Why you might need additional control over the secret event hiding inside the file object

devblogs.microsoft.com/oldnewthing/20200221-00

February 21, 2020

Raymond Chen

Some time ago, I noted that the `SetFileCompletionNotificationModes` function provides a small amount of additional control over the secret event hiding inside the file object, but I noted that I could not come up with a scenario where you would need to exercise that much control.

The purpose of the `FILE_SKIP_SET_EVENT_ON_HANDLE` flag is to prevent the kernel from messing with the secret event hiding inside the file object. But if you aren't using that secret event anyway, why does it matter what the kernel does with it?

Malcolm Smith explained to me why it matters: It's to avoid contention.

In high-performance scenarios, you may have tons of outstanding I/O operations on a handle. Those operations are all queueing to an I/O completion port. They don't need an explicit event handle, nor do they need to synchronize on the secret handle hiding inside the file object.

The normal behavior at I/O completion on an overlapped handle is that the kernel signals the event provided in the `OVERLAPPED` structure, if present. If the event handle in the `OVERLAPPED` structure is `nullptr`, then the kernel signals the secret event inside the file object.

If you're doing I/O on an overlapped handle, then the secret event inside the file object is useless once you have two outstanding I/O operations on the file handle, because they will both try to use the event and end up confusing each other. You would have to ensure that only one I/O operation is active at a time, which sort of defeats the point of overlapped I/O.

Okay, I can think of one scenario where it's useful: If you are using overlapped I/O solely for its asynchronous behavior and not for the ability to have multiple outstanding I/O at a time. But even then, just create your own event already. Don't rely on the secret event inside the file object.

In the case of overlapped I/O issued on a handle bound to an I/O completion port, you definitely don't care about the secret event hiding inside the file object, and making the kernel set it at completion is just another multithreading bottleneck. The `FILE_SKIP_SET_EVENT_ON_HANDLE` flag lets you tell the kernel to skip that step entirely.

Incorporating the `FILE_SKIP_SET_EVENT_ON_HANDLE` flag into the I/O completion process results in this pseudo-code:

```
SignalEventsWhenOverlappedIoCompleted()
{
  if (hEvent present) {
    SetEvent(hEvent);
  } else if (FILE_SKIP_SET_EVENT_ON_HANDLE is clear) {
    SetEvent(secret_event);
  } else {
    do nothing;
  }
}
```

Now, you might think you could avoid the bottleneck on the secret event hiding inside the file object by passing an event in the `OVERLAPPED` structure. According to the algorithm above, this means that the kernel will set the event in the `OVERLAPPED` structure and ignore the secret event in the file object. Since each I/O has its own event (if you know what's good for you), the `SetEvent(hEvent)` will not experience contention, so it will be fast. It's still annoying having to create an event that you have no use for, but its purpose is to be a decoy so the kernel won't try to set the secret event hiding inside the file object.

Unfortunately, this solution runs into its own bottleneck. When the I/O completes, the I/O manager returns to the original issuing thread in order to set the event,[1] and then queues the completion to the I/O completion port, This introduces an extra thread switch to the I/O operation, as well as additional contention into I/O completion bookkeeping.[2]

The secret event hiding inside the file object is useful only in the case of synchronous I/O. If you're doing asychnronous I/O, all it does is get in your way. The `FILE_SKIP_SET_EVENT_ON_HANDLE` flag lets you move it out of your way.

[1] I suspect the "return to the original issuing thread" is an artifact of the fact that completion callbacks are delivered to the original issuing thread.

[2] The cure ends up being worse than the disease if the I/O originated from an I/O completion port thread, which is highly likely if the handle is associated with an I/O completion port in the first place. I/O completion ports keep track of how many threads are running and how many are blocked, so that they don't oversubscribe the associated thread pool. When the thread is woken to set the event, the I/O completion port updates the bookkeeping to account for an idle thread being woken, and when the thread goes back to sleep after setting the

event, the I/O completion port updates the bookkeeping once again to account for the thread going back to idle. That's two more points of contention on a very busy I/O completion port. So your attempt to remove one contention point on a busy file object turned into the addition of *two* contention points on an even busier I/O completion port!

Raymond Chen

Follow