What does INIT_ONCE_CTX_RESERVED_BITS mean?

devblogs.microsoft.com/oldnewthing/20120420-00

April 20, 2012

Raymond Chen

Windows Vista adds the One-Time Initialization family of functions which address a common coding pattern: I want a specific chunk of code to run exactly once, even in the face of multiple calls from different threads. There are many implementations of this pattern, such as the infamous double-checked lock. The double-checked lock is very easy to get wrong, due to memory ordering and race conditions, so the kernel folks decided to write it for you.

The straightforward way of using a one-time-initialization object is to have it protect the initialization of some other object. For example, you might have it protect a static object:

```
INIT_ONCE GizmoInitOnce = INIT_ONCE_STATIC_INIT;
Gizmo ProtectedGizmo;
BOOL CALLBACK InitGizmoOnce(
    PINIT_ONCE InitOnce,
    PVOID Parameter,
    PVOID *Context)
{
    Gizmo *pGizmo = reinterpret_cast<Gizmo*>(Parameter);
    pGizmo->Initialize();
    return TRUE;
}
SomeFunction(...)
{
    // Initialize ProtectedGizmo if not already initialized
    InitOnceExecuteOnce(&GizmoInitOnce,
                        InitGizmoOnce,
                        &ProtectedGizmo,
                        NULL);
    // At this point, ProtectedGizmo has been initialized
    ProtectedGizmo.Something();
    ...
}
```

Or you might have it protect a dynamic object:

```
class Widget
{
    Widget()
    {
        InitOnceInitialize(&m_InitOnce);
    }
    void Initialize();
    ...
    static BOOL CALLBACK InitWidgetOnce(
        PINIT_ONCE InitOnce,
        PVOID Parameter,
        PVOID *Context)
    {
        Widget *pWidget = reinterpret_cast<Widget*>(Parameter);
        pWidget->Initialize();
        return TRUE;
    }
    SomeMethod(...)
    {
        // Initialize ourselves if not already initialized
        InitOnceExecuteOnce(&InitWidgetOnce,
                            this,
                            NULL);
        // At this point, we have been initialized
        ... some other stuff ...
    }
}
```

But it so happens that you can also have the `INIT_ONCE` object protect *itself*.

You see, once the `INIT_ONCE` object has entered the "initialization complete" state, the one-time initialization code only needs a few bits of state. The other bits are unused, so the kernel folks figured, "Well, since we're not using them, maybe the application wants to use them."

That's where `INIT_ONCE_CTX_RESERVED_BITS` comes in. The `INIT_ONCE_CTX_RESERVED_BITS` value is the number of bits that the one-time initialization code uses after initialization is complete; the other bits are free for you to use yourself. The value of `INIT_ONCE_CTX_RESERVED_BITS` is 2, which means that you can store any value that's a multiple of 4. If it's a pointer, then the pointer must be `DWORD`-aligned or better. This requirement is usually easy to meet because heap-allocated objects satisfy it, and the pointer you want to store is usually a pointer to a heap-allocated object. As noted some time ago, kernel object handles are also multiples of four, so those can also be safely stored inside the `INIT_ONCE` object. (On the other hand, USER and GDI handles are *not* guaranteed to be multiples of four, so you cannot use this trick to store those types of handles.)

Here's an example. First, the code which uses the traditional method of having the `INIT_ONCE` structure protect another variable:

```
// using the static object pattern for simplicity
INIT_ONCE PathInitOnce = INIT_ONCE_STATIC_INIT;
LPWSTR PathToDatabase = NULL;
BOOL CALLBACK InitPathOnce(
    PINIT_ONCE InitOnce,
    PVOID Parameter,
    PVOID *Context)
{
    LPWSTR Path = (LPWSTR)LocalAlloc(LMEM_FIXED, ...);
    if (Path == NULL) return FALSE;
    ... get the path in Path...
    PathToDatabase = Path;
    return TRUE;
}
SomeFunction(...)
{
    // Get the database path (initializing if necessary)
    if (!InitOnceExecuteOnce(&PathInitOnce,
                             InitPathOnce,
                             NULL,
                             NULL)) {
        return FALSE; // couldn't get the path for some reason
    }
    // The "PathToDatabase" variable now contains the path
    // computed by InitPathOnce.
    OtherFunction(PathToDatabase);
    ...
}
```

Since the object being protected is pointer-sized and satisfies the necessary alignment constraints, we can merge it into the `INIT_ONCE` structure.

```
INIT_ONCE PathInitOnce = INIT_ONCE_STATIC_INIT;
BOOL CALLBACK InitPathOnce(
    PINIT_ONCE InitOnce,
    PVOID Parameter,
    PVOID *Context)
{
    LPWSTR Path = (LPWSTR)LocalAlloc(LMEM_FIXED, ...);
    if (Path == NULL) return FALSE;
    ... get the path in Path...
    *Context = Path;
    return TRUE;
}
SomeFunction(...)
{
    LPWSTR PathToDatabase;
    // Get the database path (initializing if necessary)
    if (!InitOnceExecuteOnce(&PathInitOnce,
                             InitPathOnce,
                             NULL,
                             &PathToDatabase)) {
        return FALSE; // couldn't get the path for some reason
    }
    // The "PathToDatabase" variable now contains the path
    // computed by InitPathOnce.
    OtherFunction(PathToDatabase);
    ...
}
```

This may seem like a bunch of extra work to save four bytes (or eight bytes on 64-bit Windows), but if you use the asynchronous initialization model, then you have no choice but to use context-based initialization, as we learned when we tried to write our own lock-free one-time initialization code.

Raymond Chen

Follow