How to check for errors from SetFilePointer

devblogs.microsoft.com/oldnewthing/20070711-00

July 11, 2007

Raymond Chen

The `SetFilePointer` function reports an error in two different ways, depending on whether you passed `NULL` as the `lpDistanceToMoveHigh` parameter. The documentation in MSDN is correct, but I've discovered that people prefer when I restate the same facts in a different way, so here comes the tabular version of the documentation.

	If `lpDistanceToMoveHigh == NULL`	If `lpDistanceToMoveHigh != NULL`
If success	`retVal != INVALID_SET_FILE_POINTER`	`retVal != INVALID_SET_FILE_POINTER || GetLastError() == ERROR_SUCCESS`
If failed	`retVal == INVALID_SET_FILE_POINTER`	`retVal == INVALID_SET_FILE_POINTER && GetLastError() != ERROR_SUCCESS`

I'd show some sample code, but the documentation in MSDN already contains sample code both for the `lpDistancetoMoveHigh == NULL` case as well as the `lpDistancetoMoveHigh != NULL` case. A common mistake is calling `GetLastError` even if the return value is not `INVALID_SET_FILE_POINTER`. In other words, people ignore the whole `retVal == INVALID_SET_FILE_POINTER` part of the "did the function succeed or fail?" test. Just because `GetLastError()` returned an error code doesn't mean that the `SetFilePointer` function failed. The return value must also have been `INVALID_SET_FILE_POINTER`. I will admit that the documentation in MSDN could be clearer on this point, but the sample code hopefully resolves any lingering ambiguity. But why does `SetFilePointer` use such a wacky way of reporting errors when `lpDistanceToMoveHigh` is non-`NULL`? The MSDN documentation also explains this detail: If the file size is greater than 4GB, then `INVALID_SET_FILE_POINTER` is a valid value for the low-order 32 bits of the file position. For example, if you moved the pointer to position 0x00000001`FFFFFFFF, then `*lpDistanceToMoveHigh` will be set to the high-order 32 bits of the result (1), and the return value is the low-order 32 bits of the result (0xFFFFFFFF, which happens to be the numerical value of `INVALID_SET_FILE_POINTER`). In that case (and only in that case) does the system need to use `SetLastError(ERROR_SUCCESS)` to tell you, "No, that value is

perfectly fine. It's just a coincidence that it happens to be equal to `INVALID_SET_FILE_POINTER`". Why not call `SetLastError(ERROR_SUCCESS)` on all success paths, and not just the ones where the low-order 32 bits of the result happen to be 0xFFFFFFFF? That's just a general convention of Win32: If a function succeeds, it is not required to call `SetLastError(ERROR_SUCESS)`. The success return value tells you that the function succeeded. The exception to this convention is if the return value is ambiguous, as we have here when the low-order 32 bits of the result happen to be 0xFFFFFFFF. You might argue that this was a stupid convention, But what's done is done and until time travel has been perfected, you just have to live with the past. (Mind you, UNIX uses the same convention with the `errno` variable. Only if the previous function call failed is the value of `errno` defined.) Looking back on it, the designers of `SetFilePointer` were being a bit too clever. They tried to merge 32-bit and 64-bit file management into a single function. "It's generic!" The problem with this is that you have to check for errors in two different ways depending on whether you were using the 32-bit variation or the 64-bit variation. Fortunately, the kernel folks realized that their cleverness backfired and they came up with a new function, `SetFilePointerEx`. That function produces a 64-bit value directly, and the return value is a simple `BOOL`, which makes checking for success or failure a snap.

Exercise: What's the deal with the `GetFileSize` function?

Raymond Chen

Follow