Ready… cancel… wait for it! (part 1)

devblogs.microsoft.com/oldnewthing/20110202-00

February 2, 2011

Raymond Chen

One of the cardinal rules of the `OVERLAPPED` structure is *the `OVERLAPPED` structure must remain valid until the I/O completes*. The reason is that the `OVERLAPPED` structure is manipulated by address rather than by value.

The word *complete* here has a specific technical meaning. It doesn't mean "must remain valid until you are no longer interested in the result of the I/O." It means that the structure must remain valid until the I/O subsystem has signaled that the I/O operation is finally over, that there is nothing left to do, it has passed on: You have an ex-I/O operation.

Note that an I/O operation can complete successfully, or it can complete unsuccessfully. Completion is not the same as success.

A common mistake when performing overlapped I/O is issuing a cancel and immediately freeing the `OVERLAPPED` structure. For example:

```
// this code is wrong
 HANDLE h = ...; // handle to file opened as FILE_FLAG_OVERLAPPED
 OVERLAPPED o;
 BYTE buffer[1024];
 InitializeOverlapped(&o); // creates the event etc
 if (ReadFile(h, buffer, sizeof(buffer), NULL, &o) ||
     GetLastError() == ERROR_IO_PENDING) {
  if (WaitForSingleObject(o.hEvent, 1000) != WAIT_OBJECT_0) {
   // took longer than 1 second - cancel it and give up
   CancelIo(h);
   return WAIT_TIMEOUT;
  }
  ... use the results ...
 }
 ...
```

The bug here is that after calling `CancelIo`, the function returns without waiting for the `ReadFile` to complete. Returning from the function implicitly frees the automatic variable `o`. When the `ReadFile` finally completes, the I/O system is now writing to stack memory that has been freed and is probably being reused by another function. The result is impossible to debug: First of all, it’s a race condition between your code and the I/O subsystem, and breaking into the debugger *doesn’t stop the I/O subsystem*. If you step through the code, you don’t see the corruption, because the I/O completes *while you’re broken into the debugger*.

Here’s what happens when the program is run outside the debugger:

ReadFile	→	I/O begins
WaitForSingleObject		I/O still in progress
WaitForSingleObject times out		
CancelIo	→	I/O cancellation submitted to device driver
return		
		Device driver was busy reading from the hard drive Device driver receives the cancellation Device driver abandons the rest of the read operation Device driver reports that I/O has been canceled I/O subsystem writes `STATUS_CANCELED` to `OVERLAPPED` structure I/O subsystem queues the completion function (if applicable) I/O subsystem signals the completion event (if applicable) I/O operation is now complete

When the I/O subsystem receives word from the device driver that the cancellation has completed, it performs the usual operations when an I/O operation completes: It updates the `OVERLAPPED` structure with the results of the I/O operation, and notifies whoever wanted to be notified that the I/O is finished.

Notice that when it updates the `OVERLAPPED` structure, it’s updating memory that has already been freed back to the stack, which means that it’s corrupting the stack of whatever function happens to be running right now. (It’s even worse if you happened to catch it while it was in the process of updating the `buffer`!) Since the precise timing of I/O is unpredictable, the program crashes with memory corruption that keeps changing each time it happens.

If you try to debug the program, you get this:

ReadFile	→	I/O begins
WaitForSingleObject		I/O still in progress

WaitForSingleObject times out		
Breakpoint hit on `CancelIo` statement Stops in debugger		
Hit F10 to step over the CancelIo call	→	I/O cancellation submitted to device driver
Breakpoint hit on `return` statement Stops in debugger		
		Device driver was busy reading from the hard drive Device driver receives the cancellation Device driver abandons the rest of the read operation Device driver reports that I/O has been canceled I/O subsystem writes `STATUS_CANCELED` to `OVERLAPPED` structure I/O subsystem queues the completion function (if applicable) I/O subsystem signals the completion event (if applicable) I/O operation is now complete
Look at the `OVERLAPPED` structure in the debugger It says `STATUS_CANCELED`		
Hit F5 to resume execution No memory corruption		

Breaking into the debugger changed the timing of the I/O operation relative to program execution. Now, the I/O completes before the function returns, and consequently there is no memory corruption. You look at the `OVERLAPPED` structure and say, “See? Immediately on return from the `CancelIo` function, the `OVERLAPPED` structure has been updated with the result, and the `buffer` contents are not being written to. It’s safe to free them both now. Therefore, this can’t be the source of my memory corruption bug.”

Except, of course, that it is.

This is even more crazily insidious because the `OVERLAPPED` structure and the `buffer` are updated by the I/O subsystem, which means that it happens *from kernel mode*. This means that write breakpoints set by your debugger won’t fire. Even if you manage to narrow down the corruption to “it happens somewhere in this function”, your breakpoints will never see it as it happens. You're going to see that the value was good, then a little while later, the value was bad, and yet your write breakpoint never fired. You're then going to declare that the world has gone mad and seriously consider a different line of work.

To fix this race condition, you have to delay freeing the `OVERLAPPED` structure and the associated `buffer` until the I/O is complete and anything else that's using them has also given up their claim to it.

```
// took longer than 1 second - cancel it and give up
CancelIo(h);
WaitForSingleObject(o.hEvent, INFINITE); // added
// Alternatively: GetOverlappedResult(h, &o, TRUE);
return WAIT_TIMEOUT;
```

The `WaitForSingleObject` after the `CancelIo` waits for the I/O to complete before finally returning (and implicitly freeing the `OVERLAPPED` structure and the `buffer` on the stack). Better would be to use `GetOverlappedResult` with `bWait = TRUE`, because that also handles the case where the `hEvent` member of the `OVERLAPPED` structure is `NULL`.

Exercise: If you retrieve the completion status after canceling the I/O (either by looking at the `OVERLAPPED` structure directly or by using `GetOverlappedResult`) there's a chance that the overlapped result will be something other than `STATUS_CANCELED` (or `ERROR_CANCELLED` if you prefer Win32 error codes). Explain.

Exercise: If this example had used `ReadFileEx`, the proposed fix would be incomplete. Explain and provide a fix. Answer to come next time, and then we'll look at another version of this same principle.

Raymond Chen

Follow