Why does GlobalLock max out at 255 locks?

devblogs.microsoft.com/oldnewthing/20240604-00

June 4, 2024

Raymond Chen

Reader Declan wanted to know why the `GlobalLock` function stops incrementing the lock count when it reaches 255.

The short answer is "Because that's all the room available."

The `GlobalLock` and `GlobalUnlock` functions date back to the days of 16-bit Windows. Each time you call `GlobalLock` on a movable memory block, the function increments the lock count on the global handle and returns a pointer to the underlying memory. You can use that pointer to access the memory, and when you're done, you call `GlobalUnlock` to tell the memory manager, "I'm not using that pointer any more. It's okay to move the memory." When there are no outstanding locks, the memory manager is allowed to move the memory as part of defragmentation when it needs to satisfy a memory allocation.

The `GlobalFlags` function returns various pieces of information about a global memory handle, and the lock count is reported in the lower 8 bits. (See `GMEM_LOCKCOUNT`.) This consequently sets the maximum lock count at 255: There is no way to report any higher value.

In 16-bit Windows, once a handle's lock count reached 255, it could never return to zero, since the system doesn't know the actual number of outstanding locks. (Locking 255 times and locking 256 times both produce a lock count of 255.) The global handle was permanently locked.

Reaching a lock count of 255 is unlikely to occur in practice, because you were not supposed to leave memory blocks locked for extended periods of time. If the lock count reached 255, you probably had a lock leak, and the debugging version of Windows broke into the debugger to tell you "GlobalLock: Object usage count overflow."

Now, all this nonsense with locking became irrelevant in 1987 with the introduction of Standard Mode Windows that used the protected mode memory manager capabilities of the then-new 80286 processor. The operating system could move memory around physically without invalidating any outstanding pointers because the pointers were already going through a level of indirection at the hardware layer: The upper 16 bits of a far pointer consist of a *selector*, and the hardware used it as a lookup into a *descriptor table* which told it which physical memory each selector refers to. The operating system could move the memory around physically, and as long as it updated the entry in the descriptor table to point to where the memory got moved to, application pointers were unaffected.

Locking became even more irrelevant in 1990 with the introduction of Enhanced Mode Windows, and finally became obsolete in 1995 when Windows 95 dropped support for Real-Mode and Standard Mode Windows entirely.

Despite being obsolete for decades, 32-bit and 64-bit Windows still support the `GlobalLock` and `GlobalUnlock` functions for backward compatibility. You can allocate memory with the `GMEM_MOVEABLE` [sic] flag, which allocates a handle, and then call `GlobalLock` to lock the handle and produce a pointer, and so on, programming like it's 1984.

There is one difference between how the lock count is managed today, compared to how it was done in 16-bit Windows: In 16-bit Windows, when the lock count reached its maximum, calls to `GlobalUnlock` were ignored, since the system lost track of how many outstanding locks there were, so it just plays it safe and leaves the block locked for the remainder of its life. But today, if you max out the lock count at 255, calls to `GlobalUnlock` will still decrement it. This means that if you lock a block 256 times, it will become movable after only 255 unlocks.

This hasn't caused any problems for 30 years, so I think we dodged a bullet there.