that no one will have access to the folder.

4.Run Notepad, and enter some text into its window. Then select the Save entry in the File menu. In the File Name field of the Save dialog box, enter c:\noaccess\test.txt

5.Notepad will display an error message.

6.The message implies that C:\Noaccess does not exist.

7.The Filemon trace shows that in fact, the folder does exist but Notepad got an Access Denied trying to open it.

The error message Notepad displays, "Path does not exist", is consistent with a file-not-found error, not an access-denied error. So it appears that Notepad first tried to open the directory, and when that failed it assumed for some reason that the name C:\Noaccess\Test.txt was the name of a directory instead of a file. When it couldn't open that directory, Notepad presented the error message, but the root cause, which Filemon reveals, is the access denied error.