What did MakeProcInstance do?

devblogs.microsoft.com/oldnewthing/20080207-00

February 7, 2008

Raymond Chen

`MakeProcInstance` doesn't do anything.

```
#define MakeProcInstance(lpProc,hInstance) (lpProc)
```

What's the point of a macro that doesn't do anything?

It did something back in 16-bit Windows.

Recall that in 16-bit Windows, the `HINSTANCE` was the mechanism for identifying a data segment; i.e., a bunch of memory that represents the set of variables in use by a module. If you had two copies of Notepad running, there was one copy of the code but two sets of variables (one for each copy). It is the second set of variables that establishes the second copy of Notepad.

When you set up a callback function, such as a window procedure, the callback function needs to know which set of variables it's being called for. For example, if one copy of Notepad calls `EnumFonts` and passes a callback function, the function needs to know which copy of Notepad it is running in so that it can access the correct set of variables. That's what the `MakeProcInstance` function was for.

The parameters to `MakeProcInstance` are a function pointer and an instance handle. the `MakeProcInstance` function generated code on the fly which set the data segment register equal to the instance handle and then jumped to the original function pointer. The return value of `MakeProcInstance` is a pointer to that dynamically-generated code fragment (known as a *thunk*), and you used that code fragment as the function pointer whenever you needed another function to call you back. That way, when your function was called, its variables were properly set up. When you no longer needed the code fragment, you freed it with the `FreeProcInstance` function.

Those who have worked with ATL have seen this sort of code fragment generation already in the `CStdCallThunk` class. The operation is entirely analogous with `MakeProcInstance`. You initialize the `CStdCallThunk` with a function pointer and a `this` parameter, and it

generates code on the fly which converts a static function into a C++ member function by setting the `this` pointer before calling the function you used to initialize the thunk.

The creation of these code fragments on 16-bit Windows had to be done by the kernel because the 8086 processor did not have a memory management unit. There was no indirection through a translation table; all addresses were physical. As a result, if the memory manager had to move memory around, it also had to know where all the references to the moved memory were kept so it can update the pointers. If a data segment moved, the kernel had to go fix up all the `MakeProcInstance` thunks so that they used the new instance handle instead of the old one.

It was Michael Geary who discovered that all this `MakeProcInstance` work was unnecessary. If the callback function resided in a DLL, then the function could hard-code its instance handle and just load it at the start of the function; this technique ultimately became known as *__loadds*. Since DLLs were single-instance, the DLL already knew which set of variables it was supposed to use since there was only one set of DLL variables to begin with! (Of course, the hard-coded value had to be recorded as a fix-up since the instance handle is determined at run time. Plus the kernel needed to know which values to update if the instance handle changed values.) On the other hand, if the callback function resided in an executable, then it could obtain its instance handle from the stack selector; this technique ultimately became known as *__export*. Each program ran on a single stack (no multi-threading here), and the stack, data segment, and local heap all resided in the same selector by convention. And in a strange bit of coming full circle which I discovered as I wrote up this reminiscence, Michael Geary's copy of the original readme for his FixDS program that brought this technique to the public contains an introduction which links back to me…

Raymond Chen

Follow