The Old New Thing

Why does my asynchronous I/O complete synchronously?

23 Sep 2011 7:00 AM | 36

A customer was creating a large file and found that, even though the file was opened with `FILE_FLAG_OVERLAPPED` and the `WriteFile` call was being made with an `OVERLAPPED` structure, the I/O was nevertheless completing synchronously.

Knowledge Base article 156932 covers some cases in which asynchronous I/O will be converted to synchronous I/O. And in this case, it was scenario number three in that document.

The reason the customer's asynchronous writes were completing synchronously is that all of the writes were to the end of the file. It so happens that in the current implementation of NTFS, writes which extend the length of the file always complete synchronously. (More specifically, writes which extend the *valid data length* are forced synchronous.)

We saw last time that merely calling `SetEndOfFile` to pre-extend the file to the final size doesn't help, because that updates the file size but not the valid data length. To avoid synchronous behavior, you need to make sure your writes do not extend the valid data length. The suggestions provided in yesterday's article apply here as well.

Blog - Comment List MSDN TechNet

Comments

David Walker
23 Sep 2011 8:18 AM
#

The fact that I/O to compressed partitions are completed synchronously is one of the reasons that data files which contain SQL databases should not be on compressed partitions. SQL server really, really wants its I/O to be asynchronous. And the performance of SQL server with fully synchronous I/O would likely be poor.

In the old days, before compressed SQL backups, I tried to write a SQL backup to a compressed partition. It didn't work (delayed I/O failed).

Dave
23 Sep 2011 9:02 AM
#

From that KB article: "... because the pool of worker threads is limited (currently three on a 16MB system) ..."

Ah, 16MB systems, now those were the days! Wait, no they weren't.

Gabe
23 Sep 2011 10:16 AM
#

Does anybody know why these operations are not completed asynchronously?

Leo Davidson
23 Sep 2011 10:18 AM
#

So if you open a new, empty file and use async I/O to write data into it, all of that will become synchronous?

Async I/O seems so caveat-ridden, and almost impossible to get every possibility handled correctly (from previous posts on the subject), that I'm more and more convinced it's usually better to just spawn a thread (or pool) and use sync I/O.

Of course, that doesn't apply to every situation (and most situations don't need async at all), but I'd rather have slightly higher overheads and correct code than incorrect code.

Simon Farnsworth
23 Sep 2011 10:48 AM
#

@Gabe:

I can make some educated guesses. For compression and encryption, I would guess that the filesystem driver is simply unable to handle more than one outstanding request at a time for any one file, so going asynchronous has high costs (locking issues) and no benefits. Similar reasoning applies to extending a file, plus the additional factor that the intended consumers of async I/O typically don't extend files.

Data in cache is similarly obvious - giving you a mapping to the data in cache immediately is cheaper than handling the async completion - why pay a performance penalty when the point of async is to speed things up?

Finally, the data not in cache makes sense if you see async as a way to issue multiple requests to the hardware (which can normally handle multiple requests in parallel with a performance boost - SCSI TCQ, SATA NCQ, RAID arrays etc). No point going async if you're not getting the parallelism - Windows might as well block you as spawn a new thread that gets immediately blocked anyway.

Simon Farnsworth
23 Sep 2011 10:52 AM
#

@Leo Davidson:

It looks like Windows async I/O is meant to let you take advantage of hardware parallelism, not to act as a true async mechanism. A SCSI TCQ device can potentially

handle millions of simultaneous outstanding requests (e.g. a FC-AL attached RAID array with terabytes of cache RAM and petabytes of spinning disk). Async I/O lets you issue multiple I/O requests from one thread, as long as the software can extract parallelism from them (i.e. convert them into multiple requests to hardware); as soon as it has to sequence things, it goes synchronous, as there's no benefit to multiple outstanding requests if they must be handled one at a time.

jas88
23 Sep 2011 1:03 PM
#

@Leo Davidson: That's the scenario I had in mind: I can easily imagine a developer getting bitten by this, using overlapped IO to log activity thinking this will minimize the performance impact of log writes, when in fact every write will be converted to a synchronous one. I suspect it will actually only happen on writes which cross a cluster boundary, in reality (NTFS will be zero-filling each cluster as you start writing into it; overlapped writes within a cluster already in use won't have that issue). For that matter, storing your log files compressed seems an "obvious" improvement many system administrators might well make - then wonder why their server application is much less responsive, since it has suddenly had all its logging operations converted into synchronous ones - and the developer had carefully worked around that issue by slotting in occasional SetFileValidData calls to amortize the file expansion by using 1 or 10 Mb chunks and reduce file fragmentation at the same time.

The virtual memory system faced the same issue, but addressed it by pre-zeroing a pool of pages in the background for later use. Presumably it's the need for metadata updates (volume allocation bitmap changes, journaling and MFT record changes) which made the NTFS developers make write-extends synchronous - ironic in a way, since the extra time taken would make that particular operation an obvious candidate for asynchronous operation. With such a limited pool of threads, though, like the 3 mentioned, writes would sometimes tie up those precious resources for a while as the disk buffer is flushed (to ensure on-disk consistency with the journal).

Alex Grigoriev
23 Sep 2011 3:19 PM
#

There is a good reason for serialization of these operations.

1. All metadata modifications are serialized. File expansion needs metadata modification. While the file zeroing goes forward, all other IO to that area need to be held, too.

2. Creating new isle in a sparse file also needs metadata modification.

3. Compressed file writes need to be serialized because they produce unknown amount of hard data and need filesystem modifications.

Gabe
23 Sep 2011 3:40 PM
#

Why would the need to serialize the operations mean that they have to be synchronous? I don't see the difference between a single thread with multiple simultaneous outstanding async operations and multiple threads with simultaneous sync operations.

waleri
24 Sep 2011 12:32 AM
#

What will happen with completion port whern async I/O is converted to sync?

Simon Farnsworth
24 Sep 2011 3:49 AM
#

@Gabe:

Been thinking about this overnight (yes, I'm dull). If you see the async operations as a way to cheaply exploit command queueing, it makes sense. When the OS can convert an operation directly into a disk request, all it needs do is remember "when this I/O request completes on the disk, signal this completion"; it has to have something like that internally anyway to permit it to block one thread, execute another, and resume the first thread once its I/O has completed.

Adding asynchronous handling of serialized requests means adding fresh complexity - you need to spawn a thread, and ensure that the thread you've spawned simply runs the request synchronously and then signals the completion. However, you've now moved away from your "cheap method to exploit hardware parallelism"; you're paying the costs of spawning and destroying a thread for operations. Far easier to just not bother - if you wanted true asynchronous behaviour, you'd spawn threads anyway; without this async I/O interface, there's no way to say "if the hardware lets me issue parallel requests, I want to be asynchronous; if I'd only be exploiting software parallelism, I want to free up resources for CPU-bound processing".

JM
24 Sep 2011 8:27 AM
#

There is only a problem (from a developer POV) if operations that take a significant amount of time are performed synchronously when you wanted them to be performed asynchronously. A cache write that immediately completes is fine. There is no parallelism to exploit because the operation takes no time; you can simply go on and do whatever you intended to do while the I/O was underway. But a disk operation that completes in milliseconds is an eternity to unexpectedly block, especially if you intended to do something non-I/O related inbetween. This is not usually a problem because asynchronous I/O is usually mixed with yet more asynchronous I/O, not calculations that have nothing to do with keeping I/O flowing, which are handled by other threads.

As Simon points out, although the OS could jump through hoops to make sure an async operation is always async, this doesn't pay off. Making an inherently synchronous operation asynchronous requires overhead (although not as dire as creating and

destroying threads -- you can maintain a thread pool for it). It's intended to save resources, not to support a particular programming model. It's up to the programmers to exploit it meaningfully. If necessary, you can always layer forced asynchronous completion over synchronous operations yourself with use of the thread pool -- but since you can't reliably predict when synchronous processing will kick in, this means all operations will have to go through an extra layer. It's better to write your code in such a way that it's still correct in every way (especially with regards to timing and responsiveness) even if all I/O were synchronous, and treat asynchronous I/O as a good optimization.

Gabe
26 Sep 2011 5:52 AM
#

In the case of writing to a compressed file I can understand not spawning a new thread to perform compression. But once the data has been compressed, why not make the actual writing of the data asynchronous? At that point the CPU work has been completed and all that's left is the I/O. Does that I/O still need to be done synchronously?

Confused Developer
26 Sep 2011 2:08 PM
#

So then what is the right way to handle overlapped I/O to files on NTFS to ensure that your limited I/O threads are never blocked waiting for platters?

If the operating system itself had done the right thing and implemented serializing under the hood (while preserving async interface/behavior) it would have resulted in less error prone software and less need for each developer to reinvent the wheel.

This is the first time I've become aware of this and I can imagine there must be a lot of code out there that expects Windows to always complete I/O calls asynchronously if they will not complete immediately. All that code will probably cause unintended blocking/performance problems if used against compressed volumes.

I think this is a bad implementation decision on the part of MSFT.

cheong00
26 Sep 2011 6:46 PM
#

@Confused Developer: Why do you think it is so important? I don't think it's right to store frequently updating files in compressed folder. If they do, the performance penalty is expected (even though it is greater than absolute minimum).

Think about it, since Windows does not support CPU quota, if your little program continuously throwing "little write to large compressed file" to kernal I/O process, it would have been easy to create some form of DOS attack. (Remember NTFS compression is not in chunked format, so even altering 1 byte might require to decompress large part or even the whole file, alter the byte, and compress it back. It could be a quite expensive

task.)

Crescens2k
26 Sep 2011 9:18 PM
#

Confused Developer:

The biggest issue is that with NTFS compression, you can't guarantee that one IO operation will really result in one IO operation.

NTFS uses a form of LZ compression, and that means it does the compression in chunks of data. From what I read somewhere, it implements it so that it takes a few clusters (I think it is 16 clusters), compresses it and then stores it compressed in those cluster leaving the gap between the end of the compressed data and the next chunk as sparse. So it could turn out like

uncompressed

|data....|data....|data....|

compressed

|cdata.. |cd.. |cdata |

Where the spaces are sparse and just not in use. For reads this means two things. If you just want to read part of the file, you have to read the entire chunk to get the data you want. If the data you want spans two chunks, then you need two seperate read/decompress operations to do what you want. This is different to the uncompressed file since if you only want to read 2 bytes then it would only have to read a sector off of the disk, and if what you wanted spanned two sectors, you could read both in a single operation. (This is ignoring some hard disk hardware limitations though, but they just confuse things).

For writes this gets worse, since if you wanted to update only a couple of bytes, then you would need to read the entire data chunk, decompress, update, compress and then write. If the data spans more than one chunk then it needs to do this for each chunk affected.

So yes, while making this kind of operation may seem like a performance hit, allowing compressed file operations to be async doesn't fix anything. Async works because Windows just exploits what is available, but for operations like compressed file operations, everything changes drastically. It is no longer the case of filling out a buffer and queuing it up in the IO manager, it becomes an entire thread of work. Windows is bloated enough as it is, do you want even more. You could also run into other errors because compressed file IO is much slower than regular IO, so you could end up running out of memory because the queue got too long or something. So all you managed to do is displace the problems.

I am also curious as to how allowing async compressed file operations would make code less error prone. A normal async file write could be

```
if(!WriteFile(/*parameters*/)) //returns nonzero if completed immediately
{
   if(GetLastError() != ERROR_IO_PENDING)
```

```
  {
     //i personally use a switch here, adding cases
     //only for errors that need special attention, but
     //usually leaving everything go to default
     //an error like insufficient disk space occured
     //handle these cases
  }
}
```

Because any other error can occur and get returned via the WriteFile return value, you can't skip the check (even though people do seem to assume that file IO can't fail), so regardless you will be checking the return. So by an operation being async instead just means that the function will return an error an you have to look for it, or if it isn't async then it means the function will block and return success. So how does this make anything less error prone? Nothing would have changed from your regular async file handling.

But async IO is really there to allow you to cheaply exploit what is available in hardware. If that isn't possible because it requires multiple operations then Windows wont do it. If you look at some other operations which are never async then this should be apparent. There is a major difference between adding to a queue in the IO manager and having to create a thread to possibly do multiple file operations and all of the related work after all.

pattern
26 Sep 2011 11:49 PM
#

How about this pattern?

```
SetLastError(0);
WriteFile(/*Params*/);
switch(GetLastError){
default: // All the myriad other error codes... Maybe single out some for special processing
FatalAppExit(0,"Bang!");
abort();
case 0: PostQueuedCompletionStatus(/*params*/) or QueueUserAPC(/*params*/)
case ERROR_IO_PENDING:
}
```

cheong00
26 Sep 2011 11:52 PM
#

@Crescens2k: Ah... you're right. NTFS compress file into compression unit hence it's chunked. I must have mixed it up with something else.

Deduplicator
27 Sep 2011 12:14 AM
#

Crescens2k:"But async IO is really there to allow you to cheaply exploit what is available in hardware. If that isn't possible because it requires multiple operations then Windows wont do it. If you look at some other operations which are never async then this should be apparent. There is a major difference between adding to a queue in the IO manager and having to create a thread to possibly do multiple file operations and all of the related work after all."

@^: Does not look like Windows would have to create any new threads for that work. It could all be done on only one thread per core, possibly inheriting the priority of the highest outstanding request.

Aside from that, it is often NOT used to exploit hardware parallelization, but to make the Server/GUI/whatever responsive, and that is quite more difficult if you cannot know if Windows forces you synchronous, instead of serializing behind the scenes and out of your thread in kernel-land.

Next: One has to beware unintended deep recursion when requests return immediately and are acted upon by invoking the handler.

chaong00: "Think about it, since Windows does not support CPU quota, if your little program continuously throwing 'little write to large compressed file' to kernal I/O process, it would have been easy to create some form of DOS attack"

@^: Well, so all the work has to be debited from the originators ledger? That should be done anyway!

JM: "There is only a problem (from a developer POV) if operations that take a significant amount of time are performed synchronously when you wanted them to be performed asynchronously. [...] This is not usually a problem because asynchronous I/O is usually mixed with yet more asynchronous I/O, not calculations that have nothing to do with keeping I/O flowing, which are handled by other threads.

As Simon points out, although the OS could jump through hoops to make sure an async operation is always async, this doesn't pay off. Making an inherently synchronous operation asynchronous requires overhead (although not as dire as creating and destroying threads -- you can maintain a thread pool for it). It's intended to save resources, not to support a particular programming model. It's up to the programmers to exploit it meaningfully. If necessary, you can always layer forced asynchronous completion over synchronous operations yourself with use of the thread pool -- but since you can't reliably predict when synchronous processing will kick in, this means all operations will have to go through an extra layer. It's better to write your code in such a way that it's still correct in every way (especially with regards to timing and responsiveness) even if all I/O were synchronous, and treat asynchronous I/O as a good optimization."

@^: There are often two or more different limiting factors, so there is a problem, even if all is ordered asynchronously: e.g. File IO, Remote File IO 1..N, Vanilla Networking, Calculating, GUI, ...

And the programmer might not (be able to) know, what all is involved.

And its nice that you celebrate working around the quagmire which should not be there as prudent even in the absence of said hazard.

Simon Farnsworth
27 Sep 2011 1:05 AM
#

@Deduplicator:

Look at it this way; if you use synchronous I/O and threads, you never block (what you want for responsiveness). If you want to use synchronous I/O and threads to exploit the inherent parallelism in a big RAID array (say a nice EMC SAN), you need tens of thousands of threads, which literally exist just to issue an I/O asynchronously to the main thread.

Async I/O lets you get at a middle ground that's otherwise unavailable - the statement you make by using async I/O is "if there is hardware parallelism available, let me at it; otherwise, I'm happy going synchronous". Remember, it's designed for services like SQL Server; the pattern it's best at is "lots of I/O so that I can process for a bit, then back to lots of I/O". It's just that it issues all the I/O in parallel where possible, getting you a free speed-up as compared to synchronous I/O.

Think, for example, about a small RAID-1 array with 2 disks in it; that array is guaranteed to be able to handle two simultaneous reads without conflict, as it has two spindles; a RAID-10 with 4 disks may be able to handle two reads and two writes (depending on disk locations) simultaneously. Bigger arrays with more spindles can handle more IOPs in parallel. Async I/O lets you say "please issue lots of I/O if possible; I'll handle collating it back together later".

Deduplicator
27 Sep 2011 2:21 AM
#

@Simon:

So you are saying, it's nice you have to go needlessly multithreaded and accept all the gotchas involved, even though everything could be easily done using guaranteed user-level async calls?

Damn anyone who uses async as it reads (and works most of the time) to the pit of spuriously non-responsive and subtly broken code? (ie: the WinXP-Explorer Guys and anyone else who uses the Shell-Namespace [GetOpenFileName anyone?])

Simon Farnsworth
27 Sep 2011 3:21 AM

#

@Deduplicator

The only way to implement "guaranteed user-level async calls" is to go multithreaded and accept all the gotchas involved. Whether the OS provides this as a standard library, or whether you implement it yourself, the problems are exactly the same.

As a result, you could write yourself a library that spawns off a small number of threads that just do I/O, working on a queue of requests from the main thread, and queueing responses back to the main thread. This would be identical to what you're asking for.

However, if the async I/O interface becomes that library, you have two problems to fix:

1. Older code that uses async I/O may not be expecting it to suddenly make your process multithreaded, and could therefore be caught out by the gotchas you mention.

2. There is now no way for a process to request I/O to be done asynchronously if and only if the hardware can handle the parallelism for you - any async I/O request that can't be parallelised in hardware converts to a thread handling the request, so you can't use the async I/O facility to issue parallel I/O, blocking whenever you're reached the hardware parallelism limit for this system.

In short, why do you want to take away a useful facility for no gain to anyone?

Deduplicator
27 Sep 2011 3:42 PM
#

The only way to implement "guaranteed user-level async calls" in user-mode on top of maybe async and/or explicit sync calls is certainly going multithreaded. But that's not what I asked for. What I asked was why there is no way to have really async calls without going multithreaded. I haven't seen an answer, which doesn't say go multithreaded or tough luck, that's the way it is, therefore it's good.

Also, its extremely counter-intuitive that non-blocking methods block for minutes or even longer.

> [*We've seen earlier that it's possible for async calls to complete synchronously, even if everything supports asynchronous I/O, so you have to be ready for it one way or another. (And if you avoid using overlapped writes to extend files, that avoids the worst case.) -Raymond*]

Gabe
27 Sep 2011 9:47 PM
#

The problem isn't when your async call completes synchronously (as that happens all the time anyway, so it has to be something you expect), it's when you make async I/O calls on your UI thread because you expect async I/O not to block. The fact that async calls can actually block means that you can't keep your UI responsive simply by making all your I/O async; you have to make sure you never do any I/O on your UI thread. There being no async CreateFile should be the tip-off for this, but still many people don't know

this.

Simon Farnsworth
28 Sep 2011 6:03 AM
#

@Deuplicator:

So, very roughly (as I'm not a Microsoft developer), the reasoning works as follows:

Async I/O avoids the need for threads by exploiting the hardware's queueing. When you call a synchronous I/O call that could complete async, the following happens:

1) The kernel does the work needed to know what commands it needs to send to hardware

2) It flags your thread as "blocked, waiting for I/O" for scheduling purposes

3) It creates a completion that tells the scheduler to unblock your thread

4) It issues the commands (blocking here for as long as it takes for there to be space in the hardware queue) with the completion set up ready to be run by the interrupt handler

5) It enters the scheduler, which finds something to run.

You recover control when the I/O completes.

If you used an async call instead, the completion created in step 3 changes from "tell the scheduler to unblock this thread and reschedule" to "mark this OVERLAPPED as completed and flag this thread as having a completion to handle when it calls back in to find one". As a result, step 5 allows the scheduler to decide to continue running your thread. Remember that because the completion is called by an interrupt handler, it can't do much - it can write flags, and it can kick the scheduler, and that's about it - it can't start a new thread, it can't do complex work, and it can't rearrange your thread's execution to suddenly drop into kernel code unexpectedly.

For calls that go synchronous even when issued via the async I/O mechanism, the kernel has work to do after the hardware completes. The process then looks like:

1) The kernel does the work needed to know what commands it needs to send to hardware

2) It flags your thread as "blocked, waiting for I/O" for scheduling purposes

3) It creates a completion that tells the scheduler to unblock your thread

4) It issues the commands (blocking here for as long as it takes for there to be space in the hardware queue) with the completion set up ready to be run by the interrupt handler

5) It enters the scheduler, which runs another thread.

6) When the I/O completes, the kernel does the clean-up work it needs to do to handle this I/O.

7) It finally returns control to your thread.

Note that there is no way for the kernel to do work on your process's behalf without

either requiring you to cope with the fallout of multiple threads (as it would potentially create a thread for you at step 3, which would be unblocked when the I/O completes so that it can execute step 6 and then signal completion to your thread), or blocking you until it's completed the work it needs to do. Given that the point of async I/O is to get you parallelism without threads, it can't create a thread, so the only remaining option is to block you.

Joe
28 Sep 2011 7:52 AM
#

"We've seen earlier that it's possible for async calls to complete synchronously, even if everything supports asynchronous I/O, so you have to be ready for it one way or another."

That some calls on a overlapped handle can already be completed on return from that call does not imply that "conversions" from async to sync mode do occur. I would only take this as a sign that (for example) all the data requested by a ReadFile call is already present in the buffer cache of the OS and therefore the call can do his work without any wait.

Saying now here that the semantics of API calls in regards of blocking can change on the fly due to inadequate implementation of some detail at some I/O layer deep below this API call (you really cannot know such implementations details as an application programmer) looks flawed.

> [*But if you think about it, the two types of synchronous behavior are identical. They just have different performance characteristics. The I/O manager calls the handler's "StartAsync" function and the handler returns "Done. Oh, and by the way, it's also finished." The deal is that the "Oh, and by the way it's also finished" could be because "Oh, the answer was so easy to calculate I just did it since it was less work than setting up a full async I/O". Or it could be because "Oh, it's too hard for me to set up the async I/O for this, so I'll just do all the work inside my StartAsync function and say 'finished'." -Raymond*]

Simon Farnsworth
28 Sep 2011 8:46 AM
#

@Joe:

The MSDN docs already say that conversions from async to sync can occur, and point you to a Knowledge Base article that details exactly when they occur for specific OSes (i.e. what the implementation details are). Async I/O as documented on MSDN is strictly a performance boost, not a guarantee of non-blocking.

If you're using an API call based on what you would like it to be, not what it's documented to be, well, you're going to get burnt sooner or later.

Deduplicator

28 Sep 2011 10:29 AM
#

@Simon (long post): Ever heard about kernel-mode APCs? There's your way to schedule work after the pure IO is done, without creating new threads, and especially user-mode threads.

@Simon+Gabe: The fact that async is not guaranteed is no reason it should not be or could not be!

@raymond: you don't seriously propose that compression/decompression, encrypting/decrypting or extending a file or some such operation is so complex, it has to be handled on the kernel-mode part of the exact same user-mode thread, and additionally no part of it (the writing part for example) can actually be handed of to the IO-Subsystem to do its thing without supervision by the blocked thread hovering impatiently in the background? And arguing that behavior is the same, whether you instantly get the go ahead or have to wait the year is a bit strange.

Simon (short post): It's always the case that you have to program/design to the spec and test your contraptions as thorouly as possible, but that's neither adequate excuse for designing an api to make shooting off your own foot more likely by introducing unexpected behavior in corner-cases, nor for elliding an api which does 'The Right Thing'TM, by being non-blocking with asynchronous notifications.

Actually, there are These Ways for IO:

1 Blocking Synchronous (Old-Fashioned, Easiest)

2 Non-Blocking Synchronous (Poll or Select) (Poll would burn cycles needlessly)

3 Non-Blocking Asynchronous (Done immediately and/or asynchronous notification)

Windows Currently implements 1 and 2, NOT 3, but (roulette 1 or 3) which looks and sometimes feels like 3, until it bites you in the ass hard, meaning you have to go multithreaded.

Does someone have anything to add to the overview of IO-methodologies, or a correction?

> [*Not sure what you're asking me to say. Do you want me to say "The file system guys are lazy bums"? -Raymond*]

Deduplicator
28 Sep 2011 12:00 PM
#

@Raymond: Well, it's a start to accept that the current state of affairs is a bit suboptimal, as it forces everyone wanting a responsive UI/other Channel into taking out the big multithreading-sledgehammer.

And I actually don't think they are lazy bums, even if their decision for what they implemented might only make sense from the provider-side, eg. making their lives easier, without making all that much sense from the user-side or even the resource-side, as it doesn't allow reaping all the benefits, without using kernel-managed user-mode multi-threading as a massive clunky workaround.

BTW: Did you find a fourth IO-Method? (Not Memmap, please. That would be 1)

> [*I tend to write about practical programming, which means "The goal is to solve a problem. Arguing over whether something was a good idea doesn't help you solve your problem (at least not until time travel is perfected)." -Raymond*]

Simon Farnsworth
29 Sep 2011 1:11 AM
#

@Deduplicator:

I'm sorry; I thought this was "The Old New Thing", Raymond Chen's blog on how things actually are, and why they're imperfect. I didn't realise I'd stumbled into the "redesign Windows to remove its warts" blog.

I therefore assumed that you'd understood that this sort of post is about why something that looks odd now got into that state, and were interested in understanding what sort of historical reasoning would lead Microsoft's engineers to designing an async I/O interface with the specific warts it has, as against some form of "perfect" async I/O interface.

Having implemented such an interface on an embedded system, I sympathise with the warts the Windows async I/O interface has - they've done something simple to implement, that's usable by the intended target audience (server writers), and has one predictable gotcha, as against a myriad of slightly different weird side-effects of things not going to plan

Deduplicator
29 Sep 2011 2:37 AM
#

Yes, this is certainly the old new thing.

But it isn't only about why things came to be as they are and how they really work, but also how to hack around the limitations without quite violating the spec, and sometimes a bit about different alternatives (implemented/in progress/perhaps in the future/discarded) or completely unrelated topics.

That said, this weird side-effect could probably be removed without breaking anything, and I hope it will (some day, some way).

Even though time-travel would be nice, it's not really needed here...

Ben
29 Sep 2011 7:45 AM
#

@Simon Farnsworth: "has one predictable gotcha"

As far as I understand this all, Windows does not follow a predictable, well-documented

concept for supporting asynch I/O by "overlapped". I don't get why you seem to be happy with behavior that you cannot rely on. How do you write your server software, when the asynch behavior can be different on any machine, depending on the disk driver, filesystem provider, filter drivers or the versions of some system DLL you have never heard of?

Someone may want to use your software on exFAT or some other non-NTFS filesystem. If this does not support asynch at all ("converting" everything to sync I/O), your software can hopefully still run without errors, but maybe unusable slow.

Sure, complaining here about this does not change anything, but I take this also as a opportunity to discuss concepts of programming or APIs.

Simon Farnsworth
29 Sep 2011 12:08 PM
#

@Ben:

Writing the server software is not hard, precisely because there is only one gotcha; make sure it is correct if all the async calls turn out synchronous, and make sure it is also correct if they all turn out async. If they all turn out synchronous, you've not gained the performance boost async I/O is supposed to give you. If they all turn out async, you have.

If you need asynchronous behaviour for correctness, you use threading primitives to get you asynchronous behaviour. The OVERLAPPED async I/O API lets you exploit hardware parallelism cheaply, if (and only if) that parallelism is easy for the OS to expose.

Remember that big I/O systems are capable of huge I/O parallelism; I've not dealt with a really big one, but a small EMC SAN will happily permit you to have 60,000 I/O operations outstanding at once, while a cheap SATA disk will only allow you 30-odd operations at once (and IDE disks only let you have one operation outstanding at a time). The point of the async I/O API is to let you write code that scales trivially from 1 operation at a time to 60,000 operations at a time, depending on the hardware the user puts it on.

I'm not sure why "unusable slow" applies specifically to this API, either - a user could choose to run your software on a machine without enough RAM for you, so that it's paging all the time, or in a VM with a tiny timeslice allocated to it, so it has next to no CPU time available. You use async I/O so that if the customer puts your software on a beefy setup, they get high performance, without you having to cope with scaling between 2 threads (one for I/O, one for compute) and 60,000 threads (60 for compute, 59,940 for I/O).

Deduplicator
29 Sep 2011 12:53 PM
#

@Simon:

Well, if any client allways runs into blocking behavior, your thread-number will go through the roof, meaning you measure your performance by thread-switches/Second, the actual I/O-performance the hardware allows is completely beside the point. Real asynchronous

apis provided by kernel-land would avoid that easily.

Much worse, if only part of the requests potentially performed by a client during his session grab and strangle a thread, that would trivially lead to a really-hard-to-find denial-of-service for the service, if not the whole machine. Not only that nobody could connect anymore, but all clients not having entailed their own private thread yet will be starved to death. And those who aren't starved will move slower than any snails I've ever seen. This sudden freezing due to explosion of resource demand could also be trivially avoided using kernel-provided asynchronous apis.

Next, using the software on your pocket-calculator is a strawman if I ever heard one. And where does it say that asynchronous requests are always directly moved to dedicated hardware? Software-managed queues are always used to establish or extend queueing of requests, especially IO.

Simon Farnsworth
30 Sep 2011 4:30 AM
#

@Deduplicator:

Why exactly would my thread number go through the roof? Imagine a trivial fully async implementation based around two threads and two queues; thread 1 puts requests into one queue, and picks up completions from another queue at a time to suit it. Thread two picks up requests from one queue, executes them, and puts completions into the second queue. Voilà, you have non-blocking asynchronous I/O, with only one request outstanding in hardware at a time. If you want more requests in hardware, replace thread 2 with a constrained size pool of worker threads.

The OVERLAPPED API is still useful; it lets you give each thread in your pool a multiplier effect on beefy enough systems - you are still guaranteed that each thread can queue one request into the OS, but if some requests complete asynchronously, you get more than one request queued by each thread. In turn, this means that a tiny thread pool can, under the right conditions, queue millions of requests into hardware that's capable of handling it; compare that to trying to run millions of threads.

Moving the complexity into the kernel doesn't change anything - a kernel async API would provide you with a hidden thread pool behaving in the same way as the implementation I've described. And you can implement it as a library you take with you from project to project - if it was a common requirement, it might even become a library supplied with the Platform SDK.

And the "pocket calculator" strawman is from the "OVERLAPPED is useless" side of the argument - the claim made was that the async I/O API was useless because I could find hardware on which it was always synchronous, and thus would be unusably slow because the application would only be fast enough if the I/O was asynchronous. However, I can make the same argument about any other constrained resource, and I have examples of applications which successfully use this API to gain performance (Microsoft SQL Server), so clearly the API is usable.

I'm also unable to understand your "denial of service" rant - if the async I/O API opens up a denial of service to all users of the API, please explain how I'd use it on SQL Server. If not, it seems to be "if people write buggy code that assumes unlimited resources, things go wrong". That would still apply even if all async I/O operations were guaranteed to be async.

You're also ignoring one more part of the async I/O API - it's permissible for it to block for minutes, then return an asynchronous completion, depending on the resources available on the machine. If that's unacceptable, you need a second thread of execution, and it makes it simpler for if you have to ask for that thread explicitly, rather than having the kernel do some magic multithreading dance underneath you so that you have two threads, but you think you only have one.

Joe
30 Sep 2011 6:35 AM
#

"Moving the complexity into the kernel doesn't change anything - a kernel async API would provide you with a hidden thread pool."

First, it's already within the kernel. In the end, it has to serialize even heavy access to functionality and/or resources between different threads/processes, and therefore needs to use queues (explicit, or implicit via Wait functions) to collect and serve all the outstanding concurrent I/O requests to one resource. (Site note: There are different blocking needs which you cannot all reduce to "hardware parallism": compare "write disk block" to "rename file" to "accept TCP connection").

How it serves this queued-up requests, via worker threads or via interrupt handling or events or mutexes or what else, is really not important for the high-level APIs, like ReadFile/WriteFile/SetEndOfFile and so on.

Second, it would be much easier to use "overlapped", if there would be no weird corner cases, and if every issued overlapped operation would be completed only asynchronous (the completition ports are very handy). It should save you from coding for both cases, the sync and the async completition. It would be much easier if each and every overlapped call be non-blocking: The request is queued up to the kernel, your thread continue, and the result of the operation is signalled later on.

In the end, why code this complexity go into every application (which will get this wrong most of the time)? The overlapped concept should be implemented in the kernel in a coherent way, without special cases and without a strange mix of sync and async completition.