The NT Insider

The Truth About Cancel - IRP Cancel Operations (Part I)

(By: The NT Insider, Vol 4, Issue 6, Nov-Dec 1997 | Published: 15-Dec-97| Modified: 22-Aug-02)

Introduction

This is a two part article about I/O Request Packet Cancel operations in the Windows NT Operating system.

Part one introduces the subject, explains the mechanisms used by the operating system to support cancel operations, and provides a template for correctly supporting cancel operations in a device driver.

The second part of the article, scheduled for another edition of *The NT Insider*, will explore some of the more interesting features of cancel processing using a test driver and application. This article will provide some insight into how an engineer can explore the NT operating system using drivers as test insertion points and Windbg as an analysis tool. Source code for the driver and test application will be provided.

Cancel Operations

Windows NT provides an *asynchronous I/O* subsystem. Application threads that issue Read, Write or Ioctl operations can do so asynchronously.

In a *synchronous I/O* model, issuing an I/O request blocks the calling thread until the request is complete. The initiation and completion of the I/O request are performed in the thread context that issued the I/O request.

In an *asynchronous I/O* model, the calling thread resumes execution before the I/O request completes, and is notified later that the I/O request has completed. Much of the I/O request processing is performed in an arbitrary thread context. In particular, while the I/O request is initiated in the calling thread context, it is most likely completed in some other, arbitrary, thread context.

To support an asynchronous I/O model, the I/O subsystem in the NT operating system transforms procedural based I/O request system calls into I/O Request Packet (IRP) data structures. The IRP data structures store the stack based parameter information about an I/O request in a form that can persist and be operated on in any arbitrary kernel thread context.

To simplify our discussion of cancel processing, we will describe each asynchronous I/O Request Packet as having three states: *initiated*, *pended*, and *completed*.

An IRP is *initiated* when the NT I/O Manager creates the IRP, stores the initiating threads parameter based description of the I/O request into the IRP, and calls the top level target device driver to process the IRP.

An IRP is *pended* if any driver processing an initiated IRP decides to store the IRP and defer processing of the IRP by marking the IRP as *pending* and queuing the IRP. This corresponds to a driver calling **IoMarkIrpPending(...)**, queuing the IRP either on a driver internal queue data structure or the I/O Manager provided system queue, and returning **STATUS_PENDING** to the driver dispatch entry routine.

An IRP is *completed* when a driver decides that I/O processing for this IRP is finished and that status and data (if applicable) should be returned to the initiating thread. To change the state of an IRP to completed, a driver sets up the status fields of the IRP and calls **IoCompleteRequest()**. For a detailed discussion of completion processing in the NT operating system see the article, "How NT Handles I/O Completion" in the May 1997 issue of *The NT Insider*. Suffice it to say that the NT I/O Manger component of the operating system is responsible for delivering completion status information back to the thread that initiated the I/O request, and for deallocating the IRP that described the I/O request.

Why Cancel?

One thing that may happen after an IRP has entered the *pended* state is that the initiating thread can terminate, leaving an IRP data structure in the operating system for which there is no longer an active user mode thread. Further processing on the IRP makes little sense. In fact, one of the more common reasons why a thread terminates with I/O requests active in the operating system is that the process containing that thread has also terminated. Thus, transferring data and status to the initiating thread is problematic.

In order to clean up the IRPs associated with a terminating thread, the I/O Manager component of the NT operating system provides a Cancel facility. The simplest approach to cancellation would be for the I/O Manager to locate all the IRPs associated with a terminating thread and deallocate them. However, as a pended IRP is likely to be queued on an internal data structure inside some arbitrary device driver, a strategy of simply deallocating these IRPs on thread termination would result in stale pointers inside device drivers. As these device drivers processed the deallocated IRPs, the system would be corrupted and would soon crash and/or corrupt valuable user data.

Instead, the I/O Manager appears to implement a three phase process to cleanup the IRPs associated with a terminating thread.

Three Phased IRP Cancel Processing on Thread Termination

The following discussion of the I/O Manager?s role in cancel processing is based on the available published documentation about the Windows NT operating system and the results of the author?s investigation into cancel processing using a test driver to explore the behavior of the operating system while it is actually canceling I/O operations.

Phase One Cancel Processing is initiated as a thread terminates. Thread termination is handled by the Process Manager (e.g. through the **NtTerminateThread(...)** system call). The Process Manager calls into the I/O Manager at **IoCancelThreadIo(...)** explicitly to allow the I/O Manager to perform IRP cleanup for the terminating thread. It is important to note that the Process Manager is calling the I/O Manger in the thread context of the terminating thread, and that all of the *Phase One* Cancel Processing is performed in the terminating thread context. The thread has not terminated, it is continuing to execute in kernel mode in the I/O Manager. The process that contains the terminating thread also has not terminated, and cannot until the terminating thread has completed *Phase One* and *Phase Two* of the I/O Manager?s cancel processing.

In order for the I/O Manager to cleanup the IRPs associated with a terminating thread, it has to know what IRPs are associated with a terminating thread. This is easily accomplished by the I/O Manager, as it keeps a linked list of IRPs allocated for a thread in the **Thread Object** data structure. *Phase One* Cancel Processing runs the **IrpList** field in the terminating thread?s **Thread Object**, calling **IoCancelIrp(...)** for each IRP on the list.

IoCancelIrp(...) is a documented NT DDK I/O Manger function that acquires the global cancel spinlock, sets the **Cancel** flag of the specified IRP, and calls the **Cancel Routine**, if any, indicated in the IRP **CancelRoutine** field. A driver that has marked an IRP as pending and queued the IRP must also set the **CancelRoutine** field in that IRP to point to a **Cancel Routine** callback function in the driver.

The intention of *Phase One* Cancel Processing is that each driver that has queued an IRP associated with the terminating thread will be called at its **Cancel Routine**. The driver can then remove the cancelled IRP from the driver?s internal queue and return it to the I/O Manager by completing the IRP. We will look at the details of how a driver does this later in this article.

Phase One Cancel Processing concludes when the I/O Manger has run the entire list of IRPs associated with the terminating thread, calling **IoCancelIrp()** for each IRP. The I/O Manger then enters *Phase Two* Cancel Processing.

Phase Two Cancel Processing is a timed wait loop. The I/O Manager, still executing in the terminating thread context, waits for either the Thread Object **IrpList** to be empty, or for five minutes to elapse. If all of the drivers that have IRPs queued for the terminating thread have implemented Cancel support properly, this loop quickly terminates as the I/O Manger completes and deallocates the IRPs, emptying the Thread Object **IrpList** in the process. As an aside it should be noted that the *Phase Two* Cancel Processing runs in the terminating thread context, and so must the I/O Manger?s completion processing for asynchronous I/O requests. It is the latter function, completion processing, that empties the **IrpList**. Consequently, the I/O Manager must be preemptible while executing *Phase Two* or completion can never occur and the **IrpList** will never be emptied.

It is possible that a driver does not implement cancel support correctly (in *Phase Two*) and that as a consequence the I/O Manager, rather than quickly finding that the **IrpList** is empty, times out after five minutes with the Thread Object **IrpList** still containing IRPs. Remember that the terminating thread, and consequently the associated process, are still executing, and therefore have not finished terminating.

The I/O Manager has a dilemma. It still cannot safely deallocate the IRPs associated with the terminating thread, but it is equally unacceptable to keep the terminating thread from terminating. (Consider the case where this is an administrative request to terminate a process or shutdown the system).

The I/O Manager appears to resolve this dilemma by entering *Phase Three* Cancel Processing. After five minutes each IRP still remaining on the Thread Object?s **IrpList** is removed from this list, and as much as possible the resources associated with this IRP are deallocated. The IRP itself is not deallocated, but is disassociated with the terminating thread. Instead we can think of this IRP as being in limbo, neither dead nor alive. The IRP has no associated thread, so there is no thread to return completion status to if it ever does complete. It cannot be deallocated, as some driver somewhere may still have a pointer that references the IRP. The terminating thread however is finally allowed to complete its termination processing.

In addition to the IRP itself, the I/O Manager cannot fully dispose of the backing store for the user?s data buffer either, as the same driver that has not correctly cancelled the IRP may in fact decide to transfer data to the backing store for the user?s data buffer. These *Phase Three* IRPs appear to remain unusable until they are finally completed by the driver that has them queued or the system is reset. Any physical pages of memory allocated as non-pageable backing store for the user?s data buffer also remain unusable until the IRP is completed or the system is reset.

As part of *Phase Three* Cancel Processing, the I/O Manager attempts to notify users or administrators that there has been a problem by calling **ExRaiseHardError(...)**. This function can result in a Popup Dialogue Error Message on the Win32 Display. The offending driver is indicated by the error message. My own test results indicated that **ExRaiseHardError(...)** was always called, but that the dialogue never showed up.

Clearly correct driver cancel processing is essential. It is unacceptable to cause a thread to delay five minutes before terminating. It is even more unacceptable to consume valuable system resources like physical pages of memory and IRP data structures when there is no possibility of performing useful work. The remainder of this article looks at the correct methods for supporting Cancel Processing in an NT kernel mode driver.

Supporting Cancel Processing in a Kernel Mode Driver

This discussion applies only to a kernel mode driver that uses *internal* (or *own*) queuing rather than using the I/O Manager?s *system* queuing facility. However, in a broad sense (i.e. not the details), the discussion applies to both types of kernel mode drivers.

There are three steps to cancel processing in a driver. First, a driver has to setup cancel processing (typically in a dispatch entry routine), before changing an IRP to *pending* state and queuing the IRP internally. Second, the driver has to support one or more cancel routine callback functions. Finally, a driver must undo everything it did in step one if it is performing normal completion processing on an IRP instead of cancel processing.

```
NTSTATUS
CancelReadWrite(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    )

{
    PDEVICE_EXTENSION devExtension = (PDEVICE_EXTENSION) DeviceObject->DeviceExtension;
    KIRQL irql;

    //
    // mark the irp pending NOW before we queue this IRP
    //

    IoMarkIrpPending(Irp);

    //
    // serialize all driver activity for this device object
    //

    KeAcquireSpinLock(&devExtension->lock, &irql);

    //
    // set the cancel routine
    //

    IoSetCancelRoutine(Irp, CancelCancel);

    //
    // queue the IRP
    //

    InsertTailList(&devExtension->irpList, &Irp->Tail.Overlay.ListEntry);

    //
    // release the spinlock
    //

    KeReleaseSpinLock(&devExtension->lock, irql);

    //
    // always return status pending -
    // note that we might very well have already completed,
    // or cancelled this IRP before we get here.
    //

    return STATUS_PENDING;

}
```

Figure 1 -- Setting a Cancel Routine

Figure 1 shows a template for setting a cancel routine. An extremely simple dispatch entry point that supports both **IRP_MJ_READ** and **IRP_MJ_WRITE** operations queues the input parameter Irp IRP to an internal queue. (Some other mechanism not shown must propagate execution of the I/O request). To support cancel processing this function sets the CancelRoutine field in each IRP that it queues by calling the I/O Manager interface **IoSetCancelRoutine(...)**. As of NT4.0, a driver that uses internal queuing for IRPs should not acquire the global cancel spinlock before calling **IoSetCancelRoutine(...)**. In our code sample we acquire an internal spinlock that covers our own private Device Extension data structure before setting the cancel routine and queuing the IRP. In reality there is no particular reason to call **IoSetCancelRoutine(...)** with our spinlock held, nor, if we are executing in the thread context that issued the I/O request, need we worry about any race condition between setting the cancel routine and queuing the IRP.

Having queued the IRP, two things can happen: the IRP can be cancelled or the IRP can be completed normally. Once we have set the **CancelRoutine** field in an IRP, we can be called at our cancel routine function by the I/O Manager in order to cancel the IRP. *Figure 2* shows a sample cancel callback function.

```
VOID CancelCancel (
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp
    )

{
    PDEVICE_EXTENSION devExtension = DeviceObject->DeviceExtension;
    PLIST_ENTRY nextEl = NULL;
    PIRP cancelIrp = NULL;
    KIRQL irql;
    KIRQL cancelIrql = Irp->CancelIrql;

    //
    // release the cancel spinlock now
    //

    IoReleaseCancelSpinLock(cancelIrql);

    //
    // A thread has terminated and we should find a
    // cancelled Irp in our queue and complete it
    //

    KeAcquireSpinLock(&devExtension->lock, &irql);

    //
    // search our queue for an Irp to cancel
    //

    for (nextEl = devExtension->irpList.Flink;
        nextEl != &devExtension->irpList; )

    {

        cancelIrp = CONTAINING_RECORD(nextEl, IRP, Tail.Overlay.ListEntry);
        nextEl = nextEl->Flink;
        if (cancelIrp->Cancel) {

            //
            // dequeue THIS irp
            //

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

            //
            // and stop right here
            //
            break;

        }
        cancelIrp = NULL;

    }
   KeReleaseSpinLock(&devExtension->lock, irql);

    //
    // now if we found an irp to cancel, cancel it
    //

    if (cancelIrp) {

        //
        // this is our IRP to cancel
        //

        cancelIrp->IoStatus.Status = STATUS_CANCELLED;
        cancelIrp->IoStatus.Information = 0;
        IoCompleteRequest(cancelIrp, IO_NO_INCREMENT);

    }

    //
    // we are done.
    //

}
```

Figure 2 – Cancel Routine

There are several things that should be noted about the sample cancel callback function. First, this function releases the global cancel spinlock before searching for the cancelled IRP. While it is not necessary to do this (we could release the global spinlock after the search (but before completing the IRP)), there is no reason to hold this spinlock in this function. It should be released as implemented in the sample. In order to release the global spinlock we need the correct IRQL value as the input parameter to **IoReleaseCancelSpinLock(...)**. This is the IRQL that the I/O Manager was running at when it acquired the cancel spinlock (by calling **IoCancelIrp(...)**). Luckily the correct value has been stored in the IRP input parameter to our cancel routine. **Irp->CancelIrql** is the correct value to supply.

We of course have to acquire our own spinlock for our device object?s Device Extension before we can search our internal queue for an IRP to cancel. However, this routine does *not* search our internal queue for the IRP passed in as an input parameter. Instead it searches for the first IRP on the queue that has the **Irp.Cancel** flag set. It de-queues this IRP, terminates the search, and completes the IRP it has found with **STATUS_CANCELLED**.

The cancel routine does not cancel all IRPs that have the **Irp.Cancel** flag set. It does not restrict itself to canceling only the IRP supplied as an input parameter. It does not hold the global cancel spinlock while searching its own internal queue for an IRP to cancel. Finally, the routine cancels the *first* IRP it finds, if any, with **STATUS_CANCELLED**. At OSR we are of the opinion that these are the correct rules for cancel processing, but the second part of this article will explore *why* we think these rules are correct, and perhaps if they are correct at all.

But wait, we are not quite done. There is one more step that a driver *must* perform to correctly support cancel processing. A driver must undo what was done initially (in the dispatch routine in our sample code). Specifically a driver is required to clear the cancel routine in an IRP that the driver is removing from his internal queue. There really are two cases where this applies. First, a driver could be completing the IRP, secondly the driver could be passing the IRP down to another driver (The case where the IRP is in effect queued on the hardware will be handled separately in the second part of this article). In either case it is a potentially fatal error for the driver to not clear the **IRP.CancelRoutine** field.

To accomplish this final step in cancel processing a driver calls the **IoSetCancelRoutine(...)** supplying **NULL** as the value of the cancel callback routine (*Figure 3*).

```
//
// always remove cancel routine
// or we bugcheck in free build, assert in checked
//

(void) IoSetCancelRoutine(Irp, NULL);

//
// indicate we are finished
//

Irp->IoStatus.Status = STATUS_SUCCESS;

//
// no actual data transfer.
//

Irp->IoStatus.Information = 0;

//
// complete the request
//

IoCompleteRequest(Irp, IO_NO_INCREMENT);
```

Figure 3 -- Normal Completion

Moving Forward

The next part of this article will explore the details of cancel processing, how to examine the internal state of the operating system using the available tools, and why we think the rules for correct cancel processing are as stated above.