I/O System (I)

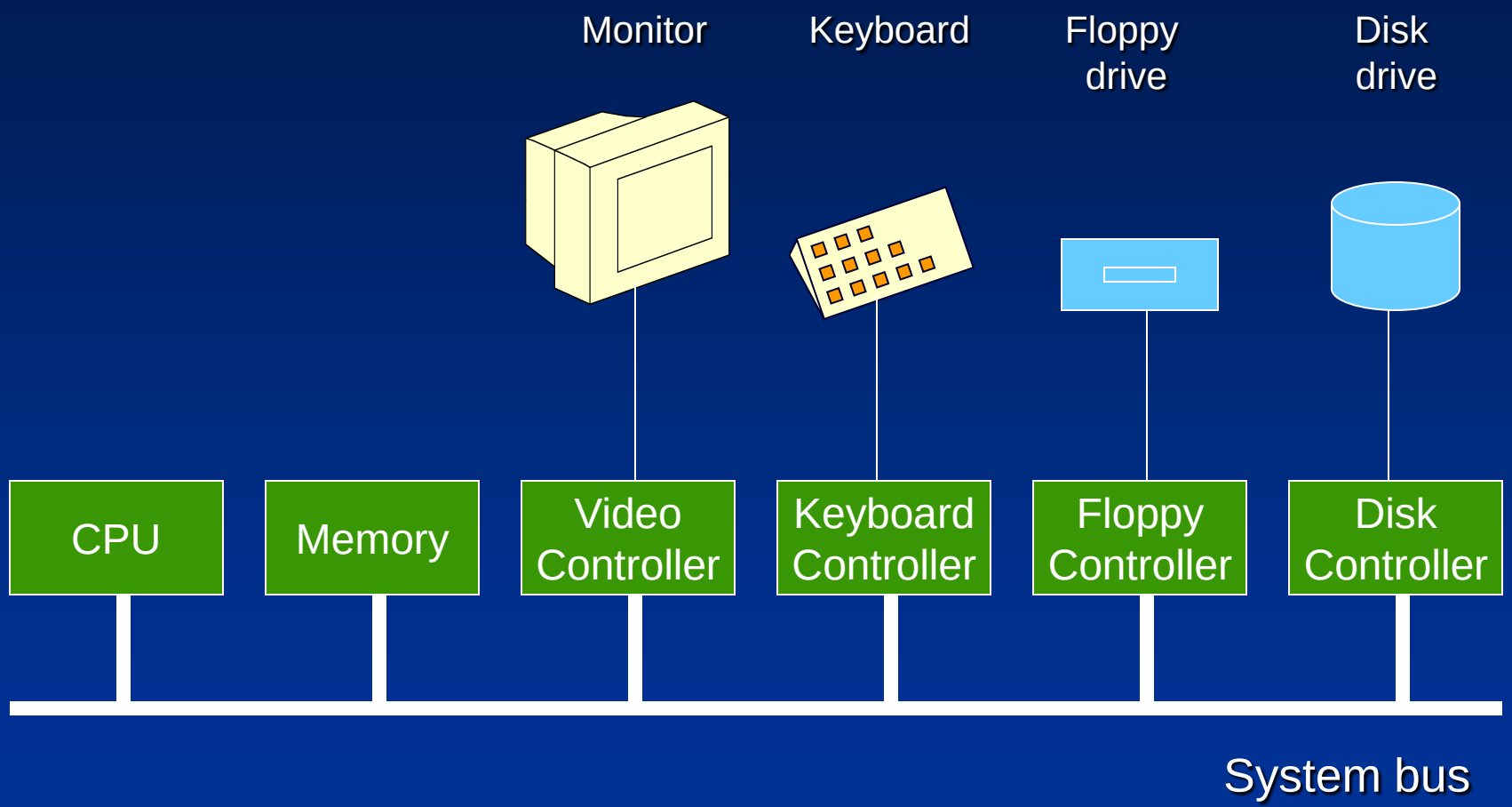
Roadmap for This Lecture

- Principles of I/O Systems
- Windows I/O Components
- Windows I/O Processing

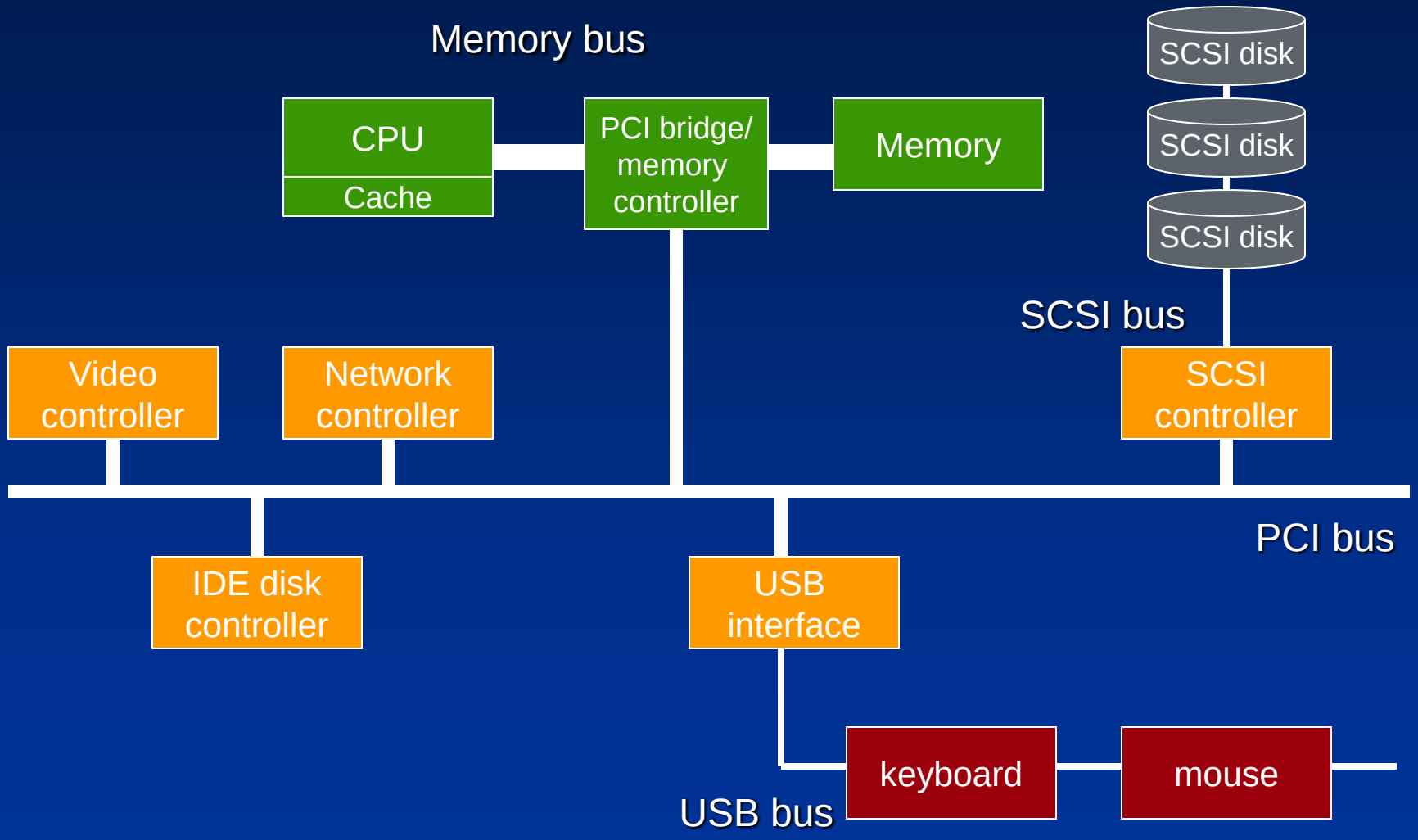
Principles of I/O Hardware

- Major components of a computer system:
  CPU, memories (primary/secondary), I/O system
- I/O devices:
  - Block devices – store information in fixed-sized blocks;
    typical sizes: 128-1024 bytes
  - Character devices – delivers/accepts stream of characters
- Device controllers:
  - Connects physical device to system bus (Minicomputers, PCs)
  - Mainframes use a more complex model:
    Multiple buses and specialized I/O computers (I/O channels)
- Communication:
  - Memory-mapped I/O, controller registers
  - Direct Memory Access - DMA

I/O Hardware - Single Bus



I/O Hardware - Multiple Buses



Diversity among I/O Devices

The I/O subsystem has to consider device characteristics:

- Data rate:
  - may vary by several orders of magnitude
- Complexity of control:
  - exclusive vs. shared devices
- Unit of transfer:
  - stream of bytes vs. block-I/O
- Data representations:
  - character encoding, error codes, parity conventions
- Error conditions:
  - consequences, range of responses
- Applications:
  - impact on resource scheduling, buffering schemes

Principles of I/O Software

- Layered organization
- Device independence

- Error handling
  - Error should be handled as close to the hardware as possible
  - Transparent error recovery at low level
- Synchronous vs. Asynchronous transfers
  - Most physical I/O is asynchronous
  - Kernel may provide synchronous I/O system calls
- Sharable vs. dedicated devices
  - Disk vs. printer

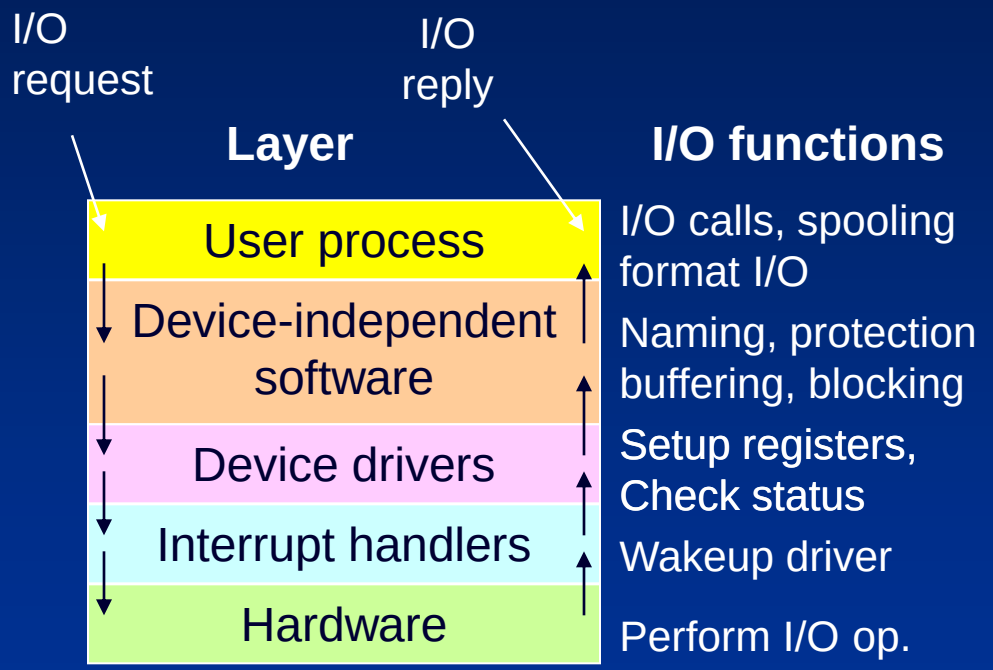
Structuring of I/O software

1. User-level software
2. Device-independent OS software
3. Device drivers
4. Interrupt handlers

Layers of the I/O System

User-Space I/O Software

- System call libraries (read, write,...)
- Spooling
  - Managing dedicated I/O devices in a multiprogramming system
  - Daemon process, spooling directory
  - lpd – line printer daemon, sendmail – simple mail transfer protocol



Application I/O Interfaces

The OS system call interface distinguishes device classes:

- Character-stream or block
- Sequential or random-access
- Synchronous or asynchronous
- Sharable or dedicated
- Speed of operation
- Read/write, read only, write only

Device-independent I/O Software

Functions of device-independent I/O software:

- Uniform interfacing for the device drivers
- Device naming
- Device protection
- Providing a device-independent block size
- Buffering
- Storage allocation on block devices
- Allocating and releasing dedicated devices
- Error reporting

Device Driver

- Contains all device-dependent code
- Handles one device
- Translates abstract requests into device commands
  - Writes controller registers
  - Accesses mapped memory
  - Queues requests
- Driver may block after issuing a request:
  - Interrupt will un-block driver (returning status information)

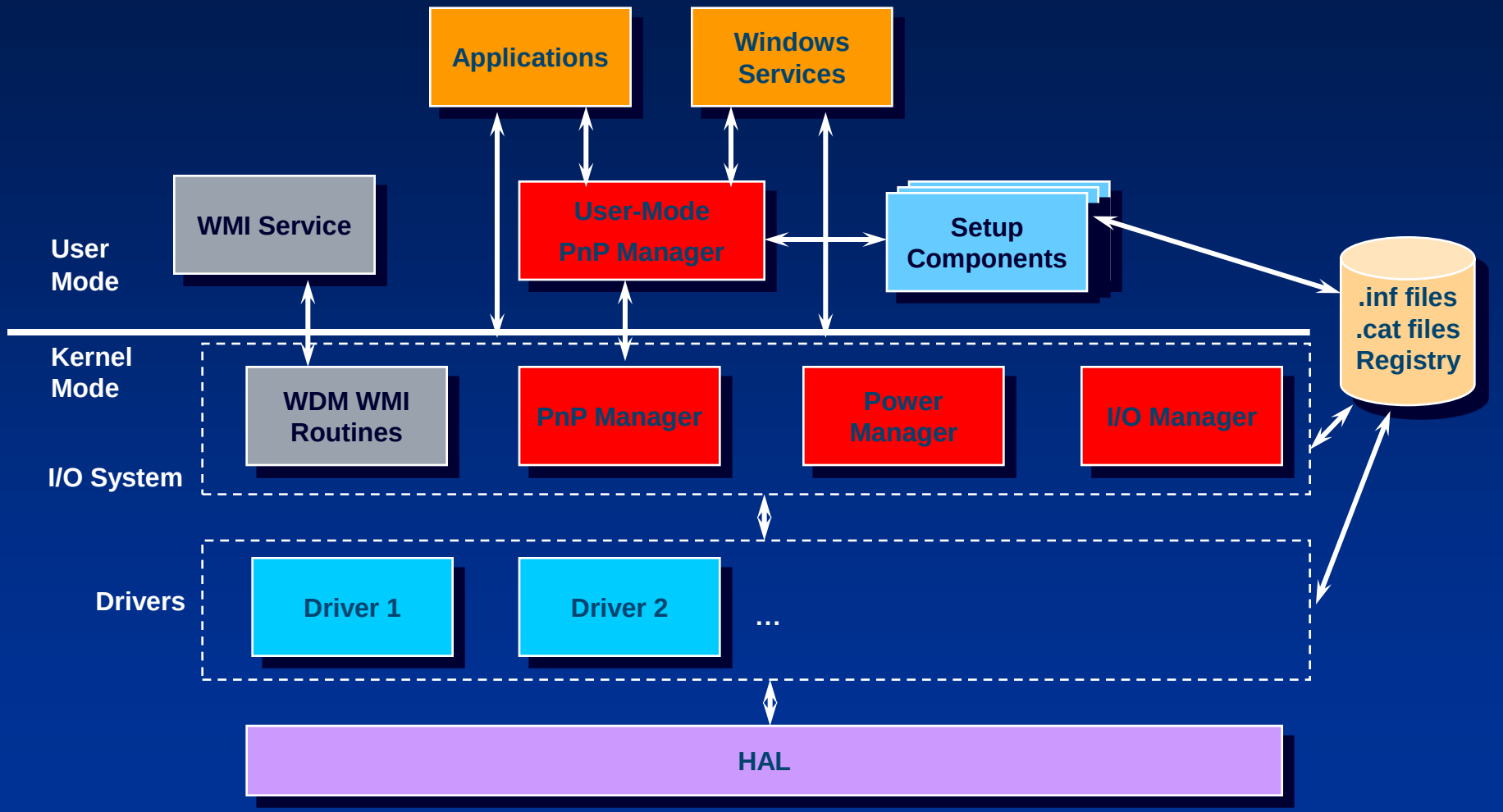
Interrupt Handlers

- Should be hidden by the operating system
- Every thread starting an I/O operation should block until I/O has completed and interrupt occurs
- Interrupt handler transfers data from device (controller) and un-blocks process

Windows I/O System Design Goals

- Fast I/O processing on single / multiprocessor systems
- Protection for shareable resources
  - Using Windows security mechanisms
- Meet requirements dictated by different subsystems
- Provide common services for device drivers
  - Ease device driver development
  - Allow drivers to be written in high-level language
- Dynamic addition/removal of device drivers
- Support multiple file systems (FAT, CDFS, UDF, NTFS)
- Provide mapped file I/O capabilities
- Windows Management Instrumentation support and diagnosability
  - Drivers can be managed through WMI applications and scripts.

I/O System Components



I/O System Components

- The I/O manager
  - Connects applications and system components to virtual, logical, and physical devices
  - Windows APIs: ReadFile, WriteFile, CreateFile, CloseFile, DeviceIoControl
  - Defines the infrastructure that supports device drivers
- Device driver
  - Provides an I/O interface for a particular type of device
  - Device drivers receive commands routed to them by the I/O manager that are directed at devices they manage, and they inform the I/O manager when those commands complete
  - Device drivers often use the I/O manager to forward I/O commands to other device drivers that share in the implementation of a device's interface or control.
  - Several types:
    - “ordinary”, file system, network, bus drivers, etc.
    - More information in I/O subsystem section

I/O System Components

- The PnP manager works closely with the I/O manager and bus driver
  - To guide the allocation of hardware resources
  - To detect and respond to the arrival and removal of hardware devices.
  - User-mode PnP manager: called when installing a PnP device for the first time
- The power manager
  - To guide the system and individual device drivers, through power-state transitions.
- Windows Driver Model (WDM) WMI support routines
  - An intermediary to communicate with the WMI service in user mode

I/O Manager

- Framework for delivery of *I/O request packets* (IRPs)
- IRPs control processing of all I/O operations
  (exception: fast I/O does not use IRPs)
- I/O manager:
  - creates an IRP for each I/O operation;
  - passes IRP to correct drivers;
  - deletes IRP when I/O operation is complete
- Driver:
  - Receives IRP
  - Performs operations specified by IRP
  - Passes IRP back to I/O manager or to another driver (via I/O manager) for further processing

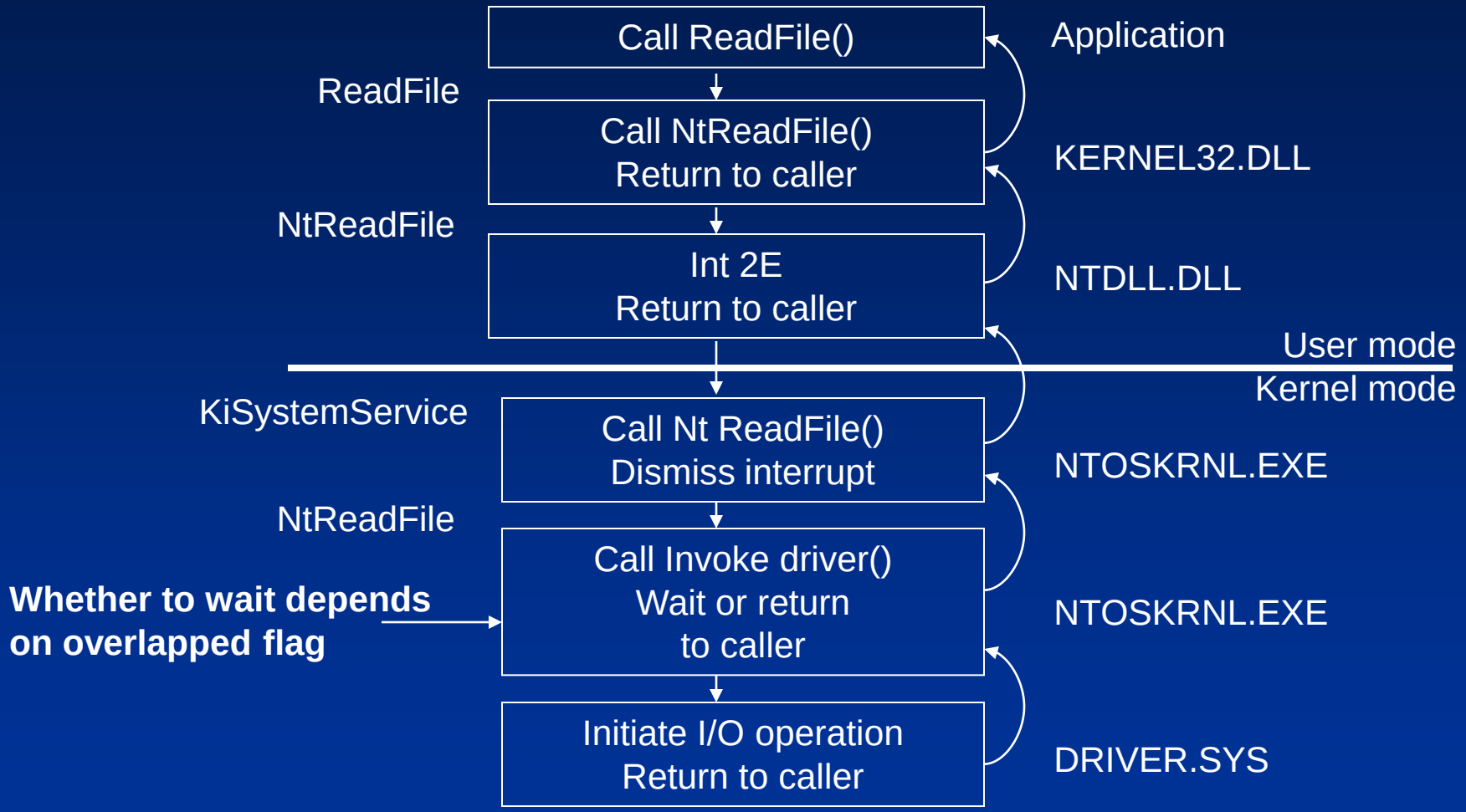
I/O Manager (contd.)

- Supplies common code for different drivers:
  - Drivers become simpler, more compact
- I/O manager:
  - Allows driver to call other drivers
  - Manages buffers for I/O requests
  - Provides time-out support for drivers
  - Records which installable file systems are loaded
  - Provides flexible I/O services to environment subsystems (Windows/POSIX asynchronous I/O)
- Layered processing of I/O requests possible:
  - Drivers can call each other (via I/O manager)

I/O Functions

- Advanced features beyond open, close, read, write:
- Asynchronous I/O:
  - May improve throughput/performance:
    continue program execution while I/O is in progress
  - Must specify FILE_FLAG_OVERLAPPED on CreateFile()
  - Programmer is responsible for synchronization of I/O requests
- Internally, all I/O is performed asynchronously
  - I/O system returns to caller only if file was opened for asynch. I/O
  - For synchronous I/O, wait is done in kernel mode depending on overlapped flag in file object
- Status of pending I/O can be tested:
  - via Windows-API function: *HasOverlappedIoCompleted()*
  - when using I/O completion ports: *GetQueuedCompletionStatus()*

Control flow for an I/O operation



Advanced I/O Functions

- **Fast I/O**
  - Bypass generation of IRPs
  - Go directly to file system driver or cache manager to complete I/O
- **Mapped File I/O and File Caching**
  - Available through Windows-API CreateFileMapping() / MapViewOfFile()
  - Used by OS for file caching and image activation
  - Used by file systems via cache manager to improve performance
- **Scatter/Gather I/O**
  - Windows-API functions ReadFileScatter()/WriteFileGather()
  - Read/write multiple buffers with a single system call
  - File must be opened for non-cached, asynchronous I/O;
    buffers must be page-aligned

HAL

- The hardware abstraction layer (HAL) insulates drivers from the specifics of the processor and interrupt controller by providing APIs that hide differences between platforms
  - in essence, the HAL is the bus driver for all the devices on the computer's motherboard that aren't controlled by other drivers
  - By programming to the HAL, drivers are source-level compatible across CPU architectures

PnP and Power

- The PnP manager
  - Handles driver loading and starting
  - Performs resource arbitration
  - It relies on the I/O Manager to load drivers and send them PnP-related commands
- The power manager controls the power state of the system
  - It relies on the I/O Manager to ask drivers if they can change power state and to inform them when they should

User-Mode Drivers

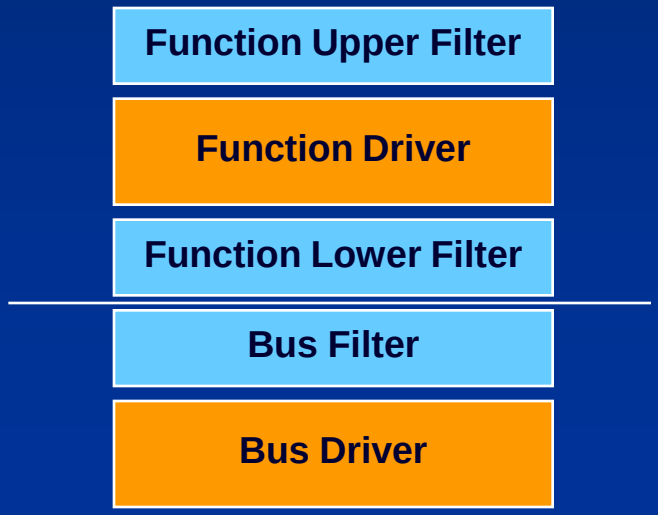
- *Virtual device drivers* (VDDs) are used to emulate 16-bit MS-DOS applications.
  - User-mode can't access hardware directly and thus must go through a real kernel-mode device driver.
  - They trap what an MS-DOS application thinks are references to I/O ports and translates them into native Windows I/O functions
- Windows subsystem *printer drivers* translate device-independent graphics requests to printer-specific commands.
  - Commands are forwarded to a kernel-mode port driver such as the parallel port driver (Parport.sys) or the universal serial bus (USB) printer port driver (Usbprint.sys)

WDM Drivers

- Windows Driver Model
  - Unified architecture for drivers
  - Originally intended to be Win9x/NT cross platform
  - Most PnP Drivers are WDM drivers
- There are three types of WDM drivers:
  - *Bus drivers* manage a logical or physical bus e.g. PCMCIA, PCI, …
  - *Function drivers* manage a particular type of device. Bus drivers present devices to function drivers via the PnP manager.
  - *Filter drivers* logically layer above or below function drivers, augmenting or changing the behavior of a device or another driver.

WDM Drivers

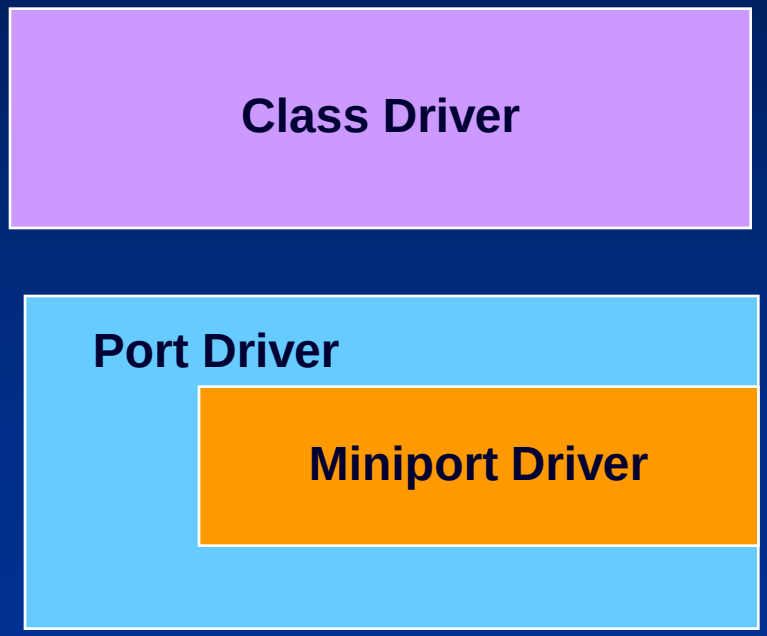
- In WDM, no one driver is responsible for controlling all aspects of a particular device.
- Bus driver: reports the devices on its bus to the PnP manager
- Function driver: manipulates the device
- Filter driver: optional



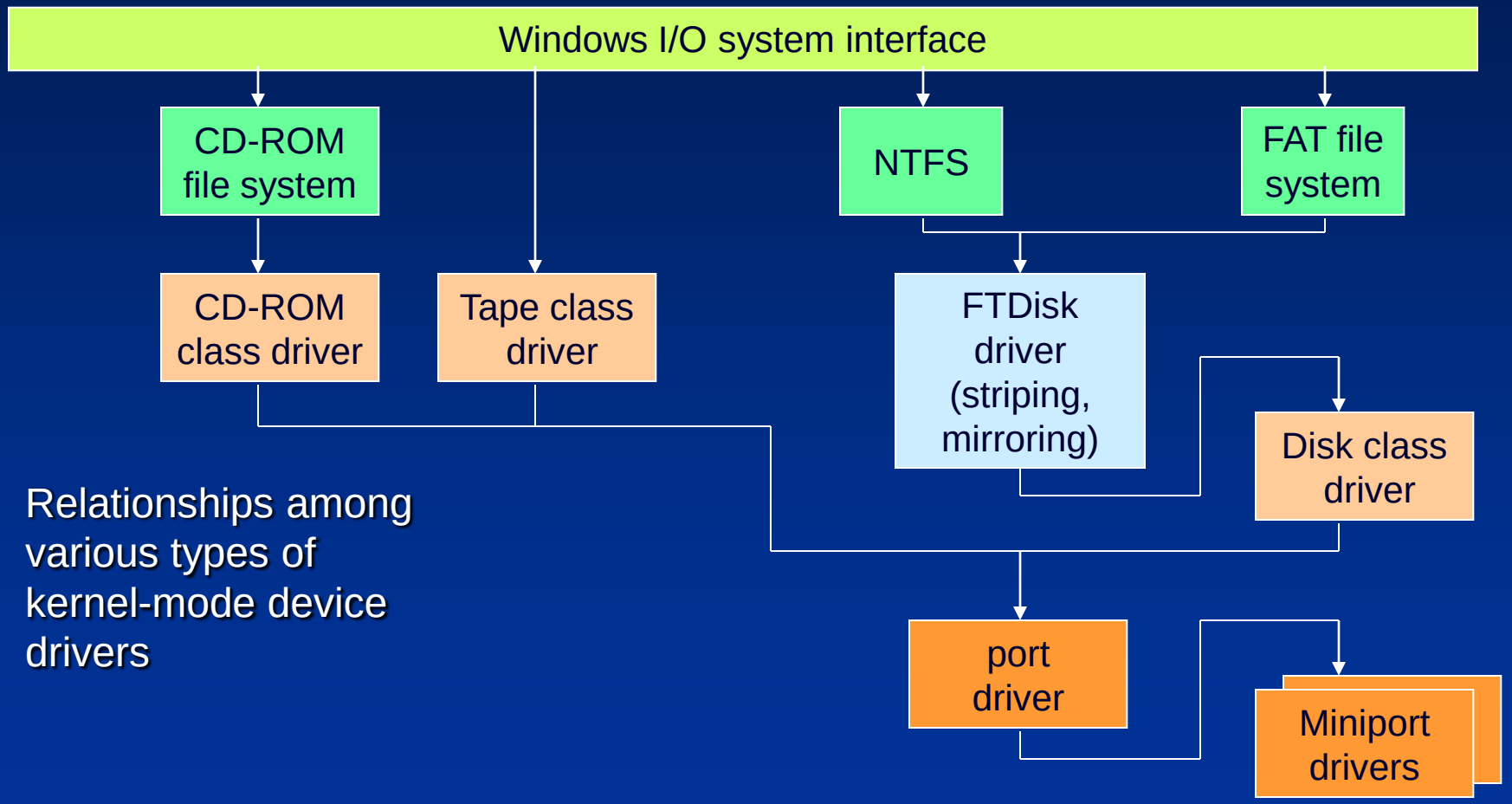
Layered Drivers

- Hardware support might be split between different modules that implement support for different levels of abstraction
  - Microsoft typically provides the drivers for the higher levels of abstraction
  - Hardware vendors provide the lowest level, which understands a particular device
- The conventional division is three levels:
  - *Class drivers* implement the I/O processing for a particular class of devices, such as disk, tape, or CD-ROM.
  - *Port drivers* implement the processing of an I/O request specific to a type of I/O port, such as SCSI, and are also implemented as kernel-mode libraries of functions rather than actual device drivers.
  - *Miniport drivers* map a generic I/O request to a type of port into an adapter type, such as a specific SCSI adapter. Miniport drivers are actual device drivers that import the functions supplied by a port driver.

Layered Drivers

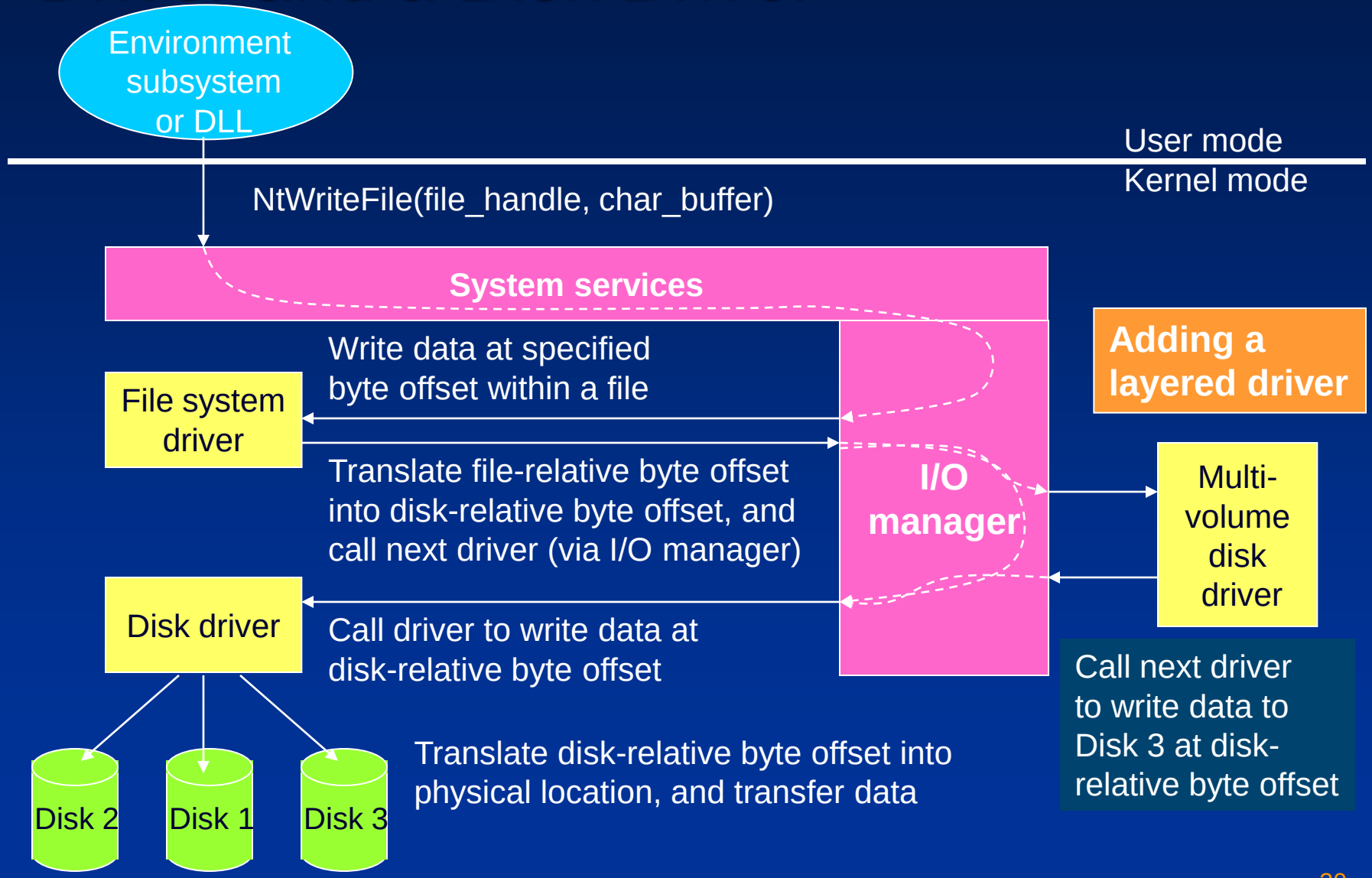


Layered Driver Structure

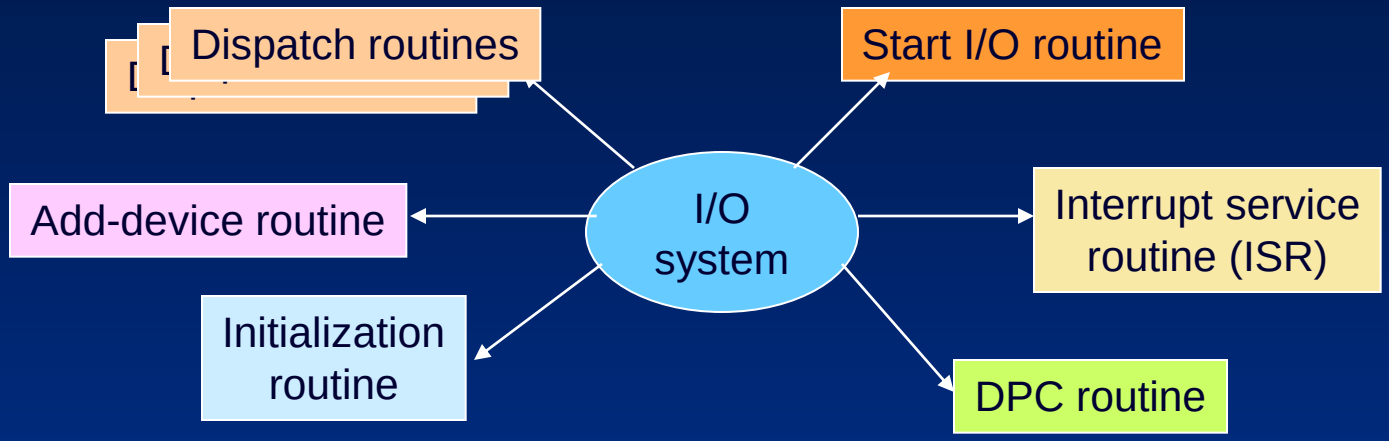


Relationships among various types of kernel-mode device drivers

Dynamically Layering a File System Driver and a Disk Driver



Primary Device Driver Routines



- I/O manager executes initialization routine when loading a driver
- PnP manager calls add-device routine on device detection
- Dispatch routines are the main functions: open(), close(), read(), write()
- Start I/O routine initiates transfer from/to a device
- ISR runs in response to interrupt; schedules DPC
- DPC routine performs actual work of handling interrupt after ISR; starts next queued I/O operation on device

Other components of device drivers

- **I/O Completion routines**
  - A layered driver may have completion routines that will notify it when a lower-level driver finishes processing an IRP
    (**I**/O **R**equest **P**acket)
- **Cancel I/O routine**
  - Assigns a cancel routine to the IRP if the I/O request can be canceled; it will be executed when the operation is canceled
- **Fast dispatch routines**
  - Drivers that use cache manager
  - To allow the kernel to bypass certain typical I/O processing when accessing the driver
  - E.g. reading and writing can access cache
- **Unload routine**
  - Releases system resources
- **System shutdown notification routine**
- **Error-logging routines**
  - Notify I/O manager to write record to error log file (e.g., bad disk block)

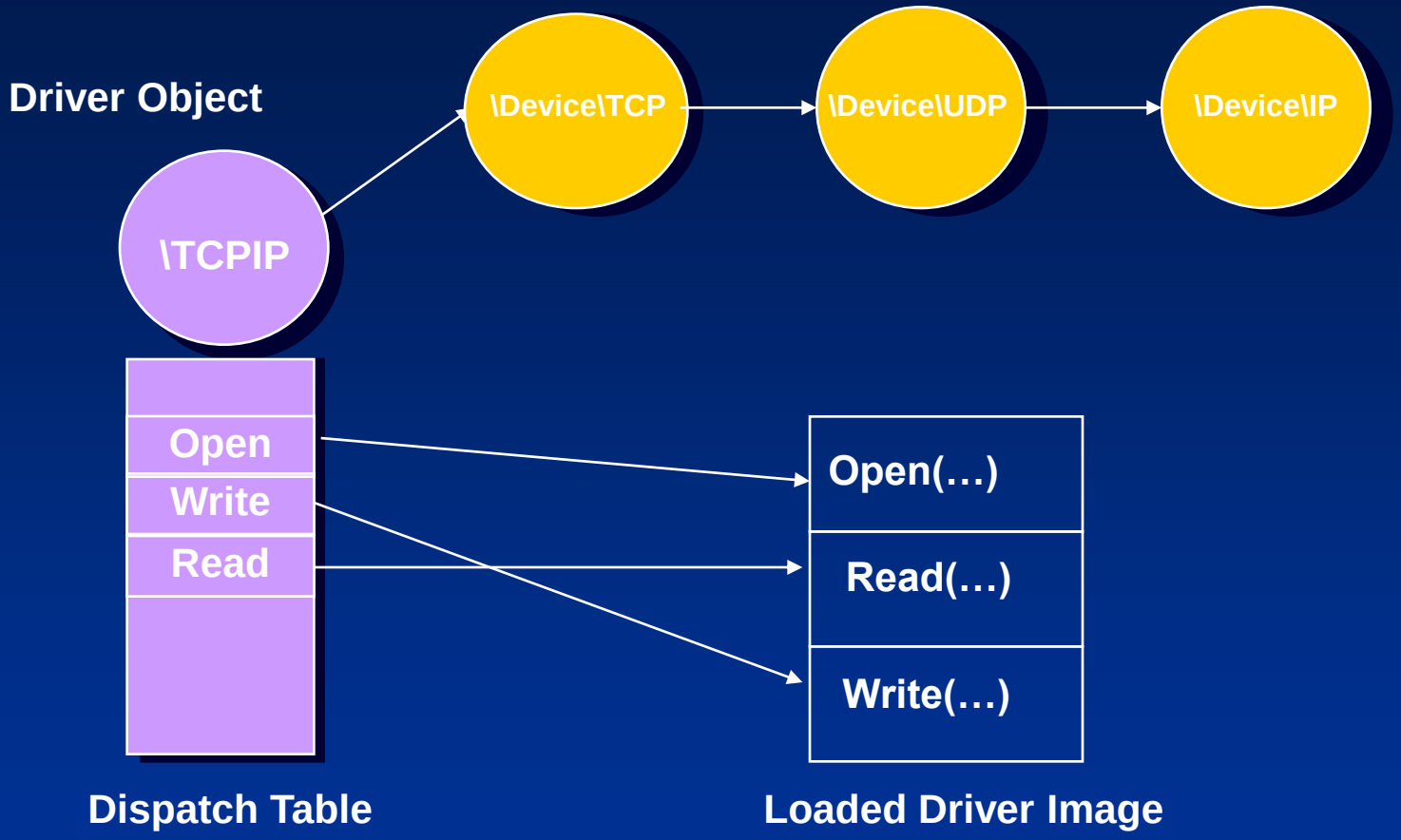
Driver Object

- A driver object represents a loaded driver
  - Names are visible in the Object Manager namespace under \Drivers
  - A driver fills in its driver object with pointers to its I/O functions e.g. open, read, write
  - When you get the “One or More Drivers Failed to Start” message its because the Service Control Manager didn’t find one or more driver objects in the \Drivers directory for drivers that *should* have started

Device Objects

- A device object represents an instance of a device
  - Device objects are linked in a list off the driver object
  - A driver creates device objects to represent the interface to the logical device, so each generally has a unique name visible under \Devices
  - Device objects point back at the Driver object

Driver and Device Objects



TCP/IP Drivers Driver and Device Objects

File Objects

- Represents ***open*** instance of a device (files on a volume are virtual devices)
  - Applications and drivers “open” devices by name
  - The name is parsed by the Object Manager
  - When an open succeeds, the object manager creates a file object to represent the open instance of the device and a file handle in the process handle table
- A file object links to the device object of the “device” which is opened
- File objects store additional information
  - File offset for sequential access
  - File open characteristics (e.g. delete-on-close)
  - File name
  - Accesses granted for convenience

I/O Request Packets

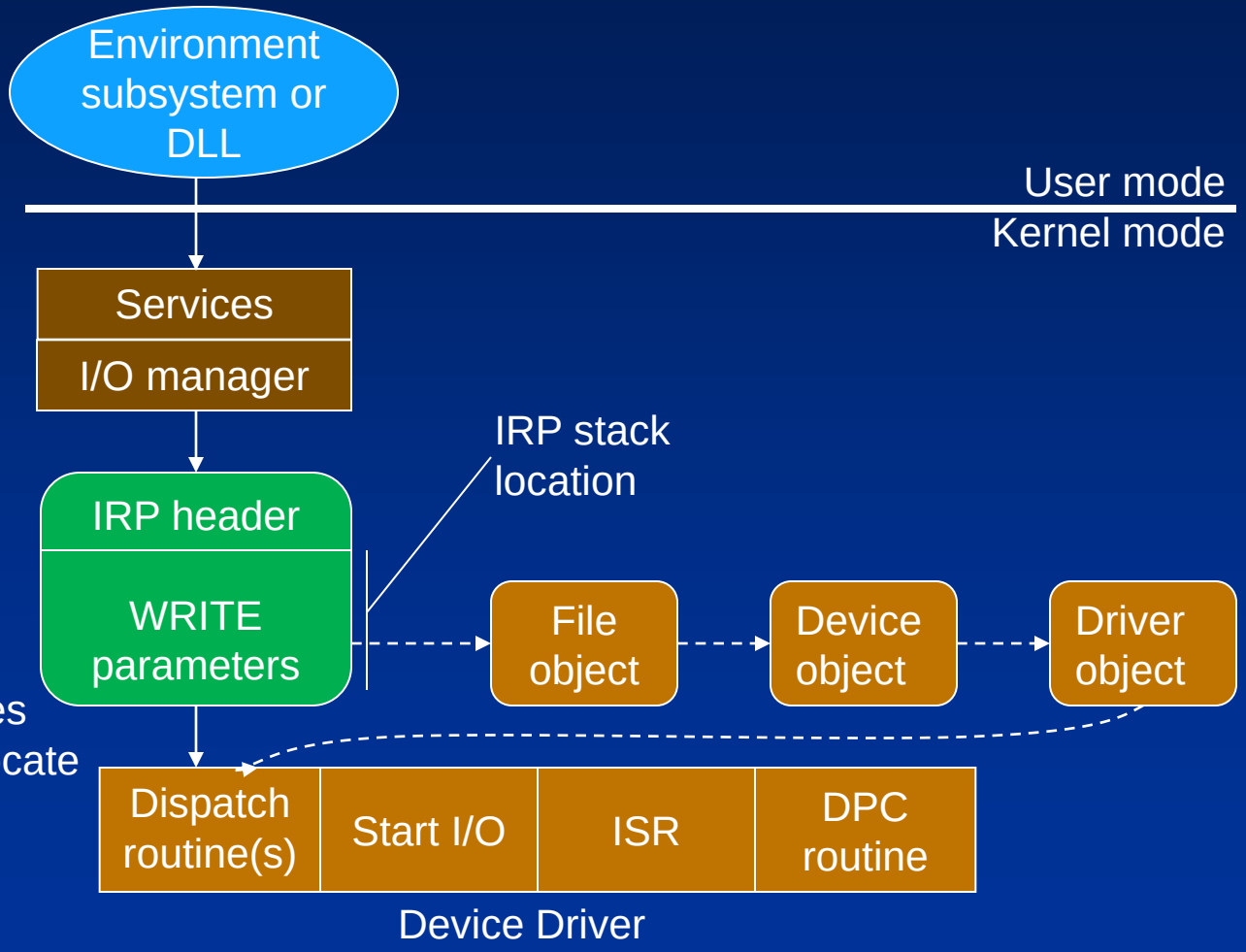
- System services and drivers allocate I/O request packets to describe I/O
- A request packet contains:
  - File object at which I/O is directed
  - I/O characteristics (e.g. synchronous, non-buffered)
  - Byte offset
  - Length
  - Buffer location
- The I/O Manager locates the driver to which to hand the IRP by following the links:

I/O Request Packet (same I/O request)

1) An application writes a file to the printer, passing a handle to the file object

2) The I/O manager creates an IRP and initializes first stack location

3) The I/O manager uses the driver object to locate the WRITE dispatch routine and calls it, passing the IRP



IRP data

IRP consists of two parts:

- Fixed portion (header):
  - Type and size of the request
  - Whether request is synchronous or asynchronous
  - Pointer to buffer for buffered I/O
  - State information (changes with progress of the request)
- One or more stack locations:
  - Function code
  - Function-specific parameters
  - Pointer to caller‘s file object
- While active, IRPs are stored in a thread-specific queue
  - I/O system may free any outstanding IRPs if thread terminates

I/O Processing – synch. I/O to a single-layered driver

1. The I/O request passes through a subsystem DLL
2. The subsystem DLL calls the I/O manager‘s NtWriteFile() service
3. I/O manager sends the request in form of an IRP to the driver (a device driver)
4. The driver starts the I/O operation
5. When the device completes the operation and interrupts the CPU, the device driver services the interrupt
6. The I/O manager completes the I/O request

Completing an I/O request

Servicing an interrupt:

- ISR schedules Deferred Procedure Call (**DPC**); dismisses int.
- **DPC** routine starts next I/O request and completes interrupt servicing
- May call completion routine of higher-level driver

I/O completion:

- Record the outcome of the operation in an I/O status block
- Return data to the calling thread – by queuing a kernel-mode Asynchronous Procedure Call (**APC**)
- **APC** executes in context of calling thread; copies data; frees IRP; sets calling thread to signaled state
- I/O is now considered complete; waiting threads are released

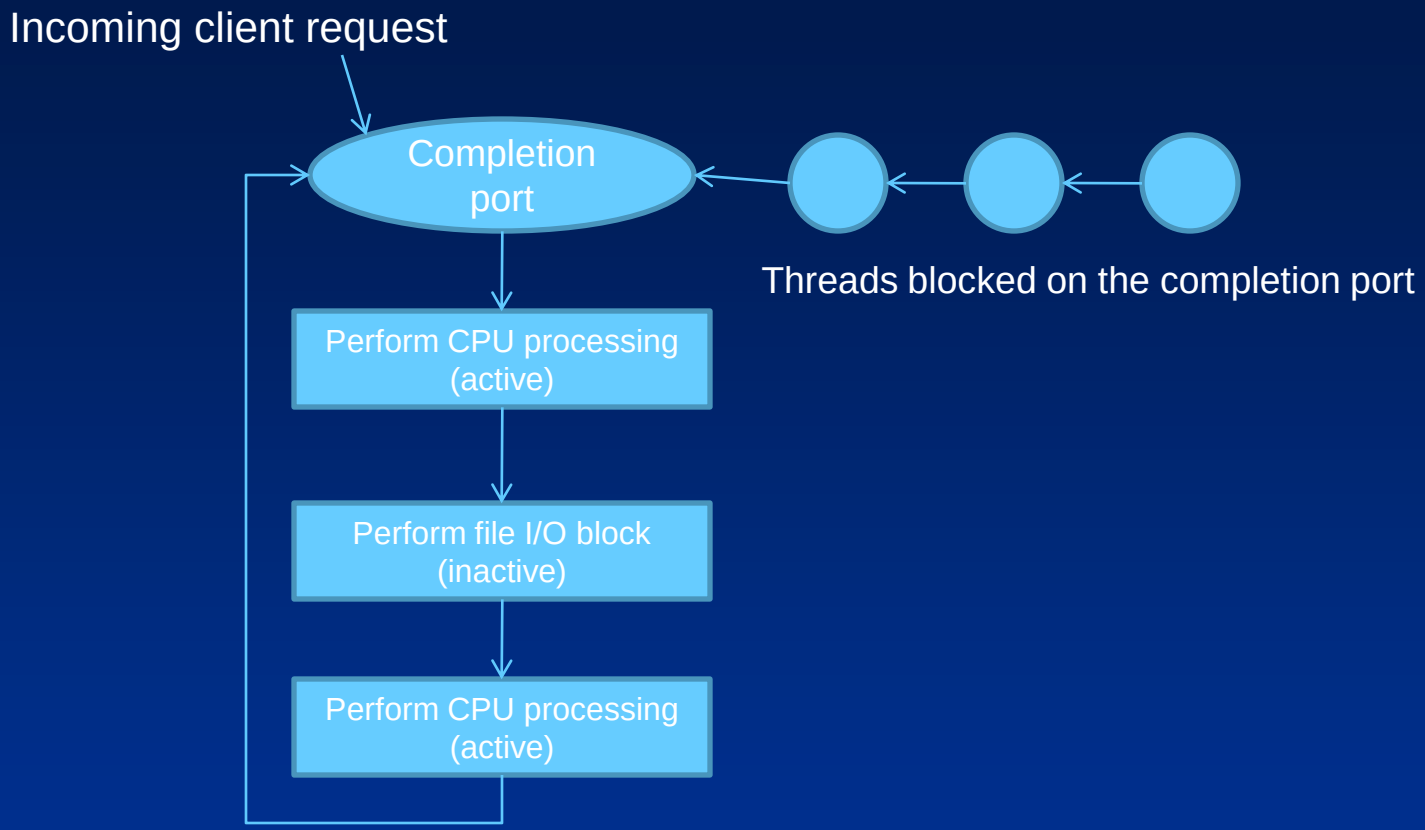
I/O Cancellation

- Example scenarios:
  - Device removal while IRPs active
  - User cancels long-running operation to a device
  - Thread/process termination
- Register an I/O cancel routine for cancellable I/O operations
- User-initiated Cancellation
  - *Cancello* to cancel async. I/Os
  - *CancelSynchronouslo* to cancel sync. I/Os
- Thread termination
  - Cancel all cancellable IRPs before terminating thread

I/O Completion Port

- *IoCompletion* Object
  - Completion of I/O associated with multiple file handles
  - Async. I/O that completes queues a completion packet to the completion port
  - Thread waiting on the completion of multiple files can wait for a completion packet on the queue
  - Similar to *WaitForMultipleObjects* API, but advantage is “concurrency” is controlled with the help of the system.
  - Concurrency value: max number of threads associated with the port that should be running at any time
- Using Completion Ports
  - Call *CreateIoCompletionPort*
  - Thread blocks on completion port are associated with the port and are woken up in LIFO order

I/O Completion Port Operation

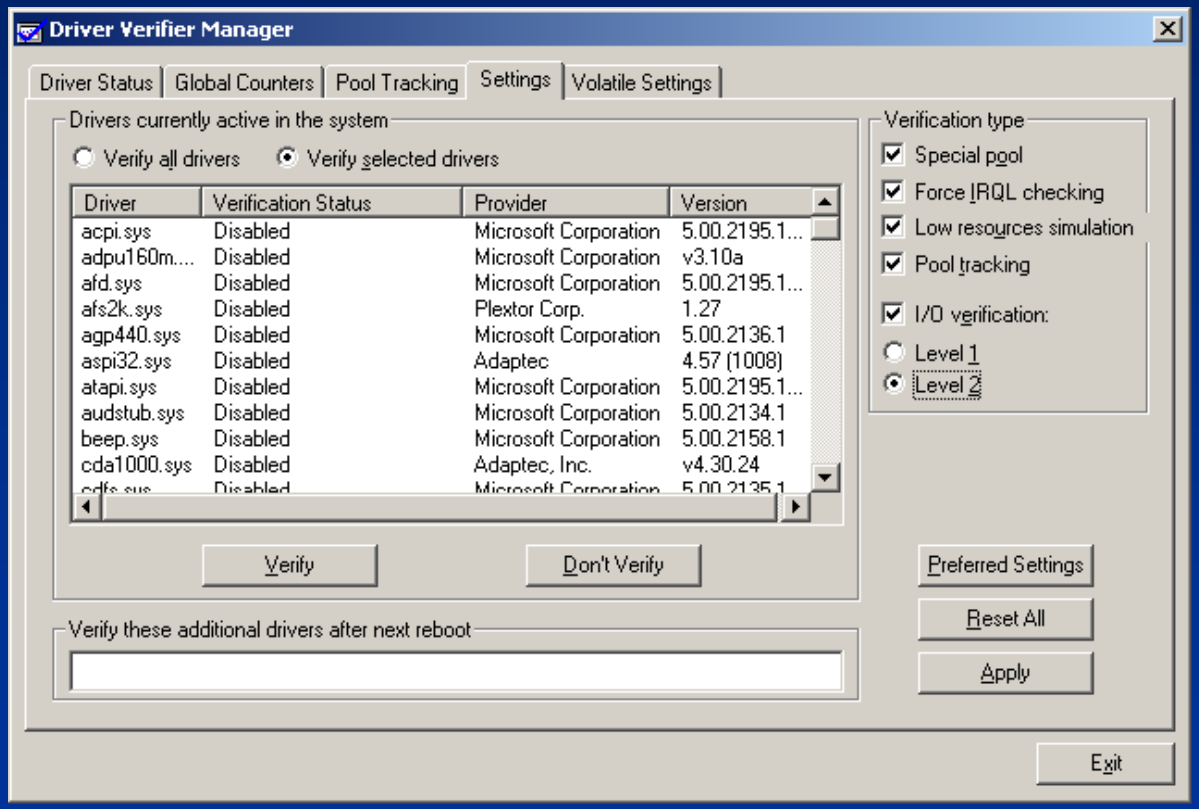


- I/O completion port minimizes dispatcher lock contention
- Doesn't acquire lock when:
  - Completion queued to a port and no threads are waiting
  - Thread calls *GetQueuedCompletionStatus* and there are items in the queue
  - Thread calls *GetQueuedCompletionStatus* with zero timeout

The Driver Verifier

- Driver Verifier is a tool introduced in Windows 2000 that helps developers test their drivers and systems administrators identify faulty drivers
  - Must be run from \windows\system32\verifier.exe (no shortcut)

This is the Windows XP GUI to driver verifier

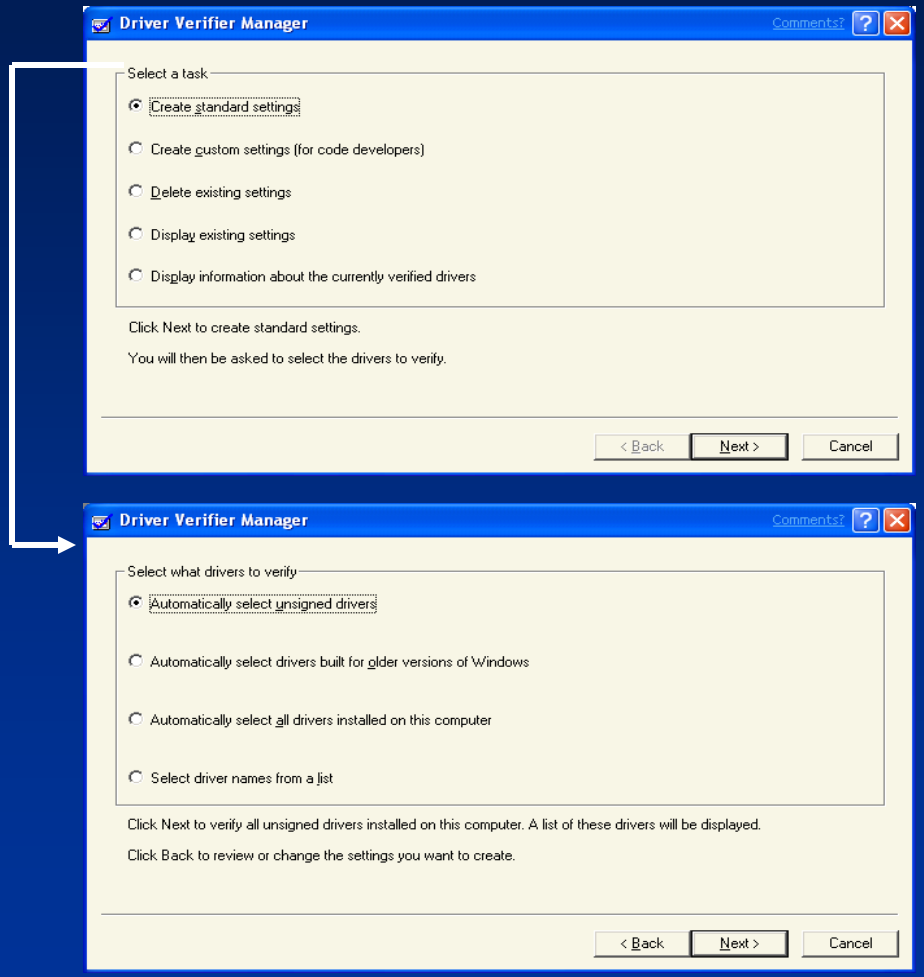


Verification Options

- Special Pool
  - The memory returned for driver memory allocations is bounded with invalid regions to catch buffer overrun and underrun
  - To be described in Crash Analysis section
- Force IRQL checking
  - Detects drivers that access paged memory when the system is in a state that can't tolerate page faults
- Low Resource Simulation
  - Randomly fails driver memory allocations
- Pool Tracking
  - Associates memory with the driver that allocated it to help identify leaks
- I/O verification
  - Ensures that I/O commands are properly formatted and handled

Driver Verifier

- Simpler wizard-style UI
  - Default is verify unsigned drivers
- Four verification options:
  - DMA verification – detects improper use of DMA buffers, adapters, and map registers
  - Deadlock detection – detects lock hierarchy violations with spinlocks, mutexes, fast mutexes
  - SCSI verification - monitors the interaction between a SCSI miniport driver and the port driver
  - Enhanced I/O Verification tests drivers' support for power management, WMI, and filters
- One new on in Server 2003:
  - Disk integrity checking - monitors a hard disk and detects whether the disk is preserving its data correctly



Windows XP GUI

Further Reading

- Mark E. Russinovich et al. Windows Internals, 5th Edition, Microsoft Press, 2009.
  - I/O system components (from pp. 537)
  - Device drivers (from pp. 541)
  - I/O Processing (from pp. 562)

Source Code References

- Windows Research Kernel sources
  - \base\ntos\io – I/O Manager
  - \base\ntos\inc\io.h – additional structure/type definitions
  - \base\ntos\verifer – Driver Verifier
  - \base\ntos\inc\verifier.h – additional structure/type definitions