The NT Insider

A Modest Proposal - A New View on I/O Cancellation

(By: The NT Insider, Vol 7, Issue 3, May-Jun 2000 | Published: 15-Jun-00| Modified: 05-Aug-02)

Probably one of the most difficult pieces of functionality to get right in any Windows NT/2000 driver is IRP cancellation. Cancel handling has also been the subject of a great deal of controversy, much of which has generated heat and little of which has generated any new light. In this article, we?ll first examine some of the issues traditionally involved in cancel processing. Then I?ll issue a new, and most likely controversial, proposal for handling IRP cancellation in NT V4 and Win2K drivers.

In this discussion, we?re going to restrict ourselves to discussing IRP cancellation for device drivers that queue their own IRPs. Therefore, we?re explicitly ignoring the issue of cancellation for drivers that use system queuing. Though we use the term ?NT? throughout this article, our comments apply equally to NT V4 and Win2K systems, in drivers that do and do not support PnP.

How NT Handles I/O Cancellation

Each time a user thread issues an I/O request, the IRP that represents that request is placed on a queue. The list head for this queue is in the **ETHREAD**, at offset **IrpList**. When the IRP is ultimately completed by IopCompleteRequest(?) (which is called as a result of **IoComplete Request(?)** finishing) the IRP is removed from the IrpList.

When a thread is aborted or exists, the I/O Manager examines the IrpList to see if there are any IRPs outstanding. If there are, the I/O Manager runs the IrpList, and calls the cancel routine (if there is one) contained in each IRP on the list. The I/O Manager then waits until all the IRPs on the IrpList have been completed (and removed from the list), or for five minutes, whichever comes first. If, after five minutes, any IRPs remain on the IrpList the I/O Manager pops up an ominous looking dialog box that reads ?The driver so-and-so failed to complete a cancelled I/O request in the allotted time.?

Not having all the outstanding IRPs for a thread complete within the requisite timeout period is bad. It represents an I/O rundown failure. While the thread associated with the I/O request *is* allowed to exit, it is important to recognize that the pages referenced by the uncompleted I/O requests will remain locked in memory (until the IRP completes? forever if necessary). And, while the locked pages could be lost for future use by the system, at least there?s no danger of memory being corrupted by an IRP that completes after the five minute I/O rundown interval.

The Problems with IRP Cancellation

IRP Cancellation in NT has always been fraught with problems. These problems typically involve race conditions, either between cancellation and I/O initiation or cancellation and IRP completion. These races typically result in bug checks for multiple IRP completions, in addition to potentially corrupted user memory.

NT?s initial attempt at helping driver writers deal with these races was the system-wide **Cancel Spin Lock**. This lock was originally to be held any time the cancel state of an IRP was changed. Since NT V4, the use of the Cancel Spin Lock for drivers that do their own queuing has been reduced as a performance optimization. It is no longer necessary to hold this lock when manipulating IRP cancel routines, for example.

An Epiphany

After thinking about this problem, and dealing with it in my own drivers, over the course of the past few years I?ve come to some new conclusions about IRP cancellation. Interestingly enough, these conclusions are in line with the guidance provided in the DDK. Will wonders never cease?! However, given the discussions I?ve seen on Usenet, my new view does not represent the conventional wisdom on this topic. Little surprise there, eh?

It is my contention that most driver writers either create, or at least exacerbate, their own problems with IRP cancellation. How? Mostly, they create problems because they try too hard. They work to cancel IRPs that, frankly, don?t even really need to be cancelled. This conclusion is based on the following facts:

- The I/O Manager will wait for up to five minutes for IRPs that have been cancelled to complete. During this wait, and even thereafter, the pages that represent the user?s I/O buffer are not removed from memory. There is, therefore, no chance of memory corruption during this period.

- I/O cancellation is a relatively rare event. Therefore, on the rare occasion that I/O cancellation does occur, it does not have to occur either quickly or efficiently.

- I have never, and I do mean not once in my 20+ years of writing device drivers, seen a non-trivial hardware device (such as a DMA adapter) on which an in-progress I/O request can *always* be cancelled without difficulty. Sure, some adapters can cancel active transfers *most* of the time. But I have never once seen a device that can?t be hung by I/O cancellation at some point under load.

Given this experience, I would recommend avoiding canceling in-progress I/O requests whenever possible. Hey, if it?s *in progress*, it?s gonna complete right quick, isn?t it? Usually, yes. If not, you have a slow device. So, you can take your time, and hammer the device?s state. Maybe even totally reset it if that?s what it takes to make the cancel work. But, in any case, you?re doing this with time on your side. As I said above, this doesn?t have to occur either quickly or efficiently.

Too many novice NT driver writers ? and more than a few journeymen ? get the mistaken impression that they must perform I/O cancellation quickly (hurry! hurry! Do it *now*! The I/O Manager is *waiting*!) as a condition of writing a proper device driver and to avoid memory corruption. This is unabashed nonsense. To quote the DDK:

> *Any driver in which IRPs can be held pending for an indefinite interval must have one or more Cancel routines. Conversely, if a driver can ensure that it will never queue more IRPs than it can complete in five minutes, it probably does not need a Cancel routine.*

Trying to handle I/O cancellation quickly or trying to handle nasty edge-conditions in I/O cancellation is what gets people into trouble.

Sure, when a user invokes a program from the command prompt that talks to your driver, and then hammers control-C to cancel it, you want the command prompt to return (signaling program exit) in a reasonably short time. But, if that?s all we?re doing cancellation for, we?re dealing with user perceivable time, measured in seconds. It seems to me that this gives us plenty of time to cancel any pending IRPs without resorting to difficult algorithms.

A Modest Proposal

It is therefore my proposal that driver writers deal with I/O cancellation in device drivers using the following rules:

1. If your driver completes all requests that it receives within a reasonably short length of time (not more than a few seconds, might be a good guideline) then your device driver probably doesn?t need a cancel routine. Save yourself the time and trouble. Don?t write one. If you don?t write such a routine, you certainly won?t have to deal with any race conditions between cancellation and completion!

   An example of a device that doesn?t need a cancel routine might be a high performance disk array. I?m thinkin? that any request a driver for such a device receives will be completed by the adapter within, oh, say, a few hundred milliseconds, *tops*. There is just no reason to bother with IRP cancellation in this situation.

2. If your driver holds requests on an internal queue or on a device for very long periods of time (more than many seconds might be a good guideline) then you?ll need a simple cancel routine. Write a nice simple one that focuses precisely on this situation: Canceling a long-pending request. Design your driver so that such requests are relatively easy to cancel.

   The DDK lists the keyboard driver as an example of such a driver. I would also mention the serial port driver. It?s possible to hang a read on a serial port, and have the request never complete? because no data arrives for input at the serial port. I?m thinkin? you?ll have to

cancel such a request. But, then again, canceling this request won?t be a big deal, right?

3. If your driver falls between the two extremes described above, it might hold requests in progress for ?more than a few seconds? but not for ?more than many seconds? (whatever those values mean). In this case, merely concentrate on handling the easy cancel cases. Go for the ?low hanging fruit.? So, maybe you miss a request occasionally and that results in an I/O request that takes many seconds to complete. Oh well. Still, no harm done.

4. If you do implement cancellation, you have to do it right. That means, no multiple IRP completions? no blue screens. Don?t mistake my advice here for advocacy in favor of people writing stupid, sloppy, code that only works ?most of the time.? Whatever you write has to work.

Most devices will likely fall within case number 1, above. Therefore, lots of driver writers who today are convinced that they have to be become experts on I/O cancellation should thus be able to sleep a whole lot easier.

All driver writers need to keep in mind that cancellation is a rare event. This means that even when you *do* have to handle cancel, you don?t have to handle it either quickly or efficiently. Remember, tailor cancellation logic to your device. If you have a very fast device, where chances of race conditions are greatest, you probably don?t need a cancel routine at all. If you have a very slow device, where requests can pend for long periods of time, your cancel logic can be similarly slow and sure-footed.

Once you understand that handling cancellation is a matter of tidiness more than a matter of proper system operation, you can start to relax about handling cancel. Either forget about it, or do only the easy cases, or implement some heavy weight, cumbersome, cancel logic that?ll work but will take a butt-load of time. In either case, your life should be much easier. And the time you save stressing over your cancel routine will be better put toward making your driver work properly in the really common cases.