Device Drivers and Asynchronous I/O

Dr. Wolfgang Koch

Friedrich Schiller University Jena

Department of Mathematics and Computer Science

Jena, Germany

wolfgang.koch@uni-jena.de

Synchronous I/O

the I/O function does not return until the operation has been completed

the execution of the calling thread can be blocked for an indefinite period

Asynchronous I/O

the I/O function can return immediately, even though the operation has not been completed.

this enables a time-consuming I/O operation to be executed in the background while the calling thread is free to perform other tasks.

Asynchronous I/O = Overlapped I/O

2

Device Drivers and Asynchronous I/O

I want to show what implications asynchronous I/O has on the driver side and what must be done in the driver to support asynchronous I/O.

Furthermore I demonstrate how to implement a bullet-proof cancel-safe wait queue in WDM.

1. Introduction: Asynchronous I/O, Drivers, WDM
2. Asynchronous I/O in the Application Program
3. Asynchronous I/O in the WDM Driver
4. A Cancel-Safe Wait Queue

Wikipedia about WDM:

I/O cancellation is almost impossible to get right.

3

Input / Output Operations

In Windows (similar to Unix) input/output devices are treated very much like ordinary files.

Application programs use the same system calls to access devices as with ordinary files:

`open(), read(), write(), close()`

in Windows: `#include <windows.h>`

```
CreateFile()
ReadFile()
WriteFile()
DeviceIoControl()
CloseHandle()
```

4

Device Driver

a software module in kernel space that controls a hardware device

drivers can be built separately from the rest of the kernel and loaded at boot time or at runtime when needed

the OS (the I/O Manager) calls the driver on behalf of a user program to perform operations that relate to the hardware device

Device Driver

a software layer that lies between the applications and the actual device –

hides the details, provides a standardized surface

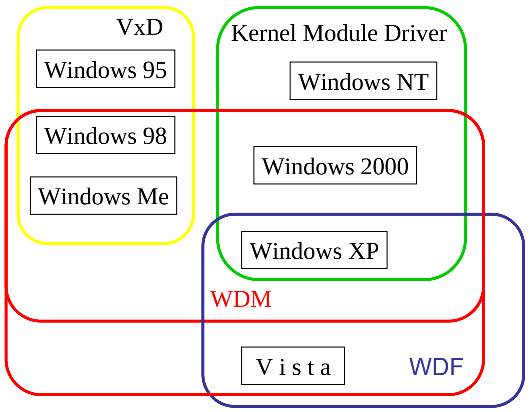
applications use system calls –

`CreateFile(), ReadFile(), WriteFile(), . . .`

to every supported function there is a counterpart (a dispatch function - `DevRead(), DevWrite(),…`) in the driver –

the I/O Manager (a part of the kernel) calls the corresponding function when serving a system call

WDM – The Windows Driver Model



WDM – The Windows Driver Model

WDM drivers:

device drivers that are source-code compatible across all Microsoft Windows operating systems (W98, W2000, forward-compatible to XP and Vista)

- Layered in Device Stacks
- Support Plug and Play (hot plugging)
- Support Power Management
- On Uniprocessor and Multiprocessor Machines

WDM – The Windows Driver Model

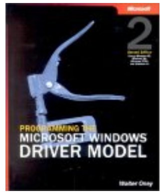
Wikipedia, the free encyclopedia states:

- I/O cancellation is almost impossible to get right.
- Interactions with power management events and Plug-and-play are difficult. This leads to a variety of situations where Windows machines cannot go to sleep or wake up correctly due to bugs in driver code.

→ Introduction of WDF (Windows Driver Foundation)

Literature

The WDM Bible:



Walter Oney
Programming the Microsoft Windows Driver Model
2nd edition

Redmond, Wash : Microsoft Press, 2003
ISBN: 0-7356-1803-8

846 p. + CD-ROM

Literature

very important : man pages and help files

Microsoft MSDN man pages:
http://msdn.microsoft.com/library

Help files of the DDK
Microsoft “white papers”

Newsgroups:
comp.os.ms-windows.programmer.nt.kernel-mode
microsoft.public.development.device.drivers

Asynchronous I/O in the Application Program

- CreateFile
- ReadFile (WriteFile, DeviceIoControl)
- OVERLAPPED Structure
- GetOverlappedResult, Status Codes
- WaitForXxx, Events
- Cancellation (`CancelIo`)

Synchronous I/O in the Application Program

```
#include <windows.h>
{
  HANDLE hd;
  char buff[100]; DWORD err, result, nrd;

  hd = CreateFile("\\\\.\\Device",
             GENERIC_WRITE | GENERIC_READ,
             FILE_SHARE_WRITE | FILE_SHARE_READ,
             NULL, OPEN_EXISTING, 0, NULL);

  if(hd == INVALID_HANDLE_VALUE){
    err = GetLastError();   . . .
  }
  . . .
```

13

Synchronous I/O in the Application Program

```
{
  HANDLE hd;
  char buff[100]; DWORD err, result, nrd;

  hd = CreateFile("\\\\.\\Device", ... ,0, NULL);

  if(hd == INVALID_HANDLE_VALUE){ … return 1;}
  . . .
  result = ReadFile( hd, buff, 30, &nrd, NULL);
  if (result) {
     buff[nrd]= 0;  . . .
  } else { err = GetLastError();  . . . }

  CloseHandle(hd);
}
```

14

Asynchronous I/O: CreateFile

```
HANDLE hd;

hd = CreateFile("\\\\.\\Device",
           GENERIC_WRITE | GENERIC_READ,
           FILE_SHARE_WRITE | FILE_SHARE_READ,
           NULL, OPEN_EXISTING,
           FILE_FLAG_OVERLAPPED, NULL);

if(hd == INVALID_HANDLE_VALUE){
  err = GetLastError();   . . .
}
```

FILE_FLAG_OVERLAPPED -

just a flag to allow asynchronous I/O

15

Asynchronous I/O: ReadFile (WriteFile)

```
char buff[100]; DWORD err, result, nrd;
OVERLAPPED  overl;

memset(&overl, 0, sizeof(OVERLAPPED));

result = ReadFile( hd, buff, 30, &nrd, &overl);

if (!result){
  if (GetLastError() == ERROR_IO_PENDING) {
    printf(" Read Async Pending %d \n", nrd);  // =0
  } else showerr("Read");
} else
{ buff[nrd]= 0;
  printf(" Read Sync: %s \n", inbu);
}
```

16

Asynchronous I/O: ReadFile (WriteFile)

```
memset(&overl, 0, sizeof(OVERLAPPED));
result = ReadFile( hd, buff, 30, &nrd, &overl);

if (GetLastError() == ERROR_IO_PENDING) { . . . }
```

if the request is pending –

… how do I know, when the request is complete ?
… the actual error code ?
… the number of bytes transferred ? (nrd = 0)

→ the OVERLAPPED structure contains the information

17

Asynchronous I/O: struct OVERLAPPED

```
typedef struct _OVERLAPPED {
  ULONG_PTR Internal;
  ULONG_PTR InternalHigh;
  DWORD Offset;
  DWORD OffsetHigh;
  HANDLE hEvent;
} OVERLAPPED,*POVERLAPPED;
```

`Internal`	the (kernel mode) error code
`InternalHigh`	the number of bytes transferred
`Offset, OffsetHigh`	the file position
`hEvent`	an event to wait for

18

Asynchronous I/O: struct OVERLAPPED

```
typedef struct _OVERLAPPED {
  ULONG_PTR Internal;       // error code
  ULONG_PTR InternalHigh;   // number of bytes
  . . .
} OVERLAPPED,*POVERLAPPED;
```

`Internal` contains `STATUS_PENDING, STATUS_SUCCESS` or whichever error occurred (kernel mode `NTSTATUS` code)

in ddk\ntstatus.h:

```
#define STATUS_SUCCESS   ((NTSTATUS)0x00000000L)
#define STATUS_PENDING   ((NTSTATUS)0x00000103L)
#define STATUS_CANCELLED ((NTSTATUS)0xC0000120L)
```

19

Asynchronous I/O: GetOverlappedResult

```
typedef struct _OVERLAPPED {
  ULONG_PTR Internal;       // error code
  ULONG_PTR InternalHigh;   // number of bytes
  . . .
} OVERLAPPED,*POVERLAPPED;
```

`Internal` and `InternalHigh` were originally reserved for system use, their behavior may change – don't use them directly →

```
result = GetOverlappedResult(hd,&overl,&nrd,FALSE);

if (GetLastError() == ERROR_IO_INCOMPLETE) . . .
```

FALSE – do not wait until the operation has been completed

20

Asynchronous I/O: GetOverlappedResult

`Internal` and `InternalHigh` were originally reserved for system use, their behavior may change – don't use them directly →

just to poll for the completion of an I/O request:

```
if ( HasOverlappedIoCompleted(&overl) ) . . .
```

```
in winbase.h:

#define HasOverlappedIoCompleted(lpOverlapped)  \
  ((lpOverlapped)->Internal != STATUS_PENDING)
```


Asynchronous I/O: Status and Error Codes

```
result = ReadFile( hd, buff, 30, &nrd, &overl);

result = GetOverlappedResult(hd,&overl,&nrd,FALSE);

 err = GetLastError();
 if ( err == ERROR_IO_INCOMPLETE ||      //  GetOverl
      err == ERROR_IO_PENDING ) . . .    //  ReadFile

in winerror.h:
 #define ERROR_OPERATION_ABORTED 995L    // 0x03E3
 #define ERROR_IO_INCOMPLETE     996L    // 0x03E4
 #define ERROR_IO_PENDING        997L    // 0x03E5
 #define ERROR_SUCCESS             0L
 #define NO_ERROR                  0L
```


Asynchronous I/O: Waiting for Completion

We can wait for the completion of an I/O request in different ways:

- call `GetOverlappedResult()` with `bWait = TRUE`
- wait for the file handle: **`WaitForSingleObject`**`(hd,INFINITE)`
- initialize and wait for the `hEvent` field in the `OVERLAPPED` structure

```
memset(&overl, 0, sizeof(OVERLAPPED));
overl.hEvent = CreateEvent(NULL, TRUE, TRUE,
               NULL); // manual-reset, signaled
ReadFile( hd, buff, 30, &nrd, &overl);
 . . .
WaitForSingleObject(overl.hEvent,INFINITE)
```

- use Completion Ports

Asynchronous I/O: Waiting for Completion

initialize and wait for the `hEvent` field in the `OVERLAPPED` structure (more specific than the file handle)

```
memset(&overl, 0, sizeof(OVERLAPPED));
overl.hEvent = CreateEvent(NULL, TRUE, TRUE,
               NULL); // manual-reset, signaled
```

Functions as `ReadFile()` set this handle to the nonsignaled state before they begin an I/O operation.
When the operation has completed, the handle is set to the signaled state.

If we have multiple I/O operations pending, we can gather their event handles in an array and use `WaitForMultipleObjects(n,arr,…)`

Asynchronous I/O: Cancellation

An application program can cancel pending I/O requests:

```
CancelIo(hd);
```

This function cancels all pending I/O operations for this file handle.

```
CancelIoEx(hd, &overl);
```

This function cancels a specific I/O operation.
It is available only in Vista.

If we call `GetOverlappedResult()` on the completed cancelled operation, we get the status code:

```
ERROR_OPERATION_ABORTED   // 0x03E3
```

25

Asynchronous I/O: Cancellation

An application program can cancel pending I/O requests:

```
CancelIo(hd);
```

this function cancels all pending I/O operations for this file handle.

```
CancelIoEx(hd, &overl);
```

this function cancels a specific I/O operation,
it is available only in Vista

If we call `GetOverlappedResult()` on the completed cancelled operation, we get the status code:

```
ERROR_OPERATION_ABORTED   // 0x03E3
```


26

Asynchronous I/O in the WDM Driver

when serving a system call
the I/O manager creates an **IRP** (**I/O Request Packet**)
and sends it to the corresponding dispatch routine
of the device driver

the dispatch routine can:

- complete the IRP immediately (→ synchronous)
- queue the IRP for later processing (→ asynchronous)
- pass the IRP down to a lower driver, with or without a Completion Routine for postprocessing

27

WDM Driver: Synchronous Buffered Read

The dispatch routine can complete the IRP immediately
→ **synchronous**

```
NTSTATUS DrvRead( IN PDEVICE_OBJECT DevObj,
                  IN PIRP          Irp    )
{ NTSTATUS status   = STATUS_SUCCESS;
  long     ReadLen;
  char *   UserBuffer;     // buffered I/O

  UserBuffer = Irp->AssociatedIrp.SystemBuffer;

  < get the data >

  RtlMoveMemory(UserBuffer, &data, ReadLen);
  . . .
```

28

WDM Driver: Synchronous Buffered Read

```
NTSTATUS DrvRead( IN PDEVICE_OBJECT DevObj,
                  IN PIRP           Irp    )
{ . . .
  RtlMoveMemory(UserBuffer, &data, ReadLen);
  // the I/O manager copies the buffer to
  // user space while completing the IRP

  // IRP Completion
  Irp->IoStatus.Status      = status;      // ???
  Irp->IoStatus.Information = ReadLen;
  IoCompleteRequest(Irp, IO_NO_INCREMENT); // ???
  return status;
}
```

29

WDM Driver: Asynchronous Buffered Read

The dispatch routine can queue the IRP
for later processing → **asynchronous**

```
NTSTATUS DrvRead( IN PDEVICE_OBJECT DevObj,
                  IN PIRP           Irp    )
{
  IoMarkIrpPending(Irp);    // ???

  StartPacket(Irp);         // queue the IRP

  return STATUS_PENDING;
}
```

30

WDM Driver: Asynchronous Buffered Read

```
StartPacket(Irp);       // queue the IRP
return STATUS_PENDING;
```

Problem: If the I/O manager gets `STATUS_PENDING`, how does it receive the real results when the IRP is completed ?

Answer: Later on, when data are available –
usually in a DPC Routine (Deferred Procedure Call),
requested by an Interrupt Service Routine (ISR) –
the data are copied to the buffer in the I/O manager and the IRP is completed using `IoCompleteRequest`

31

WDM Driver: Asynchronous Buffered Read

when data are available – usually in a DPC routine – the data are copied to the buffer in the I/O manager and the IRP is completed:

```
void DpcForIsr( void * context)
{
  < dequeue the IRP >
  RtlMoveMemory(UserBuffer, &data, ReadLen);

  // IRP Completion
  Irp->IoStatus.Status      = status;    // now
  Irp->IoStatus.Information = ReadLen;
  IoCompleteRequest(Irp, IO_NO_INCREMENT);
}
```

32

WDM Driver: Asynchronous Buffered Read

In a DPC routine the data are copied to the buffer in the I/O manager and the IRP is completed.

2nd Problem:
How can the I/O manager copy data to the requesting program's user-space buffer?
If the IRP is completed synchronously, the caller's address space is current and directly accessible – the dispatch routine runs in the **context** of the caller.
A DPC routine, on the other hand, certainly does not, it runs in an **arbitrary thread context**.

33

WDM Driver: Asynchronous Buffered Read

Problem:
The DPC routine runs in an arbitrary thread context, the caller's address space is not accessible. How can the I/O manager copy data to the caller's buffer?
Answer:
The I/O manager queues a special kernel-mode **APC** (Asynchronous Procedure Call) to the caller's thread. The APC executes later in the context of that thread, so it can transfer data to the caller's user-space buffer.

That's why `IoMarkIrpPending()` is needed – an IRP can be completed before `STATUS_PENDING` is returned.

34

WDM Driver: Interrupt Request Level (IRQL)

Kernel code runs at a certain IRQL, it can be interrupted only by an activity that executes at a higher IRQL.

3 … 26	`DIRQL`	Hardware – ISRs
2	`DISPATCH_LEVEL`	Scheduler, DPCs, code protected by a spinlock
1	`APC_LEVEL`	APC – Routines
0	`PASSIVE_LEVEL`	user code, dispatch routines, PnP routines

35

WDM Driver: Interrupt Request Level (IRQL)

Kernel code runs at a certain IRQL.
(don't confuse IRQL with thread priority)

`2 - DISPATCH_LEVEL -` Scheduler, DPCs,

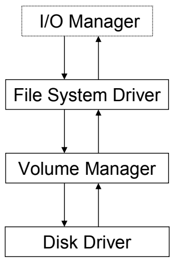
Code executing at `IRQL >= DISPATCH_LEVEL` cannot block, nor can it use pagable memory.

To use Semaphores or wait on Events, and for many other tasks (e.g. accessing disk files) driver code must run at `IRQL == PASSIVE_LEVEL` .

36

WDM Driver: Layered Drivers

WDM-Drivers usually are layered in **Device Stacks.**

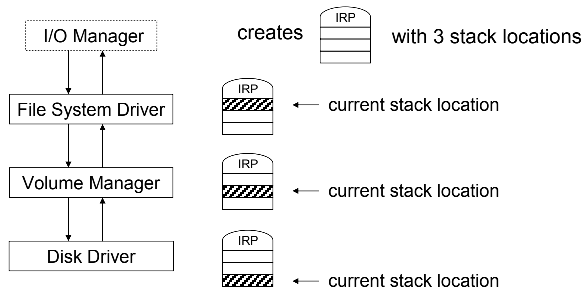


To every of these **Function Drivers** there can be one or more upper or lower **Filter Driver**. (e.g. for compression or encryption in this case)

To support Layered Drivers, an IRP consist of a fixed header (**body**) and **stack locations** – one for every driver in the stack.

WDM Driver: Layered Drivers

To support Layered Drivers, an IRP consist of a fixed header (**body**) and **stack locations** – one for every driver in the stack.



WDM Driver: Layered Drivers

The IRP body contains global information, as buffer addresses, a stack location contains a function code, function-specific parameters and driver context information.

```
NTSTATUS DrvRead( IN PDEVICE_OBJECT DevObj,
                  IN PIRP           Irp    )
{ NTSTATUS status   = STATUS_SUCCESS;
  PIO_STACK_LOCATION  pIrpStack;
  long      ReadLen;   char *   UserBuffer;

  pIrpStack  = IoGetCurrentIrpStackLocation(Irp);
  ReadLen    = pIrpStack->Parameters.Read.Length;
  UserBuffer = Irp->AssociatedIrp.SystemBuffer;
  . . .
```


WDM Driver: Layered Drivers

An upper filter driver usually does some preprocessing on the given data and then it passes the IRP down to the next lower driver with a copy of its own stack location.

```
NTSTATUS DrvWrite( IN PDEVICE_OBJECT DevObj,
                   IN PIRP           Irp    )
{ NTSTATUS status;

  < preprocessing >
  IoCopyCurrentIrpStackLocationToNext(Irp);
  // or: IoSkipCurrentIrpStackLocation(Irp);

  status = IoCallDriver(LowerDevice, Irp);
  return status;
}
```

WDM Driver: Layered Drivers

But what about postprocessing ?

```
{ NTSTATUS status;

  < preprocessing >
  IoCopyCurrentIrpStackLocationToNext(Irp);
  status = IoCallDriver(LowerDevice, Irp);

  < postprocessing ??? >

  return status;
}
```

Problem: Postprocessing doesn't work if the lower driver is asynchronous and returns `STATUS_PENDING`.

WDM Driver: Completion Routines

Problem: Postprocessing doesn't work if the lower driver is asynchronous and returns `STATUS_PENDING`.

Answer: We can install a **completion routine**, it is called by `IoCompleteRequest()`.

```
NTSTATUS DispAny(PDEVICE_OBJECT DevObj, PIRP Irp)
{
  IoCopyCurrentIrpStackLocationToNext(Irp);
  IoSetCompletionRoutine(Irp, CompletionRoutine,
                         context, TRUE,TRUE,TRUE);
  return IoCallDriver(LowerDevice, Irp);
}
```


WDM Driver: Completion Routines

The address of the **completion routine** and the **context variable** are stored in the next stack location.

```
NTSTATUS
CompletionRoutine( PDEVICE_OBJECT  DeviceObject,
                   PIRP   Irp,     PVOID  Context )
{
  NTSTATUS status = Irp->IoStatus.Status;

  if (Irp->PendingReturned)
  {   IoMarkIrpPending( Irp ); }     // !!!

  < postprocessing >
  return STATUS_CONTINUE_COMPLETION;
}
```


WDM Driver: Propagating the Pending Bit

If the completion routine returns `STATUS_CONTINUE_COMPLETION` it must propagate the Pending Bit to its stack location :

```
if (Irp->PendingReturned) IoMarkIrpPending( Irp );
```

This is one of the **strange and error-prone** things in WDM!

If there is no completion routine, propagation is done automatically.

The I/O manager uses the pending bit in the topmost stack location to decide whether an APC Routine is needed or not.

WDM Driver: Synchronous Pass Down

Since `IoCompleteRequest` usually is called in a DPC routine, the completion routine usually runs at `DISPATCH_LEVEL` .

If post processing must be done at `PASSIVE_LEVEL`, the dispatch routine can wait for completion (turning the asynchronous I/O to synchronous):

```
NTSTATUS
CompletionRoutine( PDEVICE_OBJECT  DeviceObject,
                   PIRP   Irp,     PVOID PEvent )
{
  if (Irp->PendingReturned)
    KeSetEvent (PEvent, IO_NO_INCREMENT, FALSE);

  return STATUS_MORE_PROCESSING_REQUIRED;  //  !!!
}
```


WDM Driver: Synchronous Pass Down

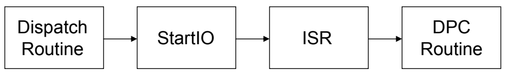
```
NTSTATUS DispAny(PDEVICE_OBJECT DevObj, PIRP Irp)
{
  NTSTATUS status;   KEVENT event;

  KeInitializeEvent(&event, NotificationEvent,FALSE);

  IoCopyCurrentIrpStackLocationToNext(Irp);
  IoSetCompletionRoutine(Irp, CompletionRoutine,
                       &event, TRUE,TRUE,TRUE );
  status = IoCallDriver(LowerDevice, Irp);

  if (status == STATUS_PENDING)
  {  KeWaitForSingleObject(&event, ... );
     status = Irp->IoStatus.Status;       }

  < postprocessing at PASSIVE_LEVEL >
  IoCompleteRequest(Irp, IO_NO_INCREMENT);  // !!!
  return status;
}
```


Cancel-Safe Wait Queue

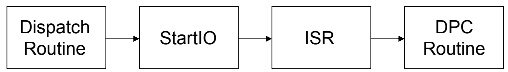
I/O devices typically are much slower then the processor.

So a drivers dispatch routine just initiates the I/O (`StartIo`). When the device completed its operation it sends an interrupt, the drivers ISR (Interrupt Service Routine) gets called.

Usually an ISR performs only the urgent operations, it then queues a DPC as its “Bottom Half” and exits. The DPC routine then does data transfer and IRP completion.



Cancel-Safe Wait Queue



I/O devices typically are much slower then the processor.

So a drivers dispatch routine just initiates the I/O.

If the device is busy, the dispatch routine puts the IRP on a wait queue (`StartPacket()`) and returns `STATUS_PENDING`.

When the previous IRP is completed, the DPC routine de-queues the next IRP (`StartNextPacket()`). If there was an IRP on the queue, it is send to `StartIo`.

Cancel-Safe Wait Queue: Dispatch Routine

Usually the dispatch routine always returns `STATUS_PENDING`.

Whether to queue the IRP or to forward it to `StartIo()`, is decided by `StartPacket()` – this enables correct serialization.

```
NTSTATUS DrvAny(PDEVICE_OBJECT DevObj, PIRP Irp)
{
  IoMarkIrpPending(Irp);
  StartPacket(DevObj->DeviceExtension, Irp);
  return STATUS_PENDING;
}
```


Cancel-Safe Wait Queue: StartPacket

A first simple approach to `StartPacket` (W. Oney calls it „naive” Start Packet) looks like:

```
VOID StartPacket_1(PWDM_DEV_EXTENSION pdx, PIRP Irp)
{
  if (pdx->DeviceBusy) {
     InsertTailList(&pdx->IrpQueue,
                    &Irp->Tail.Overlay.ListEntry);
  }
  else {
    pdx->DeviceBusy = TRUE;
    StartIo(pdx->DeviceObj, Irp);
  }
}
```


Cancel-Safe Wait Queue: StartPacket

W. Oney calls it „naive” Start Packet

```
if (pdx->DeviceBusy)
    InsertTailList(&pdx->IrpQueue, . . . );
```

At least we have to protect access to the queue and to the Busy flag against race conditions. We use a **Spinlock** (can be used at IRQL=2).

Spinlocks raise IRQL to `DISPATCH_LEVEL` on uniprocessor machines and thereby prevent preemption. On multiprocessor machines they additionally **spin** on a busy lock variable, which is **test and set** in one single atomic instruction.

Further on, we include IRP Cancellation as well as we regard important PnP and Power state transitions.

Cancel-Safe Wait Queue: StartPacket

```
VOID StartPacket_2(PWDM_DEV_EXTENSION pdx, PIRP Irp)
 { KIRQL  oldirql;
   KeAcquireSpinLock(&pdx->QueueLock, &oldirql);

   if (pdx->DeviceBusy) {
     InsertTailList(&pdx->IrpQueue,
                    &Irp->Tail.Overlay.ListEntry);
     KeReleaseSpinLock(&pdx->QueueLock, oldirql);
   }
   else {
     pdx->DeviceBusy = TRUE;
     KeReleaseSpinLock(&pdx->QueueLock, DISPATCH_LEVEL);
     StartIo(pdx->DeviceObj, Irp);
     KeLowerIrql(oldirql);          // !!!
   }
 }
```

Cancel-Safe Wait Queue: StartNextPacket

```
VOID StartNextPacket_1(PWDM_DEV_EXTENSION pdx)
{
  PLIST_ENTRY  LiEntr;
  PIRP         Irp;

  if (IsListEmpty(&pdx->IrpQueue)) {
    pdx->DeviceBusy = FALSE;
  }

  else {
    LiEntr = RemoveHeadList(&pdx->IrpQueue);
    Irp    = CONTAINING_RECORD(LiEntr, IRP,
                 Tail.Overlay.ListEntry);
    StartIo(pdx->DeviceObj, Irp);
  }
}
```

53

Cancel-Safe Wait Queue: StartNextPacket

```
VOID StartNextPacket_2(PWDM_DEV_EXTENSION pdx)
{
  PLIST_ENTRY  LiEntr;
  PIRP         Irp;
  KIRQL        oldirql;

  KeAcquireSpinLock(&pdx->QueueLock, &oldirql);

  if (IsListEmpty(&pdx->IrpQueue)) {
    pdx->DeviceBusy = FALSE;
    KeReleaseSpinLock(&pdx->QueueLock, oldirql);
  }

  else { . . . }
}
```

54

Cancel-Safe Wait Queue: StartNextPacket

```
. . .

if (IsListEmpty(&pdx->IrpQueue)) {
  pdx->DeviceBusy = FALSE;
  KeReleaseSpinLock(&pdx->QueueLock, oldirql);
}

else {
  LiEntr = RemoveHeadList(&pdx->IrpQueue);
  KeReleaseSpinLock(&pdx->QueueLock, DISPATCH_LEVEL);
  Irp    = CONTAINING_RECORD(LiEntr, IRP,
               Tail.Overlay.ListEntry);

  StartIo(pdx->DeviceObj, Irp);
  KeLowerIrql(oldirql);
}
}
```

55

Cancel-Safe Wait Queue: Cancellation

An application program can cancel pending I/O requests: `CancelIo()`

`CancelIo` calls the kernel function `IoCancelIrp(Irp)`,
which in turn calls a **Cancel Routine** in the driver.

```
VOID CancelRoutine_1(PDEVICE_OBJECT DevObj, PIRP Irp)
{
  RemoveEntryList(&Irp->Tail.Overlay.ListEntry);

  IoReleaseCancelSpinLock(Irp->CancelIrql);  // !!!

  // IRP Completion
  Irp->IoStatus.Status      = STATUS_CANCELLED;
  Irp->IoStatus.Information = 0;
  IoCompleteRequest(Irp, IO_NO_INCREMENT);
}
```

56

Cancel-Safe Wait Queue: Cancellation

`IoReleaseCancelSpinLock(Irp->CancelIrql);`

Unfortunately `IoCancelIrp(Irp)` uses a global system wide Cancel-Spinlock, which nowadays causes serious performance problems. We cannot provide another spinlock to `IoCancelIrp`.

In the next slides I show, how to work around this problem and safely use **a driver-supplied lock**. **It is really tricky.**

With Windows XP Microsoft introduced Cancel-Safe Queues, that do this in the background. The driver has to provide several callback routines. These cancel-safe queues are available in Windows 2000 too.

Anyhow: I/O cancellation is considered as almost impossible to get right.

57

Cancel-Safe Wait Queue: Cancellation

In the next slides I show, how to use a driver-supplied lock. First we need to know what `IoCancelIrp(IRP)` does.

1. Acquires the global cancel spinlock.
2. Sets the Cancel Flag in the IRP.
3. Sets the Cancel routine (if one exists) to NULL in an atomic interlocked exchange.
4. Calls the Cancel routine if one was previously set. Releases the spinlock otherwise.

This means the spinlock must be released in the cancel routine. A cancel routine for an IRP is installed by

`IoSetCancelRoutine(Irp, CancelRoutine);`

58

Cancel-Safe Wait Queue: Cancel Routine

```
VOID CancelRoutine(PDEVICE_OBJECT DevObj, PIRP Irp)
{
  PWDM_DEV_EXTENSION pdx = DevObj->DeviceExtension;

  IoReleaseCancelSpinLock(DISPATCH_LEVEL);
  KeAcquireSpinLockAtDpcLevel(&pdx->QueueLock);

  RemoveEntryList(&Irp->Tail.Overlay.ListEntry);

  KeReleaseSpinLock(&pdx->QueueLock, Irp->CancelIrql);

  // IRP Completion
  Irp->IoStatus.Status      = STATUS_CANCELLED;
  Irp->IoStatus.Information = 0;      // redundant
  IoCompleteRequest(Irp, IO_NO_INCREMENT);
}
```

59

Cancel-Safe Wait Queue: StartPacketC

```
VOID StartPacket_C(PWDM_DEV_EXTENSION pdx, PIRP Irp)
 { KIRQL  oldirql;
   KeAcquireSpinLock(&pdx->QueueLock, &oldirql);

   if (pdx->DeviceBusy) {
    IoSetCancelRoutine(Irp, CancelRoutine);

    if (Irp->Cancel && IoSetCancelRoutine(Irp,NULL)){
      KeReleaseSpinLock(&pdx->QueueLock, oldirql);
      Irp->IoStatus.Status = STATUS_CANCELLED;
      IoCompleteRequest(Irp, IO_NO_INCREMENT);

    } else {
      InsertTailList(&pdx->IrpQueue, … );
      KeReleaseSpinLock(&pdx->QueueLock, oldirql);
   } else { … StartIo(pdx->DeviceObj, Irp); … }
 }
```

60

Cancel-Safe Wait Queue: StartPacketC

```
if (pdx->DeviceBusy) {
```

The device is busy, we want to put the IRP on the queue and install a cancel routine:

```
IoSetCancelRoutine(Irp, CancelRoutine);
```

But now the IRP can be canceled.

```
if (Irp->Cancel && IoSetCancelRoutine(Irp, NULL))
```

What is this supposed to be ??

```
if (Irp->Cancel)   // IoCancelIrp is/was running
 if (IoSetCancelRoutine(Irp, NULL)){
```

If the result is not NULL, `IoCancelIrp` didn't perform step 3 yet.
It now cannot call the cancel routine, we must complete the IRP.

Cancel-Safe Wait Queue: StartPacketC

```
if (IoSetCancelRoutine(Irp,NULL)){
```

If the result is not NULL, `IoCancelIrp` didn't perform step 3 yet.
It now cannot call the cancel routine, we complete the IRP.

```
KeReleaseSpinLock(&pdx->QueueLock, oldirql);
Irp->IoStatus.Status = STATUS_CANCELLED;
IoCompleteRequest(Irp, IO_NO_INCREMENT);
```

Otherwise `IoCancelIrp` will call, or has already called the cancel routine, but is waiting on our spinlock.
We insert the IRP into the queue, the cancel routine will complete it after we released the spinlock:

```
} else {
   InsertTailList(&pdx->IrpQueue, … );
   KeReleaseSpinLock(&pdx->QueueLock, oldirql);
}
```


Cancel-Safe Wait Queue: StartNextPacketC

The old `StartNextPacket()` did the following:

```
if (IsListEmpty(&pdx->IrpQueue)) {
  pdx->DeviceBusy = FALSE;
  KeReleaseSpinLock(&pdx->QueueLock, oldirql);
}
else { … RemoveHeadList(); … StartIo(); … }
```

Now it is not sufficient to have a queue which is not empty,
we must find an IRP that is not cancelled yet. So we scan the queue:

```
Irp = NULL;
while (!IsListEmpty(&pdx->IrpQueue)) {
  < if (an Irp that is not cancelled) break; >
}

if (IsListEmpty() && !Irp) pdx->DeviceBusy = FALSE;
if (Irp){ … StartIo(); … }
```


Cancel-Safe Wait Queue: StartNextPacketC

We scan the queue:

```
KeAcquireSpinLock(&pdx->QueueLock, &oldirql);
Irp = NULL;
while (!IsListEmpty(&pdx->IrpQueue)) {

  LiEntr = RemoveHeadList(&pdx->IrpQueue);
  Irp = CONTAINING_RECORD(LiEntr, IRP,
               Tail.Overlay.ListEntry);

  < if (Irp is not cancelled) break; >

  if (IoSetCancelRoutine(Irp, NULL)) {
      break;
  } else {    // the cancel routine is running
              // but waiting on our queue spinlock

    InitializeListHead(LiEntr); Irp = NULL;
  }
}
```

Cancel-Safe Wait Queue: StartNextPacketC

We scanned the queue:

```
Irp = NULL;
while (!IsListEmpty(&pdx->IrpQueue)) {
  < if (Irp is not cancelled) break; >
}
```

Now it depends on whether we found an IRP or not:

```
if (IsListEmpty(&pdx->IrpQueue) && !Irp)
               pdx->DeviceBusy = FALSE;
if (Irp){
  KeReleaseSpinLock(&pdx->QueueLock,DISPATCH_LEVEL);
  StartIo(pdx->fdo, Irp);
  KeLowerIrql(oldirql);
} else {
  KeReleaseSpinLock(&pdx->QueueLock, oldirql);
}
```

65

Cancel-Safe Wait Queue: Cleanup

If we can have IRPs waiting in a queue, we must cancel them when the device (the handle) is closed.

The I/O manager sends `IRP_MJ_CLEANUP` just before `IRP_MJ_CLOSE`. We can write a Cleanup-Dispatch routine.

We scan the queue under the protection of our spinlock, look whether IPRs pertains to that handle (`Stack->FileObject`) and the cancel routine is not running yet. If we find such IRPs, we complete them after the spinlock is released:

```
Irp->IoStatus.Status = STATUS_CANCELLED;
IoCompleteRequest(Irp, IO_NO_INCREMENT);
```

66

Cancel-Safe Wait Queue: PnP and Power

Several PnP and Power states indicate, the device cannot serve I/O requests. Depending on the nature of that state, we abort or stall the queue.

Abortion means, no IRP is started, nor is it queued. If the queue is **stalled**, all IRPs are queued, even if the device is not busy.

```
VOID StartPacket(PWDM_DEV_EXTENSION pdx, PIRP Irp)
{ KeAcquireSpinLock(&pdx->QueueLock, &oldirql);
  NTSTATUS abortstatus = pdx->abortstatus;
  if (abortstatus) {
    KeReleaseSpinLock(&pdx->QueueLock, oldirql);
    Irp->IoStatus.Status = abortstatus;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
  } else …
```

67

Cancel-Safe Wait Queue: PnP and Power

Depending on the nature of the state, we abort or stall the queue.

```
VOID StartPacket(PWDM_DEV_EXTENSION pdx, PIRP Irp)
{ NTSTATUS abortstatus = pdx->abortstatus;
  if (abortstatus) { … }
  else if (pdx->DeviceBusy || pdx->stalled) {
    IoSetCancelRoutine(Irp, cancel);
    . . .        // queue this irp

VOID StartNextPacket(PWDM_DEV_EXTENSION pdx, PIRP Irp)
{ NTSTATUS abortstatus = pdx->abortstatus;
  while (!IsListEmpty() && !abortstatus && !pdx->stalled)
  { … }   // find an IRP to start
  if (IsListEmpty() && !Irp) pdx->DeviceBusy = FALSE;
```

68