Functions that return the size of a required buffer generally return upper bounds, not tight bounds

devblogs.microsoft.com/oldnewthing/20240214-00

February 14, 2024

Raymond Chen

There are a number of functions in Windows that are part of a three-phase operation:

1. Request the size of a buffer needed to receive some data.
2. Allocate a buffer of that size.
3. Call the function again with that buffer.

When you ask for the required size of a buffer, it is not uncommon for the function to return a value that larger than the actual value you get from step 3, when you ask for the data to be placed in the buffer. Why is that? Why did the first function lie to me?

Well, in general, you have to be prepared for the second call to return a different size from the first call because of a time-of-check-to-time-of-use (TOCTTOU) race condition. After you request the size of the buffer, the data may change, and when you get around to requesting the data to be placed in the buffer, it's possible that you get a different result size anyway, not because anybody was lying to you, but because the underlying data is different from what it was when you asked the first time.

Given that the caller has to be prepared for the size to change anyway, the "how big of a buffer do I need" call can return an over-estimate of the required size, since that will allow the second call for the data to succeed (assuming the data hasn't changed). And giving an over-estimate is often much easier than giving an exact value.

For example, a call to `GetWindowTextLengthA` can't just call `GetWindowTextLengthW` and assume that the length in bytes is always the same as the length in UTF-16 code units. One of the UTF-16 code units may require two bytes to represent in the ambient 8-bit character set; alternatively, a surrogate pair of UTF-16 code units might collapse down to a single 8-bit character (probably a question mark). Doing the calculations down to the byte would mean allocating temporary memory, reading the window text in UTF-16 into the temporary buffer, then doing the character set conversion to bytes to get the true length, and then freeing the temporary buffer.

This is a lot of wasted work, because the caller is just going to do the same thing!

So don’t be surprised when a “get required buffer size” call returns a value that is larger than necessary. Given the intended usage pattern, all these calls really need to do is give an upper bound on the buffer size, not the precise buffer size.

Bonus chatter: As a specific example, the `RegQueryInfoKey` gives the maximum lengths of all values and subkeys, but those are upper bounds and not tight upper bounds. Character set conversion estimates are not the only thing at play. Recalculating the exact maximum length when the longest value or subkey is deleted would require either a linear search through the items (to find the newest “longest” item) or maintaining a separate index which keeps the values and subkeys sorted by descending length, so that the new “longest item” can be found quickly. The internal registry code doesn’t do either of those things today. It just keeps track of a high water mark. The value you receive might be too big, but it will never be too small.