What does LockWindowUpdate do?

devblogs.microsoft.com/oldnewthing/20070219-00

February 19, 2007

Raymond Chen

Poor misunderstood `LockWindowUpdate`.

This is the first in a series on `LockWindowUpdate`, what it does, what it's for and (perhaps most important) what it's not for.

What `LockWindowUpdate` does is pretty simple. When a window is locked, all attempt to draw into it or its children fail. Instead of drawing, the window manager remembers which parts of the window the application tried to draw into, and when the window is unlocked, those areas are invalidated so that the application gets another `WM_PAINT` message, thereby bringing the screen contents back in sync with what the application believed to be on the screen.

This "keep track of what the application tried to draw while Condition X was in effect, and invalidate it when Condition X no longer hold" behavior you've seen already in another guise: `CS_SAVEBITS`. In this sense, `LockWindowUpdate` does the same bookkeeping that would occur if you had covered the locked window with a `CS_SAVEBITS` window, except that it doesn't save any bits.

The documentation explicitly calls out that only one window (per desktop, of course) can be locked at a time, but this is implied by the function prototype. If two windows could be locked at once, it would be impossible to use `LockWindowUpdate` reliably. What would happen if you did this:

```
LockWindowUpdate(hwndA); // locks window A
LockWindowUpdate(hwndB); // also locks window B
LockWindowUpdate(NULL); // ???
```

What does that third call to `LockWindowUpdate` do? Does it unlock all the windows? Or just window A? Or just window B? Whatever your answer, it would make it impossible for the following code to use `LockWindowUpdate` reliably:

```
void BeginOperationA()
{
 LockWindowUpdate(hwndA);
 ...
}
void EndOperationA()
{
 ...
 LockWindowUpdate(NULL);
}
void BeginOperationB()
{
 LockWindowUpdate(hwndB);
 ...
}
void EndOperationB()
{
 ...
 LockWindowUpdate(NULL);
}
```

Imagine that the `BeginOperation` functions started some operation that was triggered by asynchronous activity. For example, suppose operation A is drawing drag/drop feedback, so it begins when the mouse goes down and ends when the mouse is released.

Now suppose operation B finishes while a drag/drop is still in progress. Then `EndOperationB` will clean up operation B and call `LockWindowUpdate(NULL)`. If you propose that that should unlock all windows, then you've just ruined operation A, which expects that `hwndA` still be locked. Similarly, if you argue that it should unlock only `hwndA`, then only only is operation A ruined, but so too is operation B (since `hwndB` is still locked even though the operation is complete). On the other hand, if you propose that `LockWindowUpdate(NULL)` should unlock `hwndB`, then consider the case where operation A completes first.

If `LockWindowUpdate` were able to lock more than one window at a time, then the function prototype would have to have been changed so that the unlock operation knows which window is being unlocked. There are many ways this could have been done. For example, a new parameter could have been added or a separate function created.

```
// Method A - new parameter
// fLock = TRUE to lock, FALSE to unlock
BOOL LockWindowUpdate(HWND hwnd, BOOL fLock);
// Method B - separate function
BOOL LockWindowUpdate(HWND hwnd);
BOOL UnlockWindowUpdate(HWND hwnd);
```

But neither of these is the case. The `LockWindowUpdate` function locks only one window at a time. And the reason for this will become more clear as we learn what `LockWindowUpdate` is for.

Raymond Chen

Follow