Windows Kernel Internals II
Windows Driver Model

*University of Tokyo – July 2004**

Dave Probert, Ph.D.
Advanced Operating Systems Group
Windows Core Operating Systems Division
Microsoft Corporation

Windows I/O Model

Asychronous, Packet-based, Extensible

Device discovery supports plug-and-play

— volumes automatically detected and mounted

— power management support (ACPI)

Drivers attach to per device driver stacks

— Drivers can filter actions of other drivers in each stack

Integrated kernel support

— memory Manager provides DMA support

— HAL provides device access, PnP manages device resources

— Cache manager provides file-level caching via MM file-mapping

Multiple I/O completion mechanisms:

— synchronous

— update user-mode memory status

— signal events

— callbacks within initiating thread

— reaped by threads waiting on an I/O Completion Port

IO Request Packet (IRP)

IO operations encapsulated in IRPs

IO requests travel down a driver stack in an IRP

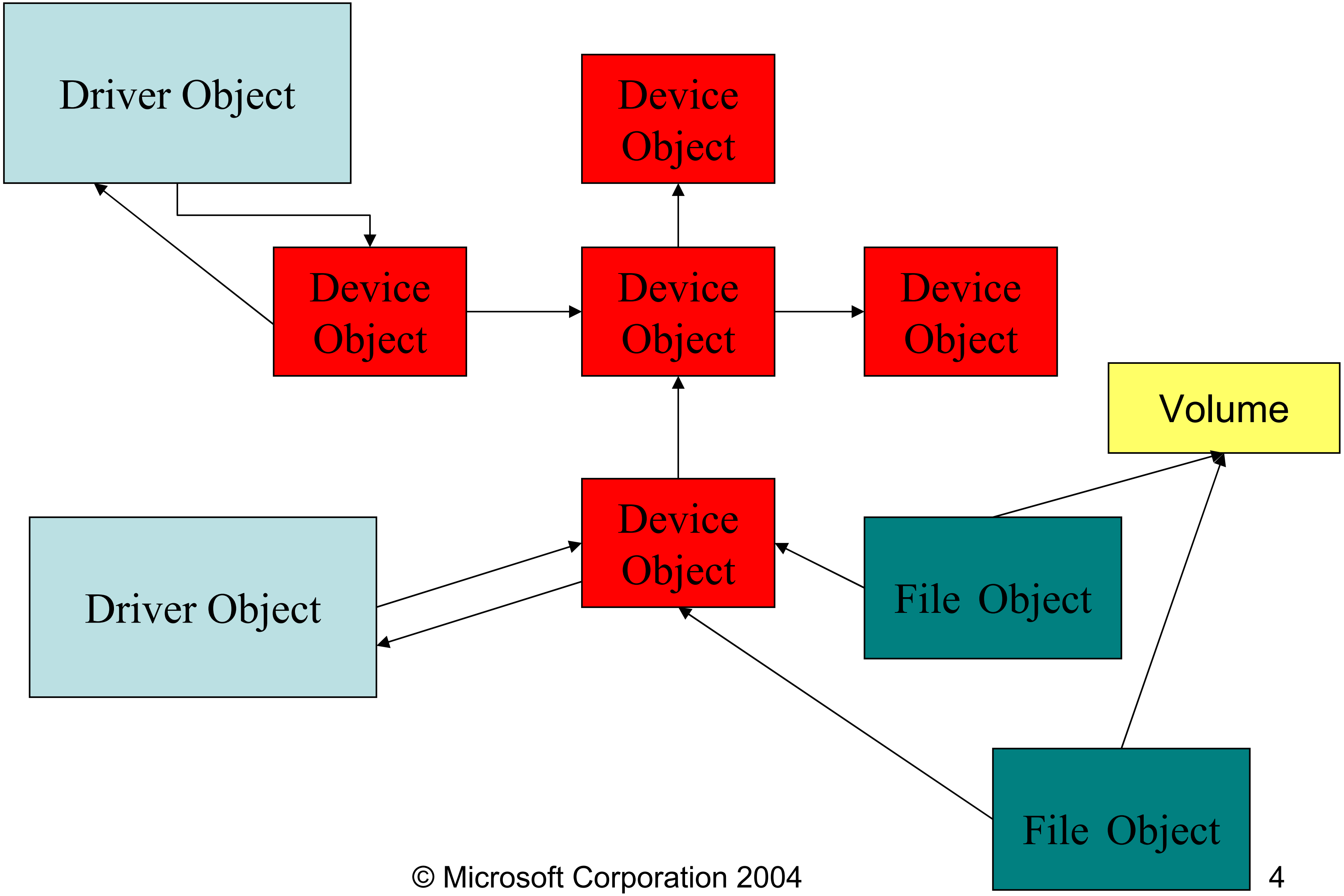
Each driver gets an IRP stack location which contains parameters for that IO request

IRP has major and minor codes to describe IO operations

Major codes include create, read, write, PNP, devioctl, cleanup and close

Irps are associated with the thread that made the IO request

Object Relationships



Layering Drivers

- **Device objects attach one on top of another using IoAttachDevice* APIs creating device stacks**
  - – IO manager sends IRP to top of the stack
  - – drivers store next lower device object in their private data structure
  - – stack tear down done using IoDetachDevice and IoDeleteDevice
- **Device objects point to driver objects**
  - – driver represent driver state, including dispatch table
- **File objects point to open files**
- **File systems are drivers which manage file objects for volumes (described by VolumeParameterBlocks)**

Loading Device Drivers

Drivers can be loaded by:

- the boot loader at boot time
- the IO manager at system initialization
- the service control manager or Plug-and-play

Driver details are obtained from the registry

Driver object is created and DriverEntry for the driver is invoked

Drivers provide dispatch routines for various IO operations. (e.g., create, read, write)

Drivers can optionally provide fast path entry points

Device Deletion and Driver Unload

- **Drivers delete devices using IoDeleteDevice**
- **Drivers are unloaded by calling NtUnloadDriver or by Plug-and-play**
- **No further opens/attaches allowed after a device is marked for deletion or unload**
- **Driver unload function is invoked when all its device objects have no handles/attaches**
- **Driver is unloaded when last reference to driver object goes away**

IRP Fields

See %DDK%¥inc¥ddk¥wnet¥ntddk.h

- flags, per-IRP pointers to buffers, an MDL, other IRPs active on thread, completion/cancel info, status, …
- union of APC control block used at completion with device queuing/communication used while active
- the stack vector with an entry for each driver in ‘stack’
  - major/minor function codes, flags and control fields
  - four words, formatted per major function code, e.g.
    - read: length, key, byteoffset
    - create: security ctx, create options, attrib, sharing, EAlen
  - deviceobject, fileobject, completion routine/context

IRP flow of control (synchronous)

- **IOMgr (e.g. IopParseDevice) creates IRP, fills in top stack location, calls IoCallDriver to pass to stack**
  - driver determined by top device object on device stack
  - driver passed the device object and IRP
- **IoCallDriver**
  - copies stack location for next driver
  - driver routine determined by major function in drvobj
- **Each driver in turn**
  - does work on IRP, if desired
  - keeps track in the device object of the next stack device
- **Calls IoCallDriver on next device**
- **Eventually bottom driver completes IO and returns on callstack**

IRP flow of control (asynch)

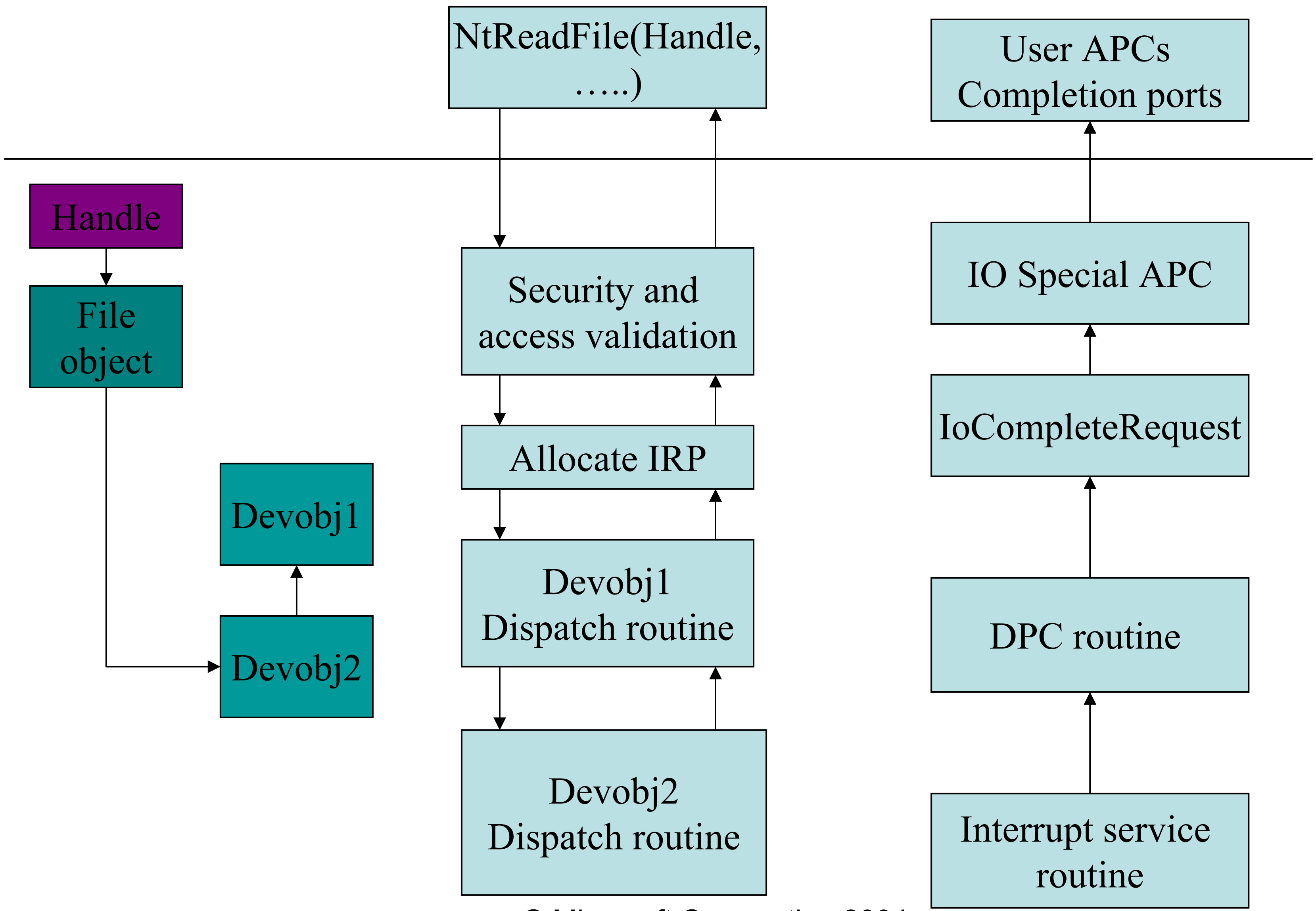
Eventually a driver decides to be asynchronous

- driver queues IRP for further processing
- driver returns STATUS_PENDING up call stack
- higher drivers may return all the way to user, or may wait for IO to complete (synchronizing the stack)

Eventually a driver decides IO is complete

- usually due to an interrupt/DPC completing IO
- each completion routine in device stack is called, possibly at DPC or in arbitrary thread context
- IRP turned into APC request delivered to original thread
- APC runs final completion, accessing process memory

Path of an Async IO request



Async IO from Win32

Applications can issue asynchronous IO requests

- to files opened with FILE_FLAG_OVERLAPPED
- passing an LPOVERLAPPED parameter to the IO API

Methods available to wait for IO completion

- Wait on the file handle
- Wait on an event handle passed in the overlapped structure
  - e.g., GetOverlappedResult(…)
- Specify a routine to be called on IO completion.
- Use completion ports

Canceling IRPs

- **IO manager provides cancellation for IRPs**
  - canceling is done on a per IRP basis
- **IO is canceled when a thread exits**
- **IO is canceled when *CancelIo* is called by the thread**
- **Drivers can cancel IRPs using *IoCancelIrp()***
- **Drivers which queue long-running IRPs must provide a cancel routine**
  - create operations must be cancellable
  - driver clears cancel routine before completing IRP
- **More recent help: Cancel Safe Queues (CSQs)**

NT IO APIs

Establish IO handles

- NtCreateFile
- NtOpenFile
- NtCreateNamedPipeFile
- NtCreateMailslotFile

IO Completion APIs

- NtCreateIoCompletion
- NtOpenIoCompletion
- NtQueryIoCompletion
- NtSetIoCompletion
- NtRemoveIoCompletion

Actual IO operations

- NtReadFile
- NtReadFileScatter
- NtWriteFile
- NtWriteFileGather
- NtCancelIoFile
- NtFlushBuffersFile

File operations

- NtLockFile
- NtUnlockFile
- NtDeleteFile

NT IO APIs - 2

Meta IO operations

NtFsControlFile
NtDeviceIoControlFile
NtQueryDirectoryFile
NtQueryAttributesFile
NtQueryFullAttributesFile
NtQueryEaFile
NtSetEaFile
NtQueryInformationFile
NtSetInformationFile
NtNotifyChangeDirectoryFile

Administrative operations

NtLoadDriver
NtUnloadDriver
NtQueryVolumeInformationFile
NtSetVolumeInformationFile
NtQueryQuotaInformationFile
NtSetQuotaInformationFile

Why is writing drivers hard?

- Driver unload routine cannot fail
- Driver image can still remain after invocation of unload routine
- Driver unload routine can race with other driver routines
- Legacy drivers should properly detach and delete device objects.
- Verifier checks for uncanceled timers and worker threads after unload

Miscellaneous Crashes

Multiple IRP completions

- Cancellation issue
- Pending flag not set correctly. If a driver returns STATUS_PENDING it should mark the IRP pending

System buffer already freed over overrun

MDL already freed.

STATUS_MORE_PROCESSING_REQUIRED should be used carefully

Drivers should watch out for IRP and MDL ownership

Spinlocks held in pageable code (verifier catches this)

Miscellaneous Crashes - 2

DRIVER_LEFT_LOCKED_PAGES bug check

- caused by lack of cancel routine
- driver locked the pages and forgot to unlock it in completion routine

Memory leaks of IO tags

- file object leaks (caused by process not closing handles)
- completion packet leaks (caused by user process not reading completion queues)
- lack of quota enforcement with pool tagging causes this
- MDL and IRP leaks (use !irpfind in debugger)

Hangs

Process stuck in kernel inside IO manager

- frequently seen as CriticalSection timeouts
- !thread shows IRP and identifies driver
- NPFS IRPs are usually hung because the consumer is another process (e.g. service hung or in debugger)
- not marking Pending flag causes hangs (verifier catches this)
- recursive locking (e.g. due to FS filter problems)
- APC deadlocks (IO issued at IRQL > PASSIVE_LEVEL) blocks IRP completion

IO Security – attack routes

How are exploits found?

- Use full crash dumps, documentation
- Probe exposed interfaces
  - IP packets, RPC interfaces, IOCTLs, etc
    - Random data, malformed data…
- Reverse engineer crashes into exploits
  - Hackers may spend months doing this!
    - Buffer overflows → Exploit
    - Double frees → Exploit
    - Synchronization bugs → Exploit

Parameter Probing

Probing ensures pointers are legal

- probe functions fail if app passes kernel addresses
  - needed as try/except won't catch writes to valid kernel addresses
- catches boundary cases, wrap-around cases
- alignment can be specified

Probing must be done for read or write as well

- probing for write handles copy-on-write cases
- missing ProbeForWrite could allow app to overwrite code in multiple processes instead of just it's own!

Missing Probe Example

```
case IOCTL_QUERY_HANDLER: {

   PVOID *EntryPoint;

   EntryPoint = irpSp→Parameters.DeviceIoControl.Type3InputBuffer;

   *EntryPoint = (PVOID)SendData;
   status = STATUS_SUCCESS;
   break;
}
```

EntryPoint not validated

- could be NULL or unmapped memory
- could be kernel address
- could be shared DLL code address
- could be misaligned

Missing Probe Example - Fixed

```
case IOCTL_QUERY_HANDLER: {

    PVOID *EntryPoint;

    EntryPoint = irpSp→Parameters.DeviceIoControl.Type3InputBuffer;

    try {
        if (Irp→RequestorMode != KernelMode) {

            ProbeForWrite( EntryPoint, sizeof(PVOID), sizeof(PVOID) );
        }

        *EntryPoint = (PVOID) SendData;
        status = STATUS_SUCCESS;
    } except(EXCEPTION_EXECUTE_HANDLER) {
        status = GetExceptionCode ();
    }
    break;
}
```

IOCTL Security

31	15	13	2
Device Number	Access	Function	Method

- Drivers encode the security requirements of IOCTLs in the 32bit code itself
- The Access mask can specify one of four rights masks:
  - openable
  - opened with FILE_READ_ACCESS
  - opened with FILE_WRITE_ACCESS
  - opened with both read and write access
- The I/O Manager won't send IOCTLs for handles with insufficient rights
- Bad IOCTLs in drivers is huge problem
  - throw garbage!

Other Common Security Issues

Validating Data That Can Change

- app can be actively modifying buffers passed to Direct and MethodNeither IOCTLs
- value validation should be done on a copy (called capturing)

```
 PortName = MmGetSystemAddressForMdl(Irp->MdlAddress);
…
//
    // Make sure the port name is properly zero
    // terminated for RtlInitUnicodeString
    //
    PortName[PortLen] = UNICODE_NULL;

    RtlInitUnicodeString( &AdapterName, PortName );
```

SuspendThread Attacks

- **An application can suspend threads running in kernel**
  - threads can be suspended indefinitely when at PASSIVE_LEVEL
- **Especially dangerous if driver grabs PASSIVE_LEVEL locks**
  - KeWaitForSingleObject(mutex, …)
- **Prevent by using KeEnterCriticalRegion and KeLeaveCriticalRegion**
  - Driver Verifier enforces this for ERESOURCE, but not other synchronization primitives

Handle Attacks

A driver might call a function that returns a handle

– ZwCreateFile, etc

By default, handle is in current process' handle table

– application could substitute it's own handle by closing a handle and opening something else
– attacker would use driver's kernel-mode access to bypass various privilege checks, etc, and use substituted handle

Pass OBJ_KERNEL_HANDLE to InitializeObjectAttributes(…)

– handle will instead be placed in the System's process table, not the applications

Memory Attacks

A driver might allocate memory in response to an IOCTL

– Attack – app calls driver until all memory is exhausted

Memory allocated on behalf of application should be done via ExAllocatePoolWithQuotaTag

Warning: Low Memory Behavior and exceptions

– ExAllocatePoolWithTag returns NULL, but ExAllocatePoolWithQuotaTag raises an exception

Class Drivers and Miniports

Drivers can be loaded as DLLs (called Class Drivers)

- allows drivers to focus on a specific flavor of a common device (called miniport drivers)
- large number of class driver/miniport driver models:
  - USB, 1394, SCSI, ATAPI, Serial, NICs
- too many class drivers using solving similar problems with slightly different approaches – more unification and simplification needed

Plug-and-Play

Basic device installation

User plugs a new device into a Bus

The Bus driver

- Notices the new device's arrival
- Enumerates the device
- Retrieves identification information for the device
  - Device and instance ID (for device node name)
  - Device capabilities (UniqueID capability indicates whether we must "unique-ify" the devnode name)
  - One or more Hardware IDs and zero or more Compatible IDs
- Passes information to Plug and Play

Plug-and-Play - 2

Windows Plug-and-play

- Searches for ID matches in the set of available INFs
- Ranks the matched ID entries in the INFs according to signature, ID match, and DriverVer date, in that order
- Selects the best ID match – identifies the INF containing that ID
- Uses the ID entry in that INF to install the driver referenced in the INF

Why is plug-and-play hard?

- **Installing drivers is privileged operation**
- **Vendors aren’t reliable about assigning IDs**
- **Devices can have multiple IDs**
  - Hardware ID – identifies a specific device
  - Compatible ID -- Used when no hardware ID match
- **ID formats are bus-specific**
  - Vendor IDs, Device IDs, Subsystem IDs, Subsystem Vendor IDs, Revision, …
- **Some devices are multi-function (combos)**
  - Drivers become artificial bus drivers

Why is plug-and-play hard? - 2

Plug-and-play uses IRP path

- IRP_MJ_PNP / IRP_MN_QUERY_ID
- Race conditions with normal IO and also power IRPs

Vendors opt for software-first installation to get right results

- Trying to defeat Pnp ranking of drivers by loading driver first
- Results in conflicts with better drivers in box or from WU vs old CD

Driver signing requirements add complexity which vendors duck

- Want only tested drivers, so Windows pops UI – which goes badly

Vendors want to add lots of user-mode software at same time

- Drivers fail to install due to user-mode configuration issues
- Compatible ID -- Used when no hardware ID match

Power Management - History

APM

- BIOS-based OS-independent Intel mechanism
  - assumed BIOS could hide details from software
  - implemented by SMM
  - not synchronized with OS, unbounded latency, not debuggable

#ifdef _PNP_POWER_

- 1st attempt at Pwr/PnP in NT
  - assumed hal/kernel could hide details from drivers

WDM - redesign of PM/PnP for NT and Win9x

ACPI - firmware interface for supporting WDM

WDF- next step in Pwr/PnP evolution

Power Management - WDM

Creates concept of “Devnode”

- PDO – Physical Device Object, represents parent bus in a device stack
- FDO – Functional Device Object, traditional device driver function
- Filters – Allow for other drivers, like ACPI, to take part in PnP/Power

Power Management split into:

- S-states, representing the entire system
  - **S0:** Working state, **S1 – S3:** Sleeping states
  - **S4:** Hibernated state, **S5:** Soft-off state
- D-states, representing single devices
  - **D0:** Working state, **D1 – D2:** Low power states, **D3:** Off state

WDM – S0 State

Devices can be in any Dx state while the machine itself is in the S0 state

- S0-D0: Device is powered on and fully active
- S0-D(1-3): Device is in a low power state, but the machine is still running. The user may not even be aware that the device is not in D0
- Devices in D(1-3) may be armed for wakeup, even though the machine is awake

WDM – S1-S3 – Sleep

- Machine appears to be off
- RAM context is maintained
- All clocks are stopped, except for RTC
- Devices may or may not have power
- Some devices may have trickle current, but not main power source
- Power in S1 >= S2 >= S3
- Differences between S1, S2 and S3 are machine specific

WDM – S4 – Hibernate

RAM context is written to disk

Machine is powered off

All devices are in the D3 state, unless they have external power sources

Machine execution resumes through NTLDR, which restores RAM context

BIOS has a chance to reprogram devices

WDM – S5 – Off

All context is lost

Machine is powered off

Resume from S5 == booting

WDM – D-states

Reasons for moving a device out of D0

- The machine is leaving S0 – time to save the device state
- The device is being ejected – time to turn off the power to it
- The device is not being used – save some power
  - Example: Ethernet PHYs consume lots of power. Moving the device to D3 when there is no cable plugged in recovers that lost power

WDM – D-states continued

Reasons for moving from D1-3 to D0

- The machine is moving from S1-4 to S0 and your device has handles open to it
- The device has received IRP_MN_START_DEVICE
- The device has been inserted and now it need to be enumerated
- Something is requesting to use the device

  <u>Example:</u> The ethernet in the laptop had no cable plugged into it. But now the user has plugged in a cable, so we need to get an IP address

Converting S IRPs To D IRPs

The WDM Power Manager sends S IRPs:

- IRP_MN_QUERY_POWER, IRP_MN_SET_POWER

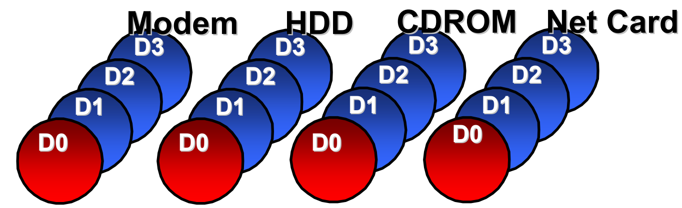
Each device stack has a "Power Policy Owner" who converts S IRPs to D IRPs

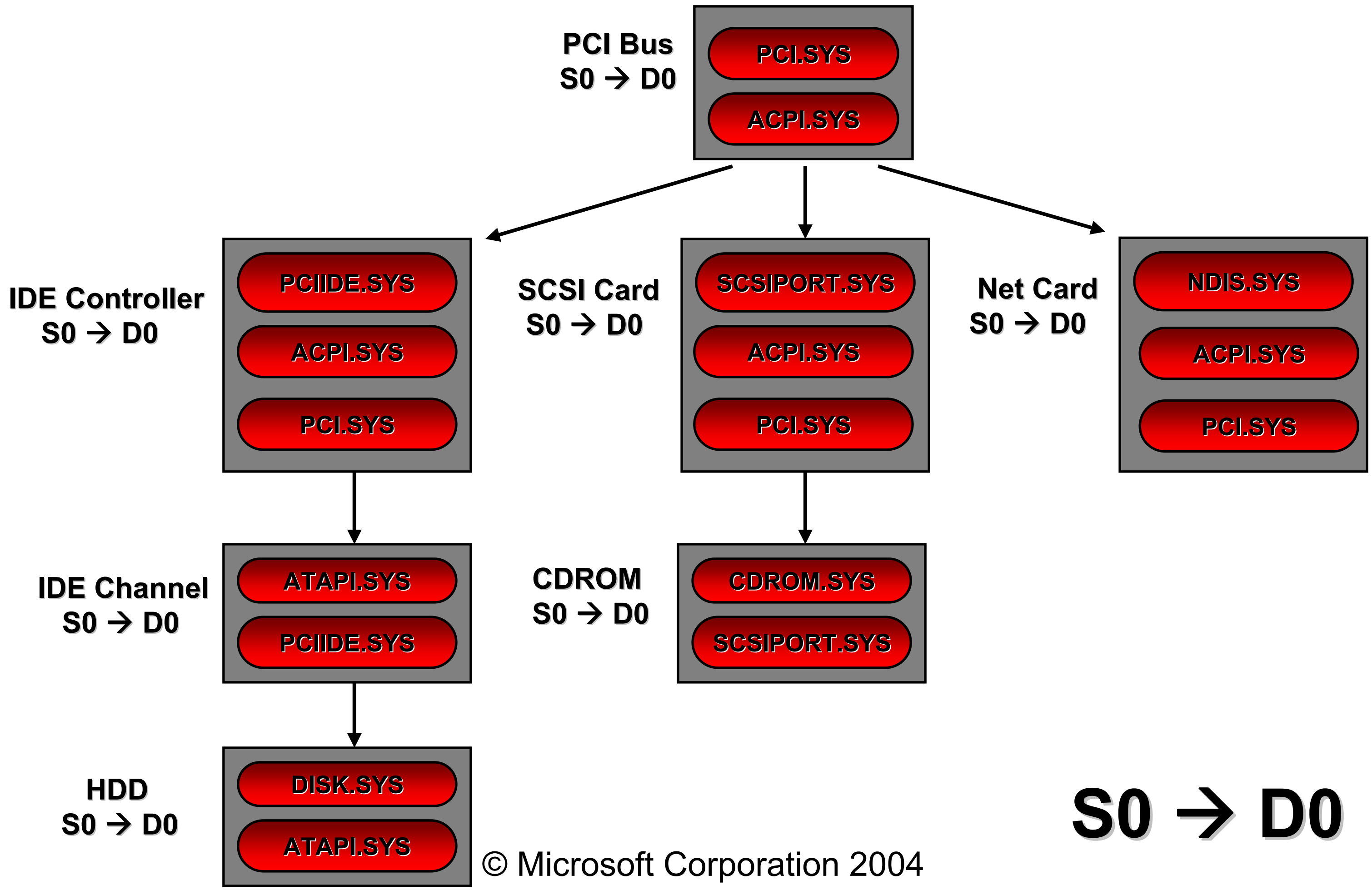
- The Power Policy Owner is typically the FDO
- The mapping comes from the S → D array stored in the IRP_MN_QUERY_CAPABILITIES structure
- Each entry calls out the lightest possible D state for a given S state
- Mappings can be rounded down (deeper)

The Power Policy Owner then uses PoRequestPowerIrp to request the appropriate D IRP

The conversion code is complicated, but most drivers can use the boilerplate code in the WDM DDK

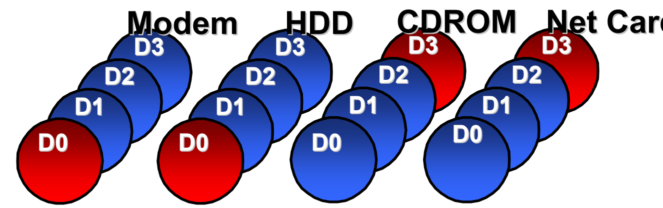
System State S0 – Working

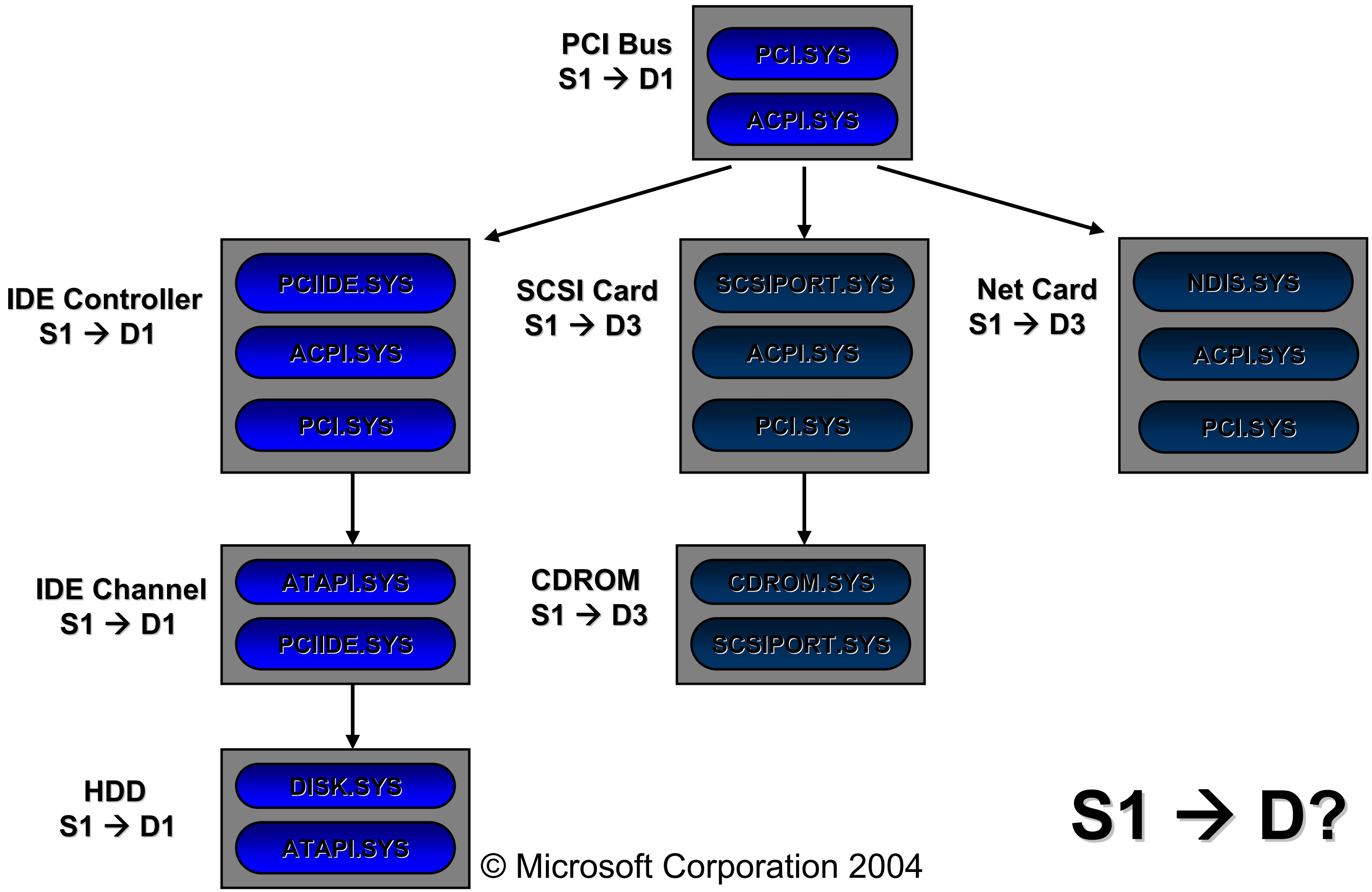




S0 → D0

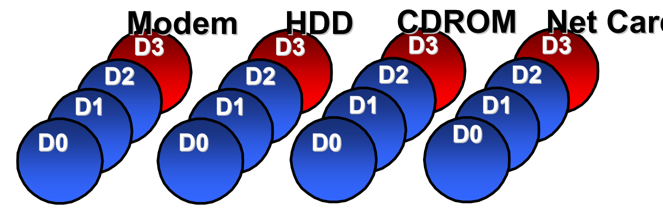
System State S1 – Standby

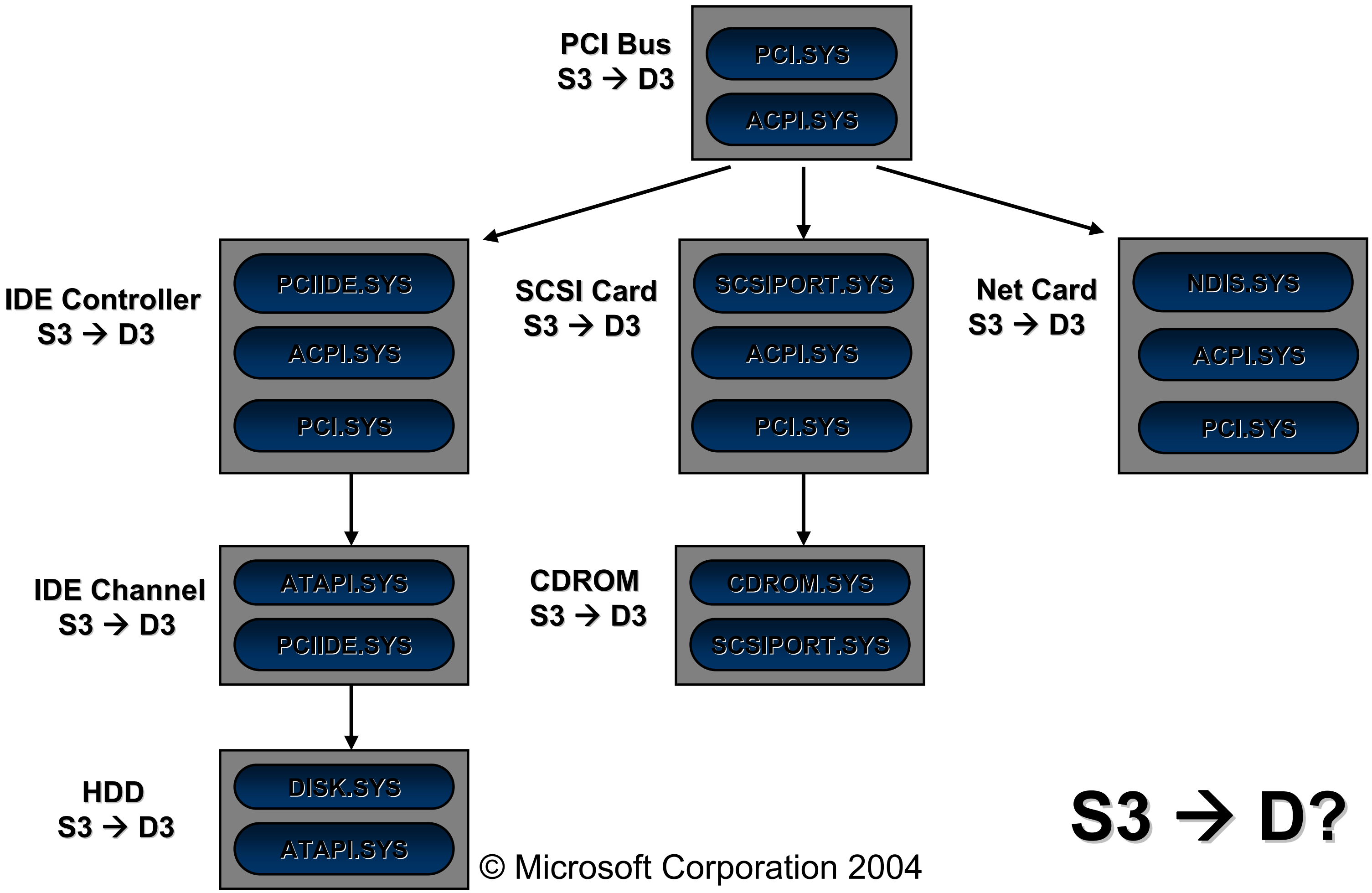




S1 → D?

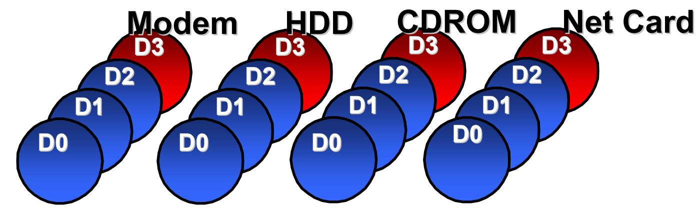
System State S3 – Standby

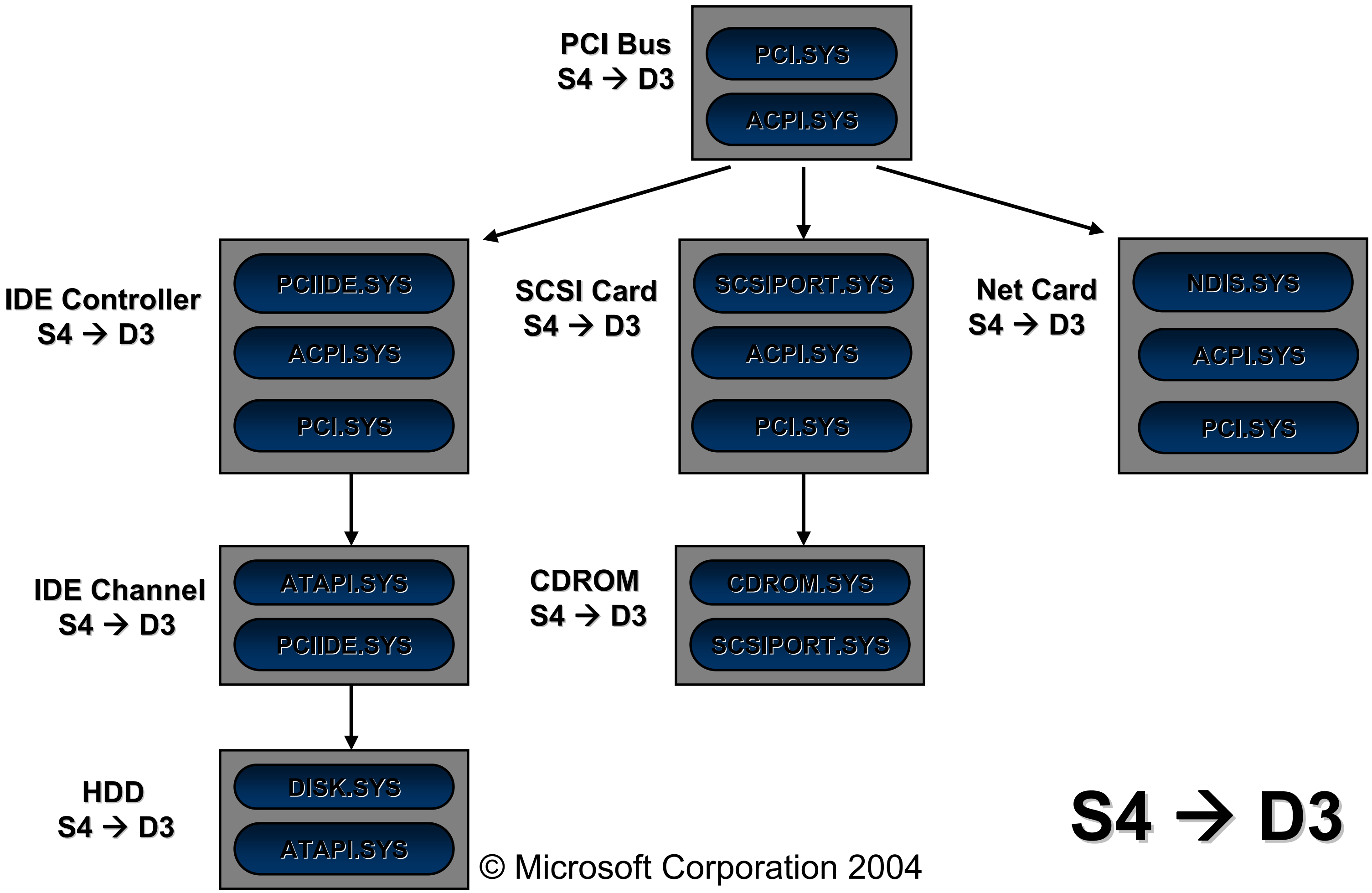




S3 → D?

System State S4 – Hibernate





S4 → D3

ACPI

- Register interface for handling power management interrupts
- Interpreted p-code language (AML)
- Human-readable specification for AML (ASL)
- Collection of firmware objects organized in namespace that describe the machine – built from AML
- Namespace structure mirrors WDM device tree
- Most devices in tree correspond to Devnodes
- Devices in tree contain child-objects that modify the properties of the device (special properties)
- Various bus specifications leave out important parts, which can be filled in with ACPI objects

ACPI Objects – Motherboard

ACPI objects can fill in machine-specific information

- If the serial port in a laptop is only exposed when connected to a dock, an ACPI object can tell us that
- If two chips are connected to the same power plane, a collection of ACPI objects can describe that
- If the CD-ROM drive can be ejected while the machine is running ACPI objects can describe that

ACPI Objects can give thermal information

- Can be used to expose temperature sensors to the OS
- Can be used to describe thermal relationships
  - Slowing the processor may also cool the CD-ROM
  - Charging the battery may overheat the processor

ACPI Objects – Motherboard - 2

ACPI Objects can abstract interfaces to batteries,

- OS need know nothing about physics or chemistry

ACPI Objects can abstract very simple devices,

- a single driver to operate with very different hardware
  - Lid Switches
  - Power Buttons
  - Fans

Why is power management hard?

System/Device states + PnP states => EXPLOSION

IRP-based communication with drivers causes races

Drivers can/will veto power state changes

- Hard hangs in drivers are common
- As is the melting laptop
- Compatible ID -- Used when no hardware ID match

Apps can/will veto power state changes

- Ditto

Flexible “power policy” results in bad decisions

Physical topology, PnP topology, power topology entertwined

Discussion