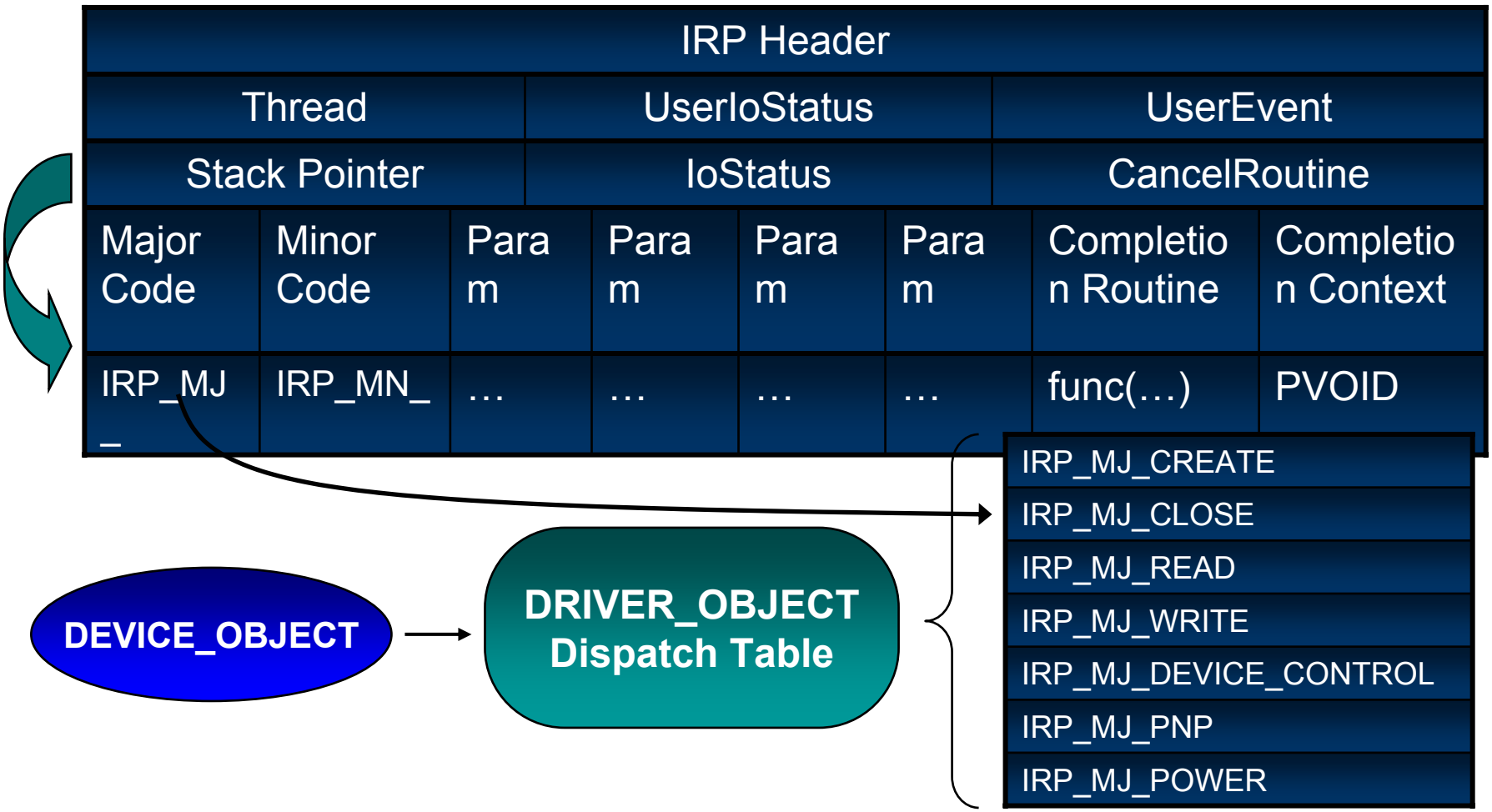
# Windows Kernel Internals
## I/O Architecture

David B. Probert, Ph.D.
Windows Kernel Development
Microsoft Corporation

# I/O Management

- Asynchronous with multiple completion mechanisms
  - Apc completion (callback)
  - Set event completion
  - I/O Completion port
- Packet based (not stack based)
- Stackable Driver model
- Fast path, synchronous in some situations
- Integrated buffer cache management

# I/O Request Packets



# IO Architecture Goals

- Provide consistent I/O abstraction to applications.
- Provide a framework to do the following.
  - Dynamic loading/unloading of drivers.
  - Driver layering.
  - Asynchronous I/O.
  - Uniform enforcement of security.

# IO Manager Objects

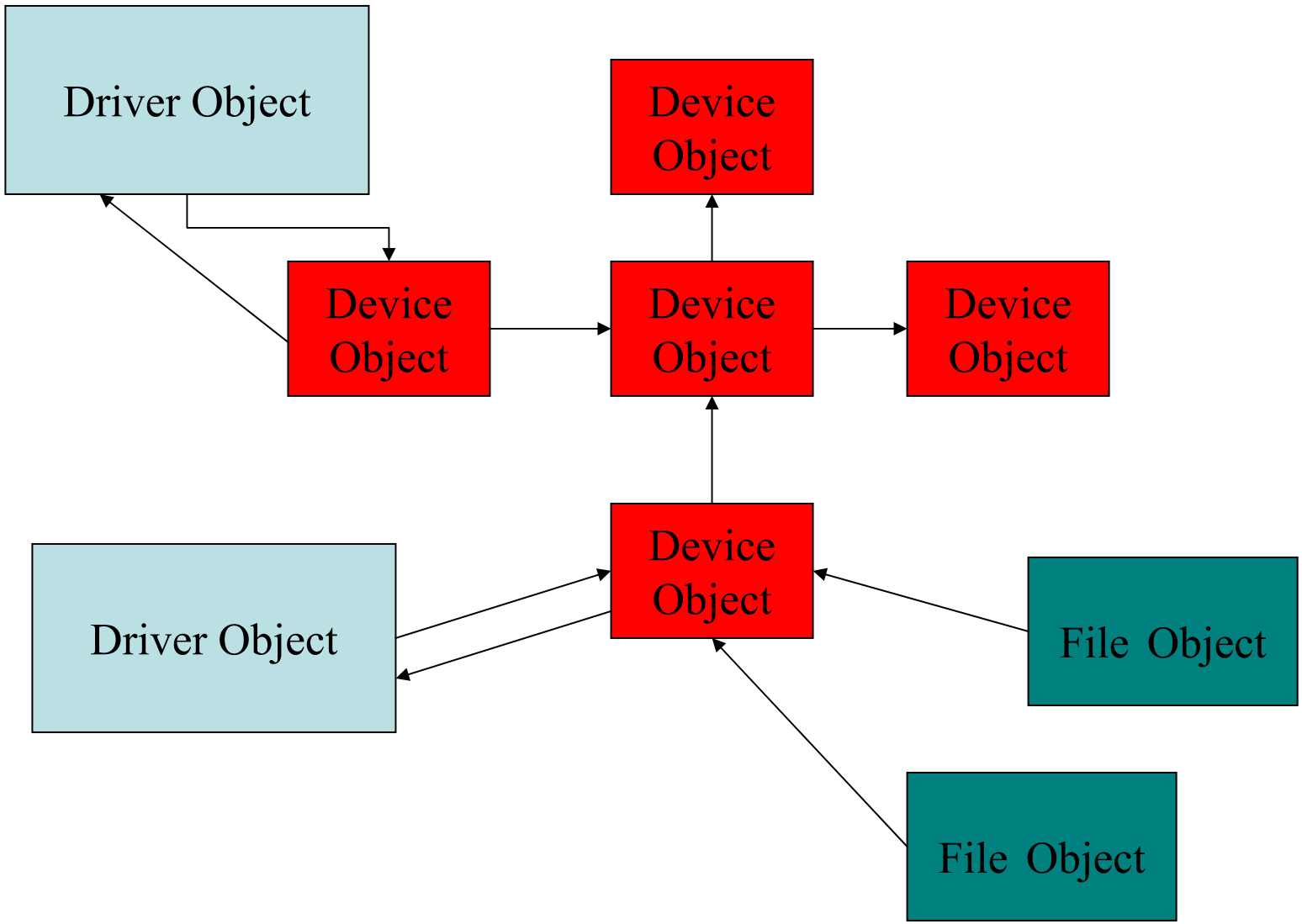
- Driver objects represent loaded drivers.
- Drivers create device objects to represent devices.
- All IO requests are made to device objects.
- File objects represent open instances of device objects.

# Layering Drivers

- Device objects can be attached one on top of another using IoAttachDevice* APIs to create 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.

# Object Relationships



# 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 PNP.
- 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.

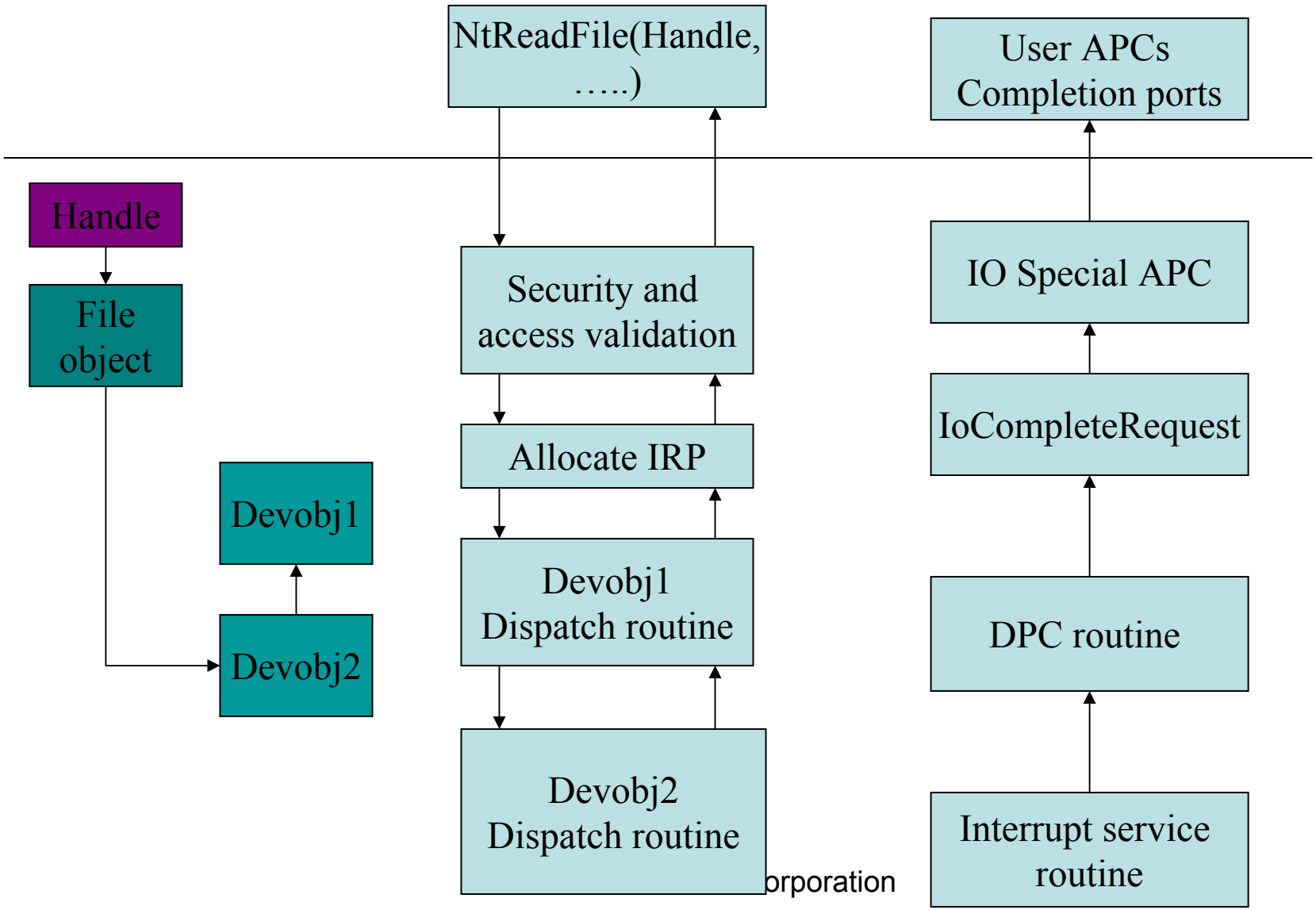
# Device Deletion and Driver Unload

- Drivers delete devices using IoDeleteDevice.
- Drivers are unloaded by calling NtUnloadDriver or by PNP.
- 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.

# IO Request Packet (IRP)

- IO operations encapsulated in IRPs.
- IO requests travel down a driver stack in an IRP.
- Each driver gets a 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 a thread that made the IO request.

# Path of an Async IO request



# Async IO (contd.)

- Applications can issue asynchronous IO requests to files opened with FILE_FLAG_OVERLAPPED and passing an LPOVERLAPPED parameter to the IO API (e.g., ReadFile(…))
- 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.

# Security and Access checks

- Detailed security and access check is done at the time of opening the file. Allowed access rights are store in the object manager and validated when an operation is performed on the handle.
- If filename exactly matches a device object then the IO manager performs the security check or if the driver sets the FILE_DEVICE_SECURE_OPEN in the device object.
- Otherwise the security check should be done by the driver.

# Canceling IRPs

- IO manager provides a mechanism to cancel IRPs queued for a long time. Canceling is done on a per IRP basis.
- IO is canceled when a thread exits or when CancelIo is called on the thread.
- Drivers can cancel IRPs using IoCancelIrp().
- Drivers which queue IRPs that can wait a long time should set a cancel routine.
- Driver clears cancel routine before completing IRP.

# 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

**Meta IO operations**

- NtFsControlFile
- NtDeviceIoControlFile
- NtQueryDirectoryFile
- NtQueryAttributesFile
- NtQueryFullAttributesFile
- NtQueryEaFile
- NtSetEaFile
- NtQueryInformationFile
- NtSetInformationFile
- NtNotifyChangeDirectoryFile

**Administrative operations**

- NtLoadDriver
- NtUnloadDriver
- NtQueryVolumeInformationFile
- NtSetVolumeInformationFile
- NtQueryQuotaInformationFile
- NtSetQuotaInformationFile

# Debugging I/O Problems

# IO Architecture Overview

- IO APIs come from many components
  - Classic IO manager
  - Plug and play manager
  - Power manager
  - Dump and Partition APIs
- Why is the IO model complex ?
  - Large number of devices to be supported
  - PnP and Power mgmt state machines
  - Lots of miniport APIs (storage, NDIS, TDI, Audio, USB, 1394)

# IO Manager Objects

- Driver objects represent loaded drivers.
- Drivers create device objects to represent devices.
- All IO requests are made to device objects.
  - Device objects have names
  - Device interface symbolic links to PDOs
- File objects represent open instances of device objects.

# IO verifier

- IO verifier started off by thunking driver calls to IO APIs (IO Verifier level 1)
  - Allocates IRP from special pool
  - Validates parameters to IO APIs
  - No performance hit
  - Path changes are local to the driver
  - Support added to unload all drivers at shutdown
  - Memory leaks caught when driver image is unloaded

# IO verifier (contd.)

- IO verifier level 2
  - Leverages IRP tracking code that was in checked build.
  - Allocates memory for each IRP and keeps track of its path and validates the parameters.
  - Catches lots of inconsistencies in the stack
  - Perf impact in terms of memory usage and long paths
  - Global to the system. Tests all drivers and reports results for driver to be verified.

# Common driver problems that crash in IO

- IOCTL system buffer already freed or overrun
- Events in IRP are allocated from stacks that have unwound. Kernel stacks pagable.
- Cancellation races
- Multiple IRP complete requests
- Driver left locked pages
- Hangs

# Driver loading issues

- Two drivers cannot be loaded from the same image
- A driver which is also a DLL (EXPORT_DRIVER) should be loaded as a driver first
- If a component is loaded as a DLL DllInitialize and DllUnload routines are invoked

# Driver unloading issues

- 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 completes.
  - Cancellation issue.
  - Pending flag not set correctly. If a driver returns STATUS_PENDING it should mark the IRP pending.
- System buffer already freed.
- 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)

# Crashes (Contd.)

- 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)

# Crashes (contd.)

- INACCESSIBLE_BOOT_DEVICE bug check
  - Status code is a bug check parameter that helps narrow it to a file system or storage problem
- KERNEL_DATA_INPAGE_ERROR bug check
  - Status code in bug check parameter helps determine the problem

# Hangs

- Critical section timeouts. Process stuck in kernel inside IO manager.
  - !thread shows IRP and identifies driver.
  - NPFS IRPs are usually hung because the consumer is another process (Services hung on another user process RPC).
  - Not marking Pending flag causes hangs (verifier catches this)
  - Recursive locking (fs filter problems)

# Hangs (contd.)

- APC deadlocks. IO system uses APCs to complete IRPs and will deadlock if IO is issued at IRQL >= APC_LEVEL
- KeStackAttachProcess can also cause deadlocks as it blocks APCs.
- DMA API hangs on PAE systems. Caused by map registers  allocated and not freed.

# Canceling IRPs

- IO manager provides a mechanism to cancel IRPs queued for a long time. Canceling is done on a per IRP basis.
- IO is canceled when a thread exits or when CancelIo is called on the thread. IO manager waits for 5 minutes for IRP to get canceled.
- Drivers can cancel IRPs using IoCancelIrp().
- Drivers which queue IRPs for a long time (> 1 second) should set a cancel routine in the IRP.
- Driver clears cancel routine before completing IRP.

# Debugging notes

- !thread lists IRPs pending for thread
- !drvobj <ptr> prints out driver object
- !devobj <ptr> prints out device object
- !object ¥??¥ prints out interesting symbolic links
- dt FILE_OBJECT <ptr> dumps fileobjects

# Discussion